# Abstract


Title of dissertation: TOWARDS A UNIFIED THEORY OF TIMED AUTOMATA

Peter Christopher Fontana,
Doctor of Philosophy, 2014

Dissertation directed by: Professor Rance Cleaveland
Department of Computer Science

Timed automata are finite-state machines augmented with special clock variables that reflect the advancement of time. Able to both capture real-time behavior and be verified algorithmically (model-checked), timed automata are used to model real-time systems. These observations have led to the development of several timed-automata verification tools that have been successfully applied to the analysis of a number of different systems; however, the practical utility of timed automata is undermined by the theories underlying different tools differing in subtle but important ways. Since algorithmic results that hold for the variant used by one tool may not apply to another variant, this complicates the application of different tools to different models. The thesis of this dissertation is this: the theory of timed automata can be unified, and a practical unified approach to timed-automata model checking can be built around the paradigm of proof search.

First, this dissertation establishes the mutual expressivity of timed automata variants, thereby providing precise characterizations of when theoretical results of one variant apply to other variants. Second, it proves powerful expressive properties about different logics for timed behavior, and as a result, enlarges the set of verifiable properties. Third, it discusses an implementation of a verification tool for an expressive fixpoint-based logic, demonstrating an application of this

newly developed theory. The tool is based on a proof-search paradigm; verifying timed automata involves constructing proofs using proof rules that enable verification problems to be translated into subproblems that must be solved. The tool's performance is optimized by using derived proof rules, thereby providing a theoretically sound basis for faster model checking. Last, this dissertation utilizes the proofs generated during verification to gain additional information about the vacuous satisfaction of certain formulae: whether the automaton satisfied a formula by never satisfying certain premises of that specification. This extra information is often obtained without significantly decreasing the verifier's performance.

# TOWARDS A UNIFIED THEORY OF TIMED AUTOMATA

by

Peter Christopher Fontana

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Advisory Committee:
Professor Rance Cleaveland, Chair
Professor Steven Marcus, Dean's Representative
Professor William Gasarch,
Professor Michael Hicks,
Professor Samir Khuller

# Preface

Welcome to my dissertation. This preface describes the purpose of this dissertation as well as some background that might be helpful to understand some of the details in the dissertation.

## Dissertation Structure

This dissertation provides motivation for my research, a description of relevant previous work as well as some background review and a discussion of my completed work. The motivation of the work is discussed Chapter 1 (Introduction). Chapter 2 (Background and Related Work) contains some background definitions and a summary of relevant related work. Some details and definitions are in discussed in the relevant chapters of the dissertation. The scope of the dissertation's contribution are split into the next four chapters (3, 4, 5, and 6), one chapter per contribution area. The contribution of Chapter 3 is a formalization of the timed automata model, reconciling different variants throughout the literature. The contribution of Chapter 4 is expressiveness proofs for a timed mu-calculus. The contribution of Chapter 5 is an extended implementation of a tool to model check properties in this timed my-calculus. The contribution of Chapter 6 is to gain additional information from the model checker while verifying formulas, leveraging that information to determine if a formula is satisfied without its premises ever being satisfied. For a detailed outline of this dissertation's material, please see the *Table of Contents* of this dissertation.

## Intended Audience

This dissertation is geared towards a broad technical audience. Its aim is that the Abstract is accessible to any researcher in any area. However, in order to better aid researchers in my field, some technical terms have been added in the Introduction. The intention is that while the Introduction can be understood without knowing these terms, knowing these terms will better allow a researcher to understand the details of the contributions.

The aim is to make the rest of the dissertation accessible to any researcher with

the mathematical maturity of 1-2 undergraduate proof-based mathematics courses and the programming knowledge of 1-2 computer science courses. Background that will help the understanding of this dissertation include:

- **Mathematical proof:** understanding proofs will be helpful when following these proofs and digesting some of the more specialized mathematics such as the logic expressivity results. Having the level of proof knowledge of a discrete math class will be helpful; two texts for discrete math courses are Epp [71], Scheinerman [142].

- **Programming:** Knowledge of programming concepts and data structures such as loops, functions, arrays, linked-lists will be helpful. Also knowing pointers may be helpful.

- **Graphs and automata:** Knowledge of **finite automata** is helpful; especially knowledge of **states**. Also, knowing what a **graph** is (here we mean a **graph** $G = (V, E)$), a directed vs. undirected graphs, and graph terms such as components and shortest paths will be helpful.

- **Logic and model checking:** Having some background in logic will be helpful. This includes understanding what the **model checking** problem is will be helpful. Also understanding temporal logics used in untimed systems such as CTL (Computation Tree Logic) will be helpful.

- **Lattices and fixpoints:** Some understanding of the mathematical construct of a **lattice** and a **complete lattice** $L = (S, \preceq, \bigcup, \bigcap)$, as well as the theory of **fixpoints** on lattices will be helpful.

- **Timed automata:** Occasionally a part of this dissertation will aim itself to fellow timed automata researchers who have an understanding of timed automata, clock zones, and region equivalence.

In order to give technical details, some parts of the dissertation give technical details aimed at the background of model checking researchers, sometimes assuming knowledge from the background areas given above.

## Acknowledgements

I would like to thank my Ph.D. Advisor: Professor Rance Cleaveland. I have learned many things from him, both directly through his advice and indirectly through working with him, which include the research process, motivating a paper, the flow of research, and how to write up results. From him I have also gained insights on how to be patient with others, how to use simple examples to illustrate points, how to give presentations, how to identify the relevant contributions of interest, and how to interact with others to bring the best out of them. Additionally, I am grateful for his patience and flexibility throughout the Ph.D. process.

I would like to thank the remaining members of my Ph.D. committee: Professor William Gasarch, Professor Michael Hicks, Professor Samir Khuller, and Professor Steven Marcus. I am grateful for their insights and feedback, which improved my dissertation. I am also grateful for their support throughout the Ph.D. process.

I would like to thank the National Science Foundation for its research funding.

I would like to thank my collaborators in the Communications Department at UMD: Professor Andrew Wolvin and Steven D. Cohen. Working with them has allowed me to grow as a researcher, improve my communication skills, and gain a better understanding "interdisciplinary."

I would like to thank the staff of the UMD Computer Science department, who include: Brandi Adams, Fatima Bangura, Brenda Chick, Felicia Chelliah, Jodie Gray, Adelaide Findlay, JoAnn Simms, and Jennifer Story. I thank them for all of their good work; they have helped make my experience in the department a more pleasant one.

I would like to thank various people whom I met in the University of Maryland

Gamer Symphony Orchestra (GSO), who include: Chris Apple, Greg Cox, Rob Garner, Sam Kretschmer, Kerry Leonard, and Katie Noble. Each of these people has contributed to my intellectual, musical, or social growth in a different way. I am grateful to have been influenced by them.

I would like to thank my friends, colleagues, and acquaintances, who include: Zamira Daw, Cody Dunne, James Ferlez, Sam Huang, Michael Lam, Christoph Schulze, Bhaskar Ramasubramanian, and Kent Wills. They have warmed up the research atmosphere and provided insightful and enjoyable conversations.

I would like to thank my best friend: Craig Greenberg. He has provided me guidance, advice, and support throughout the Ph.D. process. I have applied his insights throughout the entire Ph.D. process and utilized his friendship to grow as a person as well as a researcher. To say that he is a friend is an understatement; to me, he is family.

I would like to thank my family. First, I would like to thank my father, Bill. His insights on the working world, the research process, and presentations have permeated throughout my research career, and I am also grateful for his moral support. Next, I would like to thank my mother, Carol. Her gentleness and love encouraged me to find hope even during hard points in the process, and her constant moral support was essential for me to complete this task. Last, I would like to thank my brother, Matthew. His technical insights were invaluable, as well as his moral support; having a supportive family member who helped me grow as a researcher and a person while growing with him made him an indispensable ally throughout the Ph.D. process.

I would like to thank God; God has provided me with many blessings throughout the Ph.D. process.

In conclusion, I am grateful to everyone for so much.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Timed automata are models of real-time systems; they are used to precisely describe a system's behavior, so that desired properties of system executions may be investigated. In many cases these properties may be checked automatically, and these decidability results prompted a variety of groups to develop so-called model checkers for classes of formulas and timed automata (see Section 2.10.3). However, the use of timed automata has developed faster than its foundational theory. This creates some problems because the correctness of model checkers depend on this theory and the correctness proofs it enables. Specifically, different variants of timed automata are used in different sources. Some are equivalent; others are not. Formalizing expressiveness of temporal logics, including timed modal mu-calculi is not yet complete. By advancing the theory of timed automata further to closer meet practice, we will be able to model systems more cleanly and model check even more powerful properties. The **thesis of my dissertation** is this.

**Theorem 1.0.1** (**Thesis of dissertation**)**.** The theory of timed automata can be unified, and a practical unified approach to timed-automata model checking can be built around the paradigm of proof search.

## 1.1 The Choice of Timed Automata

In many cases, software designers wish to be able to establish that properties hold over all executions of a program. For some classes of models, researchers have developed algorithms for answering such verification questions. This is **model checking**. In order to describe systems or programs, researchers abstract the program as a **model**. In order to specify properties, researchers developed **logics** and express properties as logical formulas supported by those logics. Tools are then developed to verify these properties. In these instances, the user specifies a model, and properties it wants the model to satisfy. By showing that the model does not satisfy a desired property, the tool has identified an bug or *error* in the model.

Model checking started with finite automata as models and with various properties over untimed logics, such as Computation Tree Logic (CTL) (see Clarke et al. [54]), which include safety ("always") and liveness ("inevitable") properties. While the applicability of these models may be limited, their computational tractability is highly desirable.

One useful extension to the original theory is to support real-time constraints, both in the models and the specifications. One such way to model real-time constraints is with timed automata (see Alur and Dill [7]). Timed automata extend finite state machines with clocks that model the passage of time.

Clocks in timed automata may be viewed as restricted stopwatches. The watches all start at 0 and elapse time at the same rate. However, the watches are broken: they cannot be stopped. One can only do two things with these watches: read the current values on the stopwatches, or reset any number of the watches to 0. An example is in Figure 1.1.

To support timing constraints in the specifications, the untimed logic CTL was extended to Timed CTL (TCTL, see Alur et al. [9, 12]). (Other timed logics have also

been considered.) While verifying TCTL properties over timed automata becomes harder than verifying CTL properties over finite-state machines, model checking TCTL properties is still decidable. While timed automata are less expressive than other formalisms for real time, they afford enough capability to capture many timed systems, and they are one of the most tractable models that supports real time constraints. If one can specify a problem in terms of timed automata, one can leverage these tools to verify desirable properties.

*Remark* 1.1.1 (Fixing the stopwatches). If one were to fix the stopwatches so that the watches can stop, these automata are called stopwatch automata [89, 91]. While some stopwatch automata can be converted to timed automata [91], some cannot. This follows because reachability in stopwatch automata is undecidable [91].

## 1.2 Improving Timed Automata Model Checking

Timed automata model checking tool papers often have claims similar to the following:

> "We implement a faster timed automata model checker."

Given that model checking claims **100% correctness** when it determines if a program has a desired property, its correctness needs to be guaranteed. If these components of the claim are in question, then it is not clear if this tool meets the needs of the reader of the paper. Can that tool support one's model? Can it express the properties one wants? What information does one get from the model checker?

For instance, if a tool supports a model of timed automata different than the model defined in theory papers, it is not clear whether theoretical algorithms and their correctness results can be applied. Additionally, because a property can be

specified different ways with different logics, if a tool uses a different logic than the logic used in theory papers, it may not be clear if the property given to the tool is subtly different from the intended property written in another logic. These consequences make it unclear if the tool is 100% correct. Also, when comparing model checkers, a model checker being faster may not be relevant or comparable if the model, the model checking output, or the information produced differs.

This dissertation examines three various aspects of timed automata model checking: timed automata, timed logics, and model checker output. This dissertation discusses each of these three aspects and improves the state of the art of each of these three aspects.

### 1.2.1 Aspect 1: Timed Automata

The first aspect involves the theory of timed automata. In many timed automata papers, the models considered differ subtly, but importantly, from each other.

Timed automata were first defined in Alur and Dill [6] (in that paper they are called timed Büchi automata). This model is sometimes referred to as the "Alur-Dill" model, referencing the authors of the paper. (Alur and Dill [6] is a conference paper, and the updated journal paper is Alur and Dill [7], which also updates material presented in Alur et al. [10]). We given an example of a timed automaton defined in this model in Figure 1.1. This timed automaton is a model of a train for the Generalized Railroad Crossing (GRC) protocol (see Alur et al. [11], Heitmeyer and Lynch [84], Heitmeyer et al. [85]). The GRC protocol models the situation where a train on a railroad is crossing a road that cars drive on. The model has various trains that can cross the road, a gate, and a controller. When a train approaches, the control tells the gate to lower, and when all trains leave, the controller raises the gate. This models the interaction where the gate is down when trains are crossing the road. Notice that this model has the three nodes 0: far,

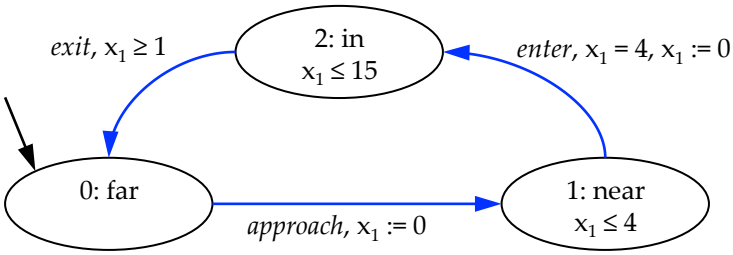


**Figure 1.1:** A timed automaton of the Alur-Dill model.

1: near, and 2: in like nodes in an untimed automaton, which represent when the train is far away from the road, is approaching the road, and as in the segment of the road. Additionally, the model has a clock $x_1$ which advances continuously.

However, let us consider models used in various tools. While the model is still considered a "timed automaton," there are some striking differences:

- **Use of variables:** The Alur-Dill model uses locations and no variables, but various tools often use variables and variable assignments.
- **Clock assignments:** In various tools, clocks are not just reset, they are assigned to the value of another clock.
- **Invariant specification:** Tools specify invariants for multiple locations at a once.
- **Initial states:** Some tools allow additional initial states. Usually, clocks have the initial value 0, but some tools allow the initial values of clocks to be user-defined.

It is not obvious that the Alur-Dill model and the models used by the various tools are "the same." In fact, some of these modeling differences are equivalent (syntactic sugar) but others are not. Some variants claimed as extensions are equivalent and other variants claimed as equivalent may be extensions.

In order to show that a tool does indeed model check timed automata, one needs to show that the model supported by the tool is equivalent to or at least as strong as the timed automata model defined in the literature. Furthermore, if the model checker uses correctness results from the theory, then the theoretical model needs to be at least as strong as the model used in the tool. If a model checker verifies a slightly different model, it may not be wise to judge the tool as better purely because it is faster.

We give formalizations for many of these variants used in the example above as well as the formalizations of the equivalences in Chapter 3. We also give a precise definition of equivalence. In some cases the equivalence is isomorphism (see Definition 3.1.2); in other cases the equivalence is weaker, but at least as strong as timed bisimulation [74].

### 1.2.2 Aspect 2: Timed Logics

The key question asked is: **what properties can a tool model check?** Different tools use different logics to specify properties and can model check different fragments of these logics. While this is usually specified in the work, theoretical results equating expressive equivalence do not always exist. Different tools can check various properties:

- Some tools mention that they can check safety properties, liveness properties, or both without giving a logic. Examples include an unnamed tool in Ehlers et al. [67].
- Other tools model check fragments of TCTL (Timed Computation Tree Logic, see Section 4.1). Examples include UPPAAL [23] and REDLIB [157].
- Other tools model check the alternation-free fragment of the timed modal-mu calculus (see Section 4.2). Examples include CMC [107] and CBWB-RT

[73, 168].

Section 2.10.3 gives a complete list of tools referenced in this dissertation. With these tools, we want to specify properties and determine properties of our model. In order to express properties in a way that a computer can understand and reason with, we use logics. These timed logics specify properties of timed automata. Some of these timed logics are:

- TCTL (Timed Computation Tree Logic). This logic has been extensively studied and is a baseline logic for model checking real-time systems.
- The timed modal mu-calculus $T_\mu$. This logic has been studied by Henzinger et al. [87, 88] and contains extensive theoretical work, but some practical work is missing. Specifically, there is no practical way of expressing an unbounded liveness property (it requires the user to guess a bound).
- The timed modal mu-calculus $L_{\nu,\mu}$. The logic $L_{\nu,\mu}$ and its greatest fixpoint fragment $L_\nu$ have been defined, but limited expressiveness proofs were provided for those logics.
- The timed modal mu-calculus $L^{rel}_{\nu,\mu}$. This is an extension of $L_{\nu,\mu}$ that extends the logic with relativized operators.

Given the ease for humans to write down desirable properties in TCTL, TCTL is widely used. Although the logics $L_{\nu,\mu}$ and $L^{rel}_{\nu,\mu}$ show promise by being able to encode many properties and be model-checked reasonably efficiently, they lack expressivity results. For instance, there is no formal proof published on how to write a safety or liveness property (easily written in TCTL) in $L^{rel}_{\nu,\mu}$ (some papers have been published that claim to have tools that check these safety and liveness properties). For instance, $L_{\nu,\mu}$ model checkers exist, (these include CMC [107] and

CWB-RT [73, 168]) but what properties can they check? Can we write a safety property? How about a TCTL property? What about $L^{rel}_{\nu,\mu}$?

This dissertation strengthens the connections between these logics compares their expressiveness in Chapter 4.

### 1.2.3 Aspect 3: Model Checker Output

For the model checking problem, the typical timed automata model checker takes in a timed automaton and a property, and gives a yes or no answer on whether the timed automaton satisfies the property. However, sometimes we can get some additional information from a model checker, such as:

- **Counterexamples.** In certain properties, such as safety properties, if the model does not satisfy the property then the tool can sometimes give a trace of the model that violates the property. This trace is sometimes called a **counterexample**.
- **Vacuous Satisfaction.** Consider an if-then property. If the if-then property is true because the if clause is always false, the property is **vacuously satisfied** [20]. A model checker might be able to tell if a formula is satisfied vacuously or satisfied non-vacuously.

When comparing model checkers, the information it outputs can enhance the experience and make up for some delay in performance.

While many timed automata model checkers only give yes or no answers, sometimes with counterexamples, our timed automata model checker also produces a **proof** of its answer. This proof contains useful information that can be used to enrich the output that the model checker provides. With this proof, we can extend the research in vacuity in untimed systems to timed systems, being the first to understand vacuity on timed systems.

We discuss how our model checker produces a proof in Chapter 5, and we discuss how we leverage this proof to get additional information on vacuous satisfaction in Chapter 6.

## 1.3 These Errors Don't Happen in Practice...Do They?

In the previous sections this dissertation discusses how the theory may differ from practice, suggesting that subtly inconsistencies in the theory can result in errors in model checking tools, and that these errors may appear when the tool is used in practice.

The first thought that may come to mind is "these errors don't happen in practice." However, problems of this sort **have** occurred **in practice**. One influential example is due to the work of Bouyer [35, 36]. One notational variant of a timed automaton used in tools is the allowing of clock constraints comparing clock differences. When model checking was proven correct in Alur et al. [9, 12], this version, the typical Alur-Dill version, did not allow clock differences. However, various tools used these clock differences. To insure that the model checking tool terminated after a finite amount of time, it would use a widening algorithm. Without clock differences in constraints, this widening algorithm is correct. While assumed to be correct (and used in tools) for over 5 years, in Bouyer [35, 36] it was proven to be incorrect when clock differences are allowed and the automaton had 4 or more clocks! As a result, model-checking tools used in practice **did not have the correctness assumed of a model checker.**

Hence, a more powerful widening method had to be invented. Two such methods (which are correct but **much** more complex) are in Bengtsson and Yi [27], Bouyer et al. [43]. However, no theoretical work has yet been done on the expressive power of these two variants. The only theoretical work (which was done

prior to this discovery) is a statement and conversion between these two variants without a good definition describing the kind of equivalence the conversion provides. This statement and conversion was done in Bérard et al. [28].

## 1.4 Contributions

The contributions of my research in my dissertation are as follows.

- It provides equivalence conversions and proofs for many of the timed automata variants used throughout the literature (Chapter 3).
- It provides additional expressivity proofs for $L_{\nu,\mu}$ and $L^{rel}_{\nu,\mu}$, including expressions for all TCTL formulas in $L^{rel}_{\nu,\mu}$ (Chapter 4).
- It completes the implementation of the Predicate Equation System (PES) model-checking engine for alternation-free $L_{\nu,\mu}$ formulas on timed automata, started by Zhang [165], Zhang and Cleaveland [167], as well as extend the implementation to model-check alternation-free $L^{rel}_{\nu,\mu}$ formulas (Chapter 5).
- It extends the research of vacuity checking in the untimed setting to the timed setting by leveraging proofs to detect vacuous subformulas as well as enhances the model-checking tool to determine if any formula is vacuously satisfied by the proof generated by the tool. (Chapter 6).

# Chapter 2

# Background and Related Work

This chapter both reviews relevant background knowledge for understanding this proposal as well as discussing previous work and the framework it has established. Review of background will discuss the model checking problem as well as additional material providing context and a framework to build up, which subsequent chapters of this proposal do. This framework includes the definition of timed automata. After providing crucial definitions for understanding this work, this dissertation continues by discussing previous work, providing citations to references.

## 2.1 Context: Logics and Model Checking

In this section we review the concept of model checking. For formal definitions of these terms, see a logic book such as Enderton [70] or a model checking book such as Baier and Katoen [17], Clarke et al. [55].

### 2.1.1 Logics and Models

To reason about objects, we use a *logic*, or a mathematical way to both write down (express) properties and to reason about them. Different logics can represent different formulas, and different logics write them down differently. Each property in a logic is a *formula*.

Furthermore, we wish to reason about objects. When reasoning about objects, we will use a collection of objects or a collections of *models*, where each object is a *model*. Often a *model* is an object, but in another sense it can be a particular world.

**Definition 2.1.1** ($M$ **satisfies** $\phi$ **(**$M \models \phi$ **or** $M \in [\![\phi]\!]$**).** Let $M$ be a model from a collection of models and $\phi$ be a formula in logic $L$. if $M$ has property $\phi$, we say that $M$ *satisfies* $\phi$, notated as $M \models \phi$. If $M$ does not satisfy $\phi$, that is notated as $M \not\models \phi$. For notational purposes, $[\![\phi]\!] = \{M \mid M \models \phi\}$; hence $M \models \phi$ iff $M \in [\![\phi]\!]$. ■

### 2.1.2 Model Checking

Now we can in broad terms define the **model checking problem**.

**Definition 2.1.2** (**Model checking problem**)**.** Given a model $M$ and a property $\phi$ specified in a logic $L$, the *model checking problem* is the following problem: does $M$ satisfy property $\phi$ (does $M \models \phi$)? ■

In this dissertation, a program is often abstracted to a model $M$. Using a spe-

cific logic $L$, write properties down that we want $M$ to have (or not have). By solving the model-checking problem, we can determine if $M$ has a desired property. If we discover that $M$ does not have a property we intended it to have, we have identified an error, either in the encoding of $M$ or the design of the original program or system.

Depending on the collection of possible models for $M$ and the logic $L$, the difficulty of this problem varies. While in general it is undecidable [145], since asking if a certain program can terminate is the undecidable Halting problem (see Sipser [145]), it is decidable for certain collections of models and certain logics $L$ [55]. For more on the motivation and history of model checking, see Clarke [53].

### 2.1.3 Satisfiability

**Definition 2.1.3** (**The satisfiability property**)**.** Given a formula $\phi$ from a logic $L$ and a collection of models, the *satisfiability problem* is: does there exist a model $M$ such that $M \models \phi$ (also notated as is $M \in [\![\phi]\!]$)? ■

This problem is more difficult than the model checking problem, because we have to determine if a model $M$ exists, often having to **find a model** $M$ that satisfies $\phi$, as well as **verifying that** $M \models \phi$.

### 2.1.4 Model Checking for Propositional Logic

Let us consider a more familiar context: **propositional logic**.

**Definition 2.1.4** (**Propositional logic formula** $\phi$)**.** Given a ground set of propositions $P$ (containing propositions such as $p$ and $q$), a *propositional logic formula* $\phi$ is

constructed as follows:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi$$

■

These operators are $\neg$ (not) and $\wedge$ (and). Propositional logic also has the derived operators $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$ (or) and $\phi_1 \rightarrow \phi_2 \equiv (\neg\phi_1) \vee \phi_2$ (if...then). A proposition can be assigned either true (`tt`) or false (`ff`), and $\phi_1 \wedge \phi_2$ is true if and only if both $\phi_1$ and $\phi_2$ are true. The definition of its semantics are omitted. For a more formal definition of propositional (also called sentential) logic, see Enderton [70] or Epp [71].

Given a set of propositions, a **model** $M$ is an assignment of `tt` or `ff` to each proposition $p \in P$. We assume that $P$ contains at least each proposition that appears in $\phi$.

The **model checking problem** is the following question: given a propositional logic formula $\phi$ and an assignment of truths to a set of atomic propositions $P$, does $M \models \phi$, or is the formula $\phi$, after assigning the values of each proposition, true? This problem can be solved in polynomial time by substituting the truths of the propositions into $\phi$ and then evaluating $\phi$.

The **satisfiability problem** is given a propositional logic formula $\phi$, does there exist an assignment of true and false to all the atomic propositions in $\phi$ such that $\phi$ is true? This is one of the well-known NP-complete problems. See Cormen et al. [58], Sipser [145].

## 2.2 Timed Automata: Baseline Definition

Throughout this dissertation, we use a **timed automaton** as our model, and the collection of models is the collection of all possible timed automata. We define this baseline timed automaton, giving its syntax and its semantics. This model is referred to the Alur-Dill model, and is based on the model in the paper by Alur and Dill [7] as well as the papers Alur [4, 5], Alur et al. [9, 12]. We use this version as the baseline version due to its clean modeling representation and its use in theoretical papers.

### 2.2.1 Syntax

Timed automata involve clocks. Also, timed automata will use clock constraints to reason with these clocks.

**Definition 2.2.1** (**Clock constraint** $\phi \in \Phi(CX)$)**.** Given a nonempty finite set of clocks $CX$ (often $CX = \{x_1, x_2, \ldots, x_n\}$) and $c \in \mathbb{Z}^{\geq 0}$ (a non-negative integer), a *clock constraint* $\phi$ may be constructed constructed using the following grammar:

$$\phi ::= x_i < c \mid x_i \leq c \mid x_i > c \mid x_i \geq c \mid \phi \wedge \phi$$

$\Phi(CX)$ is the set of all possible clock constraints over $CX$. We also use the following abbreviations: true (`tt`) for $x_1 \geq 0$, false (`ff`) for $x_1 < 0$,and $x_i = c$ as $x_i \leq c$ and $x_i \geq c$. ■

With the definition of clock constraints, we now provide the syntax for a timed automaton.

**Definition 2.2.2** (**Timed automaton**)**.** A *timed automaton* $TA = (L, L_0, L_u, \Sigma, CX, I, E)$

is a tuple where:

- $L$ is the finite set of *locations* (nodes).
- $L_0 \subseteq L$ is the nonempty set of *initial locations*.
- $L_u \subseteq L$ is the set of *urgent locations*.
- $\Sigma$ is the finite set of *action symbols*.
- $CX$ is the nonempty finite set of *clocks*. (In this dissertation, often $CX = \{x_1, x_2, \ldots, x_n\}$.)
- $I\colon L \longrightarrow \Phi(CX)$ gives a clock constraint for each location $l$. $I(l)$ is referred to as the *invariant* of $l$.
- $E \subseteq L \times \Sigma \times \Phi(CX) \times 2^{CX} \times L$ is the set of *edges*. In an edge $e = (l, a, \phi, \lambda, l')$ from $l$ to $l'$ with action $a$, $\phi \in \Phi(CX)$ is the *guard* of $e$, and $\lambda \in 2^{CX}$ represents the set of clocks to *reset* to 0.

When considering the satisfaction of logical formula, we will assume that $L_0$ is at most one location. We will refer to this initial location as $l_0$. Additionally, when convenient, we will augment a timed automaton with a set of *atomic propositions* $AP$ and a *labeling function* $Lab\colon L \longrightarrow 2^{AP}$ where $Lab(l)$ gives the subset of atomic propositions that location $l$ satisfies. While one can often represent an atomic proposition $p$ by $p = \{l \mid p \in Lab(l)\}$, the notation of $AP$ and $Lab$ will prove useful when comparing two timed automata to each other. ■

A timed automaton has a set of locations, some of which are urgent, and some of which (usually just one) are initial. The invariant of each location describes the constraints on the clocks that must be satisfied in order to remain in that location. For each edge $e = (l, a, \phi, \lambda, l')$, whenever the guard $\phi$ is true, an execution can

transition from location $l$ to location $l'$ with action symbol $a$. During this transition, all the clocks in $\lambda$ are reset to 0. Urgent locations are locations where time is not allowed to advance, not even for 0 time units.

**Example 2.2.1** (**Example of a timed automaton**)**.** Consider the timed automaton in Figure 2.1, which is similar to a model of a train in the generalized railroad crossing (GRC) protocol [85].

There are four locations: 0: far (initial location), 1: near, 2: in and 3: out, with one clock $x_1$, initial location 0: far and urgent location 3: out. There are the actions *approach*, *enter*, *exit*, and *return* in $\Sigma$. Here, location 1 has the invariant $x_1 \leq 4$ while location 0 has the vacuous invariant `tt`. The edge $(1\text{: near}, in, x_1 = 4, \{x_1\}, 2\text{: in})$ has the guard $x_1 = 4$ and resets $x_1$ to 0.

Written out formally, this timed automaton $TA_{GT}$ is:

- $L = \{0\text{: far}, 1\text{: near}, 2\text{: in}, 3\text{: out}\}$
- $L_0 = \{0\text{: far}\}$
- $L_u = \{3\text{: out}\}$
- $\Sigma = \{approach, enter, exit, return\}$
- $CX = \{x_1\}$
- $I\colon L \longrightarrow \Phi(CX)$ is the function where $I(0\text{: far}) = \texttt{tt}$, $I(1\text{: near}) = x_1 \leq 4$, $I(2\text{: in}) = x_1 \leq 15$ and $I(3\text{: out}) = \texttt{tt}$
- $E = \{(0\text{: far}, approach, \texttt{tt}, \{x_1\}, 1\text{: near}), (1\text{: near}, enter, x_1 = 4, \{x_1\}, 2\text{: in}), (2\text{: in}, exit, x_1 \geq 1, \emptyset, 3\text{: out}), (3\text{: out}, return, \texttt{tt}, \emptyset, 0\text{: far})\}$

In figures, any location without an invariant has the default invariant of true (`tt`) and any edge without a guard has the default guard of `tt`. ■

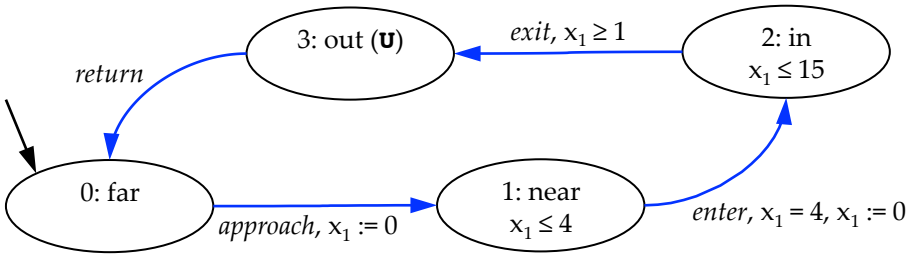


**Figure 2.1:** Timed automaton $TA_{GT}$, a model similar to the model of a train in the generalized railroad crossing (GRC) protocol.

### 2.2.2 Semantics

We represent the semantics of a timed automaton as a transition system. We do this by using valuations to give real values to the clocks and by augmenting locations with these valuations.

**Definition 2.2.3 (Clock valuation $\nu \in \mathcal{V}$).** Given a finite set of clocks $CX$, a *clock valuation* or *clock interpretation* $\nu$ is a function $\nu\colon CX \longrightarrow \mathbb{R}^{\geq 0}$ where $\nu(x)$ is the current time value of clock $x$. A valuation is an assignment of a time (or time value) to each clock in $CX$. $\mathcal{V}_{\mathcal{CX}}$ is the set of all valuations over clocks $CX$. When clear from context, the $CX$ is omitted and the set of valuations is notated as $\mathcal{V}$.

- Let $\nu[Y := c]$ denote the assignment of time $c$ to all the clocks in $Y \subseteq CX$ in the valuation $\nu$ (all other clocks' values are unchanged). Formally,

$$(\nu[Y := c])(x) = \begin{cases} \nu(x) & x \notin Y \\ c & x \in Y \end{cases} \tag{2.1}$$

  Note that $\nu[Y := 0]$ assigns 0 to every clock in $Y$ in $\nu$, leaving all other clocks' values unchanged.

- Let $\nu + \delta$ denote an increment to all clocks in the valuation $\nu$ by an amount

of $\delta$, where $\delta \in \mathbb{R}^{\geq 0}$. Formally,

$$(\nu + \delta)(x) = \nu(x) + \delta \text{ for all } x \in CX. \tag{2.2}$$

- Let $[CX := 0]$ denote the assignment of 0 to each clock.

■

With valuations, we now can define whether a valuation of clocks satisfies a clock constraint.

**Definition 2.2.4 (Valuation satisfying a clock constraint ($\nu \models \phi$)).** A valuation $\nu$ *satisfies a clock constraint* $\phi$ ($\nu \models \phi$), iff for each inequality in $\phi$, where $x_i, x_j \in CX$:

- $\nu \models x_i < c$ if and only if $\nu(x_i) < c$
- $\nu \models x_i \leq c$ if and only if $\nu(x_i) \leq c$
- $\nu \models x_i > c$ if and only if $\nu(x_i) > c$
- $\nu \models x_i \geq c$ if and only if $\nu(x_i) \geq c$
- $\nu \models \phi_1 \wedge \phi_2$ if and only if $\nu \models \phi_1$ and $\nu \models \phi_2$

■

When modeling timed automata, we make the transitions explicit and define the semantics of a timed automaton by describing it as an infinite (timed) **transition system**, or **automaton**. The definition of a transition system is below.

**Definition 2.2.5 (Transition system $TS = (Q, Q_0, \Sigma, \longrightarrow)$).** A *transition system* $TS = (Q, Q_0, \Sigma, \longrightarrow)$ is a tuple where:

- $Q$ is the set of states.
- $Q_0 \subseteq Q$ is the set of initial states.
- $\Sigma$ is the set of actions, labels or action symbols.
- $\longrightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation (need not be a function) that if $(q, a, q') \in \rightarrow$, then the TS can transition from state $q$ to state $q'$ on label $a$.

  Here $q \xrightarrow{a} q'$ is a notation for $(q, a, q') \in \longrightarrow$. A transition system is also called a *labeled transition system* (LTS) or *concrete transition system* (CTS). Additionally, when convenient, we will augment a timed automaton with a set of *atomic propositions* $AP$ and a *labeling function* $Lab\colon Q \longrightarrow 2^{AP}$ where $Lab(q)$ gives the subset of atomic propositions that state $q$ satisfies.

■

Now using the above definition, we give the semantics of a timed automaton by associating an infinite transition system to a timed automaton.

**Definition 2.2.6** (**Semantics of a timed automaton** $TS(TA)$)**.** The *semantics of a timed automaton* $TA = (L, L_0, L_u, \Sigma_{TA}, CX, I, E)$ is a transition system $TS(TA) = (Q, Q_0, \Sigma, \longrightarrow)$ given as follows:

- $Q = L \times \mathcal{V}$, where $q = (l, \nu)$ is a state.
- $Q_0 = L_0 \times [CX := 0]$ (all clocks are initially 0).
- $\Sigma = \mathbb{R}^{\geq 0} \cup \Sigma_{TA}$.
- $\longrightarrow \subseteq Q \times \Sigma \times Q$ is defined as follows:

  **Time advancement:** $(l, \nu) \xrightarrow{\delta} (l, \nu + \delta)$ if

  $l \in L, l \notin L_u$ and $\delta \in \mathbb{R}^{\geq 0}$ and $\forall t \in \mathbb{R}^{\geq 0}, 0 \leq t \leq \delta\colon \nu + t \models I(l)$.

**Action execution:** $(l, \nu) \xrightarrow{a} (l', \nu[\lambda := 0])$ if

$\exists \phi$ such that $(l, a, \phi, \lambda, l') \in E$, $\nu \models \phi$ and $\nu[\lambda := 0] \models I(l')$.

We may sometimes refer to this transition system as the *timed transition system* (as Bouyer and Laroussinie [38] also does) or *concrete transition system* for the timed automaton $TA$. If we consider timed automata augmented with a set of atomic propositions $AP$ and a labeling function $Lab$, the timed transition system has the same set of atomic propositions and the labeling function $Lab$, where $Lab(l, \nu) = Lab(l)$. ■

In the above definition, there are two kinds of transitions. The first kind, time advancement, models the transitions where time advances (elapses) in a single location. The second kind, action execution, models the transitions where edges in the timed automaton are taken.

The semantics give a *conversion* from a timed automaton to an *infinite* transition system, both with an infinite number of states and an infinite number of action symbols. The transition relation $\longrightarrow$ encompasses time advances and action executions, where each state is a (location, valuation) pair. This definition specifies the clocks to all be 0 initially.

*Remark* 2.2.1 (Time advances of 0 units). In the semantics of timed automata (Definition 2.2.6), all non-urgent locations have a time advance of 0 units, assuming that the invariant is true. Although no time advances, the modeling allows transitions of $\xrightarrow{0}$. This feature is present in all sources we consulted, including Alur [5], Baier and Katoen [17], Clarke et al. [55], Wang et al. [161]. Note that for many timed temporal formulas (such as TCTL formulas), any execution that takes a 0 time advance is equivalent to the same execution without the 0 time advance. This is not the case though for all formulas in a timed modal-mu calculus of Laroussinie

et al. [108], Sokolsky and Smolka [147], Zhang and Cleaveland [168]. For example, if we ask the formula "there exists a time advances where afterwards $x_1 = 0$", assuming $x_1$ is initially 0 is only true if we allow time advances of 0 time units.

Timed automata can also be composed in parallel ($||$), much like transition systems. Parallel composition is defined for timed automata so that for two timed automata $TA_1$ and $TA_2$, $TS(TA_1||TA_2) = TS(TA_1)||TS(TA_2)$. See Section 2.3 for a definition of parallel composition.

**Example 2.2.2 (Example 2.2.1 continued).** Again consider the timed automaton in Figure 2.1. With this timed automaton, one sequence of transitions is the sequence

$$(0\text{: far}, x_1 = 0) \xrightarrow{1.2} (0\text{: far}, x_1 = 1.2) \xrightarrow{approach} (1\text{: near}, x_1 = 0) \xrightarrow{4}$$
$$(1\text{: near}, x_1 = 4) \xrightarrow{enter} (2\text{: in}, x_1 = 0) \xrightarrow{5.713} (2\text{: in}, x_1 = 5.713) \xrightarrow{exit}$$
$$(3\text{: out}, x_1 = 5.713) \xrightarrow{return} (0\text{: far}, x_1 = 5.713).$$

Notice that when elapsing 4 time units in the location 1:near that the invariant of location 1 is always true. Also, when $x_1 = 4$, we can execute the action *in* because the guard of the outgoing edge from location 1 is satisfied. Since location 3 is urgent, time cannot advance and the action *return* must be immediately executed.

■

### 2.2.3 Timed Runs

We formally define an execution of a timed automaton (useful when discussing timed logics in Chapter 4). An execution will often be called a **timed run** or a **run of the timed automaton**.

**Definition 2.2.7 (Timed execution (run) $\pi_{tr}$).** A *timed execution (run)* $\pi_{tr}$ is a finite

or infinite sequence of transitions $q_0 \xrightarrow{\sigma} q_1 \xrightarrow{\sigma} q_2 \ldots$ (ending at $q_n$ if finite) where $q_i = (l_i, \nu_i)$ is a state, $\sigma \in \Sigma \cup \mathbb{R}^{\geq 0}$ and for all $i \geq 0$, $q_i \xrightarrow{\sigma} q_{i+1}$ is a valid transition.

We will time-stamp each position $q_i$ with a time $t_i$. We set $t_0 = 0$, $t_{i+1} = t_i + \delta$ if $\delta \in \mathbb{R}^{\geq 0}$ and $q_i \xrightarrow{\delta} q_{i+1}$, and $t_{i+1} = t_i$ otherwise. If $q_n$ is the last state in the run, then for convenience $t_{n+1} = t_n$.

A timed run $\pi_{tr}$ is *time-divergent* if and only if for every time $t \in \mathbb{R}^{\geq 0}$ there exists a position $i$ such that $t_i \geq t$. ■

We use this definition of timed run as opposed to a time-abstract run where $q_i \xrightarrow{\delta}\xrightarrow{a} q_{i+1}$ so we have an explicit list of the time advance transitions taken. This contrasts with the definition of timed runs in Baier and Katoen [17] which defines a timed run as a sequence of $\xrightarrow{\delta}\xrightarrow{a}$ transition sequences. The former definition is used because in our semantic framework, $\xrightarrow{2} \neq \xrightarrow{1}\xrightarrow{1}$ in some circumstances.

### 2.2.4 Urgent Locations

Some sources including Behrmann et al. [23], Dong et al. [65], Olderog and Dierks [131] find it convenient to allow *urgent locations* in timed automata. In such locations, time is not allowed to advance (hence, only non-urgent locations allow time advance transitions in Definition 2.2.6). UPPAAL [23] supports the implementation of urgent locations. Note that we can give an urgent location an invariant, since the invariant would then prevent action transitions into that location.

**Example 2.2.3.** Consider the timed automaton in Figure 2.2 where location 1 is an urgent location. When any execution enters location 1, because location 1 is urgent, time cannot advance and the edge going to location 2 must be taken immediately.

■

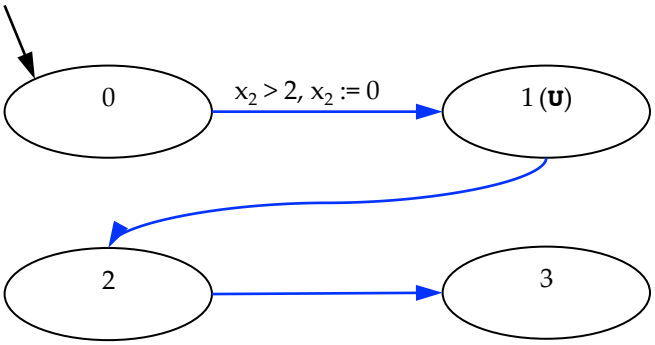


**Figure 2.2:** Timed automaton with location 1 as urgent. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

*Remark* 2.2.2 (Why urgent locations?). We can get rid of most of the urgency by following the idea in Behrmann et al. [23], as follows:

1. Add an extra clock $x_u$. This clock can be used for all urgent locations.
2. Give all urgent locations the invariant $x_u = 0$.
3. Reset $x_u$ on all incoming edges to each urgent location $l_u$.
4. Then make each urgent location non-urgent.

This is almost equivalent, but allows time advances of 0 time units. To get the power of having unsatisfied invariants being urgent locations, we must use urgent locations to prevent time advances of 0 time units. Though seemingly insignificant, these time advances influence formulas written in the timed modal-mu calculus of Laroussinie et al. [108], Sokolsky and Smolka [147]. See Remark 2.2.1.

## 2.3 Networks of Timed Automata: Parallel Composition

Much like transition systems, timed automata can be composed in parallel ($||$). This operator allows one to build networks of timed automata consisting of the

parallel composition of the timed automata. Here we formalize a variant of parallel composition allowing both asynchronous actions and synchronization on common actions.

**Definition 2.3.1 (Parallel composition $TA_1||TA_2$).** Consider two timed automata $TA_1 = (L_1, L_{0,1}, L_{u,1}, \Sigma_1, CX_1, I_1, E_1)$ and $TA_2 = (L_2, L_{0,2}, L_{u,2}, \Sigma_2, CX_2, I_2, E_2)$ where $L_1 \cap L_2 = \emptyset$ and $CX_1 \cap CX_2 = \emptyset$. The *parallel (or product) composition* automaton, denoted $TA_1||TA_2$, is $TA_1||TA_2 = (L, L_0, \Sigma, CX, I, E)$ where:

- $L = L_1 \times L_2$
- $L_0 = L_{0,1} \times L_{0_2}$
- $L_u = (L_{u,1} \times L_2) \cup (L_1 \times L_{2,u})$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $CX = CX_1 \cup CX_2$
- $I\colon L \longrightarrow \Phi(CX)$ is defined as: for any $l \in L$ where $l = (l_1, l_2)$, $I(l) = I_1(l_1) \ \wedge \ I_2(l_2)$.
- $E \subseteq L \times \Sigma \times \Phi(CX) \times CX \times L$ is defined as $E = E_{a_1} \cup E_{a_2} \cup E_s$ where:

  **Asynchronous edges for $TA_1$ ($E_{a_1}$):** $E_{a_1} = \{((l_1, l_2), a, \phi_1, \lambda_1, (l_1', l_2) \mid a \in \Sigma_1 - \Sigma_2 \wedge (l_1, a, \phi_1, \lambda_1, l_1') \in E_1\}$

  **Asynchronous edges for $TA_2$ ($E_{a_2}$):** $E_{a_2} = \{((l_1, l_2), a, \phi_2, \lambda_2, (l_1, l_2') \mid a \in \Sigma_2 - \Sigma_1 \wedge (l_2, a, \phi_2, \lambda_2, l_2') \in E_2\}$

  **Synchronous edges ($E_s$):** $E_s = \{((l_1, l_2), a, \phi_1 \wedge \phi_2, \lambda_1 \cup \lambda_2, (l_1', l_2') \mid (l_1, a, \phi_1, \lambda_1, l_1') \in E_1 \wedge (l_2, a, \phi_2, \lambda_2, l_2') \in E_2\}$

■

Whenever two automata both have an action symbol $a$, they synchronize on $a$ and produce an action $a$. An action symbol in one automaton but not the other happens asynchronously. Since $TA_1||TA_2$ is itself a timed automaton, it may be composed with other timed automata as well. By repeatedly taking compositions, and compositions of compositions, etc., any finite number of timed automata may be integrated into a single timed automaton reflecting the parallel execution of the individual automata.

Another common variant of parallel composition assumes that actions are divided into inputs ($?a$) and outputs ($!a$), with inputs and outputs on the same channel (e.g. $?a$ and $!a$) synchronizing. These are called *communicating timed automata*. In communicating timed automata, $\Sigma$ denotes the set of channels and each channel $a \in \Sigma$ has two complementary events: $?a$ and $!a$. In parallel composition, synchronization occurs when a $?a$ and a $!a$ produce either a $\tau$, an $a$, or an $!a$ (the action produced depends on the model). If there were no action labels, then communicating timed automata would be isomorphic to timed automata. This notation is used in some tools including UPPAAL [23] and CWB-RT [167], and in many other sources [46, 47, 78, 115, 131, 158].

**Example 2.3.1.** Again consider timed automata $TA_{GT}$ in Figure 2.1 and the simplified model of a gate, $TA_{gate}$, in Figure 2.3.

From the diagrams, $CX_{GT} = \{x_1\}$, and $CX_{gate} = \{x_2\}$, which are disjoint sets. Also, $\Sigma_{GT} = \{approach, enter, exit, return\}$ and $\Sigma_{gate} = \{approach, low, exit, high\}$. Hence, the action events that are synchronized on are $approach$ and $exit$.

The parallel composition, $TA_{GT}||TA_{gate}$, is given in Figure 2.4. Because the train location 3: out is urgent, time cannot advance until the composed automaton leaves that location. Furthermore, the gate requires 2 time units to raise the gate, and $x_2$ is reset to 0 when entering location $(3\text{: out}, 3\text{: raise})$. As a consequence,

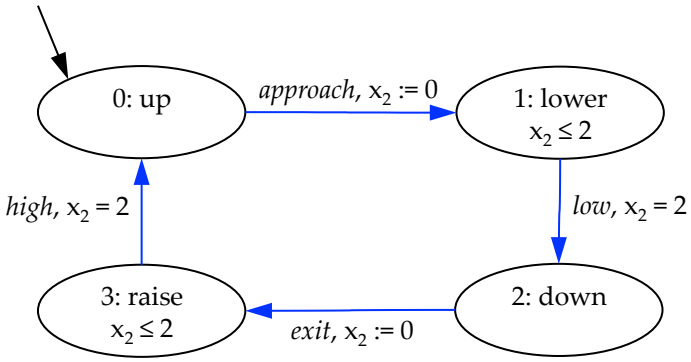


**Figure 2.3:** The model of the gate, timed automaton $TA_{gate}$. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

the component location $(3\text{: out}, 0\text{: up})$ is not reachable. Notice that given the time constraints of the automaton, component location $(2\text{: in}, 1\text{: lower})$ is also not reachable. This is because the transition to location $(1\text{: near}, 1\text{: lower})$ resets both $x_1$ and $x_2$, forcing the value of $x_1$ to be the same as the value for $x_2$. Hence, when $x_1 = 4$, $x_2 = 4$, making the invariant conjunct $x_2 \leq 2$ false. Therefore, that edge cannot be taken. ■

The following claim given in Olderog and Dierks [131] states that the semantics of the composed system is equal (isomorphic) to the parallel composition of the semantics (the transition systems).

**Claim 2.3.1** (From Olderog and Dierks [131])**.** Let $TA_1$ and $TA_2$ be two timed automata. Then
$TS(TA_1||TA_2) = TS(TA_1)||TS(TA_2)$.

Although the notion of parallel composition considered in Olderog and Dierks [131] is different than ours, their result can be adapted to our setting.

By definition, a network of timed automata formed by parallel composition is

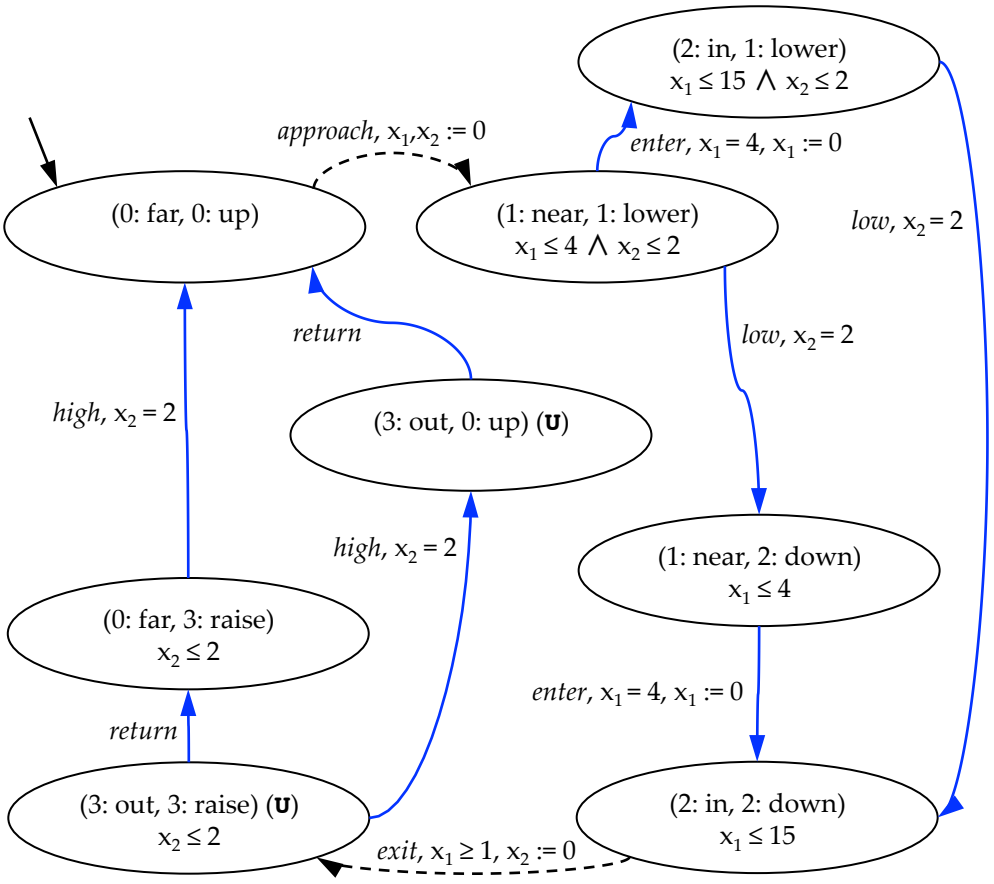


**Figure 2.4:** Timed automaton $TA_{GT}||TA_{gate}$. Dashed lines are used to represent synchronous edges. Unreachable locations are not shown. Note that the locations $(2\text{: in}, 1\text{: lower})$ and $(3\text{: out}, 0\text{: up})$, though shown, are unreachable; all other shown locations are reachable. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

also a single (much larger) timed automaton. Because of this fact, all of our results apply to timed automata formed via parallel composition. In addition, this parallel composition notion can be adapted to the variants considered in this paper such that we can convert the automata before or after composing them in parallel.

## 2.4 Timelocks, Actionlocks and Zeno Executions

When modeling programs, an assumption is often made: the program runs forever. If it is running, then after some number of actions, time is able to advance. Also, actions take some small amount of time. While the time is assumed to be negligible, this means that for a realistic execution, only a finite number of actions should be executed in a finite amount of time.

However, timed automata have no such assumptions, and it is possible to model a program that gets stuck at some point and cannot advance in time (the program stops) as well as model a program that can execute an infinite number of actions in a finite amount of time. For timed automata, these phenomenon come up in three different ways: **timelocks**, **actionlocks** and **zeno** executions.

Often, we wish to only consider **time-divergent** executions in a timed automaton, and we wish to avoid timelocks and zeno executions. While not as problematic, we should be aware of executions where time diverges but cannot have any more actions, called **pure actionlocks**. (In actionlocks that are not pure actionlocks, we can neither allow time to diverge or execute an action.) Since pure actionlocks can be states in time-divergent paths, we should check these paths with pure actionlock states where time diverges but only a finite number of actions are executed.

The goal is to use these definitions and then find ways to both detect these as well as to bypass these unrealistic executions when model checking a timed automaton.

### 2.4.1 Definitions

The definitions come from Bowman and Gómez [47] (which is a continuation of the work in Bowman [46]) as well as from Baier and Katoen [17].

**Definition 2.4.1** (**Time-convergent execution**)**.** A *time-convergent execution* is an execution that only takes a finite amount of time. A time-convergent execution can have an infinite number of time advances or an infinite number of actions. ■

Note that time-convergent executions need not have a finite number of actions. One such time-convergent execution is $\xrightarrow{1}\xrightarrow{1/2}\xrightarrow{1/4}\ldots$, since there are an infinite number of time advances in this execution, but only 2 time units are elapsed. Note that because 0-unit time advances are allowed, time-convergent paths with an infinite number of $\xrightarrow{0}$ time advances exist.

**Definition 2.4.2** (**Time-divergent execution**)**.** A *time-divergent execution* is any execution where time goes to infinity (diverges). ■

Hence, any execution that is not time-convergent is time-divergent and vice versa.

**Definition 2.4.3** (**timelock**)**.** A state $(l, v)$ in a timed automaton is in a *timelock* iff there is no time-divergent execution from that state. ■

Timelocks often arise from mismatched synchronization or parallel composition of automata. See Bowman [46] or Bowman and Gómez [47] for an example. Remember that any automata that share a symbol must synchronize to transition with that symbol.

**Definition 2.4.4** (**Timelock-free**)**.** A timed automaton is *timelock-free* iff all states reachable from the initial state do not have a timelock. ■

**Definition 2.4.5** (**Zeno execution**)**.** An execution is a *zeno execution* iff it has an infinite number of action transitions (events) in a finite amount of time. ■

Note that here a zeno execution is a time-convergent execution, but it is given its own category. Likewise, executions involving timelocks are considered a separate category. We separate out these two kinds of executions because the other time-convergent executions are easier to deal with.

**Definition 2.4.6** (**Nonzeno**)**.** A timed automaton is *nonzeno* if it has all states reachable from the initial state do not have a zeno execution. ■

**Definition 2.4.7** (**Actionlock**)**.** A state $(l, v)$ in a timed automaton is an *actionlock* if no action can be performed from that state or from any future time advance from that state. A state is a *pure actionlock* if it is an actionlock but is not in a timelock. ■

Actionlocks are not considered to be as bad as timelocks because they do not propagate globally when composed with other systems [47] as well as because actionlocks where time can diverge still form time-divergent paths. This sort of actionlock may be interpreted as a terminal state. Tripakis [150] refers to an actionlock as a deadlock, since a deadlock in the discrete sense is an actionlock in the timed sense. In general, timelocks and actionlocks are seen as kinds of deadlocks, or situations where certain progress cannot be made.

One can go further to classify timelocks and actionlocks as a *pure actionlock* (time can diverge but no action may be taken), a *zeno timelock* (time cannot diverge but an infinite number of actions can happen, so only zeno executions can happen) and an *action timelock* (time cannot diverge and no action can be perform), as is done in Bowman and Gómez [47]. Note that there can be zeno runs even in states that are not timelocked. For more information on timelocks, actionlocks and zeno runs, see Baier and Katoen [17], Bowman [46], Bowman and Gómez [47].

## 2.5 Bisimulation and Region Equivalence

We discuss two kinds of equivalences that are foundational to timed automata: bisimulation and region equivalence. Region equivalence is the relation that allows one to represent the semantics of a timed automaton as a *finite* transition system.

When constructing the equivalences, a foundational goal is to determine reachability of states. By constructing equivalence relations, we can evaluate reachability over a smaller representation of the timed automata.

### 2.5.1 Bisimulation

For two transition systems we can define *bisimulation* [124] as follows.

**Definition 2.5.1 (Bisimilarity $\sim$).** Let $TS_1 = (Q_1, Q_{0_1}, \Sigma_1, \longrightarrow_1)$ and $TS_2 = (Q, Q_{0_2}, \Sigma_2, \longrightarrow_2)$ be two transition systems with $\Sigma_1 = \Sigma_2$. A *bisimulation* $R$ is a relation on $Q_1 \times Q_2$, such that for $q_1 \in Q_1$ and $q_2 \in Q_2$, $q_1 R q_2$ implies:

1. $\forall q_1' \in Q_1, q_1 \xrightarrow{a}_1 q_1' : \exists q_2' \in Q_2 : q_2 \xrightarrow{a}_2 q_2' \wedge q_1' R q_2'$, and
2. $\forall q_2' \in Q_2, q_2 \xrightarrow{a}_2 q_2' : \exists q_1' \in Q_1 : q_1 \xrightarrow{a}_1 q_1' \wedge q_1' R q_2'$.

If only condition 1) holds, then we say that the relation is a *simulation*.

We say that $TS_1$ is *bisimilar* to $TS_2$, ($TS_1 \sim TS_2$) if and only if there is a bisimulation $R$ between $TS_1$ and $TS_2$ with the following additional properties:

1. $\forall q_1 \in Q_{0_1} : \exists q_2 \in Q_{0_2} : q_1 R q_2$, and
2. $\forall q_2 \in Q_{0_2} : \exists q_1 \in Q_{0_1} : q_1 R q_2$.

It can be shown that bisimilarity on transition systems is an equivalence relation. ■

We mention bisimulation because it has desirable properties, including:

1. Bisimulation implies language equivalence. This means that two bisimilar (timed) automata accept the same set of (timed) languages. Timed languages are outside the scope of this dissertation and are discussed in Alur and Dill [7].

2. We can use bisimulation to simplify reachability checking. Since the bisimulation relation $R$ is an equivalence relation, we can partition states into their equivalence classes with respect to $R$. Reachability over these equivalence classes is preserved: states in the same equivalence class can reach states in the same equivalence classes.

3. Many (timed) logics, including TCTL are bisimulation invariant; this means that two bisimilar (timed) automata satisfy the exact same formulas in such a (timed) logic.

Following Larsen and Wang [111], Tripakis and Yovine [152], we extend the notion of bisimulation to timed automata.

**Definition 2.5.2 (Bisimulation for timed automata ($\sim$)).** Let $TA_1$ and $TA_2$ be timed automata. $TA_1 \sim TA_2$ if and only if $TS(TA_1) \sim TS(TA_2)$. ■

One notion for weaker equivalences involves the abstracting of time. Here the amount of time advanced is abstracted away. In these time-abstract transitions:

$$q \stackrel{a}{\Longrightarrow} q' \text{ if and only if } \exists q'' \in Q, \delta \in \mathbb{R}^{\geq 0} \colon q \stackrel{\delta}{\longrightarrow} q'' \stackrel{a}{\longrightarrow} q'. \tag{2.3}$$

Using this notion, one can convert the transition system of a timed automaton into a time-abstract transition system and can weaken bisimulation to *time-abstract bisimulation*. The notion of a time-abstract transition is described in Alur [5], Larsen and Wang [111], Tasiran et al. [149] and the notion of time-abstract bisimulation

is described in Alur [5], Larsen and Yi [110], Larsen and Wang [111], Tasiran et al. [149], Tripakis and Yovine [152].

### 2.5.2 Region Equivalence

When defining region equivalence, we follow the definitions in Alur [5].

**Definition 2.5.3** (**Region equivalence relation** $\approx_c$)**.** Given a function $c\colon CX \longrightarrow \mathbb{N}$ where $c(x)$ denotes the largest constant used for clock $x$, then the *region equivalence relation* $\approx_c \subseteq \mathcal{V} \times \mathcal{V}$ is defined such that $\nu_1 \approx_c \nu_2$ if and only if all of the following hold:

1. For all $x \in CX$, either $\lfloor \nu_1(x) \rfloor = \lfloor \nu_2(x) \rfloor$ or $\nu_1(x), \nu_2(x) > c(x)$.
2. For all $x, y \in CX$ with $\nu_1(x) \leq c(x), \nu_1(y) \leq c(y)$, $fr(\nu_1(x)) \leq fr(\nu_1(y)) \leftrightarrow fr(\nu_2(x)) \leq fr(\nu_2(y))$
3. For all $x \in CX$ with $\nu_1(x) \leq c(x)$, $fr(\nu_1(x)) = 0 \leftrightarrow fr(\nu_2(x)) = 0$

For any valuation $\nu$, we let $[\nu]_{\approx_c}$ denote the equivalence class of states that $\nu$ is in considering the equivalence relation $\approx_c$.

For any number $n$, $fr(n) = n - \lfloor n \rfloor$, denotes the fractional value of $n$. ■

Applying the axioms in the above definition, if we consider clocks $x, y \in CX$ with $\nu_1(x) \leq c(x), \nu_1(y) \leq c(y)$, $fr(\nu_1(x)) = fr(\nu_1(y)) \leftrightarrow fr(\nu_2(x)) = fr(\nu_2(y))$. This follows from plugging in $fr(\nu_1(x)) \leq fr(\nu_1(y))$ and $fr(\nu_1(y)) \leq fr(\nu_1(x))$.

Sometimes, rather than having a maximum constant function $c$, one will have a maximum constant, **maxc**, where *maxc* is the maximum of the constants for each clock. Formally, $maxc = max\{(c(x) \mid x \in CX\}$.

We illustrate clock regions with two examples.

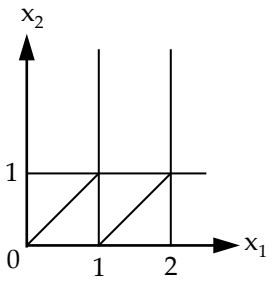


**Figure 2.5:** The set of equivalent regions for two clocks $x_1, x_2 \in CX$ where $c(x_1) = 2$ and $c(x_2) = 1$. There are 8 area regions, 6 point regions, and 14 line regions, totaling 28 clock regions.

**Example 2.5.1.** If we have two, clocks, it splits the sets of clocks into regions as shown in *Figure 2.5*, which is based off of the similar figure in Alur [5]. In the figure $c(x_1) = 2$ and $c(x_2) = 1$.

The regions are the following shapes/areas (a similar example is presented in Baier and Katoen [17]):

- The 6 corner points (such as the value $(x_1 = 2, x_2 = 1)$).
- The 9 open line segments that form the edges of the triangles but do not contain the corner points.
- The 5 rays (such as the ray $(x_1 = 2, x_2 > 1)$ that do not contain the corner points.
- The 4 triangular areas, which do not include the boundaries.
- The 4 half-rectangular areas, which do not include the boundaries.

Thus forming a total of **28** clock regions. ■

**Example 2.5.2.** If $c(x_1) = 3$ and $c(x_2) = 2$, we get a total of **60** clock regions. There

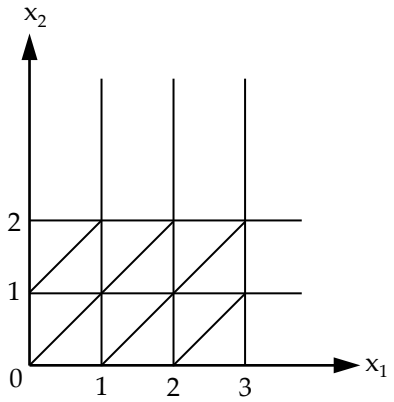


**Figure 2.6:** The set of equivalent regions for two clocks $x_1, x_2 \in CX$ where $c(x_1) = 3$ and $c(x_2) = 2$. There are 60 clock regions.

are 18 area regions, 12 point regions and 30 line regions, totaling **60** clock regions. These clock regions are shown in Figure 2.6. ■

This region equivalence results in a finite number of regions. This follows from a claim in given in Alur [5] that bounds the maximum number of regions.

**Claim 2.5.1.** Let $|CX| = k$. Then the number of regions is at most
$k! * 2^k * \prod_{x_i \in CX} (2c(x_i) + 2)$.

*Remark* 2.5.1 (Region equivalence on timed automata with clock difference constraints.). In Section 3.4.2, we will introduce different timed automata variants. One of them allows constraints to compare differences of clocks ($x_i - x_j < c$ or $x_i - x_j \leq c$). In order to have a sound region equivalence when clock difference constraints are allowed, a stronger region equivalence is needed. See Olderog and Dierks [131] for the stronger region equivalence.

**Definition 2.5.4 (Clock region).** Let $\nu \in \mathcal{V}$ and $\approx_c$ be a region equivalence relation. Then $r = [\nu]_{\approx_c}$, the equivalence class of valuations equivalent to $\nu$ is called a *clock*

*region*, and $v \in r$. Let $\mathcal{R}$ denote the set of all clock regions. ■

This groups all clock valuations into one region. We can then use each region as the new clock component. Concerning regions, there are two other regions that we are concerned about:

**Definition 2.5.5 (Successor clock region).** Let $v \in \mathcal{V}$ and $r = [v]_{\approx_c}$ denote the clock region containing $v$. Then the clock region $r'$ is the *successor clock region* of $r$ if and only if:

$$\forall v \in r\colon \exists \delta \geq 0\colon \left(v + \delta \in r'\right) \wedge \left(\forall \delta', 0 \leq \delta' \leq \delta\colon v + \delta \in (r \cup r')\right)$$

■

**Definition 2.5.6 (Boundary region).** Let $v \in \mathcal{V}$. Then $r = [v]_{\approx_c}$ is a *boundary region* if and only if:

$$\forall v \in [v]_{\approx_c}\colon \forall \delta > 0\colon v + \delta \notin [v]_{\approx_c}$$

■

From this region equivalence of clock valuations, we can extend it to form region automata: the finite equivalent to timed automata base on the region equivalence abstraction. In both cases, we work with the transition system representing the semantics of the timed automaton. Note that a different function of maximum constants for clocks will yield a slightly different region automaton.

**Definition 2.5.7 (Region automaton $Reg(TA)_c$).** Given a timed automaton $TA = (L, L_0, L_u, \Sigma_{TA}, CX, I, E)$, its semantics $TS(TA) = (Q, Q_0, \Sigma, \longrightarrow)$, and a region

equivalence relation $\approx_c$ on valuations and a region equivalence relation $\sim_c$ on states, the *region automaton of the timed automaton*,$Reg(TA)_c = (Q, Q_0, \Sigma, \rightarrow)$ is as follows:

- $Q = \{(l, [\nu]_{\approx_c}) \mid (l, \nu) \in Q \text{ and } \nu \in \mathcal{V}\} = L \times \mathcal{R}$.
- $Q_0 = \{(l_0, [\nu]_{\approx_c}) \mid (l_0, \nu) \in Q_0 \text{ and } \nu \in \mathcal{V}\} = L^0 \times \mathcal{R}$
- $\Sigma = \Sigma_{TA} \cup \{\delta\}$
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is defined as follows:

  $(l, [\nu]_{\approx_c}) \xrightarrow{\delta} (l, [\nu + \delta]_{\approx_c})$ if

  $l \in L$ and if there is a $d \in \mathbb{R}^{\geq 0}$ and $[\nu + d]_{\approx_c}$ is a successor clock region of $[\nu]_{\approx_c}$

  **(time advancement to region successor)**.

  $(l, [\nu]_{\approx_c}) \xrightarrow{a} (l', [\nu[\lambda := 0]]_{\approx_c})$ if and only if

  $\forall \nu \in [\nu]_{\approx_c} : \exists \nu' \in [\nu[\lambda := 0]]_{\approx_c} : (l, \nu) \xrightarrow{a} (l', \nu')$

  **(action or edge)**. (Here, $a \in \Sigma_{TA}$)

■

In the region automaton, we represent a set of region-equivalent valuations. From this definition, $(l_1, \nu_1) \sim_c (l_2, \nu_2)$ if and only if $l_1 = l_2$ and $\nu_1 \approx_c \nu_2$.

Each state is a location coupled with a clock region, since all the possible clock valuations have been compressed to the set of clock regions, and there is no compressions of locations. We then allow an action transition if and only if it was allowed in the Timed Kripke Structure, and we also allow a $\delta$ transition, which represents a time advance from a region to its successor region.

### 2.5.3 Region Equivalence is a Bisimulation

The power from region equivalence comes from the fact that region equivalence is time-abstract bisimulation. We state the result as the following theorem and present part of the proof omitting some details. The full proof is provided in Clarke et al. [55].

**Theorem 2.5.2** (**Region equivalence is a time-abstract bisimulation**). Let $TA = (L, L_0, L_u, \Sigma_{TA}, CX, I, E)$ be a timed automaton with its semantics as the transition system $TS(TA) = (Q, Q_0, \Sigma, \longrightarrow)$ and let $c\colon CX \longrightarrow \mathbb{N}$ be the function giving maximum constants for each clock and $\approx_c$ be the region equivalence relation. Then the relation $\sim_c \subseteq Q \times Q$ where $q = (l, \nu), q' = (l', \nu')$ and $q \sim_c q' \Leftrightarrow (l = l' \ \wedge \ \nu \approx_c \nu')$, is a time-abstract bisimulation.

To prove this theorem, we will break its proof up into the proofs of some lemmas. We start with the hardest lemma: showing that we can always have a time advance when needed.

**Lemma 2.5.3** (Picking the proper time advance.). Let $TA = (L, L_0, L_u, \Sigma_{TA}, CX, I, E)$ be a timed automaton with its semantics as the transition system $TS(TA) = (Q, Q_0, \Sigma, \longrightarrow)$, $c\colon CX \longrightarrow \mathbb{N}$ be the function giving maximum constants for each clock, $\nu_1, \nu_2 \in \mathcal{V}$ and $\approx_c$ be the region equivalence relation. Suppose $\nu_1 \approx_c \nu_2$. Then for every $i \in \mathbb{R}^{\geq 0}$, there is a $j \in \mathbb{R}^{\geq 0}$ such that $\nu_1 + i \approx_c \nu_2 + j$.

This lemma is the heart of region equivalence. This lemma states that for every time advance made by a state, there is an equivalent (though possibly a different amount of time) time advance that every region-equivalent state can take to reach a state region-equivalent to the new state.

***Proof of Lemma 2.5.3.*** We first sketch the proof, and then provide more of the details required to prove the lemma.

**Proof sketch:** It suffices to prove the result for $0 < i < 1$ (where $\nu_1' = \nu_1 + i$) because for region equivalent valuations the integer part of the time advance is matched by advancing the same amount of time for the second valuation. Next, we sort clocks by their fractional orderings, creating a permutation of the clock valuations. When $\nu_1$ advances $i$ units, this time advance produces a new ordering of valuations based on their fractional values (some fractions get smaller because those clocks are one integer larger). We then choose a time advance $j$ for $\nu_2$ to produce the same final fractional ordering of the clocks. The key idea is that adding a fractional time is like applying a cycle on the fractions, so the fractional ordering is cycled around. Thus, when a fraction $i$ is advanced, pick a $j$ that applies the same cyclic shift.

**More detailed proof**: Let $i \in \mathbb{R}^{\geq 0}$. Since for every clock $x \in CX$, if $\nu_1(x) > c(x)$, then $\nu_1(x) + i \geq c(x)$. Furthermore, by the premise premise $\nu_1 \approx_c \nu_2$, $\nu_2(x) > c$. As a result, we have that for every $j \geq 0$, $\nu_2(x) + j > c(x)$.

Because of this, we can now assume that the value of every $x \in CX$ for each valuation is at most $c(x)$. Furthermore, since there is less to prove if adding time to a clock causes the value for that clock $x$ to exceed its maximum constraint $c(x)$, we will assume that all values for each clock $x$ are still less than their maximum constraints $c(x)$ after the time advance.

There are the following cases on $i$:

**Case 1:** $i \in \mathbb{Z}$

Then choose $j = i$. Assuming no value for a clock $x$ exceeds its maximum $c(x)$ (we have less to prove if it does) because the integral parts matched before, the

integral parts will still match.

**Case 2:** $i \notin \mathbb{Z}, i > 1$

This case reduces to **Case 3**, since in the worst case, no clock value for clock $x$ exceeds its maximum constraint $c(x)$, and given that the integral parts of all valuations match before, they will all still match if **Case 3** is true when considering $fr(i) = i' = i - \lfloor i \rfloor$. After obtaining the $j$ from Case 3, the proper time advance for $\nu_2$ is $\lfloor i \rfloor + j$.

**Case 3:** $i \notin \mathbb{Z}, 0 < i < 1$ Let $|CX| = k$.

Sort the clocks in non-decreasing order of their fractional values for both $\nu_1$ and $\nu_2$, using lexicographical ordering if two clocks have equal fractional values.

Without loss of generality, we will assume that the fractional values are originally in the order of the clock indicies. Hence,

$$fr(\nu_1(x_1)) \leq fr(\nu_1(x_2)) \leq \ldots \leq fr(\nu_1(x_k))$$

Since $\nu_1 \approx_c \nu_2$ and by the definition of region equivalence, it follows that

$$fr(\nu_2(x_1)) \leq fr(\nu_2(x_2)) \leq \ldots \leq fr(\nu_2(x_k)),$$

meaning that the original fractional ordering is the same for $\nu_2$.

Now consider $\nu_1 + i$. By the property of addition, this will form a new ordering of fractions that is a cyclic permutation of the original ordering of fractions. Say that $\nu_1(x_i) + i$ is the smallest fractional-valued clock to have its integral part

increased. We then have:

$$fr(\nu_1(x_i)+i) \leq fr(\nu_1(x_{i+1})+i) \leq \ldots \leq$$
$$fr(\nu_1(x_k)+i) \leq fr(\nu_1(x_1)+i) \leq fr(\nu_1(x_{i-1})+i)$$

We now have two cases. In each case, we want $\nu_2(x_i)$ to have the same fractional ordering. Since $\nu_1(x_{i+1})$ is either not an integer or the same value of $\nu_1(x_i)$, we want the same for $\nu_2(x_{i+1})$.

**Case 3a:** $fr(\nu_1(x_i)+i) = 0$ ($\nu_1(x_i)$ is on the line.) Here, choose $j = 1 - fr(\nu_2(x_i))$, this way $\nu_2(x_i)$ is an integer, and the fraction orderings match.

**Case 3b:** $fr(\nu_1(x_i)+i) > 0$ ($\nu_1(x_i)$ crossed the integer line.) Here, choose any $1 - fr(\nu_2(x_i)) < j < 1 - fr(\nu_2(x_{i+1})$ $\big($or the smallest clock with fractional value larger than $fr(\nu_2(x_i))$ if $fr(\nu_2(x_i)) = fr(\nu_2(x_{i+1}))\big)$.

One such choice is the median of the difference

$$\frac{\big(1 - fr(\nu_2(x_i))\big) + \big(1 - fr(\nu_2(x_{i+1}))\big)}{2}$$

In both cases, we get the same cyclic ordering making sure that the same number of fractions cross the line. Therefore, in all cases, $\nu_1 + i \approx_c \nu_2 + j$. □

This next lemma states that region-equivalent valuations satisfy the same clock constraints (with integer constants up to the maximum clock constants).

**Lemma 2.5.4.** Let $\nu_1, \nu_2 \in \mathcal{V}, \phi \in \Phi(CX), c\colon CX \longrightarrow \mathbb{N}$ be the function giving maximum constants for each clock and let $\approx_c$ be a Region Equivalence relation. Suppose $\nu_1 \approx_c \nu_2$. Then $\nu_1 \models \phi \leftrightarrow \nu_2 \models \phi$.

***Proof of Lemma 2.5.4.*** This proof can be done using structural induction on the set of clock constraints, $\Phi(CX)$ (by inducting on the number of literals in the constraint). See Clarke et al. [55]. □

This next lemma allows one to construct time abstract transitions from region-equivalent states to region-equivalent states. This is the key property of a time-abstract bisimulation.

**Lemma 2.5.5.** Let $TA = (L, L_0, L_u, \Sigma_{TA}, CX, I, E)$ be a timed automaton with semantics $TS(TA) = (Q, Q_0, \Sigma, \longrightarrow)$, $c\colon CX \longrightarrow \mathbb{N}$ be the function as described above, and $\nu_1, \nu_2 \in \mathcal{V}$. Suppose $\nu_1 \approx_c \nu_2$. Then for all $l, l' \in L, \nu_1' \in \mathcal{V}$, If there exists a $\delta_1 \geq 0$ s.t. $(l, \nu_1) \xrightarrow{\delta_1, a} (l', \nu_1')$ Then there is a $\nu_2' \in \mathcal{V}$ and $\delta_2 \geq 0$ where $(l, \nu_2) \xrightarrow{\delta_2, a} (l', \nu_2')$ and $\nu_1' \approx_c \nu_2'$.

***Proof of Lemma 2.5.5.*** Let $l, l' \in L, a \in \Sigma$ and let $\delta_1 \geq 0$ such that $(l, \nu_1) \xrightarrow{\delta_1, a} (l', \nu_1')$. Also let $\nu_1, \nu_2 \in \mathcal{V}$ where $\nu_1 \approx_c \nu_2$.

By definition of $\xrightarrow{\delta_1, a}$, then this means:

$$(l, \nu_1) \xrightarrow{\delta_1} (l, \nu_1 + \delta_1) \xrightarrow{a} (l', (\nu_1 + \delta_1)[\lambda := 0]).$$

Here, $\nu_1' = (\nu_1 + \delta_1)[\lambda := 0]$.

By **Lemma 2.5.3** there is some $\nu_2 + \delta_2$ where $v_1 + \delta_1 \approx_c \nu_2 + \delta_2$. **Lemma 2.5.4** guarantees that $\nu_2 \models I(l)$ (Since $\nu_1 \models I(l)$). Now, use the same edge for $(l, \nu_2 + \delta_2)$ that is used to transition $(l, \nu_1 + \delta_1) \xrightarrow{a} (l', \nu_1')$. Let $\nu_2' = (\nu_2 + \delta_2)[\lambda := 0]$. By **Lemma 2.5.4**, $\nu_2' \models \phi$, and hence we can take that edge.

Since the same clocks are reset, the clocks that reset are the same and hence satisfy the conditions for region equivalence. The clocks that do not reset are equivalent since $\nu_1 + \delta_1 \approx_c \nu_2 + \delta_2$ and their values are unchanged by the action

transition.

Therefore, $\nu_1' \approx_c \nu_2'$. (Furthermore, by **Lemma 2.5.4**, $\nu_1' \models I(l')$ if and only if $\nu_2' \models I(l')$.) □

Now with these lemmas, we can now prove the desired theorem.

***Proof of Theorem 2.5.2.*** By definition of $\approx_c$, $\sim_c$ is an equivalence relation. Utilizing **Lemma 2.5.5** (and the case that the relation is symmetric), the time-abstract bisimulation property of $\sim_c$ is proven. □

## 2.6 Untimed Logics

This section provides definitions of Computation Tree Logic (CTL) and the untimed modal-mu calculus. CTL is a branching-time logic, and is a subset of the logic CTL*. For more information on CTL, see Clarke et al. [54, 55]. For more information on the untimed mu-calculus, see Bradfield and Stirling [49], Clarke et al. [55]. In this section, we use transition systems as our model. As with timed automata, we will augment transition systems with a set of atomic propositions $AP$ and a labeling function $\mu$. These augmented transition systems are sometimes referred to as *Kripke structures.*

### 2.6.1 Computation Tree Logic (CTL)

We begin by presenting the syntax of CTL.

**Definition 2.6.1** (**CTL syntax**). A *CTL formula* $\phi$ can be constructed using the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid EX_a(\phi) \mid AX_a(\phi) \mid E[[\phi_1] U [\phi_2]] \mid A[[\phi_1] U [\phi_2]]$$

Here, $p \in AP$ is an atomic proposition and $a \in \Sigma$ is an action. ■

CTL augments propositional logic with two path quantifiers: $E$ (there exists a path) and $A$ (for all paths), and two temporal operators: $X$ (next) and $U$ (until). Additionally, CTL requires that every temporal operator is preceded by a path quantifier, and that every path quantifier is followed by a state operator. This makes CTL easier to model check.

Similar with timed automata, we have paths over transition systems. We use these paths to describe the semantics of these formulas.

**Definition 2.6.2** (**Execution (path)** $\pi$)**.** An *execution (path)* $\pi$ is a finite or infinite sequence of transitions $q_1 \xrightarrow{\sigma} q_2 \xrightarrow{\sigma} q_3 \ldots$ (ending at $q_n$ if finite) where $q_i$ is a state, $\sigma \in \Sigma$ and for all $i$, $q_i \xrightarrow{\sigma} q_{i+1}$ is a valid transition. ■

When defining CTL, researchers sometimes assume that every path is infinite, and that every state has at least one outgoing transitions. In this situation, one will make a dead state (a state that transitions to itself) to represent getting stuck in a transition system.

Now with the definition of paths, we can provide the semantics of CTL.

**Definition 2.6.3** (**CTL semantics**)**.** Let $TS$ be a transition system with atomic proposition set $AP$ and labeling function $Lab$, and $\phi$ be a CTL formula. Then the *semantics* of $\phi$ (denoted $[\![\phi]\!]_{TS}$), the set of states in $TS$ that satisfy $\phi$, is:

- $[\![p]\!]_{TS} = \{q \in Q \mid p \in Lab(q)\}$.
- $[\![\neg\phi]\!]_{TS} = Q - [\![\phi]\!]_{TS}$.
- $[\![\phi_1 \wedge \phi_2]\!]_{TS} = [\![\phi_1]\!]_{TS} \cap [\![\phi_2]\!]_{TS}$.

- $[\![EX_a(\phi)]\!]_{TS} = \{q \mid \text{there is a state } q' \text{ such that } q \xrightarrow{a} q' \text{ and } q' \in [\![\phi]\!]_{TS}\}$.
- $[\![AX_a(\phi)]\!]_{TS} = \{q \mid \text{for all states } q' \text{ such that } q \xrightarrow{a} q', q' \in [\![\phi]\!]_{TS}\}$.
- $[\![E[[\phi_1] U [\phi_2]]]\!]_{TS} = \{q \mid \text{there is some path } \pi \text{ starting at } q \text{ where } \exists i\colon q_i \in [\![\phi_2]\!]_{TS} \wedge \forall j < i\colon q_j \in [\![\phi_1]\!]_{TS}\}$.
- $[\![A[[\phi_1] U [\phi_2]]]\!]_{TS} = \{q \mid \text{for all paths } \pi \text{ starting at } q, \exists i\colon q_i \in [\![\phi_2]\!]_{TS} \wedge \forall j < i\colon q_j \in [\![\phi_1]\!]_{TS}\}$.

$TS$ satisfies $\phi$ iff the initial state $q_0$ satisfies $\phi$. ■

We also use the following abbreviations: `ff` for $\neg$`tt`, $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, and $\phi_1 \rightarrow \phi_2$ for $\neg\phi_1 \vee \phi_2$. Abbreviations for commonly-used derived temporal operators include: $E[[\sigma_1] R [\sigma_2]]$ for $\neg(A[[\neg\sigma_1] U [\neg\sigma_2]])$, $A[[\sigma_1] R [\sigma_2]]$ for $\neg(E[[\neg\sigma_1] U [\neg\sigma_2]])$ (release), $EF[\sigma]$ for $E[[\texttt{tt}] U [\sigma]]$, $AF[\sigma]$ for $A[[\texttt{tt}] U [\sigma]]$ (eventually), $AG[\sigma]$ for $\neg EF[\neg\sigma]$, $EG[\sigma]$ for $\neg AF[\neg\sigma]$ (always), $E[[\phi_1] W [\phi_2]]$ for $\neg(A[[\neg\phi_2] U [\neg\phi_1 \wedge \neg\phi_2]])$, and $A[[\phi_1] W [\phi_2]]$ for $\neg(E[[\neg\phi_2] U [\neg\phi_1 \wedge \neg\phi_2]])$ (weak until). Additionally, we can make derived operators that are not concerned with the specific action symbols used. These are $AX(\phi)$ for $\bigwedge_{a \in \Sigma_{TA}} AX_a(\phi)$ (for all next actions) and $EX(\phi)$ for $\bigvee_{a \in \Sigma_{TA}} EX_a(\phi)$ (there exists a next action). These operators are also used for systems that do not use actions symbols.

The formula $E[[\phi_1] R [\phi_2]]$ means "there exists a path where $\phi_1$ releases $\phi_2$" ($\phi_2$ has to also be true when $\phi_1$ releases $\phi_2$ from being true).

CTL has the nice property that formulas can be expressed simply. For instance, $AG[a]$ is the safety property "always $a$" and $AF[a]$ is the liveness property "inevitably $a$." Additionally, CTL can be verified in polynomial time (polynomial in the produce of the size of the formula and the size of the state machine) [54].

### 2.6.2 Untimed Modal Mu-Calculus

We define the mu-calculus used over transition systems. To express the mu-calculus, we will use a set of variables *Var* to represent sets of states.

**Definition 2.6.4** (**Untimed mu-calculus syntax**). The *syntax* of an untimed mu-calculus formula can be constructed with the following grammar:

$$\phi ::= p \mid \neg p \mid \texttt{tt} \mid \texttt{ff} \mid cc \mid Y \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle(\phi) \mid [a](\phi)$$

Here, $p \in AP$ is an atomic proposition, $a \in \Sigma$ is an action, $Y \in Var$ is a propositional variable, and $\mu Y.[\phi]$ and $\nu Y.[\phi]$ are the least and greatest fixpoint operators over variable $Y$, respectively. ■

Notice that this version of the logic does not have the negation ()′operator. As a result, we are providing all of the dual operators. This keeps formulas in positive normal form, which will make many model-checking results easy to prove. While this mu-calculus could be defined using negation (¬), because this logic is closed, negation adds no additional expressive power. Additionally, we would need to add the restriction that that every occurrence of each variable $Y$ occurs within an even number of negations.

To interpret the meaning of a mu-calculus formula, we use an environment, which maps propositional variables to sets of states.

**Definition 2.6.5** (**Environment** $\theta$). An *environment* $\theta \colon Var \longrightarrow 2^Q$ is a function that assigns a set of states to each variable, where $\theta(Y)$ represents the set of states that

make the formula $Y$ true. For notation, we have:

$$\theta[Y := Q'](Z) = \begin{cases} \theta(Z) & \text{if } Z \neq Y \\ Q' & \text{if } Z = Y \end{cases}$$

■

Now with the environment, we can define the semantics.

**Definition 2.6.6 (Untimed mu-calculus semantics).** Given transition system $TS$ (with labeling function $Lab$), and initial environment $\theta$, the *semantics of an untimed mu-calculus formula* $\phi$, denoted $[\![\phi]\!]_{TS,\theta}$, is (easier ones omitted):

- $[\![p]\!]_{TS,\theta} = \{q \in Q \mid p \in Lab(q)\}$.
- $[\![\neg p]\!]_{TS,\theta} = \{q \in Q \mid p \notin Lab(q)\}$.
- $[\![\mathtt{tt}]\!]_{TS,\theta} = Q$.
- $[\![\mathtt{ff}]\!]_{TS,\theta} = \emptyset$.
- $[\![Y]\!]_{TS,\theta} = \theta(Y)$.
- $[\![\phi_1 \wedge \phi_2]\!]_{TS,\theta} = [\![\phi_1]\!]_{TS,\theta} \cap [\![\phi_2]\!]_{TS,\theta}$.
- $[\![\langle a \rangle(\phi_1)]\!]_{TS,\theta} = \{q \in Q \mid \exists q' \in Q\colon q \xrightarrow{a} q' \text{ and } q' \in [\![\phi_1]\!]_{TS,\theta}\}$.
- $[\![[a](\phi_1)]\!]_{TS,\theta} = \{q \in Q \mid \forall q' \in Q\colon \text{if } q \xrightarrow{a} q' \text{ then } q' \in [\![\phi_1]\!]_{TS,\theta}\}$.
- $[\![\mu Y.[\phi]]\!] =$ the least fixpoint of the function $\phi(Y') = [\![\phi]\!]_{TA,\theta[Y:=Y']}$
- $[\![\nu Y.[\phi]]\!] =$ the greatest fixpoint of the function $\phi(Y') = [\![\phi]\!]_{TA,\theta[Y:=Y']}$

Transition system $TS$ satisfies $\phi$ iff the initial state $q_0$ satisfies $\phi$.

A formula with a propositional variable can be viewed as a monotonic function on a complete lattice [56]. Due to Cousot and Cousot [59] we obtain an iterative semantics for fixpoints. Specifically, by treating the formula $\phi$ as a function on $Y$ where $\phi(Y') = [\![\phi]\!]_{TS,\theta[Y:=Y']}$ and $\phi^i(Y') = \phi(\phi^{i-1}(Y'))$:

$$[\![\mu Y.[\phi]]\!] = \bigcup_{i=0}^{\infty} \phi^i(\emptyset) \text{ and } [\![\nu Y.[\phi]]\!] = \bigcap_{i=0}^{\infty} \phi^i(Q)$$

See Kozen [100] for details. ■

The logic supports two derived operators: $[-](\phi)$ for $\bigwedge_{a \in \Sigma_{TA}} [a](\phi)$ (for all next actions) and $\langle - \rangle(\phi)$ for $\bigvee_{a \in \Sigma_{TA}} \langle a \rangle(\phi)$ (there exists a next action). These operators are also used for systems that do not use actions symbols.

The modal mu-calculus is strong enough to express all of CTL [55]. By definition, $AX_a(\phi) \equiv [a](\phi)$ and $EX_a(\phi) \equiv \langle a \rangle(\phi)$. We can write the until operators with the following equations:

$$E[[\phi_1] U [\phi_2]] \equiv \mu Y.[\phi_2 \vee (\phi_1 \wedge \langle - \rangle(Y))] \tag{2.4}$$

$$A[[\phi_1] U [\phi_2]] \equiv \mu Y.[\phi_2 \vee (\phi_1 \wedge [-](Y))]. \tag{2.5}$$

Furthermore, here are equations in the modal mu-calculus for some of the com-

mon CTL derived operators:

$$E\,[[\phi_1]\,R\,[\phi_2]] \equiv \nu Y.[\phi_2 \wedge (\phi_1 \vee \langle - \rangle(Y))] \tag{2.6}$$

$$A\,[[\phi_1]\,R\,[\phi_2]] \equiv \nu Y.[\phi_2 \wedge (\phi_1 \vee [\,-\,](Y))] \tag{2.7}$$

$$EF\,[\phi] \equiv \mu Y.[\phi \vee \langle - \rangle(Y)] \tag{2.8}$$

$$AF\,[\phi] \equiv \mu Y.[\phi \vee [\,-\,](Y)] \tag{2.9}$$

$$EG\,[\phi] \equiv \nu Y.[\phi \wedge \langle - \rangle(Y)] \tag{2.10}$$

$$AG\,[\phi] \equiv \nu Y.[\phi \wedge [\,-\,](Y)]. \tag{2.11}$$

These equivalences are taken from Clarke et al. [55]. Additionally, the modal mu-calculus is more expressive than CTL [55].

## 2.7 Related Work I: Untimed Systems and Untimed Logics

Now we begin to discuss related work in the following sections. This related work includes both related work on untimed systems and untimed logics, which also provide a rich theory to draw from and apply to timed systems (some of it has already been applied), as well as current work on timed automata. This related work will include the related work that has begun to set the framework for the theory that we plan to develop further. The theory for untimed systems is useful because much of the concepts, ideas and operations can be extended to timed automata, saving the time of developing a new theory. Two books that give expositions of much of the related work are Baier and Katoen [17], Clarke et al. [55].

### 2.7.1 Untimed Systems: Automata and Kripke Structures

The baseline untimed sytstem used is a **finite automaton**. A *finite automaton* is a collection of *nodes*, referred to as *states*, with labeled transitions between states.

When a transition is executed, the label is often referred to as the *action* or event. Using each state as a current snapshot of a system and transitions to signify a change in the program or the model, automata provide a simple model of a program. With automata, many questions about them can be asked. A common one is **reachability:** can the automaton reach a certain state from a given initial state? Another question asks about the possible sequences of actions that can happen. These collection of action sequences form a language, which can be reasoned about as well. Detailed expositions of automata and many of their properties can be found in Hopcroft et al. [93], Sipser [145].

For model checking purposes, automata are augmented so that each state is labelled with a set of atomic propositions that each state satisfies. These augmented automata form *Kripke structures* (see Clarke et al. [55]). With these propositions, we can now ask questions such as: It is always the case that proposition is $p$ is true? To do this, we will use untimed logics, described in the next section (Section 2.7.2).

### 2.7.2 Untimed Logics

Now with finite automata and Kripke structures, we wish to reason with them. The logics discussed here give ways of reasoning with these structures. They involve temporal and modal operators, which allow users to ask questions about Kripke structure executions. Through these formulas, users can ask about the current state, the next state and future states.

A common branching-time logic for model checking Kripke Structures is CTL (Computational Tree Logic), with a polynomial time model checking algorithm given in Clarke et al. [54]. Two additional temporal logics include LTL (Linear Temporal Logic) (see Clarke et al. [54]) and CTL*. We know that CTL* is a superset of both CTL and LTL [55]. However, CTL and LTL are expressibly incomparable:

see Clarke and Draghicescu [52], Emerson and Halpern [68], Lamport [104]. These logics provide temporal-based ways of expressing properties with operators such as "until," "in the next state", "for all paths," etc.

An additional logic includes the modal mu-calculus. It is a powerful fixpoint-based logic (see Clarke et al. [55]) that can express all of CTL* [31]. Additionally, a substantial fragment, called the *alternation-free* fragment (see Emerson and Lei [69]), can both express all of CTL [55] as well as be model-checked efficiently in practice. Different efficient ways to model check the modal mu-calculus come from and are discussed in Andersen [15], Cleaveland and Steffen [57], Mader [121], Mateescu and Sighireanu [123]. These efficient algorithms use Tarski's Theorem [148], the Tarski-Knaster Theorem [69] and Kleene's fixpoint theorem (see Emerson and Lei [69]) A more detailed exposition of the modal mu-calculus is in Bradfield and Stirling [48, 49].

With these logics, two subsets of interesting properties include safety properties and liveness properties. A **safety property** is "is $p$ always true"? A **liveness property** is "is $p$ inevitably true?". There are other interesting properties as well.

## 2.8 Related Work II: Timed Systems

In order to model real-time constraints, **timed systems** were developed. While the complexity of model checking is harder, these systems were developed to express real-time constraints yet still be able to be model checked.

### 2.8.1 Timed Automata Variants

The first system is the **timed automaton**, formally defined in Section 2.2. This model is the Alur-Dill model, based off of the work of Alur and Dill [7], which uses clocks and constraints on clocks to specify real-time constraints. A different

model used is where the automata are given a set of timed trajectories, and the set of timed trajectories is restricted, providing **implicit** constraints. This model is defined by Lynch and Vaandrager [118] (Part II is Lynch and Vaandrager [120] and additional versions include Lynch and Vaandrager [117, 119]). This model was extended to handle timed I/O (Input/Output) automata in Kaynar et al. [95, 96].

Even with the Alur-Dill, model, there are many variants. Many of these are discussed in Chapter 3. These variants make it easier to represent timed automata in model checking tools, and are used in tools such as UPPAAL [23]. Some variants change the syntax; others the semantics. We will examine one of the **semantic variants** here, which is the differences on how to address an unsatisfied invariant. More details are provided in Section 3.4.1.

Let us again consider the semantics of $\longrightarrow$ in the transition system for the base version of timed automata:

$\longrightarrow \subseteq Q \times \Sigma \times Q$ is defined as follows:

**Time advancement:** $(l, \nu) \xrightarrow{\delta} (l, \nu + \delta)$ if

$l \in L, l \notin L_u$ and $\delta \in \mathbb{R}^{\geq 0}$ and $\forall t \in \mathbb{R}^{\geq 0}, 0 \leq t \leq \delta \colon \nu + t \models I(l)$.

**Action execution:** $(l, \nu) \xrightarrow{a} (l', \nu[\lambda := 0])$ if

$(l, a, \phi, \lambda, l') \in E$, $\nu \models \phi$ and $\boldsymbol{\nu}[\boldsymbol{\lambda} := \mathbf{0}] \models \mathbf{I}(\mathbf{l}')$.

In the above version of the semantics, it is required that the invariant of the entering location will be true in order for the transition to be allowed. This version is used in Bengtsson and Yi [27], Bouyer and Laroussinie [38], Olderog and Dierks [131], Tripakis [151], Wang [159], Yovine [164] (some of these sources do not use urgent locations).

However, we could choose to not require the invariant to be true after performing an action. This yields the semantics:

$\longrightarrow \subseteq Q \times \Sigma \times Q$ is defined as follows:

**Time advancment:** $(l, \nu) \xrightarrow{\delta} (l, \nu + \delta)$ if

$l \in L, l \notin L_u$ and $\delta \in \mathbb{R}^{\geq 0}$ and $\forall t \in \mathbb{R}^{\geq 0}, 0 \leq t \leq \delta \colon \nu + t \models I(l)$.

**Action execution:** $(l, \nu) \xrightarrow{a} (l', \nu[\lambda := 0])$ if

$(l, a, \phi, \lambda, l') \in E$ and $\nu \models \phi$.

This interpretation is used in sources including Alur [5], Baier and Katoen [17], Behrmann et al. [23], Beyer and Noack [29], Clarke et al. [55], Zhang and Cleaveland [167]. Thus, according to the definition, if a location is entered where the invariant is false (say, $x_1 < 0$), the transition is allowed and the automaton can transition out of that location, but no time is allowed to elapse. This is the semantic equivalent of *urgency* (see Section 2.2.4). Both interpretations are used.

### 2.8.2 Other Real-Time Systems Models

One real-time system model is the *guarded-command program*, used in Henzinger et al. [88], Zhang and Cleaveland [167], as well in Henzinger et al. [87], the earlier conference version of Henzinger et al. [88]. A contribution of this dissertation, also in Fontana and Cleaveland [74], shows that a guarded-command program is isomorphic to a timed automaton. Thus, both models are completely equivalent. This is discussed in Section 3.3.3.

Another real-time system model extending timed automata is a **parametric timed automata**. A parametric timed automaton is a timed automaton that also has parameters in the constraints, allowing one to ask questions such as "for what values of these parameters is location $q$ reachable?" Parametric timed automata are described and used in Alur et al. [13], Zhang and Cleaveland [167].

In addition to timed models, for more complex systems the model of a **hybrid system** is used. Instead of clocks, hybrid automata extend expressiveness to allow the variables to grow and shrink at different rates. The variable rates are

represented as differential equations. Timed automata are a special case of hybrid automata (one kind of hybrid system). While even more expressive than timed systems, hybrid systems models are harder to model check, and in some cases (including some hybrid automata), even reachability is undecidable [91]. See Henzinger [86], Henzinger et al. [91], Platzer [137] for more information on hybrid systems.

### 2.8.3 Model-Checking Data Structures

In order to more efficiently model check automata, data structures have been used. For untimed systems, the BDD (Binary Decision Diagram) (see Clarke et al. [55]) allowed a compact representation of variables to reduce the time of model checking.

For timed systems, two abstractions were developed: **region equivalence** and **clock zones**. These abstractions allowed timed automata to be reduced to a **finite** representation and thus able to be model-checked in a **finite amount of time**. **Clock regions** are equivalence classes that group all valuations that satisfy the same clock constraints together. **Clock zones** provide an even larger abstraction that still preserves reachability by grouping regions together when there is no constraint in the automaton that distinguishes the two zones. Due to their convexity, clock zones are easy to manipulate in practice. See Alur [5], Bengtsson and Yi [27]. The first implementation for a clock zone is a DBM (difference bound matrix) (see Dill [64]). Clock zones, DBMs and other implementations are described in Chapter 5.

While good abstractions, clock regions are too hard to model check in practice (there are possibly an exponential number of clock regions [5]) and clock zones are not a guaranteed abstraction for all logical formulas [159]. Thus, one sometimes has to use a **union of clock zones**. The simplest is to use a list of DBMs.

Inspired by BDDs, two implementations were developed: the CRD (Clock Restriction Diagram) [153, 154, 155] and the CDD (Clock Difference Diagram) [21, 110]. Additional implementations developed include the Difference Detection Diagram (DDD) [126] and the Constraint Matrix Diagram (CMD) [67].

## 2.9 Related Work III: Timed Logics

Given a collection of models, **specifications** must be developed to express properties that we wish to verify. While we can verify **reachability** without a logic, we will discuss and work with logics that can write reachability questions as **safety** properties. These logics can also express other properties.

### 2.9.1 Extensions of CTL and LTL

The first logic is TCTL (Timed Computation Tree Logic), which is an extension of CTL. TCTL extends CTL to allow questions of timing intervals. The first variant adds interval timing constraints to the "until" operators while the second variant. See Alur et al. [9, 12], Penczek and Pólrola [135] for more information on TCTL, as well as Section 4.1. TCTL is used due to its ease of expressiveness for safety and liveness properties as well as its ease to verify in practice.

There are two timed extensions to LTL (Linear Temporal Logic): MTL (Metric Temporal Logic) (see Bouyer [37]) and TPTL (Timed Propositional Temporal Logic) (see Alur and Henzinger [8]). MTL extends the timing by allowing timing intervals in the "until" operators, and TPTL extends LTL by allowing **freeze quantification**. In Bouyer et al. [42, 44] it was shown that TPTL is strictly more expressive than MTL, which as a corollary, shows that the freeze-quantification variant of TCTL is more expressive than the variant with interval timing constraints. See Bouyer [37], Bouyer et al. [44], Pandya and Shah [133] for more information.

### 2.9.2 Extensions of the Modal Mu-Calculus

Extensions of the untimed modal mu-calculus have begun. The first variant of a timed modal mu-calculus is provided by Henzinger et al. [88], *called* $T_\mu$. While it can express all of TCTL in theory, it cannot express all of TCTL practically (including $AF_{<\infty}[\phi]$) [88]. The $T_\mu$ expression for $AF[\phi]$ requires the user to guess an upper bound of when $\phi$ will be true, and this upper bound may need to be much larger than the region equivalence constant $maxc$. While it does have lots of theoretical properties proven about it, lacking a practical way to model check all of TCTL is a drawback.

Another variant of the timed modal-mu calculus, given in modal equation system form, in introduced (independently) by Sokolsky and Smolka [147] and by Aceto and Laroussinie [2], but little was said in terms of expressiveness of this variant in Sokolsky and Smolka [147]. This logic considering only greatest fixpoints was also introduced in Laroussinie et al. [108]. This variant is *called* $L_{\nu,\mu}$ (taken from Aceto and Laroussinie [2]), and the variant with greatest fixpoints only is *called* $L_\nu$. $L_\nu$ is shown to be a characteristic for bisimulation in Laroussinie et al. [108]. In Aceto and Laroussinie [2], it is shown that $L_{\nu,\mu}$ is EXPTIME-complete to model check. An additional operator is added in Bouyer et al. [41] and Bouyer et al. [41] proves that the relativized operator adds expressive power to the logic $L_\nu$. We call *the relativized variant* $L^{rel}_{\nu,\mu}$. However, many desirable expressivity results, such as expressing TCTL were not proven about $L_{\nu,\mu}$ and $L^{rel}_{\nu,\mu}$. Since $L_{\nu,\mu}$ is used in Zhang and Cleaveland [168], and Zhang and Cleaveland [167, 168] use a proof-rule scheme to model check safety properties expressible in the alternation-free fragment of the timed modal-mu calculus (though the formula to express a safety property is not provided in Zhang and Cleaveland [168]), this logic is not only general but has the potential to be model checked efficiently in practice. If we

combine the results of Aceto and Laroussinie [2] and Bouyer et al. [41], model checking of the $L^{rel}_{\nu,\mu}$ is shown to be EXPTIME-complete.

We discuss these logics more in Chapter 4. There we also fill in some of the missing theory.

## 2.10 Related Work IV: Surveys, Uses, and Tools

We continue the discussion of related work by listing some surveys involving timed automata, timed automata algorithms, and modeling of timed systems. Then we list some papers that use timed automata to model systems. We conclude this section with a list of timed automata model checking tools.

### 2.10.1 Surveys, Books and Book Chapters

Book chapters covering timed automata and real-time model checking include Baier and Katoen [17], Bouyer and Laroussinie [38], Clarke et al. [55], Olderog and Dierks [131]. Fahrenberg et al. [72] presents a survey of timed automata that also includes model checking reachability. Bouyer [37] surveys different logics that can be checked by timed automata. Penczek and Półrola [134] gives a survey of verification techniques for timed automata and timed petri nets.

Furia et al. [76] surveys different models of time and timed systems. A survey of various ways to timed systems is Wang [156]. Hassine et al. [83] discusses different notations for modeling timed scenarios, and it touches upon but does not focus on using timed automata. Deligiannis and Manesis [61] discusses various types of automata, including timed automata and then discusses some points about them concerning using them to model phenomena. Another survey of different ways to model systems of timing constraints is Furia et al. [76].

### 2.10.2 Uses of Timed Automata

Sloth and Wisniewski [146] discusses a technique for using timed automata as an abstraction of continuous dynamical systems. Another technique for using timed automata to over-approximate continuous systems is discussed in Maler and Batt [122]. Hassine et al. [82] discusses how to use timed automata to model timed use case maps. Abdeddaïm et al. [1] discuses how to model and solve some scheduling problems with timed automata. Khatib et al. [97] discuss how to map a models used in the Heuristic Scheduling Testbed System (HSTS) planner into timed automata. Largouët et al. [106] uses timed automata to model ecosystems, Koltuksuz et al. [98] uses timed automata to model and verify a security protocol, Kourkouli and Hassapis [99] uses timed automata to model and verify an online transaction processing system, and Lindahl et al. [116] uses timed automata to verify a prototype gear controller. Ravn et al. [140] uses timed automata to verify the Business Agreement with Coordinator Completion (BAwCC) protocol within the WS-Business Activity (WS-BA) standard, which provides protocols used for long-lived business activities [140]. Herber et al. [92] uses timed automata to verify some SystemC designs.

### 2.10.3 Tools

Tools to model check timed automata include *UPPAAL* [22, 23, 24, 27, 113], *KRONOS* [60, 163, 164], CWB-RT/CWB-PRT [167, 168] (updated in [73]), MCTA [103], a SAT solver to model check timed automata ([130]), CMC [107], TMV [143] (also see the technical report [144]) (which is a fully symbolic model checker that can check any TCTL property), DDDLib [125] (which uses difference decision diagrams to store clock valuations when model-checking), SGM [160] (a tool that can check real-timed systems for untimed property and has a focus on requiring less

technical knowledge of model-checking to use), Synthia [136] (can check saftey properties of timed game automata), VerICS [94], a prototype (tool not named) [67] (which model checks safety and bounded liveness properties using clock matrix diagrams (see Ehlers et al. [67]) ), a prototype (tool not named) [127] (that can check safety properties by converting timed automata to finite state machines with time) and a prototype (tool not named) [34] (which can model check TCTL). Tools that can also model check linear hybrid automata as well as timed automata include *RED* [154, 155, 162] (with its library, *REDLIB* [157]), and HyTech [14, 90]. Tools that model check timed systems but use discrete-time representations include *Rabbit* [29, 30] and an unnamed tool [105]. The tool UPP2SF [132] does not model check timed automata; instead, it translates them into Stateflow models, which can be used by MATLAB Simulink.

## 2.11 Related Work V: Vacuity

### 2.11.1 Untimed Systems with Temporal Logics

The first kind of work on vacuity involves determining if a model vacuously satisfies a formula. For (untimed) temporal logics, one key paper is Beer et al. [20] (and its conference version Beer et al. [19]). Beer et al. [20] focuses its work on CTL*. Dong et al. [66] extends this work to handle the untimed modal mu-calculus. Kupferman and Vardi [102] extends the work on vacuity for CTL*, extending vacuity to also finding interesting witnesses.

An implementation of a vacuous model checker for CTL is VaqUoT, which is described in Gheorghiu and Gurfinkel [77]. The theory used by Gheorghiu and Gurfinkel [77] is described more in Gurfinkel and Chechik [80]. Di Guglielmo et al. [63] also discusses an implementation to detect vacuity, and this paper also refines some of the techniques discussed in previous works.

In the untimed setting, researchers have utilized proof-based model checkers to check for vacuity. Namjoshi [128] uses a proof-based model checker and develops the notion of a vacuous proof as well as an algorithm to develop a proof that indicates whether a formula is vacuously satisfied or not. This work is extended in Namjoshi [129]. One of the contributions of Namjoshi [129] is identifying a way to distinguish when between two kinds of vacuity related to proofs: whether a formula is vacuous within a single proof, or whether a formula is vacuous within every proof.

Ball and Kupferman [18] applies vacuity to software testing. Chockler and Strichman [50] extends the vacuity on untimed systems to handle mutual vacuity: multiple subformulas can be vacuous within a formula. This paper provides ways to find a maximal subset of sub formulas that can be substituted out and the formula remains satisfied. Kupferman [101] serveys the work on untimed vacuity and relates it to the notion of coverage in testing.

Extending this notion of formula vacuity, Gurfinkel and Chechik [81] discusses the notion of bisimulation vacuity, which aims to be more robust than the syntactic vacuity that is defined in Beer et al. [20]. The paper then applies it to CTL and CTL*. Samer and Veith [141] discusses different notions of vacuity over the untimed setting. Its contribution is relating these different notions at the semantic level.

Notice that all of these papers work only in the untimed setting.

### 2.11.2 Work Involving Real-Time Systems

Although Post et al. [138] is relevant to vacuity and works with real-time requirements, its goal for vacuity is different. Rather than determining if a model satisfies a specification vacuously, it takes a set of specifications and determines if any specifications are redundant due to vacuity. For instance, if a set of specifications

contained $AG[p \rightarrow AF[q]]$ and $AG[\neg p]$, the first specification would be redundant and deemed vacuous. Additionally, while not vacuity, Cimatti et al. [51] tries to gain more information than just a yes or no answer. Cimatti et al. [51] model checks hybrid automata and tries to explain why a certain property, explained in a message sequence chart (MSC), is infeasible.

To our knowledge, our work is the first to both extend the theory of vacuous satisfaction to timed model-checking as well as the first to provide a preliminary implementation of it.

# Chapter 3

# Timed Automata: Definitions, Variants and Equivalences

The baseline variant of timed automata is the Alur-Dill variant based off of Alur and Dill [7]. For this specific variant, strong theoretical claims have been proven, including a proof that both reachability and TCTL model checking are PSPACE-complete [12] and that reachability and TCTL model checking are preserved by region equivalence [5, 55]. This variant has strong theoretical properties. However, for ease of model checking, many tools represent this variant differently. Some of these representation changes are syntactic; others are semantic.

With these new representations, it is not clear which results proven in theory carry over. Having these theoretical results supports tools because these results can provide correctness results for algorithms used in tools. However, if the variant is changed, does the result still hold? Bouyer [36] proved that a commonly used algorithm in timed automata model checking was incorrect if timed automata were extended in a seemingly innocuous fashion. This subtlety had eluded detection for several years, and indeed the (incorrect) algorithm had been in use in several tools. The detection of this subtlety motivates a need for a cohesive framework for understanding the various versions of timed automata.

In this chapter, we not only **formally define** many of these variants but provide **formal equivalence** proofs and conversions, with the equivalences of different strengths. While the strongest equivalence is semantic isomorphism (Definition 3.1.2), all of the equivalences are strong enough to be congruences for bisimulation (Definition 2.5.2). All equivalence proofs are on the semantic level.

## 3.1 Types of Equivalence

In this chapter, we will show how different variants of timed automata can be converted into the baseline theory and vice versa. To argue for the correctness of these conversions, we will use various equivalences at the level of the transitions systems corresponding to the different automata; in particular, we will show that the transition system corresponding to a given automaton is equivalent, in a precisely defined sense, to the transition system for the translated version of that automaton.

A natural question presents itself in this setting: which equivalence should one use for these arguments? In general, our view is that the equivalence that is shown should be as strong as possible, so that one may conclude that the conversion routines at the level of timed automata preserve as much semantic information as possible.

We focus on two kinds of equivalence: *bisimulation*, and *isomorphism*. In particular, we use the equivalences of: label-preserving isomorphism (sometimes referred to as isomorphism), reachable subsystem isomorphism, and non-label-preserving isomorphism. By definition, a label-preserving isomorphism and a reachable subsystem isomorphism are bisimulations, and a non-label preserving isomorphism has most of the desirable properties of a bisimulation. The formal definition of bisimulation is given in Section 2.5.1.

### 3.1.1 Label-Preserving Isomorphism

A stronger equivalence than bisimulation is a label-preserving isomorphism. Two systems have a label-preserving isomorphism if we can convert one system to the other by renaming the states of the first system. A label-preserving isomorphism requires that the transitions have the same labels.

**Definition 3.1.1 (Isomorphism, transition systems ($TS_1 \cong TS_2$)).** Consider two transition systems $TS_1 = (Q_1, Q_{0_1}, \Sigma_1, \longrightarrow_1)$ and $TS_2 = (Q, Q_{0_2}, \Sigma_2, \longrightarrow_2)$ where $\Sigma_1 = \Sigma_2$. The two transition systems are *label-preserving isomorphic* (*isomorphic*), ($TS_1 \cong TS_2$), iff there exists a bijection $f\colon Q_1 \longrightarrow Q_2$ where :

$$q_1 \xrightarrow{a}_1 q_1' \Leftrightarrow f(q_1) \xrightarrow{a}_2 f(q_1')$$

and $q_1 \in Q_{0_1}$ if and only if $f(q_1) \in Q_{0_2}$. When clear from context, we will refer to a label-preserving isomorphism as an *isomorphism*. ■

This definition may be extended to timed automata as follows.

**Definition 3.1.2 (Isomorphism of timed automata ($TA_1 \cong TA_2$)).** Let $TA_1$ and $TA_2$ be two timed automata. The two timed automata are *label-preserving isomorphic* (*isomorphic*) ($TA_1 \cong TA_2$), if and only if $TS(TA_1) \cong TS(TA_2)$. When clear from context, we will refer to a label-preserving isomorphism as an *isomorphism*. ■

### 3.1.2 Reachable Subsystem Isomorphism

A relaxation of $\cong$ that is still stronger than bisimulation may be obtained by considering only the reachable states of a timed transition system. Intuitively, the reachable subsystem of a timed transition system contains only those states and transitions reachable from an initial state.

**Definition 3.1.3 (Reachable subsystem ($Reach(TS)$)).** Given a transition system $TS$, we form its *reachable subsystem, $Reach(TS)$*, which is a transition system $Reach(TS) = (Q_r, Q_{0_r}, \Sigma, \longrightarrow_r)$, specified as follows.

- $Q_r = \{q \mid \exists q_0 \in Q_0 \text{ s.t. } q_0 \longrightarrow^* q\}$. Here $\longrightarrow^*$ means there is a sequence of (zero or more) transitions from $q_0$ to $q$.

- $Q_{0_r} = Q_0$, since all initial states are reachable.
- $\longrightarrow_r = \{(q, a, q') \mid q, q' \in Q_r \text{ and } (q, a, q') \in \longrightarrow\}$.

■

Thus, $Q_r$ is the set of all states obtained after executing all possible transitions starting from the initial states and continuing with new transitions until there are no more transitions available.

**Definition 3.1.4 (Isomorphism of reachable subsystems ($TA_1 \cong_r TA_2$)).** Two timed automata $TA_1$ and $TA_2$ are *reachable subsystem isomorphic*, denoted $TA_1 \cong_r TA_2$ or $TS(TA_1) \cong_r TS(TA_2)$, iff $Reach(TS(TA_1)) \cong Reach(TS(TA_2))$. ■

Note that an isomorphism of reachable subsystems induces a bisimulation.

### 3.1.3 Non-Label-Preserving Isomorphism

Now we define a non-label-preserving isomorphism. The intuition is that we use two independent bijections: a bijection for the states and a bijection for the action labels. The transition relation is preserved if we relabel the states and the action symbols.

**Definition 3.1.5 (Non-label-preserving isomorphism, ($TS_1 \cong_{nl} TS_2$)).** Let $TS_1 = (Q_1, Q_{0_1}, \Sigma_1, \longrightarrow_1)$ and $TS_2 = (Q, Q_{0_2}, \Sigma_2, \longrightarrow_2)$ be two transition systems where $\Sigma_1 = \Sigma_2$. We say that the two transition systems are *non-label-preserving isomorphic* ($TS_1 \cong_{nl} TS_2$), if and only if there exists two bijections $f_q \colon Q_1 \longrightarrow Q_2$, and $f_\sigma \colon \Sigma_1 \longrightarrow \Sigma_2$ where

$$q_1 \xrightarrow{a}_1 q_1' \Leftrightarrow f_q(q_1) \xrightarrow{f_\sigma(a)}_2 f_q(q_1')$$

and $q_1 \in Q_{0_1}$ if and only if $f_q(q_1) \in Q_{0_2}$. ■

**Definition 3.1.6 (Non-label-preserving isomorphism, ($TA_1 \cong_{nl} TA_2$)).** Let $TA_1$ and $TA_2$ be two timed automata. We say that the two automata systems are *non-label-preserving isomorphic* ($TA_1 \cong_{nl} TA_2$), if and only if $TS(TA_1) \cong_{nl} TS(TA_2)$. ■

While not a bisimulation, a non-label-preserving isomorphism makes strong assertions about behavior of equivalent systems. After we relabel one of the timed automaton, the relabeled automaton is isomorphic to the other timed automaton.

Another equivalence used in many algorithms is **region equivalence.** We formally define region equivalence in Section 2.5.2. Recall that region equivalence groups clock valuations into regions whose valuations enforce the same clock constraints (up to a certain constant *maxc*). Applying the region equivalence relation to a timed automaton creates a new automaton, the region timed automaton (see Definition 2.5.7). The region automaton is both bisimilar to its timed automaton and finite (see Sections 2.5.2 and 2.5.3). This equivalence allows many properties, including reachability of timed automata, to be decidable. For more information, see Sections 2.5.2 and 2.5.3, or see Alur [5], Alur and Dill [7], Clarke et al. [55].

## 3.2 Variants and Conversions: An Overview

In this section, we overview the various variants and equivalence conversions. The details are provided in the remainder of this chapter as well as in Fontana and Cleaveland [74].

### 3.2.1 Variants

Here we summarize the variants discussed throughout this Chapter. The variants are classified according to the kind of equivalence each variant has with the baseline version (Definition 2.2.2).

First we summarize the variants semantically isomorphic (Definition 3.1.2) to the base version.

- **Disjunctive guard constraints**. Sources using this variant include Bérard et al. [28]. A disjunction of constraints for a guard on an edge is converted to a set of edges, where each disjunct of the guard constraint is converted to one edge.

- **Timed automata with variables.** Sources using this variant include Zhang and Cleaveland [167]. Sources and (especially) tool implementers like to augment timed automata with finitely-valued integer (or boolean) variables that can be assigned on edge transitions and compared on edge guards and checked in invariants. This is a useful implementation shorthand that can make model generation and model checking easier. The variables do not add expressive power because the set of variable assignments can be represented with a finite number of locations (one location per assignment).

- **Guarded-command programs.** Sources using this variant include Henzinger et al. [88]. A guarded-command program is a model used by Henzinger et al. [88] and others to represent real-time programs. Guarded-command programs do not use action symbols but can be easily augmented to allow them. Guarded-command programs are equivalent to timed automata with variables without using action symbols by converting the guards to edges (some being self loops). Since timed automata with variables are equiva-

lent to timed automata without variables, guarded-command programs are equivalent to timed automata.

Next, we describe some variants whose equivalence becomes isomorphism if we reduce both timed automata (the original and the converted) to the subsystem that can be reached from their initial states. The reachable-subsystem isomorphism equivalence is formally defined in Definition 3.1.4. Again, a summary list is provided here.

- **Unsatisfied invariants.** Sources using this variant include Alur [5] (unsatisfied invariants result in urgent states) and Bengtsson and Yi [27] (unsatisfied invariants prevent transitions). While a time advance always requires the invariant to be true, this requirement of a true invariant is sometimes waived for action transitions. The following two variants of automata are equivalent: having urgent locations and preventing action executions when invariants are unsatisfied and not having urgent locations but allowing action executions when invariants are false. Urgent locations are not needed in the latter case since they can be encoded as a location with a false invariant.

- **Clock difference inequalities in clock constraints.** Sources using this variant include Bérard et al. [28] (gives conversion of timed automata with clock difference inequalities in constraints to timed automata that do not have clock difference inequalities in constraints) and Bouyer [36] (shows that a model checking technique of widening a set of clock valuations defined by a clock constraint need to be refined for automata with clock difference inequalities). Some sources allow timed automata constraints to contain inequalities on clock differences. While we can always convert a timed automaton to eliminate these clock difference inequalities [28], the conversion

yields an exponential blowup in terms of the number of inequalities converted. What makes clock difference inequalities hard to deal with is that the standard widening method for model checking is not always sound when we allow clock difference inequalities in clock constraints [36].

Lastly, we discuss a variant that has an equivalence, but is not isomorphism. This equivalence require re-labelings to show an isomorphism, or that if we take the converted system and re-label the time advance labels, the system with the changed labels is isomorphic to the original automaton. A summary with one source is provided here.

- **Rational clock constraints**. Sources using this variant include Alur [5]. Most clock constraints allow only non-negative integers, but this is expressively equivalent to allowing non-negative rational constraints, since we can convert a timed automaton with rational constraints to one without rational constraints. This equivalence differs because it requires time advance labels to be mapped to different values to obtain the equivalence.

- **Clock assignments.** Sources using this variant include Yovine [164]. This variant extends edges to allow a clock to be assigned the current value of another clock; hence on an edge each clock can be reset to 0 or set to the pre-edge value of another clock. Bouyer et al. [39] showed that we can convert each such automaton to a bisimilar timed automaton without clock assignments.

### 3.2.2 Establishing Equivalence

For each variant $V$, we go through the following steps to establish its equivalence to the baseline variant:

1. Define a **syntactic** conversion that takes a timed automaton of variant $V$ and convert it to the baseline variant (removing the feature). This conversion is well defined in that given a timed automaton of variant $V$ always yields a timed automaton.

2. Formalize the conversion on the **semantic** level. By the definition of timed automata semantics, the timed automaton's semantics are well defined.

3. Prove the proper equivalence on the semantic level. To show isomorphism, given $TA_V$ and its converted timed automaton $CONV(TA)$, define a function $f$ that maps states of $TS(TA_V)$ to states in the converted automaton, $TS(CONV(TA_V))$. This function operates on the **semantic** level. Then prove it is an invertible bijection and that it has the morphism property. The morphism property means for all states $q$, $q'$ of the original timed automaton:

$$q \xrightarrow{\delta} q' \Leftrightarrow f(q) \xrightarrow{\delta} f(q') \; \forall \delta \in \mathbb{R}^{\geq 0} \text{ and}$$
$$q \xrightarrow{a} q' \Leftrightarrow f(q) \xrightarrow{a} f(q') \; \forall a \in \Sigma$$

   For reachable subsystem isomorphism, we utilize the fact that only reachable states are considered.

### 3.2.3 Composition of Variant Conversions

In the previous sections we discuss many variants of timed automata and show how each individual variant can be "translated away" when added to the formalism. In practice, it is possible to have a timed automaton that has many of these variant features, such as data variables, disjunctive guard constraints and clock difference inequalities allowed in clock constraints.

We can perform the conversions on extended timed automata (with more of

these variants and some other features) and compose the conversions. Furthermore, the composition preserves the minimum equivalence of the conversions and that for these conversions, their composition is commutative and associative for the semantics of the final automaton (the syntax is not necessarily commutative or associative). We also give conditions on other extensions not defined in this paper that allow our conversion functions to remove our variants from these extended timed automata. The formal theorem is given as Theorem 3.6.5.

Furthermore, we generalize this framework to consider other possible variants. Three common features, **atomic propositions** (see Alur [5]), **labeling functions** (see Alur [5]), and **clock assignments** (see Yovine [164]) can be added to timed automata without hindering this composition framework.

Full details are in Section 3.6.4.

## 3.3 Timed Automata Equivalences: (Label-Preserving) Isomorphism

This section considers several additions to the base timed automaton formalism and shows that for every timed automaton using one of these new features, there is a label-preserving isomorphic automaton (see Definition 3.1.2) in the baseline version.

### 3.3.1 Disjunctive Guard Constraints

The first extension considered allows disjunctions within transition guards.

**Definition 3.3.1** (**Disjunctive clock constraint** $\phi \in \Phi_\vee(CX)$)**.** Given a nonempty finite set of clocks $CX = \{x_1, x_2, \ldots, x_n\}$ and $c \in \mathbb{Z}^{\geq 0}$, a *disjunctive clock constraint*

$\phi$ may be constructed using the following grammar:

$$\phi ::= x_i < c \mid x_i \leq c \mid x_i > c \mid x_i \geq c \mid \phi \wedge \phi \mid \phi \vee \phi$$

$\Phi_\vee(CX)$ is the set of all possible disjunctive clock constraints over $CX$. ■

**Definition 3.3.2 (Timed automaton with disjunctive constraints).** A *timed automaton with disjunctive clock constraints* $TA = (L, L_0, L_u, \Sigma, CX, I, E_\vee)$ is a tuple where:

- $L, L_0 \subseteq L, L_u \subseteq L, \Sigma, CX$, and $I\colon L \longrightarrow \Phi(CX)$ are as in Definition 2.2.2.
- $E_\vee \subseteq L \times \Sigma \times \Phi_\vee(CX) \times 2^{CX} \times L$ is the set of *edges*. In an edge $e = (l, a, \phi, \lambda, l')$ from $l$ to $l'$ with action $a$, $\phi \in \Phi_\vee(CX)$ is the *guard* of $e$, and $\lambda$ represents the set of clocks to *reset* to 0.

■

This definition differs from that of Definition 2.2.2 in that disjunctive clock constraints are permitted in guards and transitions but not in location invariants.

The semantics of timed automata with disjunctive clock constraints in guards is the expected adaptation of the semantics of the baseline automata (see Definition 2.2.6).

**Conversion to Base Formalism**

This approach is from Bérard et al. [28]. Given an edge with a guard $\phi$ that contains a disjunction of constraints, we first use logical equivalences to convert $\phi$ in a disjunction of conjunctive clauses (*disjunctive normal form*) and then convert the edge to a set of edges such that each clause is the guard of some edge. For more information on disjunctive normal form and how to convert a formula to

disjunctive normal form, see Enderton [70]. For the remainder of this conversion, we assume that $\phi$ is in disjunctive normal form.

Formally, let $e = (l, a, \phi, \lambda, l')$ be an edge where $\phi = \bigvee_{i=1}^{m} \phi_i$, is a clock constraint in disjunctive normal form with clauses $\phi_1, \phi_2 \ldots \phi_m \in \Phi(CX)$ (no disjunctions in each $\phi_j$). Then we remove the edge $e$ and add each edge $e_j = (l, a, \phi_j, \lambda, l')$ for each $j$ from 1 to $m$ to the edge set $E$. If we take a timed automaton $TA$ that has disjunctive guard constraints and apply this conversion, the resulting timed automaton is called $DIS(TA)$. $DIS(TA)$ is a timed automaton of the base variant.

We now show that the timed automaton with disjunctive edge constraints is label-preserving isomorphic to the timed automaton with the set of edges.

**Theorem 3.3.1.** Let $TA = (L, L_0, L_u, \Sigma, CX, I, E)$ be a timed automaton with disjunctive constraints on its guards and let $DIS(TA) = (L, L_0, L_u, \Sigma, CX, I, E')$ be the converted timed automaton. Then $TS(TA) \cong TS(DIS(TA))$ ($TS(TA)$ is label-preserving isomorphic to $TS(DIS(TA))$).

***Proof of Theorem 3.3.1.*** Consider the function:

$$f : Q_{TS(TA)} \longrightarrow Q_{TS(DIS(TA))}$$
$$f\big((l, \nu)\big) = (l, \nu)$$

$f$ is the identity function; therefore, $f$ is a bijection.

We claim $f$ is the isomorphism required by Definition 3.1.1. Consider a state $(l, \nu)$ in $Q_{TS(TA)} = Q_{TS(DIS(TA))}$, and suppose $(l, \nu) \xrightarrow{a} (l', \nu')$ in $TS(TA)$. In the original automaton it took some edge $e$ with guard $\phi = \bigvee_{i=1}^{m} \phi_i$. This can happen iff there is an $e = (l, a, \phi, \lambda, l') \in E$ such that $\nu \models \phi$ and $\nu[\lambda := 0] \models I(l')$. Assuming

$\phi = \bigvee_{i=1}^{m} \phi_i$ is in DNF, this is true iff $v \models \phi_i$ for some $i$. But $e = (l, a, \phi, \lambda, l') \in E'$ and $(l, v) \xrightarrow{a} (l', v')$ in $TS(DIS(TA))$. Thus, we know that $v \models \phi$ if and only if there is some $i$ where $v \models \phi_i$. □

*Remark* 3.3.1 (Disjunctive constraints on invariants). Given the ease of converting out disjunctive constraints on guards, one may try to apply a similar procedure to remove disjunctions from invariants. However, this is not as easy. Consider the constraint $x_1 \leq 2 \vee x_2 \leq 3$. The set of clock valuations satisfying this constraint forms a *non-convex region*. One initial attempt would be to have two locations, one whose invariant is $x_1 \leq 2$ and the other whose invariant is $x_2 \leq 3$. However, in the original automaton, time advances that preserve the disjunction may invalidate one or the other of the disjuncts. Reflecting this in the new automaton would require the introduction of actions symbols to change location. This introduction of new action transitions fundamentally changes the semantics.

### 3.3.2 Timed Automata with Variables

Many sources, including Behrmann et al. [23], Ben Salah et al. [25], Bowman and Gómez [47], Gómez and Bowman [78], Morbé et al. [127], Zhang and Cleaveland [167, 168], allow timed automata to have finitely many variables whose values are drawn from a finite subset of the integers. These variables can be used in constraints and assigned new values on action executions. Morbé et al. [127] refers to these extended automata as timed automata with integer variables, while Ben Salah et al. [25] extends the definition further by distinguishing between input and output variables. In this section we show how the formalism of Morbé et al. [127] can be converted to our base formalism.

**Definitions**

For a timed automaton with variables, we augment each location with a *data valuation*: a function assigning each variable to some integer in a finite subset of integers.

**Definition 3.3.3 (Data valuation $\nu_d \in \mathcal{DV}_d$).** Let $VR$ be a finite set of variables with $VR \cap CX = \emptyset$, and let $\mathbb{Z}_f \subseteq \mathbb{Z}$ be such that $|\mathbb{Z}_f| < \infty$. A *data valuation*, or an *interpretation of data variables*, is a function $\nu_d \colon VR \longrightarrow \mathbb{Z}_f$. $\nu_d(p)$ is the value of variable $p$ assigned by $\nu_d$. $\mathcal{DV}_d$ is the set of all data valuations.

We use the following terminology and notation:

- If $V \subseteq VR$, then we refer to $dA \colon V \longrightarrow \mathbb{Z}_f$ as an *assignment function*. For such a $dA$, we use $dA[V]$ to denote that $V$ is the domain of $dA$. We use $\mathcal{DA}$ to represent the set of all data assignments.
- If $\nu_d$ is a data valuation and $dA$ an assignment function, then $\nu_d[dA]$ (or $\nu_d[dA[V]]$) denotes the data valuation after applying the assignment function $dA$ with domain $V$. This changes the values of all variables in $V \subseteq VR$ and leaves all values of variables not in $V$ unchanged. Formally,

$$\nu_d[dA](p) = \begin{cases} dA(p) & p \in V \\ \nu_d(p) & p \notin V \end{cases} \tag{3.1}$$

■

**Definition 3.3.4 (Clock constraint with variables).** Given a nonempty finite set of clocks $CX = \{x_1, x_2, \ldots, x_n\}$, a set of variables $VR$ ($CX \cap VR = \emptyset$), $c, d \in \mathbb{Z}^{\geq 0}$, and $p \in VR$, a *clock constraint with variables* $\phi$ may be constructed with the following grammar. Note that such a constraint segregates the data-variable part

of the property ($\phi_d$) from the clock part ($\phi_c$).

$$\begin{aligned}
\phi &::= (\phi_d \wedge \phi_c) \\
\phi_c &::= x_i \leq c \mid x_i \geq c \mid x_i < c \mid x_i > c \mid \phi_c \wedge \phi_c \\
\phi_d &::= \texttt{tt} \mid \texttt{ff} \mid p_j < d \mid p_j > d \mid p_j \leq d \mid p_j \geq d \mid \phi_d \wedge \phi_d \mid \phi_d \vee \phi_d
\end{aligned}$$

$\Phi(CX \cup VR)$ is the set of all clock constraints with variables. Recall that for clock constraints $\phi_c$, we have the following abbreviations: true (`tt`) for $x_1 \geq 0$, false (`ff`) for $x_1 < 0$, and $x_i = c$ for $x_i \leq c \wedge x_i \geq c$. ■

As defined in Definition 2.2.1, $\phi_c$ is a regular clock constraint. The intuition of the construction of clock constraints with variables is to restrict transitions by adding data constraints to the clock constraints. Thus, satisfaction of constraints is now $(\nu_d, \nu) \models \phi$ where $\nu_d$ is a data valuation and $\nu$ is a clock valuation, and is defined similar to the satisfaction of clock constraints (Definition 2.2.4).

We separate clock and data constraints because data constraints can be more permissive with respect to disjunction. (In the grammar above, we allow disjunction on data constraints.) In particular, when constraints are used as location invariants, disjunctions on clock constraints propose translational problems. See Remark 3.3.1.

**Definition 3.3.5 (Timed automaton with variables).** A *timed automaton with variables* $TAV = (L, L_0, L_u, VR, VR_0, \Sigma, CX, I, E)$ is a tuple where:

- $L$, $L_0$, $L_u$, $\Sigma$, and $CX$ are as in Definition 2.2.2.
- $VR$ is the set of data variables.
- $VR_0 \subseteq \mathcal{DV}_d$ is the set data valuations representing the possible initial val-

ues of the data variables (often the single assignment of 0 for all variables, provided $0 \in \mathbb{Z}_f$).

- $I\colon L \longrightarrow \Phi(CX \cup VR)$ gives the invariant for each location $l$.
- $E \subseteq L \times \Sigma \times \Phi(CX \cup VR) \times 2^{CX} \times \mathcal{DA} \times L$ is the set of edges. In an edge $e = (l, a, \phi, \lambda, dA[V], l')$ from $l$ to $l'$ with action $a$, $\phi \in \Phi(CX \cup VR)$ is referred to as the *guard* of $e$ which has constraints both on clocks and variables. $\lambda$ represents the set of clocks to reset and $dA[V]$ represents the data assignment function which gives new values to the data variables in $V \subseteq VR$.

In this definition we require $CX \cap VR = \emptyset$. ■

**Definition 3.3.6 (Timed automaton with variables semantics).** The *semantics of a timed automaton with variables* $TAV = (L, L_0, L_u, VR, VR_0, \Sigma_{TA}, CX, I, E)$ is a transition system $TS(TAV) = (Q, Q_0, \Sigma, \rightarrow)$ where:

- $Q = L \times \mathcal{DV}_d \times \mathcal{V}$ is the set of states and $q = (l, \nu_d, \nu)$ is a state consisting of a location, data valuation, and clock valuation.
- $Q_0 = L_0 \times VR_0 \times [CX := 0]$ (clocks are 0 and variables are at their initial values).
- $\Sigma = \mathbb{R}^{\geq 0} \cup \Sigma_{TA}$.
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is defined as follows:

  **Time advancement** $(l, \nu_d, \nu) \xrightarrow{\delta} (l, \nu_d, \nu + \delta)$ iff

  $l \in L, l \notin L_u$, $\delta \in \mathbb{R}^{\geq 0}$, and $\forall t \in \mathbb{R}^{\geq 0}, 0 \leq t \leq \delta\colon (\nu_d, \nu + t) \models I(l)$.

  **Action execution** $(l, \nu_d, \nu) \xrightarrow{a} (l', \nu_d[dA], \nu[\lambda := 0])$ iff

  $\exists \phi$ such that $(l, a, \phi, \lambda, dA\rangle, l') \in E$, $(\nu_d, \nu) \models \phi$, and $(\nu_d[dA], \nu[\lambda := 0]) \models I(l')$.

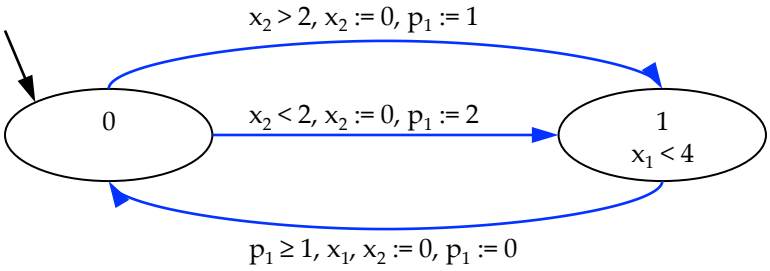


**Figure 3.1:** Timed automaton with variables $TAV_1$ with variable $p_1$. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

■

For each timed automaton with variables, the set of variables must be specified, and each variable must be given an initial value. Guards and invariants are also allowed to contain constraints on variables, and edges can assign variables new values. Assignment functions encode the changing of variables on transition executions.

For simplicity, these automata only allow variable assignments to constants. Should one wish to encode transitions such as variable $p_i$ transitions to $p_i + 1$, since $p_i$ only has a finite number of values, we can enumerate over all possible values of $p_i$ and convert this expression into a set of transitions.

**Example 3.3.1.** Consider the timed automaton with variables $TAV_1$ in Figure 3.1 with $VR = \{p_1\}$, $VR_0 = \{p_1 = 0\}$ and $\Sigma = \{a\}$ (a's omitted in diagram). If we examine the edge $(1, a, p_1 \geq 1, \{x_1, x_2\}, p_1 := 0, 0)$, $p_1 \geq 1$ is the guard (note the clock constraint is trivially true in this case and is omitted) and $p_1 := 0$ is the assignment function with $dom(p_1 := 0) = \{p_1\}$. ■

**Conversion to Base Formalism**

Any timed automaton is a timed automaton with variables where $VR = \emptyset$. To convert a timed automaton with variables to a timed automaton, we create more locations by replacing each location with a set of (location, variable assignment) pairs. We then restrict edges to locations that satisfy the variable constraints: source locations must satisfy the variable constraints in the guard, and destination locations are restricted to the source locations after executing the variable assignments on the edges.

**Definition 3.3.7 (Substituted variable constraint $\phi[VR \mapsto \nu_d]$).** Suppose variable set $VR$, clock constraint with variables $\phi \in \Phi(CX \cup VR)$, and data valuation $\nu_d$. Because by construction, $\phi = \phi_d \wedge \phi_c$, we define the *substituted variable constraint* $\phi[VR \mapsto \nu_d]$ as $\phi[VR \mapsto \nu_d] = (\phi_d[VR \mapsto \nu_d] \wedge \phi_c[VR \mapsto d])$. We call $\phi_d[VR \mapsto \nu_d]$ a *substituted data constraint*. Since $\phi_d$ is a conjunction of variable inequalities ($p$ denotes a variable), using $\bowtie \in \{<, \leq, >, \geq\}$, $\phi_d[VR \mapsto \nu_d]$ is formally defined as follows:

$$\phi_d[VR \mapsto \nu_d] = \begin{cases} \texttt{tt} & \phi_d = p \bowtie c \text{ and } \nu_d(p) \bowtie c \\ \texttt{ff} & \phi_d = p \bowtie c \text{ and } \nu_d(p) \not\bowtie c \\ \phi_1[VR \mapsto \nu_d] \wedge \phi_2[VR \mapsto \nu_d] & \phi_d = \phi_1 \wedge \phi_2 \end{cases} \tag{3.2}$$

The constraint is then simplified using logical equivalences, which by construction, will simplify to `tt` or `ff`. Since $\phi_c$ contains no data variables, $\phi_c[VR \mapsto \nu_d] = \phi_c$. In summary, $\phi[VR \mapsto \nu_d] = \phi_c$ if $\nu_d \models \phi_d$ and $\phi[VR \mapsto \nu_d] = \texttt{ff}$ otherwise ($\nu_d \not\models \phi_d$). ■

The translation procedure from a timed automaton with variables to a timed automaton is formalized as follows.

**Convert a timed automaton with variables** $TAV$ **to a timed automaton** $TA$ Given $TAV = (L_V, L_{0_V}, L_{u_V}, VR, VR_0, \Sigma_V, CX_V, I_V, E_V)$, Construct timed automaton $TA(TAV) = (L, L_0, L_u, \Sigma, CX, I, E)$ as follows:

- $L = L_V \times \mathcal{DA}[VR]$. Each location can be seen as a pair $(l, \nu_d)$, which is a location and a data valuation.
- $L_0 = \{(l_v, \nu_{d_0}) \mid l_v \in L_{0_V} \text{ and } \nu_{d_0} \in VR_0\}$ the set of locations with an initial location and the initial variable assignment.
- $L_u = L_{u_V} \times \mathcal{DV}_d$.
- $\Sigma = \Sigma_V$.
- $CX = CX_V$.
- $I \colon L \longrightarrow \Phi(CX)$, with $I(l = (l_p, \nu_d)) = I(l_p)[VR \mapsto \nu_d]$.
  (Note: $I(l_p) \in \Phi(CX \cup VR)$; hence, $I(l_p)[VR \mapsto \nu_d] \in \Phi(CX)$.)
- $E = \{((l_p, \nu_d), a, \phi[VR \mapsto \nu_d], \lambda, (l_p, \nu_d[dA])) \mid (l_p, a, \phi, \lambda, dA[V], l'_p) \in E_V, \nu_d \in \mathcal{DV}_d, \text{ and } \phi[VR \mapsto \nu_d] \neq \texttt{ff}\}$. Because multiple valuations $\nu_d$ result in guards that are not false, a single edge $e_v \in E_V$ may be converted to multiple edges in $E$ (Note: if a guard simplifies to `ff`, the edge is removed.)

The conversion makes a location for each possible (location, variable assignment) pair. The new edges are similar to the original edges except that the assignment function $dA$ and the variable assignments are substituted into the guards, invariants, and locations of the edges.

**Example 3.3.2 (Converting example 3.3.1).** Again consider the timed automaton

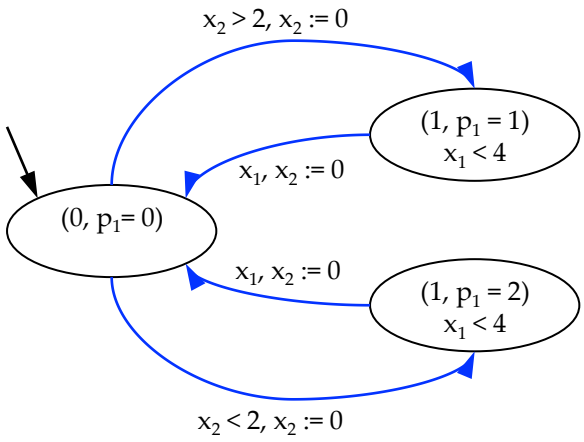


**Figure 3.2:** Timed automaton $TA(TAV_1)$. Only locations reachable from the initial location are shown. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

with variables in Figure 3.1 with $VR = \{p_1\}$ and $VR_0 = \{p_1 = 0\}$. The converted timed automaton is given in Figure 3.2. There are three reachable locations: $(0, p_1 = 0)$, $(1, p_1 = 1)$, and $(1, p_1 = 2)$. (Unreachable locations are not shown.) ■

The correctness of the conversion follows from Theorem 3.3.2, where $TA(TAV)$ is the timed automaton after applying the conversion to eliminate the variables from timed automaton with variables $TAV$.

**Theorem 3.3.2.** Let $TAV$ be a timed automaton with variables and $TA(TAV)$ be the timed automaton from our conversion. Then $TS(TAV) \cong TS(TA(TAV))$ ($TS(TAV)$ is label-preserving isomorphic to $TS(TA(TAV))$).

***Proof of Theorem 3.3.2.*** From our conversion, both automata have the same event

set $\Sigma$. Consider the function $f$:

$$f\colon Q_{TS(TAV)} \longrightarrow Q_{TS(TA(TAV))}$$
$$f\big((l, \nu_d, \nu)\big) = (l \times \nu_d, \nu) = (l, \nu_d, \nu)$$

or the identity function. The function $f$ maps $Q_{0_{TS(TAV)}}$ to $Q_{0_{TS(TA(TAV))}}$, preserving initial states. Here $f$ maps the variable assignment of $\nu_d$ to a component of the location. This results states with different data variable assignments having different locations.

**Part 1:** $f$ is a bijection.

Because $f$ is the identity function, $f$ is a bijection. Here is the definition of $f^{-1}$:

$$f^{-1}\colon Q_{TS(TA(TAV))} \longrightarrow Q_{TS(TVA)}$$
$$f^{-1}\big((l \times \nu_d, \nu)\big) = (l, \nu_d, \nu)$$

We move the assignments of variables from locations to data valuations.

**Part 2:** $f$ preserves the transition relation.

We need to show:

$$(l, \nu_d, \nu) \xrightarrow{\delta} (l', \nu'_d, \nu') \Leftrightarrow f((l, \nu_d, \nu)) \xrightarrow{\delta} f((l', \nu'_d, \nu'))\ \forall \delta \in \mathbb{R}^{\geq 0} \text{ and}$$
$$(l, \nu_d, \nu) \xrightarrow{a} (l', \nu'_d, \nu') \Leftrightarrow f((l, \nu_d, \nu)) \xrightarrow{a} f((l', \nu'_d, \nu'))\ \forall a \in \Sigma$$

Showing $f$ does not eliminate transitions, we prove the $\Rightarrow$ direction. We omit the proof of the other ($\Leftarrow$) direction; its proof is similar and (one proof) uses $f^{-1}$ instead of $f$.

**Part 2a:** Time advances: $(l, \nu_d, \nu) \xrightarrow{\delta} (l', \nu'_d, \nu') \Rightarrow f((l, \nu_d, \nu)) \xrightarrow{\delta} f((l', \nu'_d, \nu'))$.

Suppose we have a transition $(l, \nu_d, \nu) \xrightarrow{\delta} (l, \nu_d, \nu + \delta)$ in $TS(TVA)$, for some $\delta \in \mathbb{R}^{\geq 0}$. By the definition of the transition system semantics of $TS(TVA)$ (Definition 3.3.6), $\forall t, 0 \leq t \leq \delta$: $(l, \nu_d, \nu) + t \models I(l)$. By our definition of $f$, it follows that $\forall t, 0 \leq t \leq \delta$: $f((l, \nu + t)) \models I(l)$ iff $(l, \nu + t) \models I(l)$. If I(l) has no variables in the constraint this is already true. If it has variable constraints, then we know, by the definition of $f$, those variables satisfy the constraints in $I(l)$ if and only if the converted constraint $I(l)[VR \mapsto \nu_d]$ is satisfied by $f(l, \nu_d, \nu)$. Thus, by the definition of timed transition system semantics, we have the edge $f((l, \nu_d, \nu)) \xrightarrow{\delta} f((l, \nu_d, \nu + \delta))$.

**Part 2b:** Edge executions: $(l, \nu_d, \nu) \xrightarrow{a} (l', \nu'_d, \nu') \Rightarrow f((l, \nu_d \nu)) \xrightarrow{a} f((l', \nu'_d, \nu'))$.

Now suppose for some action $a$, $(l, \nu_d, \nu) \xrightarrow{a} (l', \nu_d, \nu')$, where $(\nu'_d, \nu') = (\nu_d[dA], \nu[\lambda := 0])$. By the definition of the timed automaton with variables semantics, this transition corresponds to an edge in the timed automaton with variables $(l, a, \phi, \lambda, dA[V], l')$ such that $(\nu_d, \nu) \models \phi$ and $(\nu_d[dA], \nu[\lambda := 0] \models I(l')$. By our conversion algorithm, the converted timed automaton has the edge $(l_t = (l, \nu_d), a, \phi', \lambda, l')$ with $\phi' = \phi[VR \mapsto \nu_d]$. By our definition of $f$, $f((l, \nu_d, \nu)) \models \phi'$ and $f((l', \nu_d[dA], \nu[\lambda := 0])) \models I(l[VR \mapsto \nu_d))$. The state $f((l', \nu_d[dA], \nu[\lambda := 0]))$ corresponds to $f((l, \nu_d, \nu))[\lambda := 0, dA[V]]$. Therefore, by the definition of timed automaton semantics, we have $f((l, \nu_d, \nu)) \xrightarrow{a} f((l', \nu'_d, \nu'))$. □

**Implementation Implications**

While theoretically equivalent, variables provide a compact and clean notation for states. Variables are used many in tool implementations [27, 154, 168].

### 3.3.3 Guarded-Command Programs

Guarded-command programs are a notation for real-timed systems used in Henzinger et al. [87, 88], Zhang and Cleaveland [167]. Using the equivalence of timed automata with variables to timed automata (see Section 3.3.2), it suffices to convert guarded-command programs to and from timed automata with variables. The conversion is based on the one given in Henzinger et al. [88].

#### Syntax

**Definition 3.3.8** (**Guarded-command program** $GP$)**.** A *guarded-command program* $GP = (CX, VR, G, \phi_0, \phi^{\Box})$ is a tuple where:

- $CX$ is the finite set of clocks.
- $VR$ is the finite set of variables, each of which can take one of finitely many integer values. We require $CX \cap VR = \emptyset$.
- $G \subseteq \Phi(CX \cup VR) \times 2^{CX} \times \mathcal{DA}$ is a set of guarded commands where each guarded command $g = (\psi, Y, dA[V])$, represented as $g = \psi \longrightarrow (Y, dA[V])$, has a clock constraint $\psi \in \Phi(CX \cup VR)$ (see Definition 3.3.4) as a guard, clocks in $Y \subseteq CX$ to reset and the variables $V \subseteq VR$ to assign to the integer values specified by the data assignment function $dA(V)$ (see Definition 3.3.3).
- $\phi_0 \subseteq \mathcal{DV}_d$ is the set of initial variable values.
- $\phi^{\Box} \in \Phi(CX \cup VR)$ is a constraint representing the invariant of the entire guarded-command program, containing both clock constraints and constraints on variables.

■

To represent the semantics of guarded-command programs, we use clock valuations and data valuations (Definition 3.3.3).

**Semantics**

Intuitively, each guarded command $g \in G$ represents an action whose guard $\psi$ must be true in order to execute. When this happens, the assignment function resets the subset of clocks $Y \subseteq CX$ to 0 and also assigns the subset of variables $V \subseteq VR$ to values specified by the assignment function $dA$. The assignment function can also choose to do nothing. Formally, we can encode the guarded-command program as a transition system as follows.

**Definition 3.3.9 (Guarded-command program semantics).** Consider a guarded-command program $GP = (CX, VR, G, \phi_0, \phi^{\Box})$. Then the semantics can be represented as a transition system, $TS(GP)$, where $TS(GP) = (Q, Q_0, \Sigma, \longrightarrow)$ and:

- $Q = \mathcal{DV}_d \times \mathcal{V}$. Each state is represented as $(\nu_d, \nu)$, a data valuation (Definition 3.3.3) and a clock valuation, since each state is an assignment of clock and variable values.
- $Q_0 = dA[VR := \phi_0] \times \mathcal{V}[CX := 0]$, where all data variables have their initial assignment and clock variables are 0.
- $\Sigma = \mathbb{R}^{\geq 0} \cup \{a\}$.
- $\longrightarrow \subseteq Q \times \Sigma \times Q$ is defined as follows:

  **Time advancement** $(\nu_d, \nu) \xrightarrow{\delta} (\nu_d, \nu + \delta)$ iff

  $\delta \in \mathbb{R}^{\geq 0}$ and $\forall t \in \mathbb{R}^{\geq 0}, 0 \leq t \leq \delta$: $(\nu_d, \nu + t) \models \phi^{\Box}$.

  **Action execution** $(\nu_d, \nu) \xrightarrow{a} (\nu_d[dA], \nu[Y := 0])$ iff

  $(\psi, Y, dA[V]) \in G$, $(\nu_d, \nu) \models \psi$, and $\nu_d[dA], \nu[Y := 0] \models \phi^{\Box}$.

■

**Converting a Guarded-Command Program to a Timed Automaton with Variables**

Given a guarded-command program $GP = (CX, VR_G, G, \phi_0, \phi^{\square})$, construct the timed automaton with variables $TAV = (L, L_0, L_u, VR, VR_0, \Sigma, CX, I, E)$ as follows:

- $L = \{l\}$. Make a single "generic" location $l$.
- $L_0 = \{l\}$.
- $L_u = \emptyset$; guarded-command programs do not have urgency.
- $VR = VR_G$.
- $VR_0 = \phi_0$.
- $\Sigma = \{a\}$; $a$ is the generic action symbol.
- $I \colon L \longrightarrow \Phi(CX \times VR)$ where $I(l) = \phi^{\square}$. There is only one location $l$, and $\phi^{\square}$ handles the invariants of the variables.
- $E = \{(l, a, \psi, Y, dA[V], l) \mid (\phi, Y, dA[V]) \in G\}$. The source and target location of each edge are both $l$.

We give the timed automaton with variables one location $l$, and give it the same set of variables as the guarded-command program. Each guard of the guarded-command program becomes a self-loop edge with the same guard and assignment function.

**Converting a Timed Automaton to a Guarded-Command Program**

Given a timed automaton $TA = (L, L_0, L_u, \Sigma, CX, I, E)$ (assuming $L \subseteq \mathbb{Z}$, $\Sigma = \{a\}$ or $\Sigma = \emptyset$ since the events are abstracted out, and assuming $L_u = \emptyset$), construct the

guarded-command program $GP = (CX, VR_G, G, \phi^0, \phi^\square)$ as follows:

- $VR_G = \{loc\}$.
- $G = \bigcup_{(l,a,\psi,Y,l') \in E} \big((loc = l \wedge \psi) \longrightarrow Y, dA'[loc]\big)$, where $dA'[loc]$ is defined to be the variable assignment function with domain $loc$ such that the variable $loc$ is assigned to the value of location $l'$ ($dA'(loc) = l'$).
- $\phi^0 = \bigcup_{l_0 \in L^0} (loc = l_0)$.
- $\phi^\square = \bigcup_{l \in L} (loc = l \wedge \bigwedge_{l' \neq l} (loc \neq l') \wedge I(l))$.

This conversion takes the timed automaton and introduces a variable $loc$ to represent the location as an integer variable. Transitions change location by changing the value of variable $loc$. This conversion also ignores actions.

**Correctness of Conversions**

The correctness of the conversions follow from these two theorems that show there is a label-preserving isomorphism between the semantics of the two systems. These theorems assume that the timed automaton with variables has a single action symbol $a \in \Sigma$ to accommodate the restriction of guarded-command programs not having events. The conversions still hold (with similar theorems) if guarded-command programs are augmented to have action symbols.

**Theorem 3.3.3.** Let $GP$ be a guarded-command program and $TAV(GP)$ be the timed automaton with variables from the conversion. Then $TS(GP) = TS(TAV(GP))$ ($TS(GP)$ is label-preserving isomorphic to $TS(TAV(GP))$).

***Proof of Theorem 3.3.3.*** From our conversion, both automata have the same the

event set $\Sigma = \{a, \delta\}$ with $\delta \in \mathbb{R}^{\geq 0}$. Consider the function $f$:

$$f \colon Q_{TS(GP)} \longrightarrow Q_{TS(TVA(GP))}$$
$$f\big(\nu_d, \nu\big) = (l \times \nu_d, \nu)$$

The function $f$ preserves the variable assignment and maps to the generic location $l$ (which is the generic location $loc$). In the guarded-command program, the variable assignments and clock assignments contain all of the state information. The function $f$ maps $Q_{0_{TS(GP)}}$ to $Q_{0_{TS(TVA(GP))}}$, preserving initial states. Each state $q$ in $Q_{GP}$ represents a location, an assignment to each variable, and an assignment to each clock. $f(q)$ gives us the state in $Q_{TS(TVA(GP))}$, with the variable assignments corresponding to $\nu_d$ and the clock assignments corresponding to $\nu$.

**Part 1:** $f$ is a bijection.

We know that $f$ is a bijection because our conversion converts each state in $Q_{TS(GP)}$ to one and only one state in $Q_{TS(TVA(GP))}$ (the function is like the identity but adds on a "dummy" location $l$). Here is the definition of $f^{-1}$:

$$f^{-1} \colon Q_{TS(TVA(GP))} \longrightarrow Q_{TS(GP)}$$
$$f^{-1}\big((l \times \nu_d, \nu)\big) = (\nu_d, \nu)$$

Here, $f^{-1}$ removes the extraneous single-valued variable $l$ encoding a location.

**Part 2:** $f$ preserves the transition relation.

We need to show:

$$(\nu_d, \nu) \xrightarrow{\delta} (\nu'_d, \nu') \Leftrightarrow f((\nu_d, \nu)) \xrightarrow{\delta} f((\nu'_d, \nu')) \; \forall \delta \in \mathbb{R}^{\geq 0} \text{ and}$$
$$(\nu_d, \nu) \xrightarrow{a} (\nu'_d, \nu') \Leftrightarrow f((\nu_d, \nu)) \xrightarrow{a} f((\nu'_d, \nu')) \; \forall a \in \Sigma$$

Showing $f$ does not eliminate transitions, we prove the $\Rightarrow$ direction. We omit the proof of the other ($\Leftarrow$) direction; its proof is similar and (one proof) uses $f^{-1}$ instead of $f$.

**Part 2a:** Time advances: $(\nu_d, \nu) \xrightarrow{\delta} (\nu'_d, \nu') \Rightarrow f((\nu_d, \nu)) \xrightarrow{\delta} f((\nu'_d, \nu'))$.

Since the location is an encoding of $\nu_{vr}[VR]$, and $\nu_d$ has the same variable assignments as $f((\nu_d, \nu))$, this holds.

**Part 2b:** Edge executions: $(\nu_d, \nu) \xrightarrow{a} (\nu'_d, \nu') \Rightarrow f((\nu_d, \nu)) \xrightarrow{a} f((\nu'_d, \nu'))$.

Since the location is an encoding of $\nu_d$ and $(\nu_d, \nu) = f((\nu_d, \nu))$, this holds. Each guard in the guarded-command program is an edge in the transition system, and that same guard is a self-loop in the timed automaton with variables (which its transition system represents as an edge). □

**Theorem 3.3.4.** Let $TA$ be a timed automaton with $\Sigma = \{a\}$ and $GP(TAV)$ be the the guarded-command program from the conversion. Then $TS(TA) \cong TS(GP(TA))$.

***Proof of Theorem 3.3.4.*** The proof is similar to the proof of Theorem 3.3.3. □

*Remark* 3.3.2 (Past closed invariants). Henzinger et al. [88] restricts guarded-command program invariant clock constraints to be *past-closed*. A clock constraint $\phi$ is *past-closed* if and only if for all $\nu \in \mathcal{V}$ and for all $\delta > 0$, $(\nu + \delta \models \phi) \Rightarrow (\nu \models \phi)$. This restriction is a modeling convenience. The provided conversions are correct regardless of whether the guarded-command program is past closed or not. Furthermore, any past-closed guarded-command program will be converted into a

past-closed timed automaton and vice versa. For generality, we present guarded-command programs without the restriction of past-closed invariants.

## 3.4 Timed Automata Equivalences: Isomorphism of Reachable Subsystems

This section discusses variants with a slightly weaker equivalence to the baseline timed automaton: reachable subsystem isomorphism (Definition 3.1.4).

### 3.4.1 Unsatisfied Invariants

When timed automata are defined, the transition semantics are influenced by the invariants of the source and target locations, and these invariants are used differently in different timed automata variants. While invariants can always prevent time advances in a location, invariants can only sometimes restrict action executions. Throughout the literature, there are two common variants: either the target locations' invariant must be true after an execution, or the target locations' invariant does not affect an action execution. Both invariant interpretations are widely used.

To examine this difference, consider the baseline automaton's semantics of $\longrightarrow$:

$\longrightarrow \subseteq Q \times \Sigma \times Q$ is defined as follows:

**Time advancement** $(l, \nu) \xrightarrow{\delta} (l, \nu + \delta)$ iff

$l \in L, l \notin L_u, \delta \in \mathbb{R}^{\geq 0}$ and $\forall t \in \mathbb{R}^{\geq 0}, 0 \leq t \leq \delta \colon \nu + t \models I(l)$.

**Action execution** $(l, \nu) \xrightarrow{a} (l', \nu[\lambda := 0])$ iff

there is a $\phi$ such that $(l, a, \phi, \lambda, l') \in E$, $\nu \models \phi$, and $\boldsymbol{\nu}[\boldsymbol{\lambda} := \mathbf{0}] \models \mathbf{I}(\mathbf{l}')$.

This interpretation is used in Bengtsson and Yi [27], Bouyer and Laroussinie

[38], Olderog and Dierks [131], Tripakis [151], Wang [159], Yovine [164]. In the above version, for actions to be executed, the target locations' invariants must be satisfied.

Alternatively, we could allow action executions to enter states whose location invariants are unsatisfied. This yields the semantics:

$\longrightarrow \subseteq Q \times \Sigma \times Q$ is defined as follows:

**Time advancement** $(l, \nu) \xrightarrow{\delta} (l, \nu + \delta)$ iff

$l \in L, l \notin L_u, \delta \in \mathbb{R}^{\geq 0}$, and $\forall t \in \mathbb{R}^{\geq 0}, 0 \leq t \leq \delta \colon \nu + t \models I(l)$.

**Action execution** $(l, \nu) \xrightarrow{a} (l', \nu[\lambda := 0])$ iff

there is a $\phi$ such that $(l, a, \phi, \lambda, l') \in E$, and $\nu \models \phi$.

This interpretation is used in Alur [5], Baier and Katoen [17], Behrmann et al. [23], Beyer and Noack [29], Clarke et al. [55], Zhang and Cleaveland [167]. According to the above definition, if a target location's invariant is not satisfied, action executions into it are allowed, but no time is allowed to elapse from that state. Actions can then be executed from that location. Assuming 0-unit time-advances are disallowed in urgent locations, this is the semantic equivalent of *urgency* (see Section 2.2.4 and Section 3.4.1).

**Example 3.4.1 (Unsatisfied invariant).** Consider the timed automaton in Figure 3.3. Let us examine the difference between the two interpretations of invariants. If we forbid actions into states with unsatisfied invariants, then no execution can enter location 1. On the other hand, if we treat each state with an unsatisfied invariant as urgent and allow action executions, some executions can enter location 1 and then immediately transition into location 2. From location 2, those executions can transition at will into location 3. ■

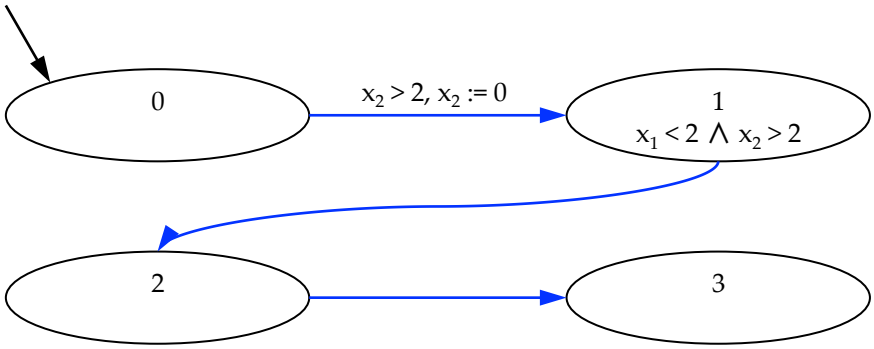


**Figure 3.3:** Timed automaton where the invariant is always initially unsatisfied at location 1. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

Our baseline theory uses the first interpretation of invariants: the target location's invariant must be satisfied for an action execution to occur. The variant presented in this section uses the second interpretation: the target location's invariant does not influence action executions.

Unlike other variants, the baseline variant is not a special case of this variant. Consequently, we not only present the conversion to the baseline formalism but also present the conversion from the baseline formalism to this variant. Because both conversions rely on the *reset predecessor* operator, we define this operator first.

**Reset Predecessor Definition**

When computing the reset predecessor operator, we perform substitution into clock constraints. Similar to substituted variable constraints (Definition 3.3.7), we define an operation that allows clocks in a clock constraint to be replaced with a constant.

**Definition 3.4.1 (Substituted clock constraint $\phi[Y \mapsto D]$).** Given a clock constraint $\phi \in \Phi(CX)$, subset of clocks $Y \subseteq CX$, and constant vector $D \in (\mathbb{Z}^{\geq 0})^{|Y|}$, the

*substituted clock constraint* $\phi[Y \mapsto D]$ is a clock constraint ($\phi[Y \mapsto D] \in \Phi(CX)$) where each clock $x_i$ in $Y$ is replaced with its relevant constant $D(x_i)$. Formally for $\bowtie \in \{<, \leq, >, \geq, =\}$:

$$\phi[Y \mapsto D] = \begin{cases} x_i \bowtie c & \phi = x_i \bowtie c \text{ and } x_i \notin Y \\ \texttt{tt} & \phi = x_i \bowtie c, x_i \in Y, \text{ and } D(x_i) \bowtie c \\ \texttt{ff} & \phi = x_i \bowtie c, x_i \in Y, \text{ and } D(x_i) \not\bowtie c \\ \phi_1[Y \mapsto D] \wedge \phi_2[Y \mapsto D] & \phi = \phi_1 \wedge \phi_2 \end{cases}$$

The resulting formula is then simplified using logical equivalences to yield a clock constraint. If the vector $D$ is a single constant $d$ for all clocks $Y$, we will write $\phi[Y \mapsto d]$.

While this substitution operator is a syntactic operator on a clock constraint, a clock constraint will often be interpreted as a set of valuations and this operator may be interpreted as changing a set of valuations into another set of valuations.

■

With this substituted clock constraint, given a clock constraint $\phi$ in $\Phi(CX)$ and a set of clocks $Y$ to reset to 0, the *reset predecessor operator*, `resetPred(`$\phi$`, `$Y$`)` is defined (and computed) as `resetPred(`$\phi$`, `$Y$`)` $= \phi[Y \mapsto 0]$. This operator converts the clock constraint $\phi$ to the clock constraint representing the precondition for $\phi$ with respect to the reset operations on the clocks in $Y$.

In order to handle substitution with clock difference constraints, we extend substitution on clock constraints to handle clock difference constraints.

**Definition 3.4.2 (Substituted clock difference constraint $\phi[Y \mapsto d]$).** Given clock constraint $\phi \in \Phi_{-}(CX)$, constant $d \in \mathbb{Z}^{\geq 0}$ and a subset of clocks $Y \subseteq CX$, the

*substituted clock difference constraint* $\phi[Y \mapsto d]$ is a clock constraint ($\phi[Y \mapsto d] \in \Phi_{-}(CX)$) where each clock $x_i$ in $Y$ is replaced with the constant $d$. Formally,

$$\phi[Y \mapsto d] = \begin{cases} x_i \bowtie c & \phi = x_i \bowtie c \text{ and } x_i \notin Y \\ d \bowtie c & \phi = x_i \bowtie c \text{ and } x_i \in Y \\ x_i - x_j \bowtie c & \phi = x_i - x_j \bowtie c \text{ and } x_i, x_j \notin Y \\ x_i - d \bowtie c & \phi = x_i - x_j \bowtie c \text{ and } x_i \notin Y, x_j \in Y \\ d - x_j \bowtie c & \phi = x_i - x_j \bowtie c \text{ and } x_i \in Y, x_j \notin Y \\ 0 \bowtie c & \phi = x_i - x_j \bowtie c \text{ and } x_i, x_j \in Y \\ \phi_1[Y \mapsto d] \wedge \phi_2[Y \mapsto d] & \phi = \phi_1 \wedge \phi_2 \end{cases}$$

Inequalities involving only constants are simplified to `tt` or `ff`. The constraint is then simplified using logical equivalences to yield a clock constraint. Within `resetPred(`$\phi$`, `$Y$`)`, we will commonly use $d = 0$.

While this substitution operator is a syntactic operator on a clock constraint, a clock constraint will often be interpreted as a set of valuations and this operator may be interpreted as changing a set of valuations into another set of valuations.

■

Hence, the extended reset predecessor operator is the same substitution as the operator when $\phi$ is not a clock difference constraint: `resetPred(`$\phi, Y$`)` $= \phi[Y \mapsto 0]$.

We prove the correctness of the extended version of `resetPred(`$\phi, Y$`)`. Since the non-extended version is a subset of these cases, its proof of correctness follows directly from this proof.

The proofs for both conversions depend on the correctness of `resetPred(`$\phi, Y$`)`, which is proven in Theorem 3.4.1. When reading this theorem, note that a clock

constraint can be interpreted as the set of valuations that satisfy the constraint.

**Theorem 3.4.1.** For any clock valuation $\nu$, any clock set $Y \subseteq CX$, and any clock constraint $\phi$, $\nu \models \texttt{resetPred}(\phi, Y)$ if and only if $\nu[Y := 0] \models \phi$.

***Proof of Theorem 3.4.1.*** Let $\nu$ be an arbitrary valuation and $Y$ be a subset of clocks (assuming the dummy clock $x_0 \notin Y$). We prove this by structural induction of $\phi$.

**Case 1:** $\phi = x \leq c$.

If $x \notin Y$, then the claim is true. If $x \in Y$, then $\texttt{resetPred}(\phi, Y) \equiv c \geq 0$. We have $\nu[Y := 0] \models \phi$ if and only if $c \geq 0$.

**Case 2:** $\phi = x < c$.

Similar to Case 1, after replacing $c \geq 0$ with $c > 0$.

**Case 3:** $\phi = x \geq c$.

Similar to Case 1. If $x \notin Y$, then the constraint is unchanged. If $x \in Y$, then the preset is equivalent to $c \leq 0$.

**Case 4:** $\phi = x > c$.

Similar to Case 3, after replacing $c \leq 0$ with $c < 0$.

**Case 5:** $\phi = x_i - x_j \leq c$.

If $x_i, x_j \notin Y$, then since the constraint is unchanged, the claim is true.

If $x_i, x_j \in Y$, then $\nu[Y := 0] \models \phi$ if and only if $c \geq 0$. Consequently, the value of $\texttt{resetPred}(\phi, Y) \equiv c \geq 0$: the same constraint.

Now assume $x_i \in Y$, $x_j \notin Y$. We know $\nu[Y := 0] \models \phi$ if and only if $-\nu(x_j) \leq c$. $\texttt{resetPred}(\phi, Y) \equiv -x_j \leq c$; hence, this case is true.

Now assume $x_i \notin Y$, $x_j \in Y$. We know $\nu[Y := 0] \models \phi$ if and only if $\nu(x_i) \leq c$. $\texttt{resetPred}(\phi, Y) \equiv x_i \leq c$; hence, this case is true.

**Case 6:** $\phi = x_i - x_j < c$.

Similar to Case 5, after replacing $\leq$ with $<$.

**Case 7:** $\phi = \phi_1 \wedge \phi_2$.

By the **Induction Hypothesis**, $\nu \models \texttt{resetPred}(\phi_1, Y)$ if and only if $\nu[Y := 0] \models \phi_1$, and $\nu \models \texttt{resetPred}(\phi_2, Y)$ if and only if $\nu[Y := 0] \models \phi_2$. By our algorithm, $\texttt{resetPred}(\phi_1 \wedge \phi_2, Y) = \texttt{resetPred}(\phi_1, Y) \wedge \texttt{resetPred}(\phi_2, Y)$. After applying the definition of $\models$ for clock valuations, this case is true. □

We use the previous Theorem to prove a useful Corollary.

**Corollary 3.4.1.** For all clock valuations $\nu$, clock sets $Y \subseteq CX$, clock constraints $\phi$ and location $l'$, $\nu \models \phi \cap \texttt{resetPred}(I(l'), Y)$ if and only if ($\nu \models \phi$ and $\nu[Y := 0] \models I(l')$). Likewise, $\nu \models \phi \cap \neg\texttt{resetPred}(I(l'), Y)$ if and only if ($\nu \models \phi$ and $\nu[Y := 0] \not\models I(l')$)

***Proof of Corollary 3.4.1.***

$$\nu \models \phi \cap \texttt{resetPred}(I(l'), Y) \Leftrightarrow \tag{3.3}$$

$$\nu \models \phi \text{ and } \nu \models \texttt{resetPred}(I(l'), Y) \Leftrightarrow \quad \text{by definition of } \models \tag{3.4}$$

$$\nu \models \phi \text{ and } \nu[Y := 0] \models \texttt{resetPred}(I(l'), Y) \quad \text{by Theorem 3.4.1} \tag{3.5}$$

The proof of the likewise statement is similar. □

**Conversion 1: Conversion to Baseline Version**

We take the variant that treats any state with an unsatisfied invariant as urgent and convert it to our baseline version, which forbids action executions into states having unsatisfied invariants. The original and converted timed automata have isomorphic reachable subsystems. This conversion, for convenience, assumes that the automaton we are converting has no urgent locations. Because giving a location the invariant `ff` is equivalent to making that location urgent, we lose no expressive power.

**Conversion Idea:** The idea underlying the construction is to make a copy of each location $l$ (named $l_u$), which represents the urgent version of $l$. Make the invariant of $l_u$ `tt`. Each edge now becomes four edges: two outgoing edges from location $l$, and two outgoing edges from location $l_u$. For location $l$, one edge goes to its destination $l'$ and a second goes to the urgent copy of $l'$, $l'_u$. To the edges going to $l'$, we add a constraint establishing $I(l')$ will be true when we enter $l'$. On the other hand, to the edges going to $l'$, we add a constraint establishing $I(l')$ will be false when we enter $l'_u$. By design, the constraint we add to the second copy is the negation of the first constraint. To avoid disjunctive constraints on guards, any edge with such a constraint is converted to a set of edges (as is done in Section 3.3.1).

**Formal Conversion:** Given $TA = (L, L_0, \Sigma, CX, I, E)$ we produce $TA' = (L', L'_0, L'_u, \Sigma, CX, I', E')$ as follows:

- $L' = L \cup \{l'_u \mid l \in L\}$.
- $L'_0$ is defined as follows. First, consider the set $L_{inv} = \{l_0 \mid l_0 \in L_0$ and $(l_0, \nu_0) \models I(l_0)\}$. The valuation $\nu_0$ is the initial valuation where all clocks are 0. Let $L_d$ be $|L_0| - |L_{inv}|$ copies of a dead state $l'_d$. Then $L'_0 = L_{inv} \cup L_d$.

- $L'_u = \{l'_u \mid l \in L\}$.
- $CX' = CX$.
- $I' \colon L' \longrightarrow \Phi(CX')$ is the function: $I'(l') = I(l')$ if $l' \in L$, $I(l'_d) = \texttt{ff}$, and $I(l') = \texttt{tt}$ otherwise ($l' \in \{l_u \mid l \in L\}$). For these urgent locations, we make the invariant true to allow all proper action executions.
- $E' =$ is the set of edges defined as follows. For each edge $(l, a, \phi, \lambda, l') \in E$, the set of edges in $E'$ includes:

$$(l, a, \psi_{l,l}, \lambda, l'), \text{ where } \psi_{l,l} = \phi \wedge \texttt{resetPred}(I(l'), \lambda)$$
$$(l, a, \psi_{l,u}, \lambda \cup \{x_u\}, l'_u), \text{ where } \psi_{l,u} = \phi \wedge \neg\texttt{resetPred}(I(l'), \lambda)$$
$$(l_u, a, \psi_{u,l}, \lambda, l'), \text{ where } \psi_{u,l} = \phi \wedge \texttt{resetPred}(I(l'), \lambda)$$
$$(l_u, a, \psi_{u,u}, \lambda, l'_u) \text{ where } \psi_{u,u} = \phi \wedge \neg\texttt{resetPred}(I(l'), \lambda).$$

  Disjunctive guard constraints may arise from negating $\texttt{resetPred}(I(l'), \lambda)$. Following the process used in Section 3.3.1, any disjunctive guard constraints is eliminated by converting the edge with such a constraint to a set of edges. Any edge with a guard constraint logically equivalent to $\texttt{ff}$ is removed from $E'$. Note: $E'$ has no outgoing edges from the dead locations $l'_d$.

If every initial state satisfies its location's invariant, then $L'_0 = L_0$. This definition of $L_0$ replaces initial states that do not satisfy their invariants with the same number of dead states. (We need to add an equal number of missing states to ensure that the function mapping converted states to original states is a bijection.) Additionally, the above conversion illustrates the (subtle) need for urgency to disallow 0-time unit advances; if urgency were to allow 0-time unit advances, then urgent states would have additional transitions: 0-unit time advances.

The converted automaton has twice as many locations and up to four times as many edges as the original automaton plus the edges that come from eliminating disjunctive guard constraints. While in theory, the negation of the reset predecessor may have up to $2|CX|$ disjunctions (consider negating the constraint $x_1 = 3 \wedge x_2 = 3 \wedge \ldots \wedge x_n = 3$ using $x_i = 3 \equiv x_i \leq 3 \wedge x_i \geq 3$), in practice the negation of a reset predecessor usually has no more than a few disjuncts; hence, in practice, the number of additional edges is small.

**Example 3.4.2** (**Continuation of Example 3.4.1**)**.** We convert the timed automaton in Figure 3.3 to our baseline formalism and add urgent copies of additional locations to represent states with unsatisfied invariants as urgent. We know that $\texttt{resetPred}(x_1 < 2 \wedge x_2 > 2, \{x_2\}) = \texttt{ff}$ and we also know that $\neg\texttt{resetPred}(x_1 < 2 \wedge x_2 > 2, \{x_2\}) = \texttt{tt}$; hence, we add the urgent location $1_u$ to mimic location 1. The converted automaton (only reachable locations shown) is in Figure 3.4. Notice that location 1 is unreachable from the initial state, and is therefore omitted, as are the urgent versions of 2 and 3. ■

Because we introduce states that are never reached when we convert the timed automaton, the original and converted automata are not isomorphic. Nevertheless, if we restrict the transition systems to only the states reachable from initial states, we can establish isomorphism. Thus, we show that the original and converted automata have a reachable subsystem isomorphism.

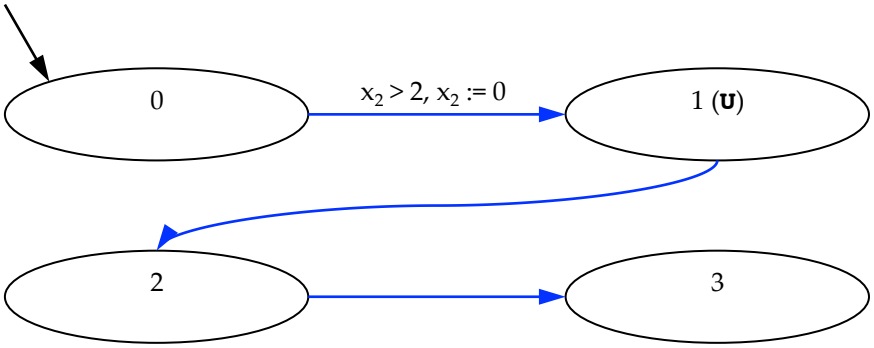


**Figure 3.4:** Timed automaton from Figure 3.3 converted into our baseline formalism with urgent location $1_u$. Only locations with states reachable from the initial state are shown. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

**Theorem 3.4.2.** Let $TA$ be a timed automaton having the semantics of allowing action executions into states with unsatisfied invariants and let $INV(TA)$ be the converted automaton having semantics disabling action executions into states with unsatisfied invariants. Their reachable subsystems are isomorphic ($TS(TA) \cong_r TS(INV(TA))$).

***Proof of Theorem 3.4.2.*** Each initial state of $TA$ that does not satisfy its location's invariant has its location replaced with its own dead state $l_d$. As a result, the states with unsatisfied invariants are mapped to the dead states. Having an equal number of dead states ensures that our function is a bijection. Also, these dead states can reach no other states because their invariants are `ff`. Hence, for the remainder of the proof, we assume that all initial states satisfy their locations' invariants.

From our conversion, both automata have the same event set $\Sigma$. Consider the

function $f$ over the reachable states:

$$f\colon Q_{TS(TA)} \longrightarrow Q_{TS(INV(TA))} \qquad \text{reachable states}$$

$$f\big((l,\nu)\big) = \begin{cases} (l,\nu), & \text{if } \nu \models I(l) \\ (l_u,\nu) & \text{otherwise.} \end{cases}$$

The function $f$ maps $Q_{0_{TS(TA)}}$ to $Q_{0_{TS(INV(TA))}}$, preserving initial states (here we use the premise that all initial states satisfy their invariants; hence, they are reachable). When the invariant is false, $f$ maps a location to its urgent copy in the converted automaton.

**Part 1:** $f$ is a bijection.

From the definition of $f$, $f$ is one-to-one.

Suppose we have a state $(l_{inv}, \nu_{inv})$ in $INV(TA)$ that is not mapped to by $f$. We claim $(l_{inv}, \nu_{inv})$ is not reachable from an initial state. By the definition of $f$, if $l_{inv}$ is not an urgent copy and $\nu_{inv} \models I(l_{inv})$, then $(l_{inv}, \nu_{inv})$ is covered by $f$. Likewise, if $l_{inv}$ is an urgent copy location $l_u$ and $(l, \nu_{inv}) \not\models I(l)$, then $(l_{inv}, \nu_{inv})$ is covered by $f$. If $l_{inv}$ is an urgent copy location $l_u$ and $(l, \nu_{inv}) \models I(l)$, then by the construction of $INV(TA)$, this state is not reachable. If $l_{inv}$ is not an urgent copy and $\nu_{inv} \not\models I(l_{inv})$, then $(l_{inv}, \nu_{inv})$ is not reachable by the definition of the semantics of $INV(TA)$. Note: in this case we use the assumption that all initial states satisfy their invariants. Thus, $f$ is onto.

Here is the definition of $f^{-1}$:

$$f^{-1}\colon Q_{TS(INV(TA))} \longrightarrow Q_{TS(TA)} \qquad \text{reachable states}$$

$$f^{-1}\big((l,\nu)\big) = (l,\nu) \qquad l \text{ not an urgent copy, and}$$

$$f^{-1}\big((l_u,\nu)\big) = (l,\nu) \qquad l_u \text{ the urgent copy of } l,$$

From the proof of $f$ being a bijection, we know either a state or its urgent copy (possibly both) is not reachable.

**Part 2:** $f$ preserves the transition relation.

We need to show:

$$(l,\nu) \xrightarrow{\delta} (l',\nu') \Leftrightarrow f((l,\nu)) \xrightarrow{\delta} f((l',\nu')) \; \forall \delta \in \mathbb{R}^{\geq 0}$$
$$(l,\nu) \xrightarrow{a} (l',\nu') \Leftrightarrow f((l,\nu)) \xrightarrow{a} f((l',\nu')) \; \forall a \in \Sigma$$

Showing $f$ does not eliminate transitions, we prove the $\Rightarrow$ direction. We omit the proof of the other ($\Leftarrow$) direction; its proof is similar and (one proof) uses $f^{-1}$ instead of $f$.

**Part 2a:** Time advances: $(l,\nu) \xrightarrow{\delta} (l',\nu') \Rightarrow f((l,\nu)) \xrightarrow{\delta} f((l',\nu'))$.

Suppose we have the transition $(l,\nu) \xrightarrow{\delta} (l,\nu+\delta)$. By definition of transition system semantics, $\nu \models I(l)$, $\nu+\delta \models I(l)$ and $\forall t, 0 \leq t \leq \delta$: $\nu + t \models I(l)$. Thus, by our definition of $f$ and our conversion, $f((l,\nu)) = (l,\nu)$ and $\nu \models I(l)$ and $\forall t, 0 \leq t \leq \delta$: $f((l,\nu+t)) = (l,\nu+t)$ and $\nu + t \models I(l)$. (To prove the $\Leftarrow$ direction, utilize that time cannot advance in an urgent state. This utilizes that urgent states prevent 0-unit time advances.)

**Part 2b:** Edge executions: $(l,\nu) \xrightarrow{a} (l',\nu') \Rightarrow f((l,\nu)) \xrightarrow{a} f((l',\nu'))$.

Now suppose we have the transition $(l,\nu) \xrightarrow{a} (l',\nu[\lambda := 0])$. By the definition of transition system semantics, $TA$ has the edge $e = (l,a,\phi,\lambda,l')$, and $\nu \models \phi$. We have two cases: either $\nu[\lambda := 0] \models I(l')$ or it does not.

**Case 2b1:** $\nu[\lambda := 0] \models I(l')$. Suppose $\nu[\lambda := 0] \models I(l')$. Now we have two cases: $\nu \models I(l)$ or $\nu \not\models I(l)$. Assume $\nu \models I(l)$. By our conversion, we have the edge $e = (l,a,\phi \cap \texttt{resetPred}(I(l'),\lambda),\lambda,l')$ in $INV(TA)$. By our conversion,

$f((l', \nu[\lambda := 0])) = (l', \nu[\lambda := 0])$. Since we assume $\nu[\lambda := 0] \models I(l')$ and $\nu \models \phi$, by Corollary 3.4.1, we know $\nu \models \phi \cap \texttt{resetPred}(I(l'), \lambda)$. Therefore, $INV(TA)$ has the transition $f((l, \nu)) \xrightarrow{a} f((l', \nu[\lambda := 0]))$.

Now assume $\nu \not\models I(l)$. Thus, $f(l, \nu) = (l_u, \nu[x_u := 0])$. By our conversion, we use the edge $e_u = (l_u, a, \phi \cap \texttt{resetPred}(I(l'), \lambda), \lambda, l')$ in $INV(TA)$. Otherwise, the proof is the same as the previous case's.

**Case 2b2:** $\nu[\lambda := 0] \not\models I(l)$. Now suppose $\nu[\lambda := 0] \not\models I(l')$. We have two cases: $\nu \models I(l)$ or $\nu \not\models I(l)$. Assume $\nu \models I(l)$. By our conversion, $f((l', \nu[\lambda := 0])) = (l'_u, \nu[\lambda := 0])$. Since $l'_u$ is the urgent copy of $l'$, we know $\nu[\lambda := 0] \models I(l'_u)$. Since $\nu \models \phi$, by Corollary 3.4.1, we know that $\nu \models \phi \cap \neg\texttt{resetPred}(I(l'), \lambda)$. By the definition of transition system semantics, $INV(TA)$ has the transition $f((l, \nu)) \xrightarrow{a} f(l', \nu[\lambda := 0])$.

Now assume $\nu \not\models I(l)$. Thus, $f(l, \nu) = (l_u, \nu)$. By our conversion, we use the edge $e_u = (l_u, a, \phi \cap \neg\texttt{resetPred}(I(l'), \lambda), \lambda, l'_u)$ in $INV(TA)$. Otherwise, the proof is the same as the previous case's. □

**Conversion 2: Conversion from Baseline Version**

In this instance, the baseline version has different semantics than the other version and cannot be considered a special case of that variant. Hence, we provide the conversion from the baseline formalism.

**Conversion Idea:** This conversion takes an automaton that disallows actions into states with unsatisfied invariants and converts the automaton to one whose semantics allow action executions into states with unsatisfied invariants. The conversion works by adding the reset predecessor of the destination state's invariant to each edge. In addition, this conversion also eliminates urgent locations from the baseline version and replaces those locations with locations with the invariant `ff`.

**Formal Conversion:** Given $TA = (L, L_0, L_u, \Sigma, CX, I, E)$ we produce $TA' = (L, L'_0, \Sigma, CX, I, E')$ as follows:

- $L'_0$ is defined as follows. First, consider the set $L_{inv} = \{l_0 \mid l_0 \in L_0 \text{ and } (l_0, \nu_0) \models I(l_0)\}$. The valuation $\nu_0$ is the initial valuation where all clocks are 0. Let $L_d$ be $|L_0| - |L_{inv}|$ copies of a dead state $l'_d$. Then $L'_0 = L_{inv} \cup L_d$.

- $I' \colon L \longrightarrow \Phi(CX)$ is the function: $I(l') = I(l')$ if $l' \notin L_u$, $I(l'_d) = \texttt{ff}$ and $I(l') = \texttt{ff}$ if $l' \in L_u$. We give the invariant `ff` to previously-urgent locations to prevent all time advances (even those of 0 units).

- $E' = \{(l, a, \phi \wedge \texttt{resetPred}(I(l'), \lambda), \lambda, l') \mid (l, a, \phi, \lambda, l') \in E\}$. Note: $E'$ has no outgoing edges from the dead locations $l'_d$.

Again, if every initial state satisfies its location's invariant, then $L'_0 = L_0$. This definition of $L_0$ replaces initial states that do not satisfy invariants with the same number of dead states. (We need to add an equal number of missing states to ensure that the function mapping converted states to original states is a bijection.)

Notice that the converted automaton has no urgent locations; the invariant `ff` handles the urgency. Given that an invariant is a clock constraint in $\Phi(CX)$, the reset predecessor will also be a clock constraint in $\Phi(CX)$; as a result, there are no disjunctive guard constraints. Thus, the converted automaton has the same number of locations and the same number of edges as the original automaton.

The correctness follows from this theorem.

**Theorem 3.4.3.** Let $TA$ be a timed automaton having the semantics of disabling action executions into states with unsatisfied invariants and let $URG(TA)$ be the converted automaton having the semantics of allowing action executions into states with unsatisfied invariants. Their reachable subsystems are isomorphic ($TS(TA) \cong_r TS(URG(TA))$).

***Proof of Theorem 3.4.3.*** For each initial state of $TA$ that does not satisfy its location's invariant, it is mapped to a dead state $l'_d$ in $URG(TA)$. Having an equal number of states allows the function to be a bijection, and by construction, neither of these states can advance time or execute an action. Hence, for the rest of the proof, we assume that all the initial states of $TA$ satisfy their locations' invariants.

From our conversions, both automata have the same event set $\Sigma$. Consider the function $f$ over the reachable states:

$$f : Q_{TS(TA)} \longrightarrow Q_{TS(URG(TA))} \qquad \text{reachable states}$$
$$f\big((l,v)\big) = (l,v)$$

or the identity function. The function $f$ maps $Q_{0_{TS(TA)}}$ to $Q_{0_{TS(URG(TA))}}$, preserving initial states (here we use the premise that all initial states satisfy their invariants; hence, they are reachable). Since we are only concerned with reachable states, we are not worried about covering every state in $Q_{TS(URG(TA))}$; states with unsatisfied invariants are not reachable.

**Part 1:** $f$ is a bijection. By definition of $f$, $f$ is one-to-one. Since all initial states satisfy their invariants, all reachable states in $Q$ and $Q_{urg}$ satisfy their invariants. Because we restrict ourselves to reachable states, $f$ is onto. Therefore, $f$ is a bijec-

tion.

Here is the definition of $f^{-1}$:

$$f^{-1}\colon Q_{TS(URG(TA))} \longrightarrow Q_{TS(TA)} \qquad \text{reachable states}$$
$$f^{-1}\big((l,\nu)\big) = (l,\nu)$$

Again, every reachable state in $Q_{TS(URG(TA))}$ satisfies its invariant.

**Part 2:** $f$ preserves the transition relation.

We need to show:

$$(l,\nu) \xrightarrow{\delta} (l',\nu') \Leftrightarrow f((l,\nu)) \xrightarrow{\delta} f((l',\nu'))\ \forall \delta \in \mathbb{R}^{\geq 0} \text{ and}$$
$$(l,\nu) \xrightarrow{a} (l',\nu') \Leftrightarrow f((l,\nu)) \xrightarrow{a} f((l',\nu'))\ \forall a \in \Sigma$$

Showing $f$ does not eliminate transitions, we prove the $\Rightarrow$ direction. We omit the proof of the other ($\Leftarrow$) direction; its proof is similar and (one proof) uses $f^{-1}$ instead of $f$.

**Part 2a:** Time advances: $(l,\nu) \xrightarrow{\delta} (l',\nu') \Rightarrow f((l,\nu)) \xrightarrow{\delta} f((l',\nu'))$.

First assume $(l,\nu) \xrightarrow{\delta} (l,\nu+\delta)$. By definition of the transition system semantics, $\forall t, 0 \leq t \leq \delta\colon \nu + t \models I(l)$. By our conversion, $\forall t, 0 \leq t \leq \delta\colon \nu + t \models I(l)$ is true in $URG(TA)$. Hence, we have the transition $f((l,\nu)) \xrightarrow{\delta} f((l,\nu+\delta))$.

**Part 2b:** Edge executions: $(l,\nu) \xrightarrow{a} (l',\nu') \Rightarrow f((l,\nu)) \xrightarrow{a} f((l',\nu'))$.

Now suppose $(l,\nu) \xrightarrow{a} (l,\nu[\lambda := 0])$. By definition of the transition system, we have an edge $e = (l,a,\phi,\lambda,l')$, $\nu \models \phi$, and $\nu[\lambda := 0] \models I(l')$. By Corollary 3.4.1, $\nu \models \phi \cap \texttt{resetPred}(I(l'),\lambda)$ and by the definition of $f$, $f((l,\nu)) = (l,\nu)$. Thus, we have the transition $f((l,\nu)) \xrightarrow{a} f((l,\nu[\lambda := 0]))$ in $URG(TA)$. □

**Subtleties with Urgency**

Urgency is used for cleaner modeling and is supported in tools. Some sources using urgency include Behrmann et al. [23], Dong et al. [65], Olderog and Dierks [131]. The version used in Behrmann et al. [23] allows 0-unit time advances while our version does not. On the one hand, these versions are operationally equivalent. On the other hand, this difference does affect the satisfaction of certain logical formulas.

If we wish to have the kind of urgency used in Behrmann et al. [23], we can use the following conversion:

1. Add an extra clock $x_u$. This clock can be used for all urgent locations.
2. Give all urgent locations the invariant $x_u = 0$.
3. Reset $x_u$ on all incoming edges to each urgent location $l_u$.
4. Make each urgent location non-urgent.

While seemingly insignificant, disallowing 0-unit time advances in urgent locations is necessary to give urgent locations enough power to represent timed automata that allow action executions into locations with unsatisfied invariants (see Section 3.4.1). In addition, these 0-unit time advances influence formulas written in the timed modal-mu calculus of Laroussinie et al. [108], Sokolsky and Smolka [147]. For example, consider the formula, "there exists a time advance to a state where $x_1 = 0$." When $x_1$ is initially 0, the formula is true if and only if we allow time advances of 0 time units. When invariants are satisfied, all sources the authors consulted, including Alur [5], Baier and Katoen [17], Clarke et al. [55], Wang et al. [161], allow 0-unit time advances in non-urgent locations.

### 3.4.2 Clock Difference Inequalities in Clock Constraints

Some sources, including Bengtsson and Yi [27], Yovine [164], allow inequalities on clock differences in clock constraints. Timed automata without clock difference inequalities in clock constraints are often called *diagonal-free* automata [27, 43].

First, we extend clock constraints to support clock difference inequalities. These constraints are called clock difference constraints.

**Definition 3.4.3 (Clock difference constraint $\phi \in \Phi_{-}(CX)$).** Given a nonempty finite set of clocks $CX = \{x_1, x_2, \ldots, x_n\}$ and $c \in \mathbb{Z}^{\geq 0}$ (a non-negative integer), a *clock difference constraint* $\phi$ may be constructed using the following grammar:

$$\phi ::= x_i < c \mid x_i \leq c \mid x_i > c \mid x_i \geq c \mid x_i - x_j < c \mid x_i - x_j \leq c \mid \phi \wedge \phi$$

$\Phi_{-}(CX)$ is the set of all possible clock difference constraints over $CX$. We use the following abbreviations: $x_i - x_j > c$ for $x_j - x_i < -c$, $x_i - x_j \geq c$ for $x_j - x_i \leq -c$, and $x_i - x_j = c$ for $x_i - x_j \leq c \wedge x_i - x_j \geq c$. ■

With these clock constraints, we can extend timed automata to support clock difference inequalities in both the invariants and the guards.

**Definition 3.4.4 (Timed automaton with clock difference constraints).** A *timed automaton with clock difference constraints* $TA = (L, L_0, L_u, \Sigma, CX, I, E)$ is defined as a timed automata (Definition 2.2.2) with the following differences:

- $I \colon L \longrightarrow \Phi_{-}(CX)$ gives a clock difference constraint for each location $l$. $I(l)$ is referred to as the *invariant* of $l$.
- $E \subseteq L \times \Sigma \times \Phi_{-}(CX) \times 2^{CX} \times L$ is the set of *edges*. Edges are the same except that guards can be clock difference constraints.

■

To define satisfaction of a clock difference constraint, we extend the definition of $\nu \models \phi$ (Definition 2.2.4) to say that $\nu \models x_i - x_j \bowtie c$ ($\bowtie \in \{<, \leq, >, \geq\}$) if and only if $\nu(x_i) - \nu(x_j) \bowtie c$. The semantics of these timed automata is then defined in a similar manner to the semantics of the baseline version (Definition 2.2.6).

In our conversion, we will often replace an inequality in a clock constraint with `tt` or `ff`. If the inequality is $x_i \bowtie c$ or $x_i - x_j \bowtie c$, we use $\phi[x_i \bowtie c \mapsto \texttt{tt}]$ or $\phi[x_i - x_j \bowtie c \mapsto \texttt{ff}]$ to replace an inequality in $\phi$ with `tt` or `ff`. If the constraint is not in $\phi$, then this operator leaves $\phi$ unchanged. This definition is similar to the definition of clock constraint substitution (Definition 3.4.1).

**Conversion**

We can convert any timed automaton with clock difference inequalities in its constraints to an equivalent timed automaton without clock difference inequalities. This conversion is taken from Bérard et al. [28].

**Conversion Idea:** Clock difference inequalities are invariant under time-passage transitions. Since these transitions cannot induce a location change in a timed automaton, it suffices to enrich the location set to have each location encode which clock differences are true. Transitions that are incident on these locations would then combine this information with any clock resets in order to determine the new target location.

**Formal Conversion.** Let $TA = (L, L_0, L_u, \Sigma_{TA}, CX, I, E)$ be a timed automaton with clock difference constraints. We convert out one inequality at a time, repeating the same procedure on the resulting automaton with the next inequality. Since all inequality types are similar, throughout the conversion denote the inequality in $TA$ that we convert out as $x - y \bowtie c$ ($\bowtie \in \{<, \leq, >, \geq\}$).

Given an inequality $x - y \bowtie c$, we produce two location components: $c_t$, representing $x - y \bowtie c$ is true; and $c_f$, representing $x - y \bowtie c$ is false. We use the notation $c_b$ ($b \in \{t, f\}$) as a variable whose value is $c_t$ or $c_f$. With the location components $c_t$ and $c_f$, we replace each location $l$ with the two locations $(l, c_t)$ and $(l, c_f)$. Throughout executions, $c_t$ and $c_f$ are used to correctly encode the truth of $x - y \bowtie c$.

To write this in shorthand, we use the *reset predecessor operator* $\texttt{resetPred}(\phi, Y)$ from Section 3.4.1 extended to handle clock differences. The operation is still substitution with $\texttt{resetPred}(\phi, Y) = \phi[Y \mapsto 0]$, and its extended definition was given in Definition 3.4.2.

Thus, we convert the timed automaton $TA = (L, L_0, L_u, \Sigma, CX, I, E)$ to the timed automaton $DF(TA) = (L_{df}, L_{0_{df}}, L_{u_{df}}, \Sigma, CX, I_{df}, E_{df})$ given as follows:

- $L_{df} = \{(l, c_t) \mid l \in L\} \cup \{(l, c_f) \mid l \in L\}$.
- $L_{0_{df}} = \{(l, c_t) \mid l \in L_0\}$ if $[CX := 0] \models x - y \bowtie c$ and $\{(l, c_f) \mid l \in L_0\}$ otherwise ($[CX := 0] \not\models x - y \bowtie c$).
- $L_{u_{df}} = \{(l, c_t) \mid l \in L_u\} \cup \{(l, c_f) \mid l \in L_u\}$.
- $I_{df} \colon L_{df} \longrightarrow \Phi(CX)$ where $I((l, c_t)) = I((l, c_f)) = I(l)$ if $x - y \bowtie c$ is not contained in $I(l)$. Otherwise, $I((l, c_t)) = I(l)[(x - y \bowtie c) \mapsto \texttt{tt}]$ and $I((l, c_f)) = I(l)[(x - y \bowtie c) \mapsto \texttt{ff}]$.
- $E_{df}$ is constructed as follows. For each edge $e = (l, a, \phi, \lambda, l')$, we construct the following edges based on $\phi$, $\lambda$, and the constraint $x - y \bowtie c$. The set $E_{df}$

includes the following edges:

$$\begin{aligned}
&\big((l,c_t),a,\psi_{t,t},\lambda,(l,c_t)\big) \text{ where } \psi_{t,t} = \phi[x-y\bowtie c \mapsto \texttt{tt}] \wedge \\
&\qquad\qquad \texttt{resetPred}(x-y\bowtie c,\lambda) \\
&\big((l,c_t),a,\psi_{t,f},\lambda,(l,c_f)\big) \text{ where } \psi_{t,f} = \phi[x-y\bowtie c \mapsto \texttt{tt}] \wedge \\
&\qquad\qquad \texttt{resetPred}(\neg(x-y\bowtie c),\lambda) \\
&\big((l,c_f),a,\psi_{f,t},\lambda,(l,c_t)\big) \text{ where } \psi_{f,t} = \phi[x-y\bowtie c \mapsto \texttt{ff}] \wedge \\
&\qquad\qquad \texttt{resetPred}(x-y\bowtie c,\lambda) \\
&\big((l,c_f),a,\psi_{f,f},\lambda,(l,c_f)\big) \text{ where } \psi_{f,f} = \phi[x-y\bowtie c \mapsto \texttt{ff}] \wedge \\
&\qquad\qquad \texttt{resetPred}(\neg(x-y\bowtie c),\lambda)
\end{aligned}$$

All edges that result in the new guard constraint being `ff` are removed.

To explain the construction, consider what happens during an execution. During a time advance, the value of $x - y \bowtie c$ remains unchanged during time advances; hence, we always correctly stay in the same location component $c_b$ during a time advance. Invariants need to be changed if and only if $x - y \bowtie c$ is in the invariant. In this case, we replace that constraint with its value based on the location component $c_t$ or $c_f$. (Invariants are not addressed in Bérard et al. [28].)

Now consider action executions. To enforce that $c_t$ and $c_f$ correctly encode the value of $x - y \bowtie c$ during action executions, we split edges containing $x - y \bowtie c$ in their guards and we choose the proper destination component: $c_t$ or $c_f$. In more detail, for every edge $e = (l, a, \phi, Y, l')$, if $x - y \bowtie c$ appears in $\phi$ we only allow transitions from $(l, c_t)$ and replace $\phi$ with $\phi[x - y \bowtie c \mapsto \texttt{tt}]$. For all edges, if the edge resets $x$ or $y$ or both, then we do the following:

- If $x, y \in Y$, then $(l, c_b, \nu) \xrightarrow{a} (l', c_i, \nu')$. Here $c_i$ represents the correct value for $0 \bowtie c$. After this transition, $x$ and $y$ are both reset to 0; hence, $x - y = 0$.

- If $y \in Y$, $x \notin Y$, then the transition we take depends on the current valuation. For these transitions, the new value of $x - y \bowtie c$ becomes $x \bowtie c$. Thus, we make two edges for each edge $e$. The first edge conjuncts $x \bowtie c$ to the guard and transitions to the component $c_t$, and the second edge conjuncts $\neg(x \bowtie c)$ (which is a single inequality) to the guard and transitions to the component $c_f$.

- If $x \in Y$, $y \notin Y$, then the transition we take depends on the current valuation. For these transitions, the new value of $x - y \bowtie c$ becomes $-y \bowtie c$. Thus, we make two edges for each edge $e$. The first edge conjuncts $-y \bowtie c$ to the guard and transitions to the component $c_t$, and the second edge conjuncts $\neg(-y \bowtie c)$ (which is a single inequality) to the guard and transitions to the component $c_f$.

Since we double the number of locations per inequality we remove, the conversion is exponential in the number of clock difference inequalities in the automaton. There can be as many inequalities as the size of the timed automaton, which results in this conversion producing a new automaton potentially exponentially larger than the original.

For notation, when we consider the final converted timed automaton (after converting out multiple constraints), we let $l_{diff}$ be the location component of $l$ that represents the truths of each clock-difference inequality. Thus, for every location $l_d \in L_{df}, l_d = (l, l_{diff})$, $l \in L$, $l_{diff} = (c_{1b}, c_{2b}, \ldots c_{ib})$, and $c_{ib}$ says that the $i^{th}$ inequality $x_j - x_k \bowtie c$ has current boolean truth $b$.

In this work, we assume that all of the clock-difference constraints are in *canonical form*, meaning that all implied constraints are explicitly represented. For example, if the constraint has $x_1 - x_2 < 0$ and $x_2 - x_3 < 0$, canonical form forces the constraint $x_1 - x_3 < 0$ to be explicitly specified. Using all-pairs shortest paths, one

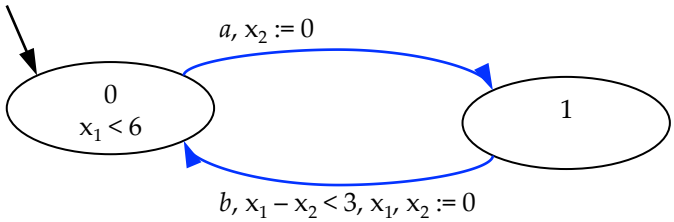


**Figure 3.5:** Timed automaton $TA_d$ with clock difference constraint $x_1 - x_2 < 3$. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

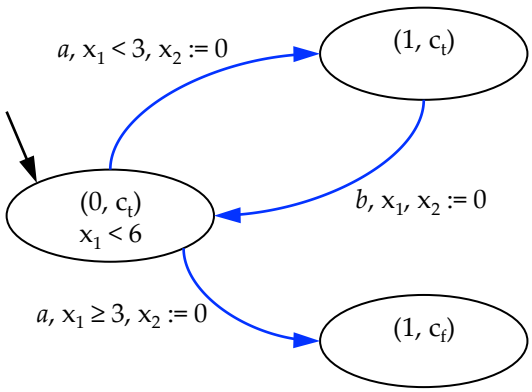


**Figure 3.6:** Diagonal-free timed automaton $DF(TA_d)$ equivalent to $TA_d$. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

can convert any constraint into canonical form in $O(|CX|^3)$ time [27].

**Example 3.4.3.** Consider the timed automaton $TA_d$ in Figure 3.5, where initially $x_1 = x_2 = 0$. As a result, initially $x_1 - x_2 < 3$. Notice that the edge from location 1 to location 0 has the clock difference constraint $x_1 - x_2 < 3$. We wish to convert the automaton to a diagonal-free automaton.

The resulting equivalent diagonal-free automaton is in Figure 3.6. The location $(0, c_f)$ is not reachable in the new automaton from the initial state and is omitted from the figure. Notice that $c_t$ encodes $x_1 - x_2 < 3$, so there is now no edge from $(1, c_f)$ to location$(0, c_t)$ since the guard $x_1 - x_2 < 3$ would be violated. ■

While the conversion is provided in Bérard et al. [28], no proof is given. Since the converted automaton has locations with components $c_t$ and $c_f$, this conversion is not isomorphic to the original automaton. As a result, we prove the equivalence of reachable subsystem isomorphism.

**Theorem 3.4.4.** Let $TA$ be a timed automaton and $DF(TA)$ be the diagonal-free automaton obtained from converting $TA$ to eliminate all of the clock difference constraints. Their reachable subsystems are isomorphic ($TS(TA) \cong_r TS(DF(TA))$).

***Proof of Theorem 3.4.4.*** From the conversion, both automata have the same event set $\Sigma$. Consider the function $f$ over the reachable states:

$$f : Q_{TS(TA)} \longrightarrow Q_{TS(DF(TA))} \qquad \text{reachable states}$$
$$f\big((l,\nu)\big) = ((l, l_{diff}), \nu)$$

The function $f$ maps $Q_{0_{TS(TA)}}$ to $Q_{0_{TS(DF(TA))}}$, preserving initial states (here we use the premise that all initial states satisfy their invariants; hence, they are reachable). The location component $l_{diff}$ represents the location components indicating the truths of clock differences. By design, for all reachable states, the correct $l_{diff}$ is paired with each valuation $\nu$.

**Part 1:** $f$ is a bijection.

Given our conversion, for all reachable states, $l_{diff}$ can be constructed from $(l,\nu)$. Consequently, we are guaranteed to cover each state in the reachable subsystem of $TS(DF(TA))$. Thus $f$ is a bijection.

Here is the definition of $f^{-1}$:

$$f^{-1}\colon Q_{TS(DF(TA))} \longrightarrow Q_{TS(TA)} \qquad \text{reachable states}$$
$$f^{-1}\big(((l, l_{diff}), \nu)\big) = (l, \nu)$$

Again, assuming $(l, \nu)$ is reachable, $l_{diff}$ can be constructed from $(l, \nu)$.

**Part 2:** $f$ preserves the transition relation.

We need to show:

$$(l, \nu) \xrightarrow{\delta} (l', \nu') \Leftrightarrow f((l, \nu)) \xrightarrow{\delta} f((l', \nu')) \; \forall \delta \in \mathbb{R}^{\geq 0} \text{ and}$$
$$(l, \nu) \xrightarrow{a} (l', \nu') \Leftrightarrow f((l, \nu)) \xrightarrow{a} f((l', \nu')) \; \forall a \in \Sigma$$

Showing $f$ does not eliminate transitions, we prove the $\Rightarrow$ direction. We omit the proof of the other ($\Leftarrow$) direction; its proof is similar and (one proof) uses $f^{-1}$ instead of $f$.

**Part 2a:** Time advances: $(l, \nu) \xrightarrow{\delta} (l', \nu') \Rightarrow f((l, \nu)) \xrightarrow{\delta} f((l', \nu'))$.

Suppose $(l, \nu) \xrightarrow{\delta} (l, \nu + \delta)$. By the definition of timed automata semantics, $\forall t, 0 \leq t \leq \delta\colon \nu + t \models I(l)$. By our conversion, $\forall t, 0 \leq t \leq \delta\colon f((l, \nu + t)) \models I(l) = I(f((l, \nu)))$. To elaborate, by our conversion we substitute in the clock difference constraints. If $\nu + t \models I(l)$, then $f((l, \nu + t)) \models I(l \times l_{diff})$ for all $0 \leq t \leq \delta$ because $\nu + t$ has the same values for the clock differences for all $0 \leq t \leq \delta$. Thus, we have the transition $f((l, \nu)) \xrightarrow{\delta} f((l, \nu + \delta))$.

**Part 2b:** Edge executions: $(l, \nu) \xrightarrow{a} (l', \nu') \Rightarrow f((l, \nu)) \xrightarrow{a} f((l', \nu'))$.

Let $(l, \nu) \xrightarrow{a} (l', \nu[\lambda := 0])$. By the definition of timed automata semantics, we have an edge $(l, a, \phi, \lambda, l')$ with $\nu \models \phi$ and $\nu[\lambda := 0] \models I(l')$. By our conversion,

$f((l,\nu)) = (l \times l_{diff}, \nu)$ and $f((l',\nu[\lambda := 0])) = (l' \times l'_{diff}, \nu[\lambda := 0])$. Since $l_{diff}$ stores the values of all clock difference inequalities, we know $\nu[\lambda = 0] \models \phi$. From our conversion and based on the clocks reset, we transition to the proper $l'_{diff}$ (reasoning is described in the conversion process). Hence, $l'_{diff}$ properly reflects the truths of the clock difference inequalities of the assignments of $\nu[\lambda := 0]$. Given we have the proper destination $f((l',\nu[\lambda := 0]))$, $f((l',\nu[\lambda := 0])) \models I(l' \times l'_{diff})$. Therefore, we have $f((l,\nu)) \xrightarrow{a} f((l',\nu[\lambda := 0]))$. □

**Algorithmic Ramifications**

Although clock difference constraints do not add expressive power to the theory of timed automata, they influence the computational techniques used for model checking. One particularly affected algorithm is the widening (normalization) algorithm. In model checking, widening is a common operation used to guarantee termination. Surprisingly, this particular widening algorithm no longer works if there are more than three clocks and clock difference inequalities in clock constraints. The full proof is given in Bouyer [35, 36], and more complex algorithms are given in Bengtsson and Yi [27], Bouyer et al. [43].

## 3.5 Timed Automata Equivalences: Other Equivalences

### 3.5.1 Rational Clock Constraints

Rather than restricting clock constants to be non-negative integers, Alur [5] allows non-negative rational ($\mathbb{Q}^{\geq 0}$) constants. Using non-negative integer constraints makes model checking easier; however, using rational numbers makes modeling easier. Equality of expressiveness is remarked on in Alur [5], and we formalize that remark. During this formalization we prove that the two models are equivalent with a non-label-preserving isomorphism.

**Conversion to Base Formalism**

Following the approach of Alur [5], we convert each timed automaton with rational constants to one with integer constants.

To perform this conversion, we find the least common positive integer, denoted $ld$, such that for all constants $c$ in constraints in the timed automaton, $ld * c$ is an integer. (This integer $ld$ will be the least common multiple of the denominators of the reduced fractions of the rationals in the constraints.) We form the new automaton by multiplying each constraint by $ld$ in the invariants and the guards. This process converts each rational constraint to an integer constraint. If $TA$ is our original timed automaton with non-negative rational constraints, then $INT(TA)$ is the converted timed automaton with only non-negative integer constraints.

In the conversion of rational clock constraints, we extend the definition of clock constraint substitution (Definition 3.4.1). Our extension extends the vector $D$ to a function that can substitute each clock with a clock times a rational constant. The typical extended substitution used is $\phi[CX \mapsto CX/ld]$.

**Formal Conversion:** Given timed automaton $TA = (L, L_0, L_u, \Sigma, CX, I, E)$ and positive integer $ld$, form timed automaton $INT(TA) = (L, L_0, L_u, \Sigma, CX, I_{INT}, E_{INT})$ as:

- $I_{INT} \colon L \longrightarrow \Phi(CX)$, such that $I_{INT}(l) = I(l)[CX \mapsto CX/ld]$.
- $E_{INT} = \{(l, a, \phi[CX \mapsto CX/ld], \lambda, l') \mid (l, a, \phi, \lambda, l') \in E\}$.

The locations, initial locations, urgent locations, set of action symbols, and set of clocks are the same. In the invariant, each constraint $x \bowtie c$ ($\bowtie \in \{<, \leq, >, \geq\}, c \in \mathbb{Q}^{\geq 0}$) is replaced with $x \bowtie (ld * c)$. For each edge in $E$, each constraint $x \bowtie c$ in the guard is replaced with with $x \bowtie (ld * c)$.

*Remark* 3.5.1. This conversion transforms a timed automaton with rational clock

constraints to a timed automaton with integer clock constraints by re-scaling time advances. When model checking satisfaction of a formula on such an automaton, one must also re-scale the constraints in the formula. Furthermore, because most model-checking algorithms work better with integer constraints, the choice for $ld$ should consider the clock constraints in the formula and be chosen such that the constants in both the converted automaton and the converted formula are non-negative integers.

Because we need to map labels with a function other than the identity function, we cannot establish a label-preserving isomorphism. However, we prove a non-label-preserving isomorphism between the two automata.

**Theorem 3.5.1.** Let $TA$ be a timed automaton with rational constraints and $ld$ be some positive integer such that for all constants $c$ in $TA$, $c * ld \in \mathbb{Z}$. Also let $INT(TA)$ be the converted automaton where each constant in a constraint is multiplied by the $ld$ (and thus all constants have non-negative integer values). There is a non-label-preserving isomorphism between $TS(TA)$ and $TS(INT(TA))$ ($TS(TA) \cong_{nl} TS(INT(TA))$).

***Proof of Theorem 3.5.1.*** From our conversion, both automata have the same event

$\Sigma$. Consider $f$: the pair of functions $f_q$ and $f_\sigma$:

$$
\begin{aligned}
&f\colon Q_{TS(TA)} \times \Sigma_{TS(TA)} \longrightarrow Q_{TS(INT(TA))} \times \Sigma_{TS(INT(TA))} \\
&f\big((l,\nu),\sigma\big) = f_q\big((l,\nu)\big) \times f_\sigma\big(\sigma\big) \\
&f_q\big((l,\nu)\big) = (l, \nu[CX := CX * ld]) \\
&f_\sigma\big(a\big) = a && a \notin \mathbb{R}^{\geq 0} \\
&f_\sigma\big(\delta\big) = \delta * ld && \delta \in \mathbb{R}^{\geq 0}
\end{aligned}
$$

The function $f$ preserves the location (though $I(f(l)) = I(f_q(l)) = I(l)[CX \mapsto CX/ld]$)), multiplies each clock's value in $\nu$ by $ld$, preserves the action labels, and maps each time advance label $\delta$ to $ld * \delta$. The function $f$ maps $Q_{0_{TS(TA)}}$ to $Q_{0_{TS(INT(TA))}}$ ($0 * ld = 0$), preserving initial states.

**Part 1:** $f$ is a bijection.

Because the real numbers are dense, $f_q, f_\sigma$, and $f$ are bijections. Here is the definition of $f^{-1}$, which is a pair of functions $f_q^{-1}$ and $f_\sigma^{-1}$:

$$
\begin{aligned}
&f^{-1}\colon Q_{TS(INT(TA))} \times \Sigma_{TS(INT(TA))} \longrightarrow Q_{TA} \times \Sigma_{TA} \\
&f^{-1}\big((l,\nu),\sigma\big) = f_q^{-1}\big((l,\nu)\big) \times f_\sigma^{-1}\big(\sigma\big) \\
&f_q^{-1}\big((l,\nu)\big) = (l, \nu[CX := CX/ld]) \\
&f_\sigma^{-1}\big(a\big) = a && a \notin \mathbb{R}^{\geq 0} \\
&f_\sigma^{-1}\big(\delta\big) = \delta/ld && \delta \in \mathbb{R}^{\geq 0}
\end{aligned}
$$

The function $f^{-1}$ preserves the location, divides each clock's value in $\nu$ by $ld$ (we know $ld \neq 0$), preserves the action labels, and maps each time advance label $\delta$ to $\delta/ld$.

**Part 2:** $f$ preserves the transition relation.

We need to show:

$$(l,\nu) \xrightarrow{\delta} (l',\nu') \Leftrightarrow f((l,\nu)) \overset{f(\delta)=\delta*ld}{\longrightarrow} f((l',\nu')) \; \forall \delta \in \mathbb{R}^{\geq 0} \text{ and}$$
$$(l,\nu) \xrightarrow{a} (l',\nu') \Leftrightarrow f((l,\nu)) \overset{f(a)=a}{\longrightarrow} f((l',\nu')) \; \forall a \in \Sigma$$

Showing $f$ does not eliminate transitions, we prove the $\Rightarrow$ direction. We omit the proof of the other ($\Leftarrow$) direction; its proof is similar and (one proof) uses $f^{-1}$ instead of $f$.

The function $f$ defines an implicit edge morphism $f_{\rightarrow}$ from the edges of $TS(TA)$ to the edges of $TS(INT(TA))$, where the edge $(q,a,q')$ is mapped as follows: $f_{\rightarrow}((q,a,q)') = (f_q(q), f_\sigma(a), f_q(q'))$.

**Part 2a:** Time advances: $(l,\nu) \xrightarrow{\delta} (l',\nu') \Rightarrow f((l,\nu)) \overset{f(\delta)=\delta*ld}{\longrightarrow} f((l',\nu'))$.

If we have $(l,\nu) \xrightarrow{\delta} (l',\nu')$ and a time advance of $\delta * ld$ from $f((l,\nu))$, then we know that $f((l,\nu)) \overset{f(\delta)=\delta*ld}{\longrightarrow} f((l',\nu'))$. Now we must show that the transition exists. By the definition of the transition system, given the transition $(l,\nu) \xrightarrow{\delta} (l',\nu')$, we know that $\forall t \in \mathbb{R}, 0 \leq t \leq \delta \colon \nu + t \models I(l)$. Because $I(f(l)) = I(l)[CX \mapsto CX/ld])$, when we map each state, we get the expression $\forall t \in \mathbb{R}, 0 \leq t \leq \delta \colon (\nu + t)[CX \mapsto CX/ld] \models I(l)[CX \mapsto CX/ld]$. This expression is equivalent to (factoring out constants) $\forall t \in \mathbb{R}, 0 \leq t \leq \delta * ld \colon (\nu[CX := CX * ld] + t \models I(l)[CX \mapsto CX/ld]$ or $\forall t \in \mathbb{R}, 0 \leq t \leq \delta * ld \colon f(\nu) + t \models I(f(l))$. Hence, we have $f((l,\nu)) \overset{f(\delta)=\delta*ld}{\longrightarrow} f((l',\nu'))$.

**Part 2b:** Edge executions: $(l,\nu) \xrightarrow{a} (l',\nu') \Rightarrow f((l,\nu)) \overset{f(a)=a}{\longrightarrow} f((l',\nu'))$.

Suppose $(l,\nu) \xrightarrow{a} (l',\nu')$. By the definition of the transition system, there is some edge $e = (l,a,\phi,\lambda,l')$ with $\nu \models \phi$ and $\nu' = \nu[\lambda := 0]$. By our conversion, $f((l,\nu)) = (l, \nu[CX := CX/ld])$ and $\nu[CX := CX : ld] \models \phi[CX \mapsto CX/ld]$. Ad-

ditionally, we have the edge $e_{INT} = (l, a, \phi[CX \mapsto CX/ld], \lambda, l')$ and the transition $(l, \nu[CX := CX * ld] \xrightarrow{a} (l', (\nu[CX := CX * ld])[\lambda := 0])$. By our definition of $f$, this is the action execution $f((l, \nu)) \xrightarrow{f(a)=a} f((l', \nu'))$. □

The corollary below illustrates the strength of a non-label-preserving isomorphism.

**Corollary 3.5.1.** Let $TA_1$ and $TA_2$ be two timed automata ($TA_1 \sim TA_2$), and let $INT(TA_1)$ and $INT(TA_2)$ be the converted automaton using the same integer $ld$ (pick a positive integer $ld$ such that both converted automata have only integral constraints. One such integer is the product of any valid $ld_1$ and any valid $ld_2$.). Then $TA_1 \sim TA_2$ if and only if $INT(TA_1) \sim INT(TA_2)$.

***Proof of Corollary 3.5.1.*** We show the $\Rightarrow$ direction; the other direction is similar. Let $TA_1$ and $TA_2$ be bisimilar timed automata. By Theorem 3.5.1 and the mapping $f(\delta) = \delta * ld$, $TA_1$ is equivalent to $INT(TA_1)$ and $TA_2$ is equivalent to $INT(TA_2)$.

We then show that $INT(TA_1) \sim INT(TA_2)$. We use the bisimulation of $TA_1$ and $TA_2$ as well as $f$ and $f^{-1}$ in the proof of Theorem 3.5.1 to extract the bisimilar transitions. Let $q_1$ be a state in $INT_{TA_1}$. Apply $f^{-1}(q_1)$ and then use the bisimilarity between $TA_1$ and $TA_2$ to get a state $s_2$ in $TA_2$. We then know $q_1$ is similar to $f(s_2)$. We follow the reverse path to show the other direction. The paths taken are sketched in Figure 3.7. □

This specific variant has a specific non-label preserving isomorphism: all of the action labels are the same and the time transitions are uniformly scaled. This specific mapping, while not a label-preserving isomorphism, is quite strong because only a scaling of time advances is required to convert the automaton to one that is label-preserving isomorphic. As a result, when model checking formulas, both

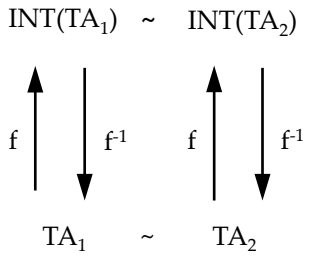


**Figure 3.7:** Diagram illustrating preservation of bisimulation. The top bisimulation can be obtained by following the other path using the bisimulation between $TA_1$ and $TA_2$. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

the automaton and the formula constraints can be rescaled to both be integers.

### 3.5.2 Clock Assignments

A clock assignment gives an edge the additional power to assign a clock to the current value of another clock. We follow the definitions in Yovine [163, 164].

**Definition 3.5.1** (**Clock assignments [164]**). The edge syntax replaces the set of clocks $\lambda$ with an *assignment* function $\gamma$, where $\gamma$ assigns each clock to itself (this "no change" in value makes $\gamma$ a total function), 0, or another clock $x' \in CX$. $\gamma(x) = x'$ indicates the assignment $x := x'$, where clock $x$ is assigned the value of clock $x'$. Assignments, like resets, are executed simultaneously.

These are the semantics of a clock assignment as provided in Yovine [164] (notation changed to match the symbols in this paper):

> Let $\nu \in \mathcal{V}$ and $\gamma$ be an assignment function. We denote by $\nu[\gamma]$ the

clock valuation such that for all $x \in CX$,

$$\nu[\gamma](x) = \begin{cases} \nu(\gamma(x)) & \text{if } \gamma(x) \in CX, \\ 0 & \text{otherwise} \end{cases}$$

■

By definition, a clock cannot be both reset to 0 and given a value of another clock on the same edge; however, that clock's value can be given to another clock before it is reset to 0. In a timed automaton with clock assignments, the set of clocks $\lambda$ in each edge is replaced with an assignment function $\gamma$.

**Definition 3.5.2 (Timed automaton with clock assignments).** A *timed automaton with clock assignments* $TA = (L, L_0, L_u, CX, I, E)$ is the same as the baseline timed automaton except for the following difference:

- $E \subseteq L \times \Sigma \times \Phi(CX) \times (CX \cup \{0\})^{CX} \times L$ is the set of *edges*. In an edge $e = (l, a, \phi, \gamma, l')$ has assignment function $\gamma \colon CX \longrightarrow CX \cup \{0\}$.

A set of clocks $\lambda$ to reset to 0 can be encoded as the following assignment function: $\gamma(x_i) = 0$ if $x_i \in \lambda$ and $\gamma(x_i) = x_i$ otherwise. ■

Concerning timed automata with clock assignments, Bouyer et al. [39] shows two properties: the decidability of reachability, and the ability to convert these automata to bisimilar automata without clock assignments. (Bouyer et al. [39] give the conversion and the bisimulation relation and claim that the proof of the relation follows from the conversion).

**Conversion to Base Form**

We use the conversion given in Bouyer et al. [39]. They apply their conversion to automata without invariants, but their conversion also handles updates of the form $x := c$ where $c \in \mathbb{Z}^0$. We take their conversion and present an adapted version that converts the previously defined variant to the baseline version.

When removing clock assignments, we extend clock constraint substitution (Definition 3.4.1) to allow a clock index to be substituted for another clock index. For example, $\phi[\{x_i\} \mapsto \{x_j\}]$ replaces every appearance of the term $x_i$ in $\phi$ with $x_j$.

**Conversion Idea:** Given that time elapses at the same rate for all clocks, after a clock assignment, the clock with the assigned value is equal to the clock it is assigned to until one of those clocks is reset. As a result, instead of performing a clock assignment, the timed automaton notes that the assigned clock is represented by the clock it was assigned. Mathematically, the locations of each timed automaton are enriched with a mapping of clocks $\sigma\colon CX \longrightarrow CX$ with $\sigma(x_i)$ denoting the clock that currently represents clock $x_i$. The mapping is changed at each transition to handle additional clock assignments and clock resets.

**Formal Conversion:** We introduce the location component set $CX^{CX}$ that stores clock mappings. This component, denoted $\sigma$, can also be viewed as a function $\sigma\colon CX \longrightarrow CX$ where $\sigma(x_i)$ denotes the clock that currently represents clock $x_i$. Hence, location component $\sigma$, also sometimes denoted $(x_{i_1}, \ldots, x_{i_n})$, means clock $x_j$ is currently represented by clock $x_{i_j}$. The identity location component is $(x_1, x_2, \ldots, x_n)$.

To handle a clock assignment $x_i := x_k$, instead of performing the assignment we set $\sigma(x_i) = x_k$ and use clock $x_k$ to represent clock $x_i$ in future transitions. Guards of outgoing edges and invariants become substituted clock constraints

where clocks are substituted with the clocks in their location components. (Invariants are not addressed in Bouyer et al. [39].) Resets and assignments on incoming edges are changed. Note that after the clock assignment, while clock $x_i$ does have a value, it is ignored, and the clock $x_i$ is treated as a spare clock.

For each edge $e = (l, a, \phi, \gamma, l')$, we form a function $\sigma_a \colon CX \longrightarrow CX$, defined as:

$$\sigma_a(x_i) = \begin{cases} x_j & \text{if } \gamma(x_i) = x_j \text{ (if } x_i := x_j) \\ x_i & \text{otherwise} \end{cases} \tag{3.6}$$

The function $\sigma_a$ handles edges without resets. To address the more complicated case of clock resets, we define a function $\Gamma_r \colon CX^{CX} \times CX \longrightarrow CX$ that takes in a function $\sigma$ and a clock $x_i$ and gives the clock that represents $x_i$. The function $\Gamma_r$ is defined as:

$$\Gamma_r(\sigma, x_i) = \begin{cases} x_c & \text{if } \gamma(x_i) = 0, x_c \notin \sigma(CX), \text{ and } \forall x_j \neq x_i, \Gamma_r(\sigma, x_j) \neq x_c \\ \sigma(x_i) & \text{if } \gamma(x_i = 0) \text{ and } \sigma(CX) = [CX] \\ \sigma(x_i) & \text{otherwise} \end{cases} \tag{3.7}$$

For notational purposes, $\Gamma_r(\sigma)$ denotes the function $\Gamma_r(\sigma) \colon CX \longrightarrow CX$ such that $\Gamma_r(\sigma)(x) = \Gamma_r(\sigma, x)$. For each $\sigma$, there may be multiple functions $\Gamma_r$ because there may be multiple clocks $x_c$ to choose from in the first case. Additionally, the choice of clocks $x_c$ is chosen such that for each $\sigma$, clock $x_c$ is the output of $\Gamma_r(\sigma)$ for at most one clock $x_c$. This constraint is represented by the third conjunct of the first case. The clocks $x_c$ are the "spare" clocks that were not currently representing any clock. These clocks can be used to represent clocks after a reset.

For each edge, given the current mapping $\sigma_{cx}$, the function we obtain from $\sigma_{cx}$ is $\sigma'_{cx}\colon CX \longrightarrow CX$ where $\sigma'_{cx}(x_i) = \Gamma_r(\sigma_a \circ \sigma, x_i)$. The timed automaton uses substitution for invariant and guard constraints in the new locations and the edges.

Using $\sigma'_{cx}$, given a timed automaton with clock assignments
$TA = (L, L_0, L_u, \Sigma, CX, I, E)$, we form the timed automaton
$ASN(TA) = (L_{ASN}, L_{0_{ASN}}, L_{u_{ASN}}, \Sigma, CX, I_{ASN}, E_{ASN})$ as follows:

- $L_{ASN} = L \times CX^{|CX|}$, and each location is $(l, \sigma_{cx})$, $l \in L$, and $\sigma_{cx} \in CX^{|CX|}$. We will refer to a state as $(l, \sigma_{cx}, \nu)$.
- $L_{0_{ASN}} = L_0 \times \{(x_1, x_2, \ldots, x_n)\}$
- $L_{u_{ASN}} = L_u \times \{(x_1, x_2, \ldots, x_n)\}$
- $I_{ASN}\colon L_{ASN} \longrightarrow \Phi(CX)$, such that $I_{ASN}(l \times \sigma_{cx}) = I(l)[CX \mapsto \sigma_{cx}]$.
- $E_{ASN} = \{((l, \sigma_{cx}), a, \phi[CX \mapsto \sigma_{cx}], \lambda[CX \mapsto \sigma'_{cx}], (l', \sigma'_{cx})) \mid (l, a, \phi, \gamma, l') \in E\}$. For each edge, $\sigma'_{cx} = \Gamma_r(\sigma_a \circ \sigma_{cx})$. Note that $\sigma'_{cx}$ is based off $\gamma$ but is computed before $\lambda$.

We define function composition $f \circ g$ as $(f \circ g)(x) = f(g(x))$.

From the conversion, the resulting automaton has a number of states that is exponential in the number of clocks in the timed automaton. To illustrate the conversion we provide two examples.

**Example 3.5.1.** Consider the timed automaton $TA_5$ in Figure 3.8 (top). The converted timed automaton is $ASN(TA_5)$ in Figure 3.8 (bottom).

Consider the edge $e = (0, a, x_2 \geq 2, (x_1 := 0, x_2 := x_1), 1)$. To determine $\sigma'_{cx}$, we first compute $\sigma_a$. For this edge, $\sigma_a(x_1) = x_2$ and $\sigma_a(x_2) = x_2$. Then, we compute $\Gamma_r(\sigma_a, x_1) = x_2$ and $\Gamma_r(\sigma_a, x_2) = x_1$. In this case $x_1$ is the "spare clock" we can use

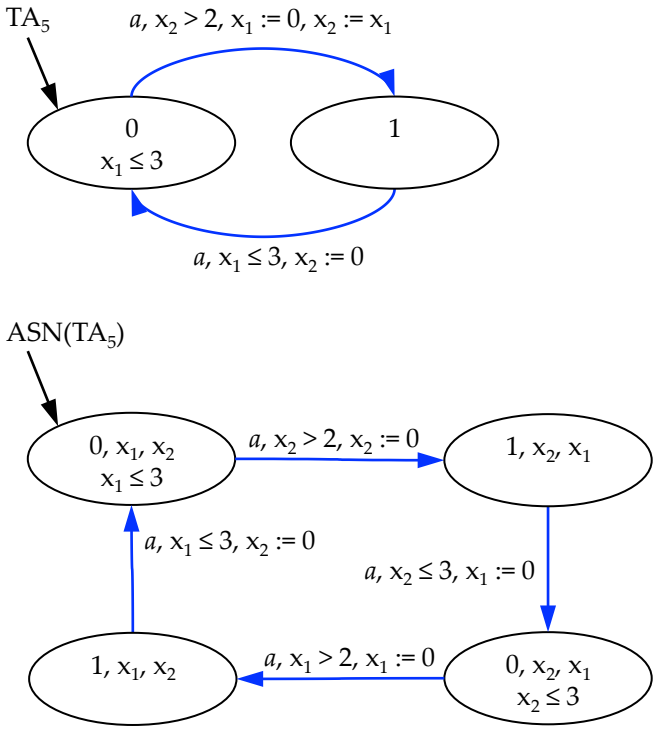


**Figure 3.8:** Timed automaton $TA_5$ with clock assignments (top) and the timed automaton $ASN(TA_5)$ after performing the conversion (bottom). In $ASN(TA_5)$, only the states reachable from the initial state are shown. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

to track $x_2$ after $x_2$ is reset to 0. Hence, for that edge, $\sigma'_{cx}(x_1) = x_2$ and $\sigma'_{cx}(x_2) = x_1$.

■

**Example 3.5.2.** Consider the timed automaton $TA_6$ in Figure 3.9 (left). The converted timed automaton is $ASN(TA_6)$ in Figure 3.9 (right).

■

Bouyer et al. [39] give the conversion, the bisimulation relation, and from the construction claim (without proof) that the relation is indeed a bisimulation. We

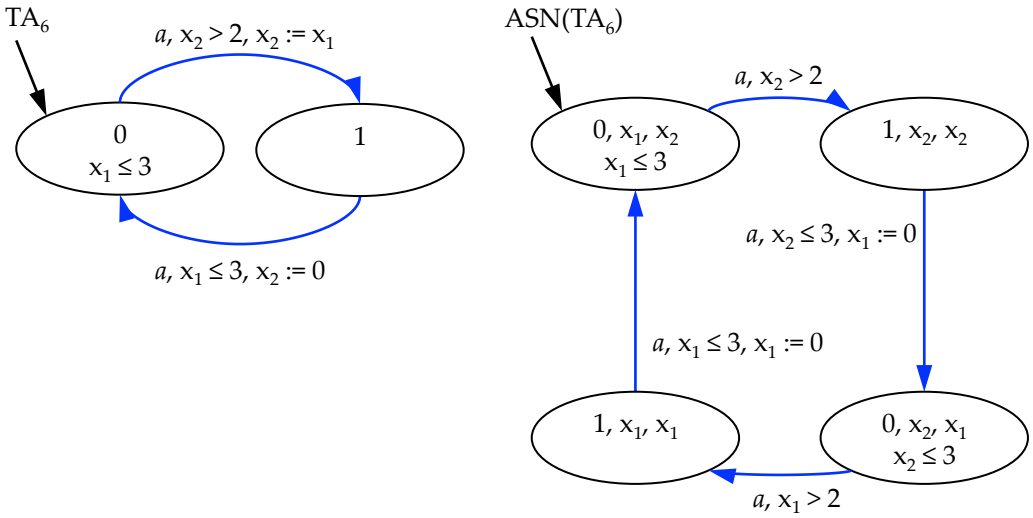


**Figure 3.9:** Timed automaton $TA_6$ with clock assignments (left) and the timed automaton $ASN(TA_6)$ after performing the conversion (right). In $ASN(TA_6)$, only the states reachable from the initial state are shown. Figure is used and adapted from Fontana and Cleaveland [74] with permission.

present the relation from Bouyer et al. [39] and prove that it is a bisimulation.

**Theorem 3.5.2.** Let $TA$ be a timed automaton, and $ASN(TA)$ be the timed automaton obtained from converting $TA$ to eliminate all clock assignments. Then they are bisimilar ($TS(TA) \cong_r TS(ASN(TA))$).

***Proof of Theorem 3.5.2.*** Both timed automata have the same event set $\Sigma$. Consider the relation $R$ on $Q_{TS(TA)} \times Q_{TS(ASN(TA))}$ defined as follows:

$$R = \{((l, \nu), (l, \sigma, \nu \circ \sigma))\}$$

In this definition, $\nu \circ \sigma$ is defined as $(\nu \circ \sigma)(x_i)) = \nu(\sigma(x_i))$. By definition, since the initial $\sigma$ is the identity function, initial states in $TS(TA)$ are mapped to initial

states in $TS(ASN(TA))$.

Now we show that $R$ is the bisimulation relation. To show that $R$ is a bisimulation, we are only concerned with the states reachable from an initial state.

We need to show for all $((l,\nu),(l_R,\sigma_R,\nu_R)) \in R$:

$\Rightarrow$:

$\forall \delta \in \mathbb{R}^{\geq 0} : (l,\nu) \xrightarrow{\delta} (l',\nu') \Rightarrow$

$(l_R,\sigma_R,\nu_R) \xrightarrow{\delta} (l'_R,\sigma'_R,\nu'_R)$ and $((l',\nu'),(l'_R,\sigma'_R,\nu'_R)) \in R$

$\forall a \in \Sigma : (l,\nu) \xrightarrow{a} (l',\nu') \Rightarrow$

$(l_R,\sigma_R,\nu_R) \xrightarrow{a} (l'_R,\sigma'_R,\nu'_R))$ and $((l',\nu'),(l'_R,\sigma'_R,\nu'_R)) \in R$

$\Leftarrow$:

$\forall \delta \in \mathbb{R}^{\geq 0} : (l_R,\sigma_R,\nu_R) \xrightarrow{\delta} (l'_R,\sigma'_R,\nu'_R) \Rightarrow$

$(l,\nu) \xrightarrow{\delta} (l',\nu')$ and $((l',\nu'),(l'_R,\sigma'_R,\nu'_R)) \in R$

$\forall a \in \Sigma : (l_R,\sigma_R,\nu_R) \xrightarrow{a} (l'_R,\sigma'_R,\nu'_R)) \Rightarrow$

$(l,\nu) \xrightarrow{a} (l',\nu')$ and $((l',\nu'),(l'_R,\sigma'_R,\nu'_R)) \in R$

We prove the $\Rightarrow$ direction. We omit the similar proof of the other ($\Leftarrow$) direction.

**Part 1a:** Time advances: $((l,\nu),(l_R,\sigma_R,\nu_R)) \in R$ and $(l,\nu) \xrightarrow{\delta} (l',\nu') \Rightarrow$ $(l_R,\sigma_R,\nu_R) \xrightarrow{\delta} (l'_R,\sigma'_R,\nu'_R)$ and $((l',\nu'),(l'_R,\sigma'_R,\nu'_R)) \in R$.

Suppose $((l,\nu),(l_R,\sigma_R,\nu_R)) \in R$ and $(l,\nu) \xrightarrow{\delta} (l,\nu+\delta)$. By definition of $R$, $l_R = l$ and $\nu_R = \nu \circ \sigma_R$. By construction, $I((l_R,\sigma_R)) = I(l)[CX \mapsto \sigma_R(CX)]$. Hence, for all $\delta', 0 \leq \delta' \leq \delta$, $\nu+\delta' \models I(l)$ if and only if $\nu_r + \delta' = (\nu+\delta') \circ \sigma_R \models I(l)[CX \mapsto \sigma_R]$. Furthermore, by definition, $(\nu+\delta)\circ\sigma = (\nu\circ\sigma)+\delta$. Hence, $(l_R,\sigma_R\nu_R) \xrightarrow{\delta}$ $(l_R,\sigma_R\nu_R+\delta)$ and $((l,\nu+\delta),(l_R,\sigma_R,\nu_R+\delta)) \in R$.

**Part 1b:** Edge executions: $((l,\nu),(l_R,\sigma_R,\nu_R)) \in R$ and $(l,\nu) \xrightarrow{a} (l',\nu') \Rightarrow$ $(l_R,\sigma_R,\nu_R) \xrightarrow{a} (l'_R,\sigma'_R,\nu'_R)$ and $((l',\nu'),(l'_R,\sigma'_R,\nu'_R)) \in R$.

Suppose $((l,\nu),(l_R,\sigma_R,\nu R)) \in R$ and $(l,\nu) \xrightarrow{a} (l',\nu')$. By definition, since the transition was taken, $\nu' = \nu \circ \gamma$, $\nu \models \phi$, and $\nu' \models I(l')$. By definition of $R$, $(l_R,\sigma_R,\nu_R) = (l,\sigma_R,\nu \circ \sigma_R)$. Since $\nu \models \phi$, $\nu \circ \sigma_R \models \phi[CX \mapsto \sigma_R]$. Now we examine $\sigma'$ formed by the construction, which is $\sigma' = \Gamma_r(\sigma_a \circ \sigma_R)$. By definition $\sigma_a \circ \sigma_R$ and $\gamma$ agree on all clocks that are not reset. For any clocks $x_i$ that are reset, if $\sigma_a \circ \sigma_R$ is onto, then $(\sigma_a \circ \sigma_R)(x_i) = \sigma_R(x_i)$. Else, then we know there is at least one spare clock $x_c$ and $\sigma'(x_i) = x_c$. Since all clocks $x_c$ were not in any constraints, this clock can be set to 0 and used for future states. Because $\nu \circ \sigma' \models I(l')[CX \mapsto \sigma']$, we have the transition $(l,\sigma_R,\nu \circ \sigma_R) \xrightarrow{a} (l',\sigma',\nu' \circ \sigma')$. □

## 3.6 Composition of Variant Conversions

Throughout this chapter we discuss many variants of timed automata and show how each individual variant can be "translated away" and converted to the baseline formalism. In practice, it is possible to have timed automata with a combination of these variants. One combination is: data variables, disjunctive guard constraints, and clock difference inequalities allowed in clock constraints. In this section we show how to utilize the conversions discussed in this paper to convert an automaton with a combination of these features into the baseline automaton. The key: compose the conversions. Furthermore, we show that the compositions of these functions preserve the minimum equivalences of the conversions and that their compositions are commutative and associative. The commutativity and associativity is argued at the semantic level (commutativity does not always hold at the syntactic level). We also remark on applying these conversions to timed automata that contain features in addition to the variants we convert out. Two such

extensions are atomic propositions and clock assignments.

### 3.6.1 Extending the Conversion Functions

For each variant, the conversion function has the domain of set of timed automata with that variant and has the range of the set of baseline timed automata. (Guarded-command programs are the exception: that function's range is the set of timed automata with variables.) However, we can take the same conversion function and extend the domain and range. By changing the domain to the set of automata of interest and the range as the set of automata with the one specified variant converted out, we can extend our conversion functions to handle extended automaton. While we may occasionally have to alter a conversion function to handle an extension, in many cases the functions can be used unchanged. When extending the function, be sure that the domain is well defined, the range is well defined, and the conversion converts an element of the domain into an element in the range.

### 3.6.2 Extended Functions Preserve Equivalence

Here we take each variant and discuss extended functions. The typical extension is extending the function to handle automata with a combination of the variants we discussed in this paper. Because the conversion is (mostly) unchanged, we can apply the previously-discussed proofs of equivalence to the extended functions. To use the same proof for the additional variants, there may be some conditions we need to be aware of. These conditions for each variant are:

- **Disjunctive guard constraints**: There must be a way to convert each guard constraint $\phi$ to disjunctive normal form: $\phi'$ where $\phi' = \bigvee_{i=1}^{k} \phi'_i$, each $\phi'_i$ is disjunction free, and each $\phi_i$ has the following property: for any state $(l, \nu)$, $(l, \nu) \models \phi'$ if and only if there is some $i$ where $(l, \nu) \models \phi'_i$. For

disjunctive guard constraints, one defines `resetPred(`$\phi, Y$`)` to first convert $\phi$ to disjunctive normal form ($\phi = \bigvee_{i=1}^{k} \phi_i$) form and `resetPred(`$\phi, Y$`)` = $\bigvee_{i=1}^{k}$ `resetPred(`$\phi_i, Y$`)`.

- **Timed automata with variables:** For every data valuation $v_d$ and every constraint $\phi$ in the automaton, we must be able to compute $\phi[VR := v_d]$. Furthermore, $\phi[VR := v_d]$ must have no variables in it.

- **Guarded-command program:** Guarded-command programs are a different notation for timed automata with variables; hence, they require the same conditions.

- **Unsatisfied invariants:** First, the reset predecessor method `resetPred(`$\phi, Y$`)` must be well-defined and computable. Second, we must have urgent locations (that also prevent 0-unit time advances), such as those in the baseline version, in the automaton.

- **Clock difference inequalities:** Time-elapses preserve clock-differences (clocks must elapse at the same rate). Furthermore, when converting out clock differences, edges must be "updated" and/or changed to correctly encode which clock difference inequalities are true and which are false. (This is a condition on additional variants; no combination of these variants requires a changed in this aspect of the conversion.)

- **Rational clock constraints:** First, we use the density of real numbers for our time advances. Second, if we can make a time advance of $\delta$ from a state, then we must be able to advance any time $\delta'$, $0 \leq \delta' \leq \delta$.

*Remark* 3.6.1 (Atomic propositions and clock assignments). One common feature is atomic propositions. Since atomic propositions are semantic shorthand for sets of

locations, when we have automata that have atomic propositions and whenever we copy (duplicate) a state, we also copy over the atomic propositions that it satisfies. Thus, for any labeling function $\mu$, location $l$ and copy $l_c$, $\mu(l_c) = \mu(l)$. We do not change the atomic propositions for unchanged locations. For the added location components $c_t$ and $c_f$ in the clock-difference conversion, we define $\mu((l, c_t)) = \mu((l, c_f)) = \mu(l)$.

Another commonly-used variant is clock assignments. They are described in Section 3.5.2. The details of the clock-difference inequalities conversion must be changed to accommodate the changing of clock difference inequalities when clocks assignments are executed on a transition. Furthermore, for unsatisfied invariants, the `resetPred`$(\phi, Y)$ must be augmented to handle the predecessor of clock assignments.

### 3.6.3 Composition Preserves Equivalences

Because label-preserving isomorphism is an equivalence relation (reflexive, symmetric and transitive) [75], if we compose two or more label-preserving isomorphic conversions, then the original and the final automaton are semantically label-preserving isomorphic.Therefore, we have the following claims:

**Claim 3.6.1.** Let us apply (compose) two or more conversion functions for label-preserving isomorphic variants (apply the functions compositionally in any order). Then the original automaton is semantically label-preserving isomorphic to the final converted automaton.

***Proof of Claim 3.6.1.*** Follows from transitivity of label-preserving isomorphism. □

**Claim 3.6.2.** Let us apply (compose) two or more conversion functions for variants

with label-preserving isomorphism or (label-preserving) reachable subsystem isomorphism (apply the functions compositionally in any order). Then the reachable subsystem of the original automaton is semantically label-preserving isomorphic to the reachable subsystem of the final converted automaton.

***Proof of Claim 3.6.2.*** Follows from transitivity of isomorphism of reachable subsystems, which comes from transitivity of isomorphism. □

**Claim 3.6.3.** Suppose we have a timed automaton with rational clock constraints and other label-preserving isomorphic variants. Then the original automaton is non-label-preserving isomorphic to the final converted automaton.

***Proof of Claim 3.6.3.*** By applying the rational-clock constraints conversion first, we get a non-label preserving isomorphism, which is a transitive relation (compose the label changes). Since all of the other conversions are label-preserving isomorphic, they are also non-label-preserving (with the identity mapping of labels). Hence, by transitivity, we have a non-label preserving isomorphism. □

**Claim 3.6.4.** Suppose we have a timed automaton with rational clock constraints and other variants for which we provided conversion functions. Then the reachable subsystem of the original automaton after a label remapping is semantically isomorphic to the reachable system of the final converted automaton. I.e. the reachable subsystems are non-label-preserving isomorphic.

***Proof of Claim 3.6.4.*** By applying the rational-clock constraints conversion first, we get a non-label preserving isomorphism. By definition, any reachable subsystem isomorphism is a non-label-preserving reachable subsystem isomorphism (use the identity mapping for the labels), by transitivity the reachable subsys-

tems of the original and converted automata have a non-label-preserving isomorphism. □

### 3.6.4 Commutativity and Associativity of Semantics

When we apply multiple conversions to convert out multiple variants, we are equivalently applying the function that is the composition of the applied conversion functions. We show that for these variants and *these conversion functions* on the semantic level (transition systems of the timed automata), the composition of these commutative and associative. (Depending on the composition order, the syntax may not be the same). We denote the function composition $f \circ g$ as $(f \circ g)(x) = f(g(x))$.

**Theorem 3.6.5** (**Commutativity and associativity**). Let $f, g, h$ be three extended conversion functions (extended properly when needed) for three different variant features that have conversion functions in this paper. Then:

$$f \circ g = g \circ f \qquad \text{(Commutativity)}$$

$$(f \circ g) \circ h = f \circ (g \circ h) \qquad \text{(Associativity)}$$

***Proof of Theorem 3.6.5.*** The proof is done, at the semantic level, on a case-by-case basis of the different variants.

First, any function is commutative and associative with itself.

Second, for disjunctive guard constraints, timed automata with variables and guarded-command programs, the conversion function is the identity function. By definition, any function composed with the identity function is the original func-

tion. Hence, any case involving the identity function is either commutative and associative or reduces to a simpler case. We start with commutativity.

**Commutativity Case 1: Unsatisfied Invariants (f) and Clock Difference Inequalities (g)**. Let $f$ be the conversion function that eliminates unsatisfied invariants and $g$ be the conversion function that eliminates clock difference constraints. Thus:

$$f((l,\nu)) = \begin{cases} (l,\nu) & \text{if } \nu \models I(l) \\ (l_u,\nu) & \text{otherwise} \end{cases}$$

$$g((l,\nu)) = (l \times l_{diff},\nu)$$

Applying the compositions:

$$(f \circ g)(l,\nu) = \begin{cases} (l \times l_{diff},\nu) & \text{if } \nu \models I(l) \\ ((l \times l_{diff})_u,\nu) & \text{otherwise} \end{cases}$$

$$(g \circ f)(l,\nu) = \begin{cases} (l \times l_{diff},\nu) & \text{if } \nu \models I(l) \\ (l_u \times l_{diff},\nu) & \text{otherwise} \end{cases}$$

which are the same. By definition, the $u$ represents an "urgent copy." Furthermore, if any component of a location is urgent, then the location is urgent. (Note: while the compositions are semantically equivalent, these compositions have different labeling (syntax) in which different locations have urgent copies.)

**Commutativity Case 2: Unsatisfied Invariants (f) and Rational Clock Constraints (g)** Let $f$ be the conversion function that eliminates unsatisfied invariants, $g$ be the conversion function that eliminates rational clock constraints, and $ld$ be the

positive integer used in the relabeling of $g$.

$$f((l,\nu)\times\sigma) = \begin{cases} (l,\nu)\times\sigma & \text{if } \nu \models I(l) \\ (l_u,\nu)\times\sigma & \text{otherwise} \end{cases}$$

$$g((l,\nu)\times\sigma) = \begin{cases} (l,\nu[CX := CX * ld])\times\sigma * ld & \text{if } \sigma \in \mathbb{R}^{\geq 0} \\ (l,\nu[CX := CX * ld])\times\sigma & \text{otherwise } (\sigma \in \Sigma) \end{cases}$$

Applying the compositions:

$$(f\circ g)((l,\nu)\times\sigma) = \begin{cases} (l,\nu[CX := CX * ld])\times\sigma * ld & \text{if } \nu \models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ (l_u,\nu[CX := CX * ld])\times\sigma * ld & \text{if } \nu \not\models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ (l,\nu[CX := CX * ld])\times\sigma & \text{if } \nu \models I(l) \text{ and } \sigma \in \Sigma \\ (l_u,\nu[CX := CX * ld])\times\sigma & \text{otherwise} \end{cases}$$

$$(g\circ f)((l,\nu)\times\sigma) = \begin{cases} (l,\nu[CX := CX * ld])\times\sigma * ld & \text{if } \nu \models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ (l_u,\nu[CX := CX * ld])\times\sigma * ld & \text{if } \nu \not\models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ (l,\nu[CX := CX * ld])\times\sigma & \text{if } \nu \models I(l) \text{ and } \sigma \in \Sigma \\ (l_u,\nu[CX := CX * ld])\times\sigma & \text{otherwise} \end{cases}$$

which are the same. Since $f$ only alters the location $l$, and $g$ alters only the valuation $\nu$ and the action symbol $\sigma$, commutativity follows .

**Commutativity Case 3: Rational Clock Constraints (f) and Clock Difference Inequalities (g)**. Let $f$ be the conversion function that eliminates rational clock constraints, $ld$ be the positive integer used in the relabeling of $f$, and $g$ be the

conversion function that eliminates clock difference constraints.

$$f((l,\nu)\times\sigma) = \begin{cases} (l,\nu[CX := CX * ld])\times\sigma * ld & \text{if } \sigma \in \mathbb{R}^{\geq 0} \\ (l,\nu[CX := CX * ld])\times\sigma & \text{otherwise } (\sigma \in \Sigma) \end{cases}$$

$$g((l,\nu)\times\sigma) = (l \times l_{diff},\nu)\times\sigma$$

Applying the compositions:

$$(f\circ g)((l,\nu)\times\sigma) = \begin{cases} (l \times l_{diff},\nu[CX := CX * ld])\times\sigma * ld & \text{if } \sigma \in \mathbb{R}^{\geq 0} \\ (l \times l_{diff},\nu[CX := CX * ld])\times\sigma & \text{otherwise} \end{cases}$$

$$(g\circ f)((l,\nu)\times\sigma) = \begin{cases} (l \times l_{diff},\nu[CX := CX * ld])\times\sigma * ld & \text{if } \sigma \in \mathbb{R}^{\geq 0} \\ (l \times l_{diff},\nu[CX := CX * ld])\times\sigma & \text{otherwise} \end{cases}$$

which are the same. Since $f$ only alters the valuation $\nu$ and the action symbol $\sigma$, and $g$ only alters the location $l$, commutativity follows .

**Associativity Case: Rational Clock Constraints (f), Unsatisfied Invariants (g) and Clock Difference Inequalities (h)**. Using commutativity proofs of $f, g$ and $h$ and using the fact that the identity function is associative, we can reduce all other cases of associativity to the commutativity cases or to this case.

Let $f$ be the conversion function for rational clock constraints, $ld$ be the positive integer used in the relabeling of $f$, $g$ be the conversion function for unsatisfied

invariants, and $h$ be the conversion function for clock difference inequalities.

$$f((l,\nu)\times\sigma) = \begin{cases} (l,\nu)\times\sigma & \text{if } \nu \models I(l) \\ (l_u,\nu)\times\sigma & \text{otherwise} \end{cases}$$

$$g((l,\nu)\times\sigma) = \begin{cases} (l,\nu[CX := CX * ld])\times\sigma * ld & \text{if } \sigma \in \mathbb{R}^{\geq 0} \\ (l,\nu[CX := CX * ld])\times\sigma & \text{otherwise } (\sigma \in \Sigma) \end{cases}$$

$$h((l,\nu)\times\sigma) = (l \times l_{diff},\nu)\times\sigma$$

Computing $(f \circ g) \circ h$ is, we get from composing $f \circ g$

$$((f\circ g)\circ h)((l,\nu)\times\sigma) = \left( \begin{cases} (l,\nu[CX := CX * ld])\times\sigma * ld & \text{if } \nu \models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ (l_u,\nu[CX := CX * ld])\times\sigma * ld & \text{if } \nu \not\models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ (l,\nu[CX := CX * ld])\times\sigma & \text{if } \nu \models I(l) \text{ and } \sigma \in \Sigma \\ (l_u,\nu[CX := CX * ld])\times\sigma & \text{otherwise } (\nu \not\models I(l) \text{ and } \sigma \in \Sigma) \end{cases} \right) \circ h$$

yielding

$$((f \circ g) \circ h)((l, \nu) \times \sigma) = \begin{cases} (l \times l_{diff}, \nu[CX := CX * ld]) \times \sigma * ld \\ \qquad \text{if } \nu \models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ (l_u \times l_{diff}, \nu[CX := CX * ld]) \times \sigma * ld \\ \qquad \text{if } \nu \not\models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ (l \times l_{diff}, \nu[CX := CX * ld]) \times \sigma \\ \qquad \text{if } \nu \models I(l) \text{ and } \sigma \in \Sigma \\ (l_u \times l_{diff}, \nu[CX := CX * ld]) \times \sigma \\ \qquad \text{otherwise } (\nu \not\models I(l) \text{ and } \sigma \in \Sigma) \end{cases}$$

Now computing $f \circ (g \circ h)$, we get from composing $g \circ h$

$$(f \circ (g \circ h))((l, \nu) \times \sigma) = f \circ \left( \begin{cases} (l \times l_{diff}, \nu) \times \sigma & \text{if } \nu \models I(l) \\ ((l \times l_{diff})_u, \nu) \times \sigma & \text{otherwise} \end{cases} \right)$$

yielding

$$(f \circ (g \circ h))((l, \nu) \times \sigma) = \begin{cases} (l \times l_{diff}, \nu[CX := CX * ld]) \times \sigma * ld \\ \qquad \text{if } \nu \models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ ((l \times l_{diff})_u, \nu[CX := CX * ld]) \times \sigma * ld \\ \qquad \text{if } \nu \not\models I(l) \text{ and } \sigma \in \mathbb{R}^{\geq 0} \\ (l \times l_{diff}, \nu[CX := CX * ld]) \times \sigma \\ \qquad \text{if } \nu \models I(l) \text{ and } \sigma \in \Sigma \\ ((l \times l_{diff})_u, \nu[CX := CX * ld]) \times \sigma \\ \qquad \text{otherwise } (\nu \not\models I(l) \text{ and } \sigma \in \Sigma) \end{cases}$$

which is the same as the previous composition. □

*Remark* 3.6.2 (Guarded-command programs). Guarded-command programs are a syntactic relabeling of timed automata with variables and the conversion functions to and from guarded-command programs and timed automata with variables are both the identity function. As a consequence of the results in this section, we can both extend guarded-command programs to have the features we can convert out as well as convert those features out of guarded-command programs. We can choose to first convert a guarded-command program to a timed automaton with variables and then convert out the features or we can choose to first convert out the features and then convert the guarded-command program to a timed automaton. From either order, we will end up with (semantically) the same automaton.

### 3.6.5 Putting it All Together

We give an example showing how to convert out multiple variants.

**Example 3.6.1.** Let $TA$ be a timed automaton with the following three extensions: disjunctive clock constraints in guards, variables, and clock difference inequalities in clock constraints. We will convert these out one variant at a time. Given that the extended functions are commutative and associative, we can apply them in any order.

We start by eliminating the variables and converting them to locations. The semantic conversion function is $f((l, \nu_d, \nu)) = ((l, \nu_d), \nu)$ The other properties of the automaton do not influence this and are thus left unchanged. Then we wish to get rid of the disjunctive clock constraints. We now apply the syntactic conversion of converting the edges to set of edges. The semantic function $f_2((l, \nu)) = (l, \nu)$ and is unchanged.

Lastly, we now have the timed automaton in our base form with the sole extension of clock difference inequalities. We apply the conversion in Section 3.4.2. ■

## 3.7 Summary of Established Equivalences

We summarize the equivalences established in this chapter in Table 3.1.

## 3.8 Dissertation Contributions

### 3.8.1 Contributions

These are my contributions discussed in this chapter:

- We gave a formal baseline definition for a timed automata based on definitions of others.
- We provided formal definitions for the following variants: disjunctive guard constraints, timed automata with variables and different semantics for unsatisfied invariants.

**Table 3.1:** Summary of timed automata variants and their equivalences.

| Variant | Equivalence Proved | Comments |
|---|---|---|
| Disjunctive guard constraints | label-preserving isomorphism | Map each edge to a set of edges going, mapping each disjunct of the guard to its own edge. |
| Timed automata with variables | label-preserving isomorphism | Make each assignment of variables a location. |
| Guarded-command programs | label-preserving isomorphism | Convert the guards to edges, making a timed automata with variables from the guarded command program. |
| Differing invariant semantics | reachable-subsystem isomorphism | Disable transitions by pushing the reset predecessor of the entering location's invariant onto the guard. |
| Clock-difference constraints | reachable-subsystem isomorphism | Encode the truth of a clock different constraint in the locations by adding a location component representing the truth of each clock-difference constraint. |
| Rational constants | non-label preserving isomorphism | Uniformly scale the constants by multiplying them all by some integer such that all constants become non-negative integers. |
| Clock assignments | bisimulation | Keep track of clock assignments by augmenting locations with a record of which clocks in the automata are representing multiple other clocks. |

- For timed automata with disjunctive guard constraints, timed automata with variables, and guarded-command programs, we show those variants are isomorphic to the baseline formalism and give a conversion translating out each variant.

- For the different unsatisfied invariant semantics and allowing clock differ-

ences in clock constraints, we show that the reachable subsystems of those variants are isomorphic to the reachable subsystem of the baseline formalism and give a conversion translating out each variant.

- For rational clock constraints, we give a non-label preserving isomorphism to the baseline formalism (uses integer constants only) and give a conversion translating out the rational constants.

- We showed how the above conversions are composable, not only for timed automata with these features but also for timed automata with even more features. We give a framework, a composable timed automata, that give sufficient conditions describing extensions that still allow the equivalent variants to be converted out. We then showed that these conversions are commutative and associative at the semantic level.

### 3.8.2 Future Work

Future work includes allowing the initial state to have clock values other than 0, and potentially to allow a set of initial states whose clock values are defined by a clock zone or a union of clock zones. Additionally, future work includes handling disjunctive constraints in invariants. While these constraints cannot be converted in a fashion similar to converting out disjunctive constraints in guards, future work involves determining the expressiveness of this additional feature. Disjunctive constraints in invariants are used to express timed automata with deadlines (see Bornot and Sifakis [32], Bornot et al. [33], Bowman [46], Bowman and Gómez [47], Gómez and Bowman [78]).

# Chapter 4

# Timed Logics and Expressivity Results

With a timed automaton defined, we can now begin to ask properties about it. To do this, we work with relevant timed logics. The goal is to expand upon their theory, allowing us to say more about timed automata. The most commonly used timed logic is TCTL (Timed Computation Tree Logic) Alur et al. [12], a timed extension of CTL (Computation Tree Logic). It is the logic used to describe properties in various tools including both UPPAAL [23] and RED [157]. There are also two timed modal-mu calculi that were developed. The first is $T_\mu$, developed by Henzinger et al. [88]. While $T_\mu$ is expressive, it is impractical to write an unbounded liveness formulas in $T_\mu$ (writing a liveness formula in $T_\mu$ requires the bound to be guessed [88]). This makes $T_\mu$ sometimes undesirable. The second timed modal-mu calculus developed is $L_{\nu,\mu}$, introduced independently by Sokolsky and Smolka [147] and Aceto and Laroussinie [2] (the version with just greatest fixpoints was introduced in Laroussinie et al. [108]). To provide additional power, a relativized operator was added by Bouyer et al. [41] and was shown to give additional power. This logic is called $L^{rel}_{\nu,\mu}$. While this logic is promising to model check, many expressiveness properties, including expressions of TCTL formulas, have not been obtained. This chapter focuses on studying TCTL, $T_\mu$, $L_{\nu,\mu}$ and $L^{rel}_{\nu,\mu}$ and compares their expressive powers by (in many cases) showing whether we can write any formula in one logic in another logic or not. Among other properties, by proving that $L^{rel}_{\nu,\mu}$ can express any TCTL formula (which includes any safety and liveness property), we add the theory that provides the ability for $L^{rel}_{\nu,\mu}$ to be used.

## 4.1 Timed Computation Tree Logic (TCTL)

Timed computation tree logic (TCTL), is a branching-time logic used for a dense-time representation of properties.

Following the definition in Alur et al. [12], we present the branching-time logic Timed Computation Tree Logic (TCTL).

**Definition 4.1.1 (TCTL syntax).** A *TCTL formula* can be constructed with the following grammar:

$$\phi ::= p \mid cc \mid \texttt{tt} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]] \mid E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$$

Here, $p \in 2^L$ is an atomic proposition (a subset of locations); $cc \in \Phi(CX)$ is a clock constraint; $\bowtie$ is any one of the operators $=, <, >, \leq, \geq$; and $c \in \mathbb{Z}^{\geq 0} \cup \{\infty\}$. ■

**Definition 4.1.2 (TCTL semantics).** Let $TA$ be a timed automaton and $\phi$ be a TCTL formula. Then the *semantics* of $\phi$ (denoted $[\![\phi]\!]_{TA}$), the set of states in $TA$ that satisfy $\phi$, is:

- $[\![p]\!]_{TA} = \{(l, \nu) \in Q \mid l \in p\}$.
- $[\![cc]\!]_{TA} = \{(l, \nu) \in Q \mid \nu \models cc\}$.
- $[\![\texttt{tt}]\!]_{TA} = Q$.
- $[\![\neg\phi]\!]_{TA} = Q - [\![\phi]\!]_{TA}$.
- $[\![\phi_1 \wedge \phi_2]\!]_{TA} = [\![\phi_1]\!]_{TA} \cap [\![\phi_2]\!]_{TA}$.
- $[\![E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]]\!]_{TA} = \{(l, \nu) \in Q \mid \exists$ time-divergent run $\pi_{tr} : q_0 = (l, \nu) \wedge \exists i \geq 0 : \big(\exists d, t_i \leq d \leq t_{i+1} : \big(d \bowtie c \wedge (l_i, (\nu_i + (d - t_i)) \models \phi_2\big)$ and

$\forall d' < d, t_i \leq d' \leq t_{i+1}\colon \big((l_i, \nu_i + (d' - t_i)) \models \phi_1\big)$ and $\forall j \leq i\colon \big(\forall d' \leq d, t_j \leq d' \leq t_{j+1}\colon \big((l_j, \nu_j + (d' - t_j)) \models \phi_1\big)\big)\big)\big)\}$.

- $[\![E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]]\!]_{TA} = \{(l, \nu) \in Q \mid \exists$ time-divergent run $\pi_{tr} : q_0 = (l, \nu) \wedge \forall i \geq 0\colon \big(\forall d, t_i \leq d \leq t_{i+1}\colon d \bowtie c \rightarrow (l_i, (\nu_i + (d - t_i)) \models \phi_2)$ or $\exists d' < d, t_i \leq d' \leq t_{i+1}\colon \big((l_i, \nu_i + (d' - t_i)) \models \phi_1\big)$ or $\exists j \leq i\colon \big(\exists d' \leq d, t_j \leq d' \leq t_{j+1}\colon \big((l_j, \nu_j + (d' - t_j)) \models \phi_1\big)\big)\big)\}$.

$TA$ satisfies $\phi$ iff the initial state $(l_0, \nu_0)$ satisfies $\phi$. ■

We also use the following abbreviations: $\texttt{ff}$ for $\neg\texttt{tt}$, $\phi_1 \vee \phi_2$ for $\neg(\neg\phi_1 \wedge \neg\phi_2)$, and $\phi_1 \rightarrow \phi_2$ for $\neg\phi_1 \vee \phi_2$. Abbreviations for commonly-used derived temporal operators include: $A\,[[\sigma_1]\,U_{\bowtie c}\,[\sigma_2]]$ for $\neg(E\,[[\neg\sigma_1]\,R_{\bowtie c}\,[\neg\sigma_2]])$, $A\,[[\sigma_1]\,R_{\bowtie c}\,[\sigma_2]]$ for $\neg(E\,[[\neg\sigma_1]\,U_{\bowtie c}\,[\neg\sigma_2]])$ (all paths operators), $EF_{\bowtie c}\,[\sigma]$ for $E\,[[\texttt{tt}]\,U_{\bowtie c}\,[\sigma]]$, $AF_{\bowtie c}\,[\sigma]$ for $A\,[[\texttt{tt}]\,U_{\bowtie c}\,[\sigma]]$ (eventually), $AG_{\bowtie c}\,[\sigma]$ for $\neg EF_{\bowtie c}\,[\neg\sigma]$, $EG_{\bowtie c}\,[\sigma]$ for $\neg AF_{\bowtie c}\,[\neg\sigma]$ (always), $E\,[[\phi_1]\,W\,[\phi_2]]$ for $\neg(A\,[[\neg\phi_2]\,U\,[\neg\phi_1 \wedge \neg\phi_2]])$, and $A\,[[\phi_1]\,W\,[\phi_2]]$ for $\neg(E\,[[\neg\phi_2]\,U\,[\neg\phi_1 \wedge \neg\phi_2]])$ (weak until).

The formula $E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$ means "there exists a path where $\phi_1$ releases $\phi_2$" ($\phi_2$ has to also be true when $\phi_1$ releases $\phi_2$ from being true).

In the semantics of $E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]$, the $i$ is the state in the timed run corresponding after $i-1$ transitions are taken (some of them action executions, some time advances), and $d$ is the amount of time elapsed before $\phi_2$ is true. $d \bowtie c$ means that $d$ satisfies the constraint $\bowtie c$. $d - t_i$ is the amount of time elapsed in the $i^{th}$ transition before $\phi_2$ is satisfied.

To explain the intuition of the TCTL $E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]$ semantics, we refer to Figure 4.1 and its two path prefixes. In the first (upper) path, $\phi_2$ becomes true *immediately after an action is executed*. The first state where $\phi_2$ is true is the state at position $i$, $(l_i, \nu_i)$. Notice that $\phi_1$ must be true up until time $t_{i-1}$ (in position

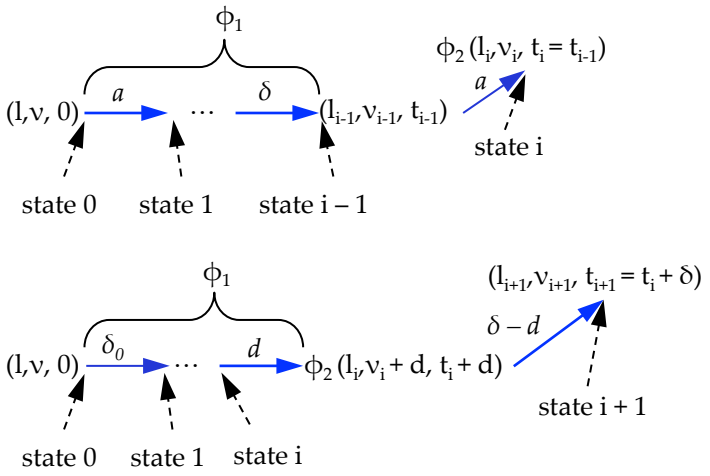


**Figure 4.1:** Two path-prefix types satisfying TCTL formula $E\,[[\phi_1]\,U\,[\phi_2]]$.

$i-1$). Since no time elapses during action executions, $t_{i-1} = t_i$ and $\phi_1$ must be true before the action into position $i$. In the second (lower) path, $\phi_2$ becomes true *during a time advance*. State $i$ is the state before the time advance, the state $i+1$ is the state after the entire time advance, and the state $(l_i, v_i + d)$ at global time $t_i + d$ is the first state (at the first time instance) when $\phi_2$ is true. Notice that $\phi_1$ has to be true for all times $t_i$ to $t_i + d$ except the time $t_i + d$.

TCTL has the nice property that formulas can be expressed simply. For instance, $AG_{<\infty}\,[a]$ is the safety property "always $a$" and $AF_{<\infty}\,[a]$ is the liveness property "inevitably $a$."

*Remark* 4.1.1 (On the definition of $A\,[[\phi_1]\,U\,[\phi_2]]$ and $E\,[[\phi_1]\,U\,[\phi_2]]$). For the until ($A\,[[\phi_1]\,U\,[\phi_2]]$, $E\,[[\phi_1]\,U\,[\phi_2]]$) operator semantics, we keep $\phi_1$ as the required precursor formula following, Alur et al. [12] as opposed to $\phi_1 \vee \phi_2$ as is used by Baier and Katoen [17], Henzinger et al. [88]. Note that in CTL, $A\,[[\phi_1]\,U\,[\phi_2]] \equiv A\,[[(\phi_1 \vee \phi_2)]\,U\,[\phi_2]]$ and $E\,[[\phi_1]\,U\,[\phi_2]] \equiv E\,[[(\phi_1 \vee \phi_2)]\,U\,[\phi_2]]$, but in TCTL when clock constraints are allowed in $\phi_1$ and $\phi_2$, this no longer holds. If one wishes to use the other semantics, simply change the formula to $A\,[[(\phi_1 \vee \phi_2)]\,U_{\bowtie c}\,[\phi_2]]$ or

$E\,[[(\phi_1 \vee \phi_2)]\,U_{\bowtie c}\,[\phi_2]]$ as desired.

## 4.2 Timed Modal-Mu Calcului $L_{\nu,\mu}$ and $L^{rel}_{\nu,\mu}$

The modal mu-calculus $L_{\nu,\mu}$ is discussed in Sokolsky and Smolka [147]; its greatest fixpoint fragment, $L_\nu$, is discussed in Bouyer et al. [45], Laroussinie et al. [108, 109]; and the relativized greatest fixpoint fragment, $L^{rel}_\nu$, is discussed in Bouyer et al. [45].

### 4.2.1 $L^{rel}_{\nu,\mu}$ Syntax and Semantics

**Definition 4.2.1** ($L_{\nu,\mu}$ **and** $L^{rel}_{\nu,\mu}$ **syntax).** The *syntax* of a $L_{\nu,\mu}$ formula can be constructed with the following grammar:

$$\phi ::= p \mid \neg p \mid \texttt{tt} \mid \texttt{ff} \mid cc \mid Y \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle(\phi) \mid$$
$$[a](\phi) \mid \exists(\phi) \mid \forall(\phi) \mid z.(\phi) \mid \mu Y.[\phi] \mid \nu Y.[\phi]$$

Here, $p \in 2^L$ is an atomic proposition, $cc \in \Phi(CX)$ is a clock constraint, $Y \in Var$ is a variable ($Var$ is the set of formula variables), $a \in \Sigma_{TA}$ is an action, and $\mu Y.[\phi]$ and $\nu Y.[\phi]$ are the least and greatest fixpoint operators over variable $Y$, respectively. For freeze quantification, $z$ is a clock in $CX_f = \{z, z_1, z_2, \ldots\}$. We assume $CX_f \cap CX = \emptyset$.

The *relativized timed modal mu-calculus* $L^{rel}_{\nu,\mu}$ *syntax* replaces $\exists(\phi)$ and $\forall(\phi)$ with $\exists_{\phi_1}(\phi_2)$ and $\forall_{\phi_1}(\phi_2)$, where $\phi_1$ is also a $L^{rel}_{\nu,\mu}$ formula. ■

To handle the clocks used in freeze quantification ($z.(\phi)$), we extend the timed automaton's states $(l, \nu)$ to *extended states* $(l, \nu, \nu_f)$ using the additional valuation component $\nu_f : CX_f \longrightarrow \mathbb{R}^{\geq 0}$. This formalism comes from Bouyer et al. [45]. When clear from context, we will refer to an extended state as $(l, \nu)$ and omit the explicit

notation of $\nu_f$.

**Definition 4.2.2 (Environment $\theta$).** An *environment* $\theta\colon Var \longrightarrow 2^Q$ is a function that assigns a set of states to each variable, where $\theta(Y)$ represents the set of states that make the formula $Y$ true. For notation, we have:

$$\theta[Y := Q'](Z) = \begin{cases} \theta(Z) & \text{if } Z \neq Y \\ Q' & \text{if } Z = Y \end{cases}$$

■

**Definition 4.2.3 ($L^{rel}_{\nu,\mu}$ semantics).** Given freeze clock set $CX_f = \{z, z_1, z_2, \ldots\}$, timed automaton $TA$ (with extended states), and initial environment $\theta$, the *semantics of a $L_{\nu,\mu}$ formula $\phi$*, denoted $[\![\phi]\!]_{TA,\theta}$, is (easier ones omitted):

- $[\![p]\!]_{TA,\theta} = \{(l, \nu, \nu_f) \in Q \mid l \in p\}$.
- $[\![Y]\!]_{TA,\theta} = \theta(Y)$.
- $[\![\phi_1 \wedge \phi_2]\!]_{TA,\theta} = [\![\phi_1]\!]_{TA,\theta} \cap [\![\phi_2]\!]_{TA,\theta}$.
- $[\![\langle a \rangle(\phi_1)]\!]_{TA,\theta} = \{(l, \nu, \nu_f) \in Q \mid \exists l_1 \in L, (\nu_1, \nu_{1_f}) \in \mathcal{V}\colon$
  $(l, \nu, \nu_f) \xrightarrow{a} \left(l_1, \nu_1, \nu_{1_f}\right)$ and $\left(l_1, \nu_1, \nu_{1_f}\right) \in [\![\phi_1]\!]_{TA,\theta}\}$.
- $[\![[a](\phi_1)]\!]_{TA,\theta} = \{(l, \nu, \nu_f) \in Q \mid \forall l_1 \in L, (\nu_1, \nu_{1_f}) \in \mathcal{V}\colon$
  if $(l, \nu, \nu_f) \xrightarrow{a} \left(l_1, \nu_1, \nu_{1_f}\right)$ then $\left(l_1, \nu_1, \nu_{1_f}\right) \in [\![\phi_1]\!]_{TA,\theta}\}$.
- $[\![\exists(\phi_1)]\!]_{TA,\theta} = \{(l, \nu, \nu_f) \in Q \mid \exists \delta \geq 0\colon (l, \nu, \nu_f) \xrightarrow{\delta} (l, \nu + \delta, \nu_f + \delta)$
  and $(l, \nu + \delta, \nu_f + \delta) \in [\![\phi_1]\!]_{TA,\theta}\}$.
- $[\![\forall(\phi)_1]\!]_{TA,\theta} = \{(l, \nu, \nu_f) \in Q \mid \forall \delta \geq 0\colon$ if $(l, \nu, \nu_f) \xrightarrow{\delta} (l, \nu + \delta, \nu_f + \delta)$
  then $(l, \nu + \delta, \nu_f + \delta) \in [\![\phi_1]\!]_{TA,\theta}\}$.

- $[\![z.(\phi_1)]\!]_{TA,\theta} = \{(l, \nu, \nu_f) \mid (l, \nu, \nu_f[z := 0]) \in [\![\phi]\!]_{TA,\theta}\}$.
- $[\![\mu Y.[\phi]]\!] =$ the least fixpoint of the function $\phi(Y') = [\![\phi]\!]_{TA,\theta[Y:=Y']}$

The *semantics of a relativized formula in* $L^{rel}_{\nu,\mu}$ *is*:

- $[\![\exists_{\phi_1}(\phi_2)]\!]_{TA,\theta} = \{(l, \nu, \nu_f) \in Q \mid \exists \delta \geq 0 : ((l, \nu + \delta, \nu_f + \delta) \models \phi_2 \wedge \forall \delta', 0 \leq \delta' < \delta : (l, \nu + \delta', \nu_f + \delta') \models \phi_1)\}$
- $[\![\forall_{\phi_1}(\phi_2)]\!]_{TA,\theta} = \{(l, \nu, \nu_f) \in Q \mid \forall \delta \geq 0 : ((l, \nu + \delta, \nu_f + \delta) \models \phi_2 \vee \exists \delta', 0 \leq \delta' < \delta : (l, \nu + \delta', \nu_f + \delta') \models \phi_1)\}$

Timed automaton $TA$ satisfies $\phi$ iff the initial (extended) state $(l_0, \nu_0, [CX_f := 0])$ satisfies $\phi$.

A formula with a propositional variable can be viewed as a monotonic function on a complete lattice (monotonicity follows from extending the proof of Cleaveland [56] to handle the timed operators). Due to Cousot and Cousot [59] we obtain an iterative semantics for fixpoints. Specifically, by treating the formula $\phi$ as a function on $Y$ where $\phi(Y') = [\![\phi]\!]_{TA,\theta[Y:=Y']}$ and $\phi^i(Y') = \phi(\phi^{i-1}(Y'))$:

$$[\![\mu Y.[\phi]]\!] = \bigcup_{i=0}^{\infty} \phi^i(\emptyset) \text{ and } [\![\nu Y.[\phi]]\!] = \bigcap_{i=0}^{\infty} \phi^i(Q)$$

Here $\infty$ may be transfinite to handle the case when the function $\phi(Y')$ is not continuous. See Kozen [100] for details. ■

The logic supports two derived operators: $[-](\phi)$ for $\bigwedge_{a \in \Sigma_{TA}} [a](\phi)$ (for all next actions) and $\langle - \rangle(\phi)$ for $\bigvee_{a \in \Sigma_{TA}} \langle a \rangle(\phi)$ (there exists a next action). These operators are also used for systems that do not use actions symbols. Furthermore, $\exists_{\mathtt{tt}}(\phi)$ is the same formula as $\exists(\phi)$, and $\forall_{\mathtt{ff}}(\phi)$ is the same formula as $\forall(\phi)$.

The formula $\exists_{\phi_1}(\phi_2)$ means, "there exists a time advance where $\phi_2$ is true and $\phi_1$ is true for all previous times", and $\forall_{\phi_1}(\phi_2)$ means, "either $\phi_2$ is true for all time

advances or $\phi_1$ releases $\phi_2$ from being true after some time advance." *We define* $\forall_{\phi_1}(\phi_2)$ *as* $\neg(\exists_{\neg\phi_1}(\neg\phi_2))$, *the dual of* $\exists_{\phi_1}(\phi_2)$.

The definitions of $\exists_{\phi_1}(\phi_2)$ and $\forall_{\phi_1}(\phi_2)$ are subtle. In $\exists_{\phi_1}(\phi_2)$, $\phi_1$ is not required to be true the time instant $\phi_2$ is true, and in $\forall_{\phi_1}(\phi_2)$, both $\phi_1$ and $\phi_2$ must be true at the time instant $\phi_1$ releases $\phi_2$ from being true. To illustrate these subtleties, consider a state $q = (l, x_1 = 0)$ that allows time to diverge; $q$ satisfies $\exists_{x_1<2}(x_1 \geq 2)$, $q$ does not satisfy $\forall_{x_1\geq 2}(x_1 < 2)$, and $q$ satisfies $\exists_{x_1<2}(x_1 \geq 2) \vee \forall(x_1 < 2)$.

We define $\exists_{\phi_1}(\phi_2)$ to have the definition of $\phi_1 \, [\delta\rangle_s \, \phi_2$ in Bouyer et al. [45]. Using the derivation in Bouyer et al. [45], we can derive $\phi_2 \, [\delta\rangle_w \, \phi_1$ of Bouyer et al. [45] as $\exists_{\phi_2}(\phi_1) \vee \forall(\phi_2)$ (note the inverted positions of $\phi_1$ and $\phi_2$ in $\phi_2 \, [\delta\rangle_w \, \phi_1$).

### 4.2.2 $L^{rel}_{\nu,\mu}$ Modal Equation Systems

For shorthand, we often write formulas as a system of equations. We write $X_1 \stackrel{\nu}{=} \phi$ for $\nu X_1.[\phi]$ and $X_1 \stackrel{\mu}{=} \phi$ for $\mu X_1.[\phi]$, putting the outermost fixpoints on top. This form is called a modal equation system (MES). Using the same proof as for the untimed modal mu-calculus (see Bradfield and Stirling [49], Cleaveland and Steffen [57]), the modal equation system form is equally as expressive.

When defining MES for $L^{rel}_{\nu,\mu}$, we leverage the definitions of $L^{rel}_{\nu,\mu}$ in the previous subsection. The following definition of $L^{rel}_{\nu,\mu}$ uses the modal-equation system (MES) format used in Cleaveland and Steffen [57] for untimed systems and in Zhang and Cleaveland [167, 168] for $L_{\nu,\mu}$.

The idea is that we separate $L^{rel}_{\nu,\mu}$ formulas into two components: basic formulas (the formulas without fixpoints) and equation systems. The basic formulas are the operations excluding the fixpoints, and the equations embed the fixpoints.

**Definition 4.2.4** (**$L_{\nu,\mu}$, $L^{rel}_{\nu,\mu}$ basic formula syntax**). Let $CX = \{x_1, x_2, \ldots\}$ and $CX_f = \{z, z_1, \ldots\}$ be disjoint sets of clocks. Then the syntax of a $L_{\nu,\mu}$ *basic formula* is given

by the following grammar:

$$\phi ::= p \mid \neg p \mid \texttt{tt} \mid \texttt{ff} \mid cc \mid Y \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle(\phi)$$
$$\mid [a](\phi) \mid \exists(\phi) \mid \forall(\phi) \mid z.(\phi)$$

Here, $p \in 2^L$ is an atomic proposition, $cc \in \Phi(CX)$ is a clock constraint over clock set $CX$, $Y \in Var$ is a propositional variable ($Var$ is the set of propositional variables), and $a \in \Sigma_{TA}$ is an action. In formula $z.\phi$ the $z.$ operator is often referred to as *freeze quantification*, and each $z$ is a clock in $CX_f$.

The *relativized timed modal-mu calculus* $L^{rel}_{\nu,\mu}$ *syntax* replaces $\exists(\phi)$ and $\forall(\phi)$ with $\exists_{\phi_1}(\phi_2)$ and $\forall_{\phi_1}(\phi_2)$. ■

As with $L^{rel}_{\nu,\mu}$ in the original form, formulas are interpreted with respect to states (i.e. (location, clock valuation) pairs) of a timed automaton and an environment $\theta$.

$L^{rel}_{\nu,\mu}$ MESs are mutually recursive systems of equations whose right-hand sides are basic formulas as specified above. The formal definition of the equation systems follows.

**Definition 4.2.5** ($L^{rel}_{\nu,\mu}$ **MES syntax).** Let $X_1, X_2, \ldots, X_v$ be propositional variables, and let $\phi_1, \ldots \phi_v$ all be $L^{rel}_{\nu,\mu}$ basic formulae. Then *a* $L^{rel}_{\nu,\mu}$ *modal equation system (MES)* is an ordered system of equations as follows, where each equation is labeled with a *parity* ($\mu$ for least fixpoint, $\nu$ for greatest fixpoint): $X_1 \stackrel{\mu/\nu}{=} \phi_1, X_2 \stackrel{\mu/\nu}{=} \phi_2, \ldots, X_v \stackrel{\mu/\nu}{=} \phi_v$.

In our MES, we will assume that all variables are *bound* (every variable in the right of the equation appears as some left-hand variable). ■

When defining MES, we follow the definitions found in Zhang and Cleaveland

[167, 168]; we also leverage the semantics of these formulas. We use the same lattice interpretation as for the inline form. We now treat each equation in the system as a function over this lattice. As with the inline form, each function is monotonic. The parity of the equation is the fixpoint type that we interpret the equation over. Specifically, given MES $X_1 \overset{\mu/\nu}{=} \phi_1, \ldots, X_v \overset{\mu/\nu}{=} \phi_v$, we may construction a function that, given a set of states for $X_1$, returns the set of states satisfying $\phi_1$, where the values for $X_2, \ldots, X_v$ have been computed recursively. This function is monotonic, and therefore has a unique least and greatest fixpoint. If the parity for $X_1$ is $\mu$, then the set of states satisfying $X_1$ is the least fixpoint of this function, while if the parity is $\nu$ then the set of states satisfying $X_1$ is the greatest fixpoint. By convention, the meaning of a MES is the set of states associated with $X_1$, the first left-hand-side in the sequence of equations. However, in the MES, each variable $X_i$ can be interpreted as its own subformula; this interpretation will prove useful constructing proofs that a state satisfies a MES.

In this dissertation, we often consider MESs that are *alternation-free*. Intuitively, an MES is alternation free if there is no mutual recursion involving variables of different parities. For more information on the notion, see Emerson and Lei [69]. We denote the alternation-free fragment of $L^{rel}_{\nu,\mu}$ as $L^{rel,af}_{\nu,\mu}$. This restriction is not prohibitive because for any timelock-free nonzeno timed automaton (see Bowman and Gómez [47]), we can express any TCTL formula into a $L^{rel,af}_{\nu,\mu}$ MES. See Section 4.7 of this dissertation.

## 4.3 Timed Modal-Mu Calculus $T_\mu$

We now define $T_\mu$, the timed modal mu-calculus of Henzinger et al. [88]; see Henzinger et al. [88], Penczek and Pólrola [135] for details.

**Definition 4.3.1** ($T_\mu$ **syntax and semantics**). The *syntax* of a $T_\mu$ formula is con-

structed with the following grammar:

$$\phi ::= p \mid cc \mid Y \mid \phi \wedge \phi \mid \neg(\phi) \mid \phi_1 \rhd \phi_2 \mid z.(\phi) \mid \mu Y.[\phi]$$

Except $\phi_1 \rhd \phi_2$, all syntactic terms are the same as in $L_{\nu,\mu}$ (see Definition 4.2.4). Given the $\neg$, formulas are restricted to be monotonic. The *semantics* of $\rhd$ is:

- $[\![\phi_1 \rhd \phi_2]\!]_{TA,\theta} = \{(l,\nu,\nu_f) \in Q \mid \exists (l',\nu',\nu'_f) \in Q, \exists a \in \Sigma_{TA}$, and $\exists \delta \geq 0$ : $((l,\nu,\nu_f) \xrightarrow{\delta} (l',\nu',\nu'_f)$ or $(l,\nu,\nu_f) \xrightarrow{\delta}\xrightarrow{a} (l',\nu',\nu'_f)) \wedge (l,\nu',\nu'_f) \models \phi_2 \wedge \forall \delta', 0 \leq \delta' \leq \delta : (l,\nu+\delta',\nu_f+\delta') \models \phi_1 \vee \phi_2\}$

■

The intuition of $\phi_1 \rhd \phi_2$ is as follows: after some time advance (possibly zero) and (possibly) a single action, $\phi_2$ is true, and at $\phi_1 \vee \phi_2$ is true for all states up to and including the state $\phi_2$ is satisfied.

## 4.4 Region and Logical Equivalence

To show that one logic is as expressive or more expressive than another logic, we use the notion of logical equivalence.

**Definition 4.4.1 (Logical equivalence ($\phi_1 \equiv_{\mathcal{M}} \phi_2$) and logical expressiveness ($L_1 \subseteq_{\mathcal{M}} L_2$), from Bouyer et al. [45]).** Two formula $\phi_1$ and $\phi_2$ are *logically equivalent* for a set of models $\mathcal{M}$ ($\phi_1 \equiv_{\mathcal{M}} \phi_2$) if and only if for all $M \in \mathcal{M}$, $M \models \phi_1$ iff $M \models \phi_2$. Over a set of models $\mathcal{M}$, a logic $L_1$ is *as least as expressive* as a logic $L_2$ ($L_1 \subseteq_{\mathcal{M}} L_2$) if for all formulas $\phi_1 \in L_1$, there exists a formula $\phi_2 \in L_2$ such that $\phi_1 \equiv_{\mathcal{M}} \phi_2$. We say $L_1 =_{\mathcal{M}} L_2$ iff $L_1 \subseteq_{\mathcal{M}} L_2$ and $L_2 \subseteq_{\mathcal{M}} L_1$. ■

In order to establish logical equivalence over timed automata, we would like

to establish a finite search space. Since logical formulas are commonly interpreted over the lattice of sets of states, our goal is to group the states into a finite set so that the lattice formulas search over is finite To establish a finite lattice, we use region equivalence, which is formally defined in Section 2.5.2. As a summary, region equivalence groups clock valuations into regions where all valuations in the same region enforce the same clock constraints (up to a certain constant *maxc*). Applying the region equivalence relation to a timed automaton creates a new automaton, the region timed automaton. The region automaton is both bisimilar to its timed automaton and *finite*; finite means that the semantics, the transition system, has a finite number of states and transitions, providing us the finite lattice that we desire. In this chapter, we leverage the finiteness of this lattice to establish continuity of monotonic functions over that lattice. This continuity allows us to use the constructive fixpoint semantics, making the search space finite and the proofs executable.

We now define a notion of invariance to a relation to provide a means of showing that the region automaton satisfies the same logical formulas as the original timed automaton.

**Definition 4.4.2** (**Logic invariance**)**.** A logic $L$ is *invariant with respect to equivalence relation* $R$, or $R$ *invariant*, if for all formulas $\phi$ in logic $L$ and for any models $M_1, M_2$ such that $M_1 R M_2$, $M_1 \models \phi \Leftrightarrow M_2 \models \phi$. ■

In this definition, $R$ is a generic relation, and throughout the chapter, we will substitute a relation such as region equivalence for $R$ and we will use terms such as *bisimulation invariant* and *region equivalence invariant* when $R$ is replaced with bisimulation or region equivalence, respectively. For $\mathcal{M}$, we typically use the set of all possible timed automata ($\mathcal{TA}$).

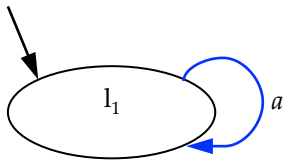


**Figure 4.2:** Timed automaton with $CX = \{x_1\}$ and a coarse bisimulation.

## 4.5 $L^{rel}_{\nu,\mu}$ is Region Equivalence Invariant

**Definition 4.5.1 (Respects $\phi$).** For any formula $\phi$, we say a bisimulation $R$ *respects clock constraints in* $\phi$ iff for all states $q_1, q_2$, if $q_1 R q_2$ then for all clock constraints $cc$ that are subformulae of $\phi$, $q_1 \models cc \Leftrightarrow q_2 \models cc$. Furthermore, a bisimulation $R$ *respects* $\phi$ if $R$ respects clock constraints in $\phi$ and for all atomic propositions $p$, $q_1 R q_2$ implies $q_1 \models p \Leftrightarrow q_2 \models p$. ■

In the above definition, the set of atomic propositions are assumed to represent bisimilar states: i.e. the set of states represented by $p$ in $TA_1$ are the set of states bisimilar to those represented by $p$ in $TA_2$.

Due to subtleties with formulas allowing (possibly unobservable) atomic propositions and clock constraints, many logics are not invariant over all bisimulations. However, this subtlety is handled by using a bisimulation that respects $\phi$.

**Claim 4.5.1.** $L_{\nu,\mu}$, $L^{rel}_{\nu,\mu}$, or any timed logic that allows direct expression of clock constraints using timed automata clocks is not bisimulation invariant.

***Proof of Claim 4.5.1.*** Consider the timed automaton with clock $x_1$ in Figure 4.2. This timed automaton is bisimilar to one with a single state where every action execution and every time advance is a self loop. However, only some of the bisimilar states satisfy $x_1 \leq 2$. □

However, this is a subtlety handled by using a specific bisimulation. If we

assume that the bisimulation respects all clock constraints, then we prove that $L^{rel}_{\nu,\mu}$ is invariant with respect to that bisimulation. Furthermore, while bisimulations also need to respect all atomic propositions (labels for subsets of locations) in a formula, bisimulations are often constructed to have bisimilar states satisfy the same propositions.

**Theorem 4.5.2.** Let $\phi$ be any $L^{rel}_{\nu,\mu}$ formula, and let $TA_1$ and $TA_2$ be timed automata with states $q_1$, $q_2$, respectively, such that $q_1 R q_2$ in a bisimulation that respects $\phi$. Then $q_1 \models \phi$ if and only if $q_2 \models \phi$.

***Proof of Theorem 4.5.2.*** We induct on the size of the formula $\phi$. From Bradfield and Stirling [49], we know that the untimed modal mu-calculus is bisimulation invariant. This handles all cases (including greatest and least fixpoints) except for atomic propositions, clock constraints, and the operators $\exists(\phi), \forall(\phi)$, $\exists_{\phi_1}(\phi_2)$, $\forall_{\phi_1}(\phi_2)$, $\langle - \rangle(\phi)$, $[-](\phi)$, $\langle a \rangle(\phi)$, $[a](\phi)$ and $z.(\phi)$.

The proofs for clock constraints and atomic propositions hold because the bisimulation respects all clock constraints and all atomic propositions. We prove bisimulation invariance for $\exists_{\phi_1}(\phi_2)$; the other operators are similar.

Let $q_1 = (l_1, \nu_1)$, $q_2 = (l_2, \nu_2)$ and $q_1 \sim q_2$. Suppose $q_1 \models \exists_{\phi_1}(\phi_2)$. By definition of $\exists_{\phi_1}(\phi_2), \exists \delta \geq 0 \colon (l_1, \nu_1 + \delta) \models \phi_2 \wedge \forall \delta', 0 \leq \delta' < \delta \colon (l, \nu + \delta') \models \phi_1$. Since $q_1 \sim q_2$, we know $q_2 \xrightarrow{\delta} (l_2, \nu_2 + \delta)$ and $(l_1, \nu_1 + \delta) \sim (l_2, \nu_2 + \delta)$ and by the definition of $\sim$, $(l_1, \nu_1 + \delta') \sim (l_2, \nu_2 + \delta')$. By our inductive hypothesis, we are done, since if $(l_1, \nu_1 + \delta) \models \phi_2$, then $(l_2, \nu_2 + \delta) \models \phi_2$. The same reasoning can be used to show that $\phi_1$ is true for all states $(l, \nu + \delta')$ for all time advances $\delta' < \delta$. (When proving $\phi_1$ holds for all time advances $\delta' < \delta$, we use that $\sim$ respects all clock constraints). If $q_1 \not\models \exists_{\phi_1}(\phi_2)$, the proof that $q_2 \not\models \exists_{\phi_1}(\phi_2)$ is similar. $\square$

Even though region equivalence is a time-abstract bisimulation, because it respects all clock constraints (assuming we pick a large enough maximum constant) and all atomic propositions, we can adapt this theorem to get this corollary:

**Corollary 4.5.1.** Let $\phi$ be an $L^{rel}_{\nu,\mu}$ formula, $TA$ be a timed automaton, $\sim_r$ be the region equivalence relation, and $TA_R$ be its region automaton (formed using a constant large enough such that $\sim_r$ respects $\phi$). For all states $q_1 \in TA, q_2 \in TA_R$, if $q_1 \sim_r q_2$ then $q_1 \models \phi$ if and only if $q_2 \models \phi$.

***Proof of Corollary 4.5.1.*** Similar to the proof of Theorem 4.5.2 with the following adaptation. Whenever there is a time advance $\delta$ in a run on one timed automaton, instead of just advancing $\delta$, use the region equivalence relation to advance a proper amount of time. Selecting this time advance is exactly the same as in the proof of Lemma 44 of Clarke et al. [55]. □

## 4.6 $T_\mu \subseteq_{\mathcal{TA}} L^{rel}_{\nu,\mu}$

Henzinger et al. [88] give a TCTL formula for the $T_\mu$ operator $\phi_1 \rhd \phi_2$. While we could use that TCTL representation and then embed it into $L^{rel}_{\nu,\mu}$ using the $L^{rel}_{\nu,\mu}$ equivalents of TCTL formulas in Section 4.7, by embedding $\phi_1 \rhd \phi_2$ directly into $L^{rel}_{\nu,\mu}$, we can get a simpler expression.

**Claim 4.6.1.** $T_\mu \subseteq_{\mathcal{TA}} L^{rel}_{\nu,\mu}$, using:

$$\phi_1 \rhd \phi_2 \equiv \exists_{\phi_1 \vee \phi_2}(\phi_2 \vee ((\phi_1 \vee \phi_2) \wedge \langle - \rangle(\phi_2))) \tag{4.1}$$

***Proof of Claim 4.6.1.*** Follows from the operators' definitions. □

Likewise, the dual of $\triangleright$ has the analogous $L^{rel}_{\nu,\mu}$ expression $\forall_{\phi_1 \wedge \phi_2}(\phi_2 \wedge ((\phi_1 \wedge \phi_2) \vee [-](\phi_2)))$.

## 4.7 TCTL $\subseteq_{\mathcal{TA}} L^{rel}_{\nu,\mu}$

### 4.7.1 Incorrect Attempts to Show TCTL $\subseteq_{\mathcal{TA}} L^{rel}_{\nu,\mu}$

When trying to write TCTL formulas in $L^{rel}_{\nu,\mu}$, based on the ease of expressing analogous untimed CTL formulas in the modal mu-calculus, the following reasonable $L_{\nu,\mu}$ and $L^{rel}_{\nu,\mu}$ formulas may be suggested (and some have been!):

$$E\,[[\phi_1]\,U\,[\phi_2]] \stackrel{?}{\equiv}_{\mathcal{TA}} Y \stackrel{\mu}{=} \phi_2 \vee (\phi_1 \wedge \exists(\phi_1 \wedge \langle - \rangle(Y)) \tag{4.2}$$

$$E\,[[\phi_1]\,U\,[\phi_2]] \stackrel{?}{\equiv}_{\mathcal{TA}} Y \stackrel{\mu}{=} \phi_2 \vee (\phi_1 \wedge (\exists(Y) \vee \langle - \rangle(Y))) \tag{4.3}$$

$$AF\,[\phi] \stackrel{?}{\equiv}_{\mathcal{TA}} Y \stackrel{\mu}{=} \phi \vee (\forall(Y) \wedge [-](Y)) \tag{4.4}$$

$$EG\,[\phi] \stackrel{?}{\equiv}_{\mathcal{TA}} Y \stackrel{\nu}{=} \exists_{\phi}\Big(\phi \wedge \langle - \rangle(Y)\Big) \vee \forall(\phi) \tag{4.5}$$

$$A\,[[\phi_1]\,W\,[\phi_2]] \stackrel{?}{\equiv}_{\mathcal{TA}} Y \stackrel{\nu}{=} \phi_2 \vee (\phi_1 \wedge (\forall(Y) \wedge [-](Y))) \tag{4.6}$$

$$E\,[[\phi_1]\,W\,[\phi_2]] \stackrel{?}{\equiv}_{\mathcal{TA}} Y \stackrel{\nu}{=} \exists_{\phi_1}(\phi_2 \vee \langle - \rangle(Y)) \tag{4.7}$$

While these attempts are reasonable, *they are all incorrect*.

**Example 4.7.1 (Need to check in-between time advances).** Consider the timed automaton $TA_1$ in Figure 4.3 and the TCTL formula $E\,[[EF_{=0}\,[l_1]]\,U\,[l_2]]$.

To see that the initial state $(l_0, x_1 = 0)$ of $TA_1$ does not satisfy $E\,[[EF_{=0}\,[l_1]]\,U\,[l_2]]$, notice that at the state $(l_0, x_1 = 4.2)$, $EF_{=0}\,[l_1]$ is false; that state cannot immediately transition to location $l_1$. Also, because no state in location $l_0$ can transition to location $l_2$ until $x_1 \geq 5$, every path from location $l_0$ that reaches location $l_2$ must pass through $(l_0, x_1 = 4.2)$.

Now, we show that the initial state of $TA_1$ satisfies the formula written using

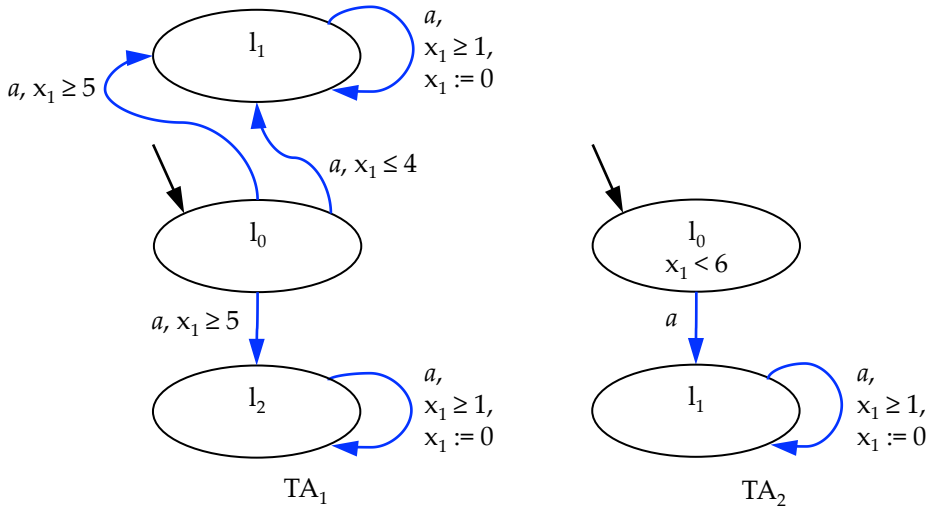


**Figure 4.3:** Timed automata $TA_1$ and $TA_2$.

Equation 4.2. The reasoning to show that the initial state of $TA_1$ satisfies the formula written using Equation 4.3, is similar. The formulation using Equation 4.2, assuming an oracle $L^{rel}_{\nu,\mu}$ formula for $EF_{=0}[l_1]$, is:

$$Y \stackrel{\mu}{=} l_2 \vee (EF_{=0}[l_1] \wedge \exists(EF_{=0}[l_1] \wedge \langle - \rangle(Y)) \tag{4.8}$$

Consider the path prefix $(l_0, x_1 = 0) \xrightarrow{5} (l_0, x_1 = 5) \xrightarrow{a} (l_2, x_1 = 5)$. By construction of Equation 4.8, the premise $EF_{=0}[l_1]$ is only checked at time 0 and time 5, and at those two times,$EF_{=0}[l_1]$ is true. Then, at time 5, the action $a$ is executed and location $l_2$ is reached. As a result, the initial state of $TA_1$ does satisfy Equation 4.8.

*Intuition Summary:* $\phi_1$ must be checked in between the initial time and the final time. ■

**Example 4.7.2 (Need to handle time-convergent circularities).** Now consider the timed automaton $TA_2$ in Figure 4.3 and the TCTL formula $AF[l_1]$.

To see that the initial state $(l_0, x_1 = 0)$ of $TA_2$ satisfies $AF[l_1]$, note that because

of the invariant $x_1 < 6$, all time-divergent paths must leave location $l_0$. Since there is no guard on the edge to location $l_1$, there is a path that can transition to $l_1$. Hence, all time-divergent paths take that edge and eventually transition to location $l_1$.

Now, we show that the initial state of $TA_2$ satisfies the formula written using Equation 4.4. The formulation written using Equation 4.4 is:

$$Y \overset{\mu}{=} l_1 \vee (\forall(Y) \wedge [-](Y)) \tag{4.9}$$

Consider the following time-convergent path: $(l_0, x_1 = 0) \xrightarrow{0} (l_0, x_1 = 0) \xrightarrow{0} (l_0, x_1 = 0) \xrightarrow{0} \ldots$. While not time-divergent, because of the $\forall(Y)$ branch of Equation 4.9, the state $(l_0, x_1 = 0)$ will be visited again. This results in a least fixpoint circularity, which indicates a false path. Since we are conjuncting over a least-fixpoint circularity, this formula returns `ff`.

*Intuition Summary*: the time-convergent execution $\xrightarrow{0}\xrightarrow{0} \ldots$ creates a `ff` circularity in a, $\wedge$ formula.

■

**Example 4.7.3 (Need to exclude time-convergent paths).** Again, consider the timed automaton $TA_2$ in Figure 4.4 but consider the TCTL formula $EG\,[l_0 \wedge x_1 < 6]$.

To see that the initial state of $TA_2$ does not satisfy $EG\,[l_0 \wedge x_1 < 6]$, notice that for time to diverge, any path must leave location 0. Time cannot diverge in location $l_0$.

However, the initial state of $TA_2$ satisfies the formula written using Equation 4.5. Written out, we get the formula:

$$Y \overset{\nu}{=} \exists_{(l_0 \wedge x_1 < 6)}\Big((l_0 \wedge x_1 < 6) \wedge \langle - \rangle(Y)\Big) \vee \forall((l_0 \wedge x_1 < 6)) \tag{4.10}$$

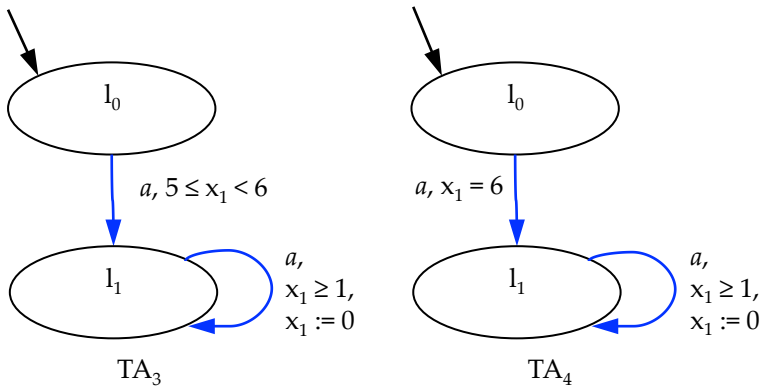


**Figure 4.4:** Timed automata $TA_3$ and $TA_4$.

Consider the time convergent path from the initial state $(l_0, x_1 = 0) \xrightarrow{3} (l_0, x_1 = 3) \xrightarrow{3/2} (l_0, x_1 = 4.5) \xrightarrow{3/4} \ldots$. This path satisfies $\forall(l_0 \wedge x_1 < 6)$, and due to the invariant of $l_0$, the state $(l_0, x_1 = 0)$ satisfies $\forall(l_0 \wedge x_1 < 6)$. Because this path is time-convergent, it should be rejected (but it is not).

*Intuition Summary:* Paths in locations where time cannot advance are considered valid paths, but due to being time-convergent should be rejected. ■

**Example 4.7.4 (Need to check in-between time advances (ii)).** Consider timed automaton $TA_3$ in Figure 4.4 and the TCTL formula $A\,[[x_1 < 6]\,W\,[5 \leq x_1 \leq 6 \vee l_1]]$. Using Equation 4.6, the $L^{rel}_{\nu,\mu}$ formula is:

$$Y \overset{\nu}{=} (5 \leq x_1 \leq 6 \vee l_1) \vee (x_1 < 6 \wedge (\forall(Y) \wedge [-](Y))) \tag{4.11}$$

While $TA_3$ satisfies $A\,[[x_1 < 6]\,W\,[5 \leq x_1 \leq 6 \vee l_1]]$, $TA_3$ does not satisfy Equation 4.11.

*Intuition Summary:* during the transition $\xrightarrow{7}$, after 6 time units the run is released from needed $\phi_1$ or $\phi_2$ to be true. ■

**Example 4.7.5 (Need to check before and after action executions).** Now consider timed automaton $TA_4$ in Figure 4.4 and the TCTL formula $E\,[[AG_{=0}\,[l_0]]\,W\,[l_1]]$. The $L^{rel}_{\nu,\mu}$ formula formed using Equation 4.7 is:

$$Y \stackrel{\nu}{=} \exists_{AG_{=0}[l_0]}(l_1 \vee \langle - \rangle(Y)) \tag{4.12}$$

$TA_4$ does not satisfy $E\,[[AG_{=0}\,[l_0]]\,W\,[l_1]]$. Assuming an oracle $L^{rel}_{\nu,\mu}$ formula for $AG_{=0}\,[l_0]$, $TA_4$ satisfies Equation 4.12.

*Intuition Summary:* $\phi_1$ needs to be checked after the time advance but before the action is executed (the definition of $\exists_{\phi_1}(\phi_2)$ does not check $\phi_1$ at the final time).

■

### 4.7.2 Converting Interval Timing Bounds

The first component of encoding TCTL is to write timing constraints using freeze quantification. Bouyer et al. [44] showed that freeze quantification is strictly more expressive than timing constraints. They did this in the context of timed extensions of LTL: MTL (Metric Temporal Logic), which uses timing interval constraints; and TPTL (Timed Propositional Temporal Logic), which uses freeze quantification. We use their [44] solution to express timing constraints with freeze quantification, which is as follows.

Introduce a freeze quantification clock $z$. Then, given $\bowtie \in \{=, <, \leq, >, \geq\}$ and $c \in \mathbb{Z}^{\geq 0} \cup \{\infty\}$, write:

$$E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]] \equiv_{\mathcal{TA}} z.(E\,[[\phi_1]\,U_{<\infty}\,[(\phi_2 \wedge z \bowtie c)]])$$
$$E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]] \equiv_{\mathcal{TA}} z.(E\,[[\phi_1]\,R_{<\infty}\,[(z \not\bowtie c \vee \phi_2)]])$$

In these formulas, $z \not\bowtie c$ is the negated inequality of $z \bowtie c$. The constraint is

negated before it is put in the formula.

### 4.7.3 Expressing TCTL in $L^{rel}_{\nu,\mu}$

We can embed TCTL into $L^{rel}_{\nu,\mu}$; here $\bowtie \in \{=, <, \leq, >, \geq\}$ and $c \in \mathbb{Z}^{\geq 0} \cup \{\infty\}$. We first provide simpler $L^{rel}_{\nu,\mu}$ formulas with the timelock-free and nonzeno assumptions. We remove these assumptions in Section 4.7.4.

**Theorem 4.7.1.** For any timed automaton $TA$, any of its states $(l, \nu)$, and any TCTL formulas $\phi_1$ and $\phi_2$:

if $TA$ is timelock-free, then

$$(l,\nu) \models E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]] \Leftrightarrow$$

$$(l,\nu,[CX_f := 0]) \models \begin{cases} Y_1 \stackrel{\mu}{=} z.(Y_2) \\ Y_2 \stackrel{\mu}{=} \exists_{\phi_1}\Big((\phi_2 \wedge z \bowtie c) \vee (\phi_1 \wedge \langle - \rangle (Y_2))\Big) \end{cases} \tag{4.13}$$

and if $TA$ is timelock-free and nonzeno, then

$$(l,\nu) \models E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]] \Leftrightarrow$$

$$(l,\nu,[CX_f := 0]) \models \begin{cases} Y_1 \stackrel{\nu}{=} z_1.(Y_2) \\ Y_2 \stackrel{\nu}{=} \exists_{z_1 \not\bowtie c \vee \phi_2}\Big((\phi_1 \wedge (z_1 \not\bowtie c \vee \phi_2)) \vee \\ \qquad ((z_1 \not\bowtie c \vee \phi_2) \wedge \langle - \rangle (Y_2))\Big) \vee \\ \qquad \Big(\forall(z.(\exists(z \geq 1))) \wedge \forall(z_1 \not\bowtie c \vee \phi_2)\Big) \end{cases} \tag{4.14}$$

Here are the $L^{rel}_{\nu,\mu}$ formulas for $E\,[[\phi_1]\,U\,[\phi_2]]$ and $E\,[[\phi_1]\,R\,[\phi_2]]$ without timing constraints:

$$E\,[[\phi_1]\,U\,[\phi_2]] \equiv_{\mathcal{TA}} Y \stackrel{\mu}{=} \exists_{\phi_1}\Big(\phi_2 \vee (\phi_1 \wedge \langle - \rangle(Y))\Big) \tag{4.15}$$

$$E\,[[\phi_1]\,R\,[\phi_2]] \equiv_{\mathcal{TA}} Y \stackrel{\nu}{=} \exists_{\phi_2}\Big((\phi_1 \wedge \phi_2) \vee (\phi_2 \wedge \langle - \rangle(Y_2))\Big) \vee \Big(\forall(z.(\exists(z \geq 1))) \wedge \forall_{\phi_1}(\phi_2)\Big) \tag{4.16}$$

and here are $L^{rel}_{\nu,\mu}$ formulas for some other TCTL operators:

$$A\,[[\phi_1]\,U\,[\phi_2]] \equiv_{\mathcal{TA}} Y \stackrel{\mu}{=} \forall_{\phi_2}\Big((\phi_1 \vee \phi_2) \wedge (\phi_2 \vee [-](Y_2))\Big) \wedge \Big(\exists(z.(\forall(z < 1))) \vee \exists_{\phi_1}(\phi_2)\Big) \tag{4.17}$$

$$A\,[[\phi_1]\,R\,[\phi_2]] \equiv_{\mathcal{TA}} Y \stackrel{\nu}{=} \forall_{\phi_1}\Big(\phi_2 \wedge (\phi_1 \vee [-](Y))\Big) \tag{4.18}$$

$$EF\,[\phi_1] \equiv_{\mathcal{TA}} Y \stackrel{\mu}{=} \exists(\phi_1 \vee \langle - \rangle(Y)) \tag{4.19}$$

$$EG\,[\phi_1] \equiv_{\mathcal{TA}} Y \stackrel{\nu}{=} \exists_{\phi_1}\Big(\phi_1 \wedge \langle - \rangle(Y)\Big) \vee \Big(\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_1)\Big) \tag{4.20}$$

$$AF\,[\phi_1] \equiv_{\mathcal{TA}} Y \stackrel{\mu}{=} \forall_{\phi_1}\Big(\phi_1 \vee [-](Y_2)\Big) \wedge \Big(\exists(z.(\forall(z < 1))) \vee \exists(\phi_1)\Big) \tag{4.21}$$

$$AG\,[\phi_1] \equiv_{\mathcal{TA}} Y \stackrel{\nu}{=} \forall(\phi_1 \wedge [-](Y)) \tag{4.22}$$

To prove our embedding of $E\,[[\phi_1]\,R\,[\phi_2]]$ in $L^{rel}_{\nu,\mu}$, we use the following three lemmas:

**Lemma 4.7.2.** Let $TA$ be a timed automaton and $(l, \nu)$ be one of $TA's$ states. Then $(l, \nu) \xrightarrow{1} \xrightarrow{1} \xrightarrow{1} \ldots$ if and only if $(l, \nu, [CX_f := 0]) \models \forall(z.(\exists(z \geq 1)))$.

The formula $\forall(z.(\exists(z \geq 1)))$ means: "time can elapse in the current location

by an infinite amount."

***Proof of Lemma 4.7.2.*** Time can diverge in $(l, \nu)$ if and only if $(l, \nu)$ has the (infinite) sequence of time advances $(l, \nu) \xrightarrow{1} \xrightarrow{1} \xrightarrow{1}$. This is true if and only if for all $\delta \geq 0$, $(l, \nu + \delta) \xrightarrow{1} (l, \nu + \delta + 1)$, which by definition of timed automata invariants and time advance transitions is true if and only if there is a $\delta' \geq 0$ such that $(l, \nu + \delta) \xrightarrow{1+\delta'} (l, \nu + \delta + 1 + \delta')$. □

**Lemma 4.7.3.** In TCTL, $E\,[[\phi_1]\,R\,[\phi_2]] \equiv_{\mathcal{TA}} E\,[[\phi_2]\,U\,[\phi_1 \wedge \phi_2]] \vee EG\,[\phi_2]$

***Proof of Lemma 4.7.3.*** Similar to the proof in Clarke et al. [55] for the analogous derivation of $E\,[[\phi_1]\,R\,[\phi_2]]$ in CTL. □

**Lemma 4.7.4.** For any timed automaton $TA$, any of its states $(l, \nu)$, and any TCTL formula $\phi_1$: if $TA$ is nonzeno, then

$$\begin{aligned}&(l, \nu) \models EG\,[\phi_1] \Leftrightarrow \\ &(l, \nu, [CX_f := 0]) \models Y \stackrel{\nu}{=} \exists_{\phi_1}\Big(\phi_1 \wedge \langle - \rangle(Y)\Big) \vee \Big(\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_1)\Big)\end{aligned} \tag{4.23}$$

***Proof of Lemma 4.7.4.*** We use the same premises and the same region automaton construction from $TA$ as in Theorem 4.7.1. This proof is inspired by the analogous CTL proof in Clarke et al. [55].

**Proving EG $[\phi_1]$:**

$$EG\,[\phi_1] \equiv Y \stackrel{\nu}{=} \exists_{\phi_1}\Big(\phi_1 \wedge \langle - \rangle(Y)\Big) \vee \Big(\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_1)\Big) \tag{4.24}$$

**Fixpoint:** First, we show that $EG\,[\phi_1]$ is a fixpoint; i.e.

$$EG\,[\phi_1] = \exists_{\phi_1}(\phi_1 \wedge \langle - \rangle (EG\,[\phi_1])) \vee (\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_1)) \tag{4.25}$$

$\Rightarrow \big(EG\,[\phi_1] \subseteq \exists_{\phi_1}(\phi_1 \wedge \langle - \rangle (EG\,[\phi_1])) \vee (\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_1))\big)$: Suppose $(l, \nu) \models EG\,[\phi_1]$. By definition of $EG\,[\phi_1]$, we have a time-divergent path from $(l, \nu)$ where $\phi_1$ is always true, which by definition means: either the path diverges to $\infty$ in location $l$, or the path takes a time advance and an action to a state $(l', \nu')$ that satisfies $EG\,[\phi_1]$. By Lemma 4.7.2, the first path occurs if and only if $(l, \nu) \models (\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_1))$. The second path occurs if and only if $(l, \nu) \models \exists_{\phi_1}(\phi_1 \wedge \langle - \rangle (EG\,[\phi_1]))$.

$\Leftarrow \big(\exists_{\phi_1}(\phi_1 \wedge \langle - \rangle (EG\,[\phi_1])) \vee (\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_1)) \subseteq EG\,[\phi_1]\big)$: similar to $\Rightarrow$.

**Greatest fixpoint:** Now we show that $EG\,[\phi_1]$ is the greatest such fixpoint. To do this, we use the constructive semantics of $\nu$. Defining our fixpoint function $\phi(Y) = \exists_{\phi_1}(\phi_1 \wedge \langle - \rangle (Y)) \vee (\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_1))$, we now show that $EG\,[\phi_1] = \bigcap_{i=0}^{\infty} \phi^i(2^Q)$.

$\Rightarrow \big(EG\,[\phi_1] \subseteq \bigcap_{i=0}^{\infty} \phi^i(2^Q)\big)$: similar to the proof of $(\bigcup_{i=0}^{\infty} \phi^i(\emptyset) \subseteq E\,[[\phi_1]\,U\,[\phi_1]])$.

$\Leftarrow \big(\bigcap_{i=0}^{\infty} \phi^i(2^Q) \subseteq EG\,[\phi_1]\big)$: Suppose $(l, \nu) \in \bigcap_{i=0}^{\infty} \phi^i(2^Q)$. Because our lattice is finite, there exists some iteration $f$ such that $\phi^{f+1}(2^Q = \phi^f(2^Q)$ Because $(l, \nu)$ is in every iteration $i$, $(l, \nu) \in \phi^{f+1}(2^Q)$ and $(l, \nu) \in \phi^f(2^Q)$. By definition of our function $\phi$, since $(l, \nu) \in \phi(\phi^f(2^Q))$, $l_\nu \models \exists_{\phi_1}(\phi_1 \wedge \langle - \rangle (\phi^f(2^Q))) \vee (\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_1))$. Using the definitions of the formulas, this means that $(l, \nu) \models \phi_1$, and there exists a $\delta$ and an action $a$ such that for all $\delta' \leq \delta$, $(l, \nu + \delta') \models \phi_1$ and $(l, \nu + \delta) \xrightarrow{a} (l', \nu')$ and $(l', \nu') \in \phi^f(2^Q)$. Because $f$ is the fixpoint iteration and $\phi$ is monotonic, $(l', \nu') \in \bigcap_{i=0}^{\infty} \phi^i(2^Q)$. Hence, by repeating iterations, we have found an infinite sequence of states starting at $(l, \nu)$ that always satisfies $\phi_1$. Since this

infinite sequence contains an infinite number of actions, by the **nonzeno assumption**, this sequence is a time-divergent path. Hence, $(l, \nu) \models EG\,[\phi_1]$. □

The formulations are subtle, especially for $EG\,[\phi]$ (and hence $E\,[[\phi_1]\,R\,[\phi_2]]$). From the definition of $EG\,[\phi]$, there are two different types of *time-divergent* paths that can be chosen to satisfy $EG\,[\phi]$.

1. The path allows time to diverge in a single location.

2. The path visits some state infinitely often.

Even assuming that the timed automaton is *timelock-free* and *nonzeno*, ensuring that the formula always finds a time-divergent path is subtle.

To detect paths of condition (1), we utilize the formula in Lemma 4.7.2, which determines if there is a time-divergent path that never leaves the current location (not state). With this formula, the partial formula $\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi)$ determines that if such a time-divergent path exists, $\phi$ is always true. While $\forall(\phi)$ may seem sufficient, it considers additional time-convergent paths that stay in that current location. Using this simplification would result in our formula giving a time-convergent path instead of a time-divergent path. Example 4.7.3 in Section 4.7.1 is a counterexample showing that $\forall(\phi)$ is not sufficient for this purpose.

The other disjunct, $\exists_{\phi_1}(\phi_1 \wedge \langle - \rangle(Y))$ covers paths of condition (2). Many of the subtleties that resulted in this formulation requiring the relativization operator are demonstrated and discussed in the previously discussed counterexamples in Section 4.7.1. (These subtleties also apply to $E\,[[\phi_1]\,U\,[\phi_2]]$.)

Now, using the above Lemmas, we prove Theorem 4.7.1.

***Proof of Theorem 4.7.1.*** This proof is inspired by the analogous CTL proof given in Clarke et al. [55]. The correctness for handling the timing constraints $\bowtie c$ is

argued in Section 4.7.2; therefore, we show the correctness for formulas without time bounds.

**Establishing the finite lattice:** Let $TA$ be an arbitrary timed automaton and $\phi_1, \phi_2$ be arbitrary TCTL formulas. Because $L^{rel}_{\nu,\mu}$ is region equivalence invariant, by Corollary 4.5.1, $TA$ and its region automaton $TA_R$ ($TA_R$ is formed using a constant high enough to respect $\bowtie c$, $\phi_1$ and $\phi_2$) either both satisfy the given $L^{rel}_{\nu,\mu}$ formula or they both do not. Since TCTL properties also respect region equivalence (assuming $TA_R$ also respects the formula's constants) [152], showing $TA$ satisfies the TCTL formula if and only if $TA$ satisfies the corresponding $L^{rel}_{\nu,\mu}$ formula is equivalent to showing $TA_R$ satisfies the TCTL formula if and only if $TA_R$ satisfies the corresponding $L^{rel}_{\nu,\mu}$ formula. Because all region timed automata are *finite* [5] and $L^{rel}_{\nu,\mu}$ formulas are monotonic, the lattice is finite and all monotonic functions over the lattice are continuous. Hence, we can use the iterative semantics in Definition 4.2.3 where iterations only need to go up to order $\omega$ (finitely many iterations suffice) to establish the equivalence.

In these proofs, we assume there are no urgent locations in $TA$. We handle urgent locations in Remark 4.7.1.

**Proving $\mathbf{E}\,[[\boldsymbol{\phi_1}]\,\mathbf{U}\,[\boldsymbol{\phi_2}]]$ (Equation 4.15):**

**Fixpoint:** We show that $E\,[[\phi_1]\,U\,[\phi_2]]$ is a fixpoint; i.e.

$$E\,[[\phi_1]\,U\,[\phi_2]] = \exists_{\phi_1}(\phi_2 \vee (\phi_1 \wedge \langle - \rangle (E\,[[\phi_1]\,U\,[\phi_2]]))) \tag{4.26}$$

$\Leftarrow \left(\exists_{\phi_1}(\phi_2 \vee (\phi_1 \wedge \langle - \rangle (E\,[[\phi_1]\,U\,[\phi_2]]))) \subseteq E\,[[\phi_1]\,U\,[\phi_2]]\right)$: Suppose $(l, \nu) \models \exists_{\phi_1}(\phi_2 \vee (\phi_1 \wedge \langle - \rangle (E\,[[\phi_1]\,U\,[\phi_2]])))$. If $(l, \nu) \models \exists_{\phi_1}(\phi_2)$, then by definition this means that there is a time advance $\delta$ such that $(l, \nu + \delta) \models \phi_2$ and for all previous times $\delta' < \delta$, $(l, \nu + \delta') \models \phi_1$. By **the timelock-free assumption**, there is a time-

divergent path from $(l,\nu)$ with this path prefix. Hence, $(l,\nu) \models E\,[[\phi_1]\,U\,[\phi_2]]$. Else, $(l,\nu) \models \exists_{\phi_1}(\phi_1 \wedge \langle - \rangle (E\,[[\phi_1]\,U\,[\phi_2]]))$. Hence, there is some path prefix with a (possibly 0) time-advance $\delta$ and an action $a$ such that for all $\delta' \leq \delta$ $(l,\nu+\delta') \models \phi_1$ and after taking the action $a$, the resulting state satisfies $E\,[[\phi_1]\,U\,[\phi_2]]$. By definition, this path prefix concatenated with the path from the state satisfying $E\,[[\phi_1]\,U\,[\phi_2]]$ satisfies $E\,[[\phi_1]\,U\,[\phi_2]]$.

$\Rightarrow \left(E\,[[\phi_1]\,U\,[\phi_2]] \subseteq \exists_{\phi_1}(\phi_2 \vee (\phi_1 \wedge \langle - \rangle (E\,[[\phi_1]\,U\,[\phi_2]])))\right)$: similar to $\Leftarrow$. (Note: the $\Rightarrow$ direction does not require the timelock-free assumption.)

**Least fixpoint:** Now we show that $E\,[[\phi_1]\,U\,[\phi_2]]$ is the least such fixpoint. To do this, we use the constructive semantics of $\mu$, define $\phi(Y) = \exists_{\phi_1}(\phi_2 \vee (\phi_1 \wedge \langle - \rangle (Y)))$, and show that $E\,[[\phi_1]\,U\,[\phi_2]] = \bigcup_{i=0}^{\infty} \phi^i(\emptyset)$.

$\Leftarrow \left(\bigcup_{i=0}^{\infty} \phi^i \subseteq E\,[[\phi_1]\,U\,[\phi_2]]\right)$: Similar to the analogous CTL proof in Clarke et al. [55].

$\Rightarrow \left(E\,[[\phi_1]\,U\,[\phi_2]] \subseteq \bigcup_{i=0}^{\infty} \phi^i\right)$: Let $(l,\nu) \models E\,[[\phi_1]\,U\,[\phi_2]]$. By definition and the finiteness of region automata, there is some time-divergent path that satisfies $E\,[[\phi_1]\,U\,[\phi_2]]$ and $\phi_2$ is satisfied within a finite number of actions. We induct on the number of these *action executions*.

**Proof by Strong Induction on $k$, the number of actions**.

**Base case:** $k = 0$**.** Proof omitted.

**Strong Induction Hypothesis:** *For all states $(l,\nu)$ satisfying $E\,[[\phi_1]\,U\,[\phi_2]]$ with a path taking $k-1$ or fewer actions to a state satisfying $\phi_2$, $(l,\nu) \in \bigcup_{i=0}^{\infty} \phi^i$.*

**Induction step.** Let $(l,\nu) \models E\,[[\phi_1]\,U\,[\phi_2]]$, and let it have a time-divergent path that satisfies $E\,[[\phi_1]\,U\,[\phi_2]]$ with a prefix of exactly $k$ action steps to the state that satisfies $\phi_2$ (here $k \geq 1$). *By definition of $E\,[[\phi_1]\,U\,[\phi_2]]$*, $(l,\nu)$ satisfies $E\,[[\phi_1]\,U\,[\phi_2]]$ in $k$ steps (actions) iff $(l,\nu) \models \phi_2$ or there is a $\delta$ where $(l,\nu+\delta) \models \phi_2$ and for all $\delta', 0 \leq \delta' < \delta$, $(l,\nu+\delta') \models \phi_1$ or there is a $\delta$ and an action $a$ where $(l,\nu+\delta) \xrightarrow{a}$

$(l', \nu')$ such that $(l', \nu') \models E\,[[\phi_1]\,U\,[\phi_2]]$ in $k-1$ action execution steps and for all $\delta', 0 \leq \delta' \leq \delta$, $(l_i, \nu_i + \delta') \models \phi_1$. *After distributing the wording*, we get: either $(l, \nu) \models \phi_2$; or there exists a $\delta$ where $(l, \nu + \delta) \models \phi_2$ and for all $\delta' < \delta, (l, \nu + \delta') \models \phi_1$); or $(l, \nu + \delta) \xrightarrow{a} (l', \nu')$ such that $(l', \nu') \models E\,[[\phi_1]\,U\,[\phi_2]]$ in $k-1$ or fewer action executions, and for all $\delta', 0 \leq \delta' \leq \delta$, $(l, \nu + \delta') \models \phi_1$. By definition, the first two disjuncts are the subformula $\exists_{\phi_1}(\phi_2)$, and the third disjunct is the subformula $\exists_{\phi_1}(\phi_1 \wedge \langle - \rangle(Y))$. Our formula in Equation 4.13 is the disjunct of these two formulas, which by definition, is the wording above. By **the strong induction hypothesis,** if $(l', \nu') \models E\,[[\phi_1]\,U\,[\phi_2]]$ in $k-1$ or fewer action steps, then $(l', \nu') \in \bigcup_{i=0}^{\infty} \phi^i$.

**Proving E $[[\boldsymbol{\phi_1}]$ R $[\boldsymbol{\phi_2}]]$** (Equation 4.16):

We reduce this formulation to the disjunction of $E\,[[\phi_2]\,U\,[\phi_1 \wedge \phi_2]] \vee EG\,[\phi_2]$ by utilizing logical equivalences. We begin with *using **absorption** ($p \equiv p \vee (p \wedge q)$) to add a least fixpoint variable* $Y_2$ as follows:

$$
\begin{aligned}
Y \overset{\nu}{=} \exists_{\phi_2}\Big((\phi_1 \wedge \phi_2) \vee (\phi_2 \wedge \langle - \rangle(Y)) \vee (\phi_2 \wedge \langle - \rangle(Y_2))\Big) \\
\vee\ (\forall(z.(\exists(z \geq 1))) \wedge \forall_{\phi_1}(\phi_2)) \\
Y_2 \overset{\mu}{=} \exists_{\phi_2}\Big((\phi_1 \wedge \phi_2) \vee (\phi_2 \wedge \langle - \rangle(Y_2))\Big)
\end{aligned}
$$

If we examine $Y_2$ and (for now) ignore the fixpoints on the variables of $Y$ and $Y_2$, we notice that $Y_2$ is a subformula of $Y$. Therefore, if $Y_2$ were a $\nu$ fixpoint, we would be done. By Tarski's theorem [148] and the construction of our lattice, the greatest fixpoint of any formula $\phi$ contains all of the states of the least fixpoint of $\phi$. Hence, $(\phi_2 \wedge \langle - \rangle(Y_2))$ with a least fixpoint is a subformula of $(\phi_2 \wedge \langle - \rangle(Y_2))$ with a greatest fixpoint, which is a subformula of $(\phi_2 \wedge \langle - \rangle(Y))$.

Using additional logical equivalences, we get:

$$
\begin{aligned}
&Y' \stackrel{\mu/\nu}{=} Y \vee Y_2 \\
&Y \stackrel{\nu}{=} \exists_{\phi_2}\Big((\phi_2 \wedge \langle - \rangle(Y))\Big) \vee \Big(\forall(z.(\exists(z \geq 1))) \wedge \forall(\phi_2)\Big) \\
&Y_2 \stackrel{\mu}{=} \exists_{\phi_2}\Big((\phi_1 \wedge \phi_2) \vee (\phi_2 \wedge \langle - \rangle(Y_2))\Big)
\end{aligned}
$$

By the first part of this Theorem and by **the timelock-free assumption**, $Y_2 \equiv_{\mathcal{TA}} E\,[[\phi_2]\,U\,[\phi_1 \wedge \phi_2]]$. By Lemma 4.7.4 and by **the nonzeno assumption**, $Y \equiv_{\mathcal{TA}} EG\,[\phi_2]$. Therefore, $Y' \equiv_{\mathcal{TA}} E\,[[\phi_2]\,U\,[\phi_1 \wedge \phi_2]] \vee EG\,[\phi_1]$. By Lemma 4.7.3, $Y' \equiv_{\mathcal{TA}} E\,[[\phi_1]\,R\,[\phi_2]]$. □

*Remark* 4.7.1 (Handling urgent locations). For urgent locations, there are no time advances. Technically, we must add the subformula inside the $\exists$ (of $E\,[[\phi_1]\,U\,[\phi_2]]$ or $E\,[[\phi_1]\,R\,[\phi_2]]$) as a disjunct outside of the $\exists$. This would result in the formulas becoming the longer formulas below.

$$
\begin{aligned}
E\,[[\phi_1]\,U\,[\phi_2]] \equiv_{\mathcal{TA}} Y \stackrel{\mu}{=} &\Big(\phi_2 \vee (\phi_1 \wedge \langle - \rangle(Y))\Big) \vee \\
&\exists_{\phi_1}\Big(\phi_2 \vee (\phi_1 \wedge \langle - \rangle(Y))\Big) \qquad (4.27) \\
E\,[[\phi_1]\,R\,[\phi_2]] \equiv_{\mathcal{TA}} Y \stackrel{\nu}{=} &\Big((\phi_1 \wedge \phi_2) \vee (\phi_2 \wedge \langle - \rangle(Y_2))\Big) \vee \\
&\exists_{\phi_2}\Big((\phi_1 \wedge \phi_2) \vee (\phi_2 \wedge \langle - \rangle(Y_2))\Big) \vee \\
&\Big(\forall(z.(\exists(z \geq 1))) \wedge \forall_{\phi_1}(\phi_2)\Big) \qquad (4.28)
\end{aligned}
$$

However, since tools can handle the urgency internally by removing the $\exists$ and examining the subformula when encountering an urgent state, such a subtlety is addressed without having to further complicate the formula. Hence, the proof assumes no urgent locations and allows the simpler formulas.

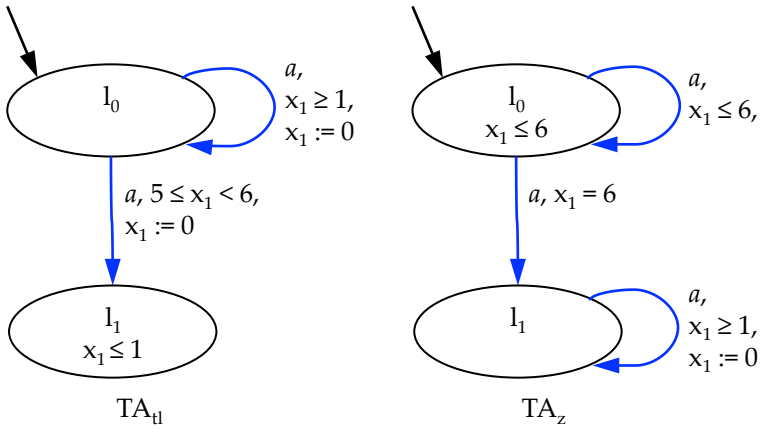


**Figure 4.5:** Timed automaton $TA_{tl}$ with a timelock and zeno timed automaton $TA_z$ with zeno timed runs.

In Theorem 4.7.1, the formula for $E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]$ requires that the timed automaton be timelock-free and the formula for $E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$ requires that the timed automaton be timelock-free and nonzeno. We give examples illustrating that for these formulas, these assumptions are necessary.

**Example 4.7.6 (Necessity of assumptions).** Consider timed automata $TA_{tl}$ and $TA_z$ in Figure 4.5, and consider the following TCTL formulas and their $L^{rel}_{\nu,\mu}$ equivalents from Theorem 4.7.1:

$$E\,[[l_0]\,U\,[l_1]] \equiv_{\mathcal{TA}} Y \overset{\mu}{=} \exists_{l_0}(l_1 \vee (l_0 \wedge \langle - \rangle(Y))) \tag{4.29}$$

$$E\,[[l_1]\,R\,[l_0 \vee x_1 \leq 0]] \equiv_{\mathcal{TA}} Y \overset{\nu}{=} \exists_{l_0 \vee x_1 \leq 0}((l_1 \wedge (l_0 \vee x_1 \leq 0))$$
$$\vee\ ((l_0 \vee x_1 \leq 0) \wedge \langle - \rangle(Y))) \vee (\forall(z.(\exists(z \geq 1))) \wedge \forall(l_0 \vee x_1 \leq 0)) \tag{4.30}$$

$$EG\,[l_0] \equiv_{\mathcal{TA}} Y \overset{\nu}{=} \exists_{l_0}(l_0 \wedge \langle - \rangle(Y)) \vee (\forall(z.(\exists(z \geq 1))) \wedge \forall(l_0))) \tag{4.31}$$

While $TA_{tl}$ does not satisfy $E\,[[l_0]\,U\,[l_1]]$, $TA_{tl}$ satisfies Equation 4.29 with the time locked execution $(l_0, x_1 = 0) \xrightarrow{5} (l_0, x_1 = 5) \xrightarrow{a} (l_1, x_1 = 0) \ldots$. Likewise, while $TA_{tl}$ does not satisfy $E\,[[l_1]\,R\,[l_0 \vee x_1 = 0]]$, $TA_{tl}$ satisfies Equation 4.30 with the

time locked execution $(l_0, x_1 = 0) \xrightarrow{5} (l_0, x_1 = 5) \xrightarrow{a} (l_1, x_1 = 0) \ldots$ (at this point $x_1 = 0$ and $l_1$ are true). $TA_z$ does not satisfy $EG\,[l_0]$, but $TA_z$ satisfies Equation 4.31 with the zeno execution $(l_0, x_1 = 0) \xrightarrow{a} (l_0, x_1 = 0) \xrightarrow{a} (l_0, x_1 = 0) \ldots$ (repeated $a$ actions).

*Intuition Summary:* In all three cases, the only executions satisfying the properties are time-convergent, yet the fixpoint formulas consider these executions valid (circularity does not consider time-divergence of the rest of the path). ■

### 4.7.4 Removing Timelock-free and Nonzeno Assumptions

**Theorem 4.7.5.** TCTL $\subseteq_{\mathcal{TA}} L^{rel}_{\nu,\mu}$.

***Proof of Theorem 4.7.5.*** Theorem 4.7.1 showed how to encode TCTL formulas for timed automata satisfying timelock-free and nonzeno assumptions. Consider the TCTL-like formula $EG\,[z.(F\,[z = 1])]$ from Henzinger et al. [88]. This formula translated to $L^{rel}_{\nu,\mu}$ (using Claim 4.6.1) is

$$\begin{aligned} Y_t &\stackrel{\nu}{=} z.(Y_2) \\ Y_2 &\stackrel{\mu}{=} (z = 1 \wedge Y_t) \vee \exists(Y_2 \vee \langle - \rangle(Y_2)) \end{aligned} \tag{4.32}$$

To remove the need for the timelock-free assumption: replace $\phi_2$ in $E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]$ with $\phi_2 \wedge Y_t$, and replace $\phi_1$ in $E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$ with $\phi_1 \wedge Y_t$.

Here we eliminate the need for the nonzeno assumption for $E\,[[\phi_1]\,R_{\bowtie}\,[\phi_2]]$ ($E\,[[\phi_1]\,U_{\bowtie}\,[\phi_2]]$ does not assume the nonzeno property). Our formula is $Y_A \vee (Y_B \wedge Y_C)$. $Y_A$ is $E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$ with the fixpoint inverted from a $\nu$ to a $\mu$. $Y_B$ is

our formula for $E\,[[\phi_1]\,R_{\bowtie}\,[\phi_2]]$, and $Y_C$ is the MES:

$$\begin{aligned} Y_C &\stackrel{\nu}{=} z.(\phi_2 \wedge Y_{c2}) \\ Y_{c2} &\stackrel{\mu}{=} (\phi_2 \wedge z = 1 \wedge Y_C) \vee \exists(Y_{c2} \vee \langle - \rangle(Y_{c2})) \end{aligned} \tag{4.33}$$

$Y_C$, similar to $EG\,[\phi_2 \wedge z.(F\,[z = 1])]$, means, "there exists a cycle of time advances and actions such that: $\phi_2$ is always true and time can always advance by one unit" (proof omitted).

To sketch the correctness of $Y_A \vee (Y_B \wedge Y_C)$: subformula $Y_A$ is true if and only if $E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$ is true from finding a path using $\phi_1$ to release $\phi_2$, or $\phi_2$ is always true in a path ending in a state where time diverges to $\infty$. If $Y_A$ is true, then this path is non-zeno by construction and we are done ($Y_B$ will also be true). Else, we check our original formula $E\,[[\phi_1]\,R_{\bowtie}\,[\phi_2]]$. Given $Y_A$ is false, $Y_B$ is true if and only if we have a cycle (involving action executions) where $\phi_2$ is always true. If $Y_B$ is false, then the formula is false and we are done. If $Y_B$ is true, then the satisfying path is a cycle (involving action transition) such that $\phi_2$ is always true. Such a nonzeno path or cycle exists if and only if $Y_C$ is true. □

## 4.8 $L_{\nu,\mu} \not\subseteq_{\mathcal{TA}} TCTL$ and $TCTL \not\subseteq_{\mathcal{TA}} L_{\nu,\mu}$

### 4.8.1 Expressive power of fixpoints: $L_{\nu,\mu} \not\subseteq_{\mathcal{TA}} TCTL$

**Claim 4.8.1.** $L_{\nu,\mu} \not\subseteq_{\mathcal{TA}} TCTL$, because over timed automata, TCTL cannot express the $L_{\nu,\mu}$ formula:

$$\begin{aligned} Y_1 &\stackrel{\nu}{=} z.(Y_2) \\ Y_2 &\stackrel{\nu}{=} z \leq 0 \wedge \forall(z \leq 0 \wedge [-](Y_2)) \end{aligned} \tag{4.34}$$

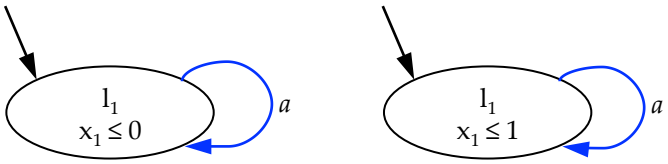


**Figure 4.6:** The left timed automaton with invariant $x_1 \geq 0$ does not allow time to advance; the right timed automaton with invariant $x_1 \geq 1$ does.

***Proof of Claim 4.8.1.*** The formula in Claim 4.8.1 means "for all executions, time never advances." Consider the two timed automata in Figure 4.6. The left automaton satisfies the formula, but the right automaton does not. Since there are no time-divergent paths, TCTL can only ask about the initial state $(l_1, x_1 = 0)$, which is the same for both automata. □

There is another non-constructive way to show that $TCTL \not\subseteq_{\mathcal{TA}} L_{\nu,\mu}$ that follows from specification complexity results in Aceto and Laroussinie [2]. The specification complexity of the model-checking problem is the complexity of model-checking any formula on the single-location *nil* timed automaton: a timed automaton with one location $l_0$ with the invariant `tt` and no edges. There can be an arbitrary number of clocks. The following claim is proven in Aceto and Laroussinie [2].

**Claim 4.8.2** (Specification complexity results [2])**.** The model-checking problem for $L_\nu$ over*nil* timed automata is PSPACE-complete. The model-checking problem for $L_{\nu,\mu}$ over *nil* timed automata is EXPTIME-complete.

Hence, even over an extremely simplified space of models, model-checking $L_{\nu,\mu}$ formulas is still EXPTIME-complete. The complexity arises from the power of freeze-quantification with the time -advance modal operators. Without fixpoints, these alone can be used to ask whether exact integer time-unit time advances

happened. Somewhat surprisingly, as shown in Aceto and Laroussinie [2], these formulas with these transitions can be used to encode the a quantified boolean formula, which is a PSPACE-complete problem. Allowing fixpoints makes the formulas even more powerful.

While model-checking a $L_{\nu,\mu}$ formula over the *nil* automaton is EXPTIME-complete, the problem is much simpler for TCTL formulas without freeze quantifications. This yields a *non-constructive proof* of the above claim, expressed in the following corollary:

**Corollary 4.8.1.** $L_{\nu,\mu} \not\subseteq TCTL$

***Proof of Corollary 4.8.1.*** Without action transitions, there is only one path: waiting. Since TCTL only involves path operators involving location constraints, any location constraint is either true or false, regardless of the amount of time spent. Hence, over these *nil* automata, waiting neither makes a property true or false. Therefore, model-checking TCTL over *nil* automata is in $P$. Since there is only one path, the path operator $E$ or $A$ can be ignored. Since waiting cannot help, $[[\phi_1]\, U\, [\phi_2]]$ is equivalent to $\phi_2$ regardless of the timing operator.

By Claim 4.8.2 [2], the model-checking problem for $L_{\nu,\mu}$ over *nil* automata is EXPTIME-complete. Since $P \neq EXPTIME$ by the Deterministic Time Hierarchy Theorem (see Arora and Barak [16]), we know that the model-checking problem over *nil* automata for $L_{\nu,\mu}$ is harder than for $TCTL$ formulas. Hence, there exists a formula that $L_{\nu,\mu}$ can say that TCTL cannot. □

### 4.8.2 Necessity of Relativization for TCTL: TCTL $\not\subseteq_{\mathcal{TA}} L_{\nu,\mu}$

**Claim 4.8.3.** $L_{\nu,\mu}$ cannot express $\forall_{\langle b\rangle(\mathtt{tt})}([a](\mathtt{ff}))$; therefore, $L_{\nu,\mu} \not=_{\mathcal{TA}} L^{rel}_{\nu,\mu}$.

***Proof Sketch of Claim 4.8.3.*** Bouyer et al. [45], show that $L_\nu$ cannot express the

formula $\forall_{\langle b \rangle(\mathtt{tt})}([a](\mathtt{ff}))$. We adapted their proof to cover least fixpoints by strengthening *Gluing Everything* of Bouyer et al. [45]. See Section 4.9 for details. □

**Theorem 4.8.4.** There exist $L_{\nu,\mu}$ formulas $\phi_1$ and $\phi_2$ such that $L_{\nu,\mu}$ cannot express $A\,[[\phi_1]\,R\,[\phi_2]]$. Hence, TCTL $\not\subseteq_{\mathcal{TA}} L_{\nu,\mu}$.

***Proof of Theorem 4.8.4.*** Suppose not. Then consider $\phi_{act}$, defined as $\neg\, l_0 \vee (z > 0 \wedge \bigvee_{x_i \in CX}(x_i \leq 0))$, and the formula: $z.(A\,[[\phi_1 \vee \phi_{act}]\,R\,[\phi_2 \vee \phi_{act}]])$.

$(l_0, \nu_0, [CX_f := 0]) \models z.(A\,[[\phi_1 \vee \phi_{act})]\,R\,[\phi_2 \vee \phi_{act}]])$ iff $(l_0, \nu_0, [CX_f := 0]) \models \forall_{\phi_1}(\phi_2)$. This contradicts Claim 4.8.3. Since this formula can be used for all timed automata (details omitted), TCTL $\not\subseteq_{\mathcal{TA}} L_{\nu,\mu}$. □

## 4.9 Proving $L_{\nu,\mu} \neq_{\mathcal{TA}} L^{rel}_{\nu,\mu}$

### 4.9.1 Summary of Previous Work

Bouyer et al. [40, 45] proved that $L^{rel}_{\nu} \neq L_{\nu}$ by showing that one cannot write $\forall_{\langle b \rangle(\mathtt{tt})}([a](\mathtt{ff}))$ in $L_{\nu}$. We adapt this proof. In order to show which parts need to be adapted when least fixpoints are added, we summarize the proof given in Bouyer et al. [40, 45], following the version presented in Bouyer et al. [40]. In Section 4.9.2 we prove the components of this proof that need to be proven when least fixpoints are added.

The proof of Bouyer et al. [40] begins by constructing two families of timed automata. Any automaton in the former satisfies $\forall_{\langle b \rangle(\mathtt{tt})}([a](\mathtt{ff}))$, but no automaton in the latter family does. Note: these automata uses rational constants in the automata's clock constraints.

They then suppose that there is a $L_{\nu}$ formula $\phi$ that is equivalent to the formula

$\forall_{\langle b \rangle(\mathtt{tt})}([a](\mathtt{ff}))$ over timed automata. As a result, it is equivalent for the subclass of the two families of timed automata.

Given the particular families of automata, through a series of lemmas and arguments (the proofs of the lemmas are mostly omitted), they argue that there must be a formula $\Psi \equiv \phi$ for this class of automata, where $\Psi$ is constructed from the following *simplified* grammar:

$$\Psi ::= p \mid \mathtt{tt} \mid \mathtt{ff} \mid [a](\mathtt{ff}) \mid \langle a \rangle(\mathtt{tt}) \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \mathsf{G}^+\Psi \mid \mathsf{F}^+\Psi \mid Y$$

In this grammar, $p$ is a proposition, $Y$ is an equation variable, $\mathsf{F}^+\Psi$ means $\Psi$ is true in some time successor, and $\mathsf{G}^+\Psi$ means $\Psi$ is true in any time successor. The argument in Bouyer et al. [40] does not involve circularity or fixpoints.

Since the previous argument uses region equivalence and successor regions to simplify the language, $\mathsf{F}^+$ and $\mathsf{G}^+$ are operators nearly identical in function to $\exists()$ and $\forall()$, respectively. Note that any region is a time successor of itself. Note that $[a](\mathtt{tt}) \equiv \mathtt{tt}$ and $\langle a \rangle(\mathtt{ff}) \equiv \mathtt{ff}$.

At this point, no part of the proof relied on circularity or fixpoints, and can be used without adaptation when least fixpoints are added to the logic.

The next part, *Gluing Everything* from Bouyer et al. [40], is part of the proof of Bouyer et al. [40] that handles the fixpoints and needs to be adapted for least fixpoints.

[Copied verbatim (sic) from Bouyer et al. [40].]

***Gluing everything.*** The formula $\Psi$ can be written in normal form as a system of equations $(X_i = f_i(X_1, ..., X_n))_{1 \leq i \leq n}$ and $\Psi = X_1$. We assume that each formula $f_i(X_1, ..., X_n)$ is a boolean combination of subformulas $\alpha_i^j$ (which can be either some formula $\mathsf{F}^+\beta_i^j$, or $\mathsf{G}^+\beta_i^j$, or some atomic-like formula $\langle a \rangle(\mathtt{tt})$, $[a](\mathtt{ff})$, $\mathtt{tt}$ or

`ff`, or some fix-point variable $X_i^j$):

$$\begin{cases} X_1 =_\nu b_1(\alpha_1^1, \ldots, \alpha_1^{k_1}) \\ \quad \vdots \\ X_n =_\nu b_n(\alpha_n^1, \ldots, \alpha_n^{k_1}) \end{cases}$$

Without loss of generality, we assume that no subformula $\alpha_i^j$ is a fix-point variable. The following lemma justifies this fact:

**Lemma 7.** *We assume that $\alpha_i^j = X_k$ (with $i \neq k$). Then the new formula obtained by replacing $X_k$ by its definition formula is equivalent to the previous formula. If $\alpha_i^j = X_i$, then the new formula obtained by replacing this variable $X_i$ by `tt` is equivalent to the initial formula.*

Thus, each $\alpha_i^j$ is either an atomic proposition, or its negation, or a formula $\mathsf{F}^+\phi$ or a formula $\mathsf{G}\phi$.
[End of quote of Bouyer et al. [40].]

This is the portion of the proof of Bouyer et al. [40] that needs to be adopted and proven when least fixpoints are added. Since this lemma eliminates the circularity by simplifying the logic to one without circularity, The remainder of the proof of Bouyer et al. [40] does not contain circularity and can be adopted without change. After this adaptation, we know that $L_{\nu,\mu} \not\equiv_{\mathcal{TA}} L^{rel}_{\nu,\mu}$.

### 4.9.2 Adaptation of Proof

To handle least fixpoints (and alternations that result), we strengthen and prove the *Gluing Everything* and Lemma 7 of Bouyer et al. [40]. We strengthen the Lemma in order for it to be applied without change in the remainder of the proof of Bouyer et al. [40].

First, by arguing that the $\beta_i^j$ subformulas do not have fixpoint variables in them, we make the lemma stronger. Given Lemma 8 in the paper (it is not proven in Bouyer et al. [40]), we think that the condition we argue above is argued by them. While it may not be, for us to use Lemma 8 without adopting the proof, we must argue this condition.

As a result, the Lemma is changed. When we substitute circularity, we substitute `tt` when we see a $\nu$ variable $X_i$, and `ff` when we see a $\mu$ variable $X_i$.

We use the following replacement algorithm:

**When $X_k$ has parity $\nu$ (greatest fixpoint):**

- $\alpha_i^j = X_i \equiv \texttt{tt}$.
- $\beta_i^j = \mathsf{G}^+\phi \equiv \mathsf{G}^+\phi'$, and $\phi' = \phi[X_i \mapsto RHS(X_i)[X_i \mapsto \texttt{tt}]]$, which is $\phi$ after each $X_i$ is replaced with its right hand side equation after substituting `tt` where $X_i$ occurs in its own formula.
- $\beta_i^j = \mathsf{F}^+\phi \equiv \mathsf{F}^+\phi'$, and $\phi' = \phi[X_i \mapsto \texttt{tt}]$, which is $\phi$ with every instance of $X_i$ substituted with `tt`, replacing $X_i$ with its formula.

**When $X_k$ has parity $\mu$ (least fixpoint):**

- $\alpha_i^j = X_i \equiv \texttt{ff}$.
- $\beta_i^j = \mathsf{G}^+\phi \equiv \mathsf{G}^+\phi'$, and $\phi' = \phi[X_i \mapsto \texttt{ff}]$, which is $\phi$ with every instance of $X_i$ substituted with `ff`, replacing $X_i$ with its formula.
- $\beta_i^j = \mathsf{F}^+\phi \equiv \mathsf{F}^+\phi'$, and $\phi' = \phi[X_i \mapsto RHS(X_i)[X_i \mapsto \texttt{ff}]]$, which is $\phi$ after each $X_i$ is replaced with its right hand side equation after substituting `ff` where $X_i$ occurs in its own formula.

We first replace the $\alpha$ subformulas and then replace the $\beta$ subformulas. The lemma we now get is:

**Lemma 4.9.1.** Suppose $\alpha_i^j = X_k$, $\beta_i^j = \mathsf{G}^+\phi$, or $\beta_i^j = \mathsf{F}^+\phi$ (with $i \neq k$ and $X_k$ within $\phi$). Then the new formula obtained by replacing each instance of $X_k$ with its definition formula is equivalent to the previous formula. If $\alpha_i^j = X_i$, $\beta_i^j = \mathsf{G}^+\phi$, or $\beta_i^j = \mathsf{F}^+\phi$ where $X_i$ is within $\phi$, then the new formula obtained by the algorithm is equivalent to the initial formula.

***Proof of Lemma 4.9.1.*** We argue each case for each subformula in the above algorithm.

**First, we suppose $X_i$ is a greatest fixpoint $\nu$ variable.**

- **Case 1: $\alpha_i^j = X_i$.** Since $X_i$ is outside of a time modality, if we execute it again we will reach $X_i$ and obtain a circularity. For a $\nu$ variable, a circularity means `tt`.

- **Case 2: $\beta_i^j = \mathsf{G}^+\phi$.** By reaching $X_i$ in the $\mathsf{G}^+\phi$ formula, we consider all successor states in the subformula $X_i$. Thus, we must unravel the formula once. If we do not reach $X_i$, then circularity is not used. That means that the unraveling of $X_i$ correctly solved the formula. If we do reach $X_i$, then we have a $\nu$ circularity, which results in $X_i$ being `tt`.

- **Case 3: $\beta_i^j = \mathsf{F}^+\phi$.** Since $\mathsf{F}^+$ always allows a transition to the same region (the same set of states), for all subformula we can choose the 0-unit advance and return to the same formula. Since all $\alpha$ terms were already substituted for, by definition we will return to the $X_i$ within the $\beta_i^j$ subformula and obtain a $\nu$ circularity.

**Now we suppose $X_i$ is a least fixpoint $\mu$ variable.**

- **Case 1b: $\alpha_i^j = X_i$.** Similar to **Case 1**. Since $X_i$ is outside of a time modality, if we execute it again we will reach $X_i$ and obtain a circularity. For a $\mu$ variable, a circularity means `ff`.

- **Case 2b:** $\beta_i^j = \mathsf{G}^+\phi$**.** Similar to **Case 3**. Since $\mathsf{G}^+$ includes a transition to the same region (the same set of states), for all subformula we examine the 0-unit advance and return to the same formula. Since all $\alpha$ terms were already substituted for, by definition we will return to the $X_i$ within the $\beta_i^j$ subformula and obtain a $\mu$ circularity.

- **Case 3b:** $\beta_i^j = \mathsf{F}^+\phi$**.** Similar to **Case 2**. The change in the formula is that with the $\exists$, if we reach a circularity if means that that trace is $\mathtt{ff}$. Since any finite sequence of time elapses can be simulated by a single time advance, we only need to unroll once.

Without alternations,we can start at the head variable $X_1$ (assumed to be the head variable without loss of generality) and unroll the formula. Since the algorithm renders circularity without expressive power, even with alternations, the circularity has no additional expressive power. Hence, the variables can be replaced (details omitted). □

There are two keys to the correctness of this lemma:

1. *The logic is simplified and has no* $[a](Y)$ *or* $\langle a\rangle(Y)$. Furthermore, it is impossible for a variable $Y$ to be inside an action modality. $Y$ can only appear outside time and action modalities or inside a time modality.

2. *These time modalities equate to a sequence of time advances.* Due to time additivity where $\xrightarrow{\delta_1} + \xrightarrow{\delta_2} \approx \xrightarrow{\delta_1+\delta_2}$, whenever a sequence wishes to perform two time elapses of $\delta_1$ followed by $\delta_2$ units, it can equivalently perform a time elapse of $\delta_1 + \delta_2$ units. Because the operators are not relativized, we are not concerned with what happens in-between the advances.

With Lemma 4.9.1, the $L_{\nu,\mu}$ formula now has no circularity and is in the form of the equivalent $L_\nu$ formula. The remainder of the proof of Bouyer et al. [40] goes

through. Hence, $L_{\nu,\mu} \not\equiv_{\mathcal{TA}} L^{rel}_{\nu,\mu}$.

## 4.10 Additional Expressivity Results

Here we give additional expressivity results we have shown involving TCTL, $T_\mu$, $L_{\nu,\mu}$ and $L^{rel}_{\nu,\mu}$.

### 4.10.1 Set of Next States

The key idea is that while timed automata do not have a "next state", we can give a definition of a *set of next states* by looking at the concrete transition systems that represents the semantics.

**Theorem 4.10.1.** Let $TA$ be a timed automaton and let $TS(TA)$ be the transition system representing the semantics of $TA$, and let $(l, \nu)$ be a state of $TA$. Then we have the assertion of the timed modal equation system $(l, \nu) \models X \overset{\mu/\nu}{=} \forall(\phi) \wedge [-](\phi)$ with respect to $TA$ if and only we have the assertion of the untimed modal equation system $(l, \nu) \models X \overset{\mu/\nu}{=} [-](\phi)$ with respect to the transition system $TS(TA)$.

***Proof of Theorem 4.10.1.***

$$(l, \nu) \models X = [-](\phi) \text{ in } TS(TA) \Leftrightarrow$$
$$\forall \sigma \in \Sigma, (l, \nu) \xrightarrow{\sigma} (l', \nu') \colon (l', \nu') \models X \overset{\mu/\nu}{=} \phi \Leftrightarrow$$

(Definition of $[-](\phi)$ for untimed MES)

$$\left(\forall \delta \in \mathbb{R}^{\geq 0}, (l,\nu) \xrightarrow{\delta} (l',\nu') \colon (l',\nu') \models X \overset{\mu/\nu}{=} \phi\right) \wedge$$
$$\left(\forall a \in \Sigma_{TA}, (l,\nu) \xrightarrow{a} (l',\nu') \colon (l',\nu') \models X \overset{\mu/\nu}{=} \phi\right) \Leftrightarrow$$

(Enumeration over $\sigma$ in $TS(TA)$, distributivity of $\wedge$)

$$(l,\nu) \models X = \forall(\phi) \wedge [-](\phi) \text{ in } TA$$

(Definition of $\forall(\phi)$ and $[-](\phi)$ in timed MES) □

**Theorem 4.10.2.** Let $TA$ be a timed automaton and let $TS(TA)$ be the transition system representing the semantics of $TA$, and let $(l,\nu)$ be a state of $TA$. Then we have the assertion of the timed modal equation system $(l,\nu) \models X \overset{\mu/\nu}{=} \exists(\phi) \vee \langle - \rangle(\phi)$ with respect to $TA$ if and only we have the assertion of the untimed modal equation system $(l,\nu) \models X \overset{\mu/\nu}{=} \langle - \rangle(\phi)$ with respect to the transition system $TS(TA)$.

***Proof of Theorem 4.10.2.***

$$(l,\nu) \models X = [-](\phi) \text{ in } TS(TA) \Leftrightarrow$$
$$\exists \sigma \in \Sigma, (l,\nu) \xrightarrow{\sigma} (l',\nu') \colon (l',\nu') \models X \overset{\mu/\nu}{=} \phi \Leftrightarrow$$

(Definition of $\langle - \rangle(\phi)$ for untimed MES)

$$\left(\exists \delta \in \mathbb{R}^{\geq 0}, (l,\nu) \xrightarrow{\delta} (l',\nu') \colon (l',\nu') \models X \overset{\mu/\nu}{=} \phi\right) \vee$$
$$\left(\exists a \in \Sigma_{TA}, (l,\nu) \xrightarrow{a} (l',\nu') \colon (l',\nu') \models X \overset{\mu/\nu}{=} \phi\right) \Leftrightarrow$$

(Enumeration over $\sigma$ in $TS(TA)$, distributivity of $\vee$)

$$(l,\nu) \models X = \exists(\phi) \wedge \langle - \rangle(\phi) \text{ in } TA$$

(Definition of $\exists(\phi)$ and $\langle - \rangle(\phi)$ in timed MES) □

From Theorem 4.10.1 and Theorem 4.10.2, we now can emulate $[-](\phi)$ and $\langle - \rangle(\phi)$ on $TS(TA)$, thus covering "for all next actions" or "there is some next action." With this, the timed version thus covers the "set of states after one action." Keep in mind that using these subformulas will cover *all* executions, including time-convergent paths. Thus, when expressing TCTL, we are careful so that these potential time-convergent executions are excluded.

### 4.10.2 Detecting and Bypassing Timelocks and Actionlocks

As motivated in Section 2.4, there are three kinds of situations that are undesirable: timelocks, actionlocks and zeno executions. We discuss how to handle timelocks and actionlocks, both to detect them as well as to bypass them when model checking formulas.

Using an alternation, we can detect timelock freedom by using the formula (taken from Henzinger et al. [88]):

$$\nu X.[z.(EF\,[z = 1 \wedge X])] \equiv \nu X.[z.(\mu Y.[(z = 1 \wedge X) \vee (\mathtt{tt} \rhd Y)])]$$

This translated to $L^{rel}_{\nu,\mu}$ is

$$Y_1 \stackrel{\nu}{=} t.(Y_2)$$
$$Y_2 \stackrel{\mu}{=} (t = 1 \wedge Y_1) \vee \exists(Y_2 \vee \langle - \rangle(Y_2))$$

Notice that this formula has an alternation, and that it is similar to the formula used to eliminate the timelock assumption in Section 4.7.4.

Any state that satisfies the above formula is timelock free. To remove the timelock assumption, in $E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]$ , replace $\phi_2$ with "timelock-free" $\wedge\,\phi_2$. For $E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$, replace $\phi_1$ with "timelock-free" $\wedge\,\phi_1$ (notice that with correcting for timelocks, duality is no longer preserved).

More simply and without alternation, we can detect actionlocks. The formula

$$X \stackrel{\mu/\nu}{=} \forall([-](\texttt{ff})) \tag{4.35}$$

returns the set of action-locked states, and $X \stackrel{\mu/\nu}{=} \exists(\langle - \rangle(\texttt{tt}))$ returns the set of states that are not action-locked. The equation $X \stackrel{\mu}{=} \forall([-](X))$ returns the set of states that can reach an action-locked state. To ignore actionlocks for the formulas $E\,[[\phi_1]\,U_{\bowtie_c}\,[\phi_2]]$, and for $E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$, replace $\phi_2$ with "not action-locked" $\wedge\,\phi_2$.

## 4.11 Summary of Established Expressiveness Results

We summarize the logical equivalences established in this chapter in Table 4.1.

## 4.12 Dissertation Contributions

### 4.12.1 Contributions

These are my contributions discussed in this chapter:

**Table 4.1:** Summary of logical equivalences and expressiveness results.

| **Expressiveness Result** | **Assumptions** | **Comments** |
|---|---|---|
| $T_\mu \subseteq L^{rel}_{\nu,\mu}$ | none | Utilize the definition of $\triangleright$ of $T_\mu$ to write the $\triangleright$ operator using $L^{rel}_{\nu,\mu}$ operators. No additional fixpoint variables are needed for the conversion. |
| $TCTL \subseteq L^{rel,af}_{\nu,\mu}$ | timed automata are nonzeno and timelock-free | Utilize region automata to establish a finite search space, and then extend the result that $CTL$ is a subset of the alternation-free untimed mu-calculus. |
| $TCTL \subseteq L^{rel}_{\nu,\mu}$ | none | Utilize alternations to write the constraints for nonzeno and timelock-free. |
| $L^{af}_{\nu,\mu} \not\subseteq TCTL$ | none | Give a $L^{af}_{\nu,\mu}$ formula that cannot be written in $TCTL$. |
| $L^{rel,af}_{\nu,\mu} \not\subseteq L^{af}_{\nu,\mu}$ | none | Adapt the proof of Bouyer et al. [40, 45] to handle least fixpoints. |
| $L^{rel}_{\nu,\mu} \not\subseteq L_{\nu,\mu}$ | none | Adapt the proof of Bouyer et al. [40, 45] to handle least fixpoints and alternations. |
| $TCTL \not\subseteq L_{\nu,\mu}$ | none | Utilizing that $L^{rel}_{\nu,\mu} \not\subseteq L_{\nu,\mu}$ Show that if we can write the $TCTL$ formula $A\,[[\phi_1]\,R\,[\phi_2]]$ in $L_{\nu,\mu}$ then we can write the relativization operators of $L^{rel}_{\nu,\mu}$ in $L_{\nu,\mu}$. |
| $L_\nu \subseteq L^{af}_{\nu,\mu}$ | none | Follows from definitions of fixpoints. |

- With a common assumption regarding atomic propositions, we show that $L^{rel}_{\nu,\mu}$, $L_{\nu,\mu}$ and $T_\mu$ are bisimulation invariant. Additionally, for the region equivalence relation, we show that $L^{rel}_{\nu,\mu}$ is invariant.

- We show $T_\mu \subseteq L^{rel}_{\nu,\mu}$ Furthermore, we show this result without requiring additional fixpoints, thus keeping the complexity simple.

- We show $TCTL \subseteq L^{rel}_{\nu,\mu}$ For $E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]$ we assume the timelock-free assuption and for $E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$ (and its dual, $A\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]$), we assume both a timelock-free assumption and a nonzeno assumption.

We then show, using a formula from Henzinger et al. [88] that with an alternation, we can both detect timelocks as well as bypass timelocked states, thus removing the need for the timelock-free assumption.

- We show $L_{\nu,\mu} \not\subseteq TCTL$.
- We show $TCTL \not\subseteq L_{\nu,\mu}$, showing that expressing all of TCTL requires the additional power of the relativization.
- We give a way of writing the *set of next states* of $TS(TA)$ in $L_{\nu,\mu}$.

### 4.12.2 Future Work

Future work includes answering some of the unanswered expressivity questions of various timed logics:

- Can we detect zeno executions in $L^{rel}_{\nu,\mu}$? Can we write formulas to bypass zeno executions?
- Is $L^{rel}_{\nu,\mu} \subseteq T_\mu$? We conjuecture no, but are not sure.
- Is $TPTL \subseteq L^{rel}_{\nu,\mu}$? Since TPTL is MTL with freeze quantification instead of timing intervals, determining if $TPTL \subseteq L^{rel}_{\nu,\mu}$ is similar to determining if $MTL \subseteq L^{rel}_{\nu,\mu}$.
- Is $L^{rel}_{\nu,\mu} \subseteq TPTL$? What about $L_{\nu,\mu}$ and $T_\mu$?

Answers to these items will allow us to better decide if $L^{rel}_{\nu,\mu}$ can verify the formulas we want or if we have to leverage properties of additional logics when verifying $L^{rel}_{\nu,\mu}$ formulas.

# Chapter 5

# Model Checking $\mathsf{L}^{\mathsf{rel}}_{\nu,\mu}$ with Predicate Equation Systems (PES)

Now that we have formalized a time automata model (Chapter 3) and shown that $L^{rel,af}_{\nu,\mu}$ can express properties we want to express (Chapter 4), we now wish to implement a tool that can model check $L^{rel,af}_{\nu,\mu}$ formula over timed automata. To implement this verifier, we take the model checking framework using Predicate Equation Systems (PES) from Zhang [165], Zhang and Cleaveland [167, 168] and build upon it. First, we provide an implementation implementation for all the proof rules in Zhang [165], Zhang and Cleaveland [167, 168], resulting in a timed automata model checker supporting all of $L^{af}_{\nu,\mu}$. To verify additional properties, we develop sound and complete proof rules for the relativization operators, completing the proof-rule theory for $L^{rel,af}_{\nu,\mu}$. When we implement these proof rules, we use select derived proof rules to improve the performance. We discuss some other implementation details, the most substantial being the choice of data structure to store and encode clock values in.

With these implementations we give two preliminary evaluations: the first compares our PES tool to UPPAAL, the most widely-used timed automata model checker. We compare our tool to theirs on the properties both support and then give a sampling of properties that UPPAAL cannot verify (the PES tool can verify any property that UPPAAL can). The second evaluation compares the different implementations of the clock data structures within the PES tool to each other.

## 5.1 Predicate Equation Systems

**Predicate Equation System (PES)** are a first-order logic invented independently by Groote and Willemse [79] (as parameterized boolean equation systems) and by Zhang and Cleaveland [167, 168]. PES have the advantage of being general enough to represent a variety of systems including timed automata [167], parametric timed automata [168] and presburger systems [166], yet efficient enough to check formulas in practice [165, 167, 168]. Currently sound and complete rules are available for $L_{\nu,\mu}$ only and the system was only implemented to check safety properties.

Predicate Equation Systems are in modal equation form (see Section 4.2.2 for more information on modal equation systems). PES use a fixpoint logic more general than, but similar to $L_{\nu,\mu}$. It differs from a $L_{\nu,\mu}$ formula in that unlike a logic, it encodes both the **property and the system**. Hence, a PES represents a timed automaton and an $L_{\nu,\mu}$ formula. A PES is defined to be true if and only if the initial states of the timed automaton satisfy the $L_{\nu,\mu}$ formula. For the formal PES syntax and semantics, see Zhang [165], Zhang and Cleaveland [167, 168].

This dissertation only utilize them for timed automatas, but PES have been used to model check timed automata. Since our tool handles the special case of using the PES framework to encode a timed automata and an MES, we do not describe PES in detail. However, the model checking algorithm is extremely similar.

## 5.2 Model-Checking Algorithm

We discuss the model-checking algorithm for timed automata and MES. We begin by describing the PES algorithm that was provided to us from Zhang [165], Zhang and Cleaveland [167, 168]. This briefly describes the PES framework and how the PES system can encode the formula and the timed automaton in one PES.

Then, we describe how we specialize the framework to model check timed automata and $L^{af}_{\nu,\mu}$ formulas. By specializing, we can leverage specific optimizations to improve the performance of the tool.

### 5.2.1 PES Model Checking Algorithm

We begin with the predicate equation systems to model-check $L^{rel}_{\nu,\mu}$ formulas for timed automata. In this case, the model checking algorithm takes a timed automaton (with variables) and an alternation-free $L^{rel}_{\nu,\mu}$ formula and converts it to a PES.

Model checking $L^{rel}_{\nu,\mu}$ formulas on timed automata with PES is as follows:

1. We take a timed automaton $TA$ and an $L^{af}_{\nu,\mu}$ formula and convert it to a predicate equation system.

2. We now ask if we have a true predicate equation system. The PES starts with the initial state and using proof rules, asks if it has a true PES.

3. From the theory of fixpoints we can correctly terminate when we find a circularity. Additionally, we use caching to efficiently and correctly utilize previously proven sequents as well as to speed up circularity detection.

This process is equivalent to taking the timed automaton, starting from the initial state and applying proof rules on the alternation-free $L^{af}_{\nu,\mu}$ formula using fixpoint knowledge to correctly terminate when reaching circularity. Given the complexity of $L_{\nu,\mu}$ we restrict model checking to the alternation-free fragment, $L^{af}_{\nu,\mu}$.

### 5.2.2 Conversion to PES

Here is the overview of the conversion:

1. For each location $l$ in the timed automaton and each variable $Y$ in the $L_{\nu,\mu}$ formula, we will make a variable $V_{l,Y}$ and an equation with that of the left hand variable.

2. With these larger variables, we construct the PES environment $\theta$ using this larger set of variables.

3. We embed the timed automaton into the PES by converting its states to proposition variable assignments, using substitution in the PES to account for transitions.

4. We then convert each $L_{\nu,\mu}$ formula into the PES inductively. We enumerate the possible transitions (since the $TA$ is presumed to be finite) between locations to eliminate the $[-]()$, $\langle - \rangle()$, $[a]()$ and $\langle a \rangle()$ operators.

For initial model-checking, as is done in Zhang and Cleaveland [167], the conversion is done prior to model checking. Hence, all of the transitions are enumerated and put on the stack when dealing with the $[-]()$, $\langle - \rangle()$, $[a]()$ and $\langle a \rangle()$ operators. One way we plan to improve performance is when model-checking is to convert the PES on the fly (or equivalently use $L^{rel}_{\nu,\mu}$ proof rules similar to the PES rules) and then (conservatively) eliminate many transitions. For instance, when making a transition, we only care about the edges coming from the location we are leaving. Not including other transitions will shrink the PES.

Conversion details for $L_{\nu,\mu}$ formulas and timed automata are in Zhang and Cleaveland [167, 168].

*Remark* 5.2.1 (PES equation variables vs. locations). Zhang and Cleaveland [167, 168] give one PES variable per location. Alternatively, we could convert the timed automaton to a PES using propositions for variable assignments and thus have fewer predicate equation variables. However, both are equivalent.

Therefore, in the conversion process, we can either give one predicate variable per (location, $L_{\nu,\mu}$ variable) pair or we can give one predicate variable per $L_{\nu,\mu}$ variable and use more location variables to separate the location. It is not known which conversion leads to faster model checking in practice.

### 5.2.3 Timed Automata Model Checker: Adaptations from PES Tool

The proof rule framework in Zhang and Cleaveland [167, 168] used the general framework called Predicate Equation Systems (PES). PES involved fixpoint equations over first-order predicates and used the proof-search to establish the validity of a PES. For practical reasons, however, one generally wishes to avoid the construction of the PES explicitly; this dissertation adopts this point of view, and the proof rules that it presents involve explicit mention of timed-automata notions, including location and edge. Hence, rather than converting a timed automaton and an $L^{af}_{\nu,\mu}$ formula to a PES and then checking if the PES is true, with minimal changes we adapt the proof rules to be for $L^{af}_{\nu,\mu}$ over timed automata. Consequently, all proof rules are written in the timed automata framework.

The adaptation is as follows:

1. We encode locations as proposition variables rather than equation variables, as described in Remark 5.2.1.

2. Given the definitions of PES and $L^{af}_{\nu,\mu}$ formulas, the conversion is mostly a syntactic change. The first semantic change is that the operators $[-](\phi)$ and $\langle - \rangle(\phi)$ are encoded by enumerating over all the transitions. The PES handles transitions through assignments of proposition variables and assignments of clock variables to 0.

3. We represent the freeze operator $z.(\phi)$ as $\phi[z := 0]$, the reset operator.

4. While $\forall$ and $\exists$ in PES are general operators, in the MES framework they only encode time advances. Hence, we replace the variable substitution with the advancement of time, grouping times together for improved performance.

5. The fixpoint logic for PES and MES is identical, so no change to the fixpoint logic is necessary.

This adaptation allows for the following enhancements:

- We can directly encode $[-](\phi)$ and $\langle - \rangle(\phi)$ as formula operators rather than having to encode the enumeration of the transitions in the PES conversion.

- This allows us to separate the model description and the formula description, because the transitions can now be provided outside of the equation. This also lets the tool optimize the model checking for transitions.

- This also allows us to separate out comparisons from the invariant and the guards from the PES formula. As a result, the invariants are separated from the PES formula and their model-checking is accounted for during time advances.

## 5.3 The Proof-Based Approach and Proof Rules

We describe the proof-based model checking approach that we use and extend to model check timed automata against $L^{rel,af}_{\nu,\mu}$ specifications. The $L^{rel,af}_{\nu,\mu}$ model-checking problem for timed automata may be specified as follows: given timed automaton $TA = (L, l_0, \Sigma_{TA}, CX, I, E)$, atomic-proposition interpretation function $Lab$, and $L^{rel,af}_{\nu,\mu}$ formula $\phi$ with initial environment $\theta$, determine if the initial state of $TA$ satisfies $\phi$, i.e.: is $(l, \nu) \in [\![\phi]\!]_{TA,\theta}$?

To answer this question posed in the notation of denotational semantics, we aim to construct a proof. The proof consists of proof-rules, or sequences of *judg-*

*ments* (i.e. the statements that can be proved using the proof system). Judgments are *sequents* of the form $(l, cc) \vdash \phi$, where $l \in L$ is a location, $cc \in \Phi(CX \cup CX_f)$ is a clock constraint, and $\phi$ is a $L^{rel,af}_{\nu,\mu}$ formula. Note that $cc$ includes clocks from the timed automaton as well as any clocks used in freeze quantification (such as $z$). A clock constraint $cc$ can also be viewed as the set of valuations $cc = \{\nu \mid \nu \models cc\}$; likewise, we can encode a valuation $\nu$ as the clock constraint $cc_\nu = x_1 = \nu(x_1) \wedge \ldots \wedge x_n = \nu(x_n)$ (we also intersect the values of the clocks used in freeze quantification). A proof rule is one or more premise sequents and a conclusion sequent. In this dissertation we will write a proof rule as:

$$\frac{\text{Premise 1} \quad \ldots \quad \text{Premise } n}{\text{Conclusion}} \ (\textit{Rule Name})$$

Each premise, and the conclusion, is a judgment; the intended reading of the rule is that if each premise is valid, then so is the conclusion. Some proof rules, *leaves*, have no premises and only require a check on the sequent. The verifier then builds a *proof* by chaining these proof rules together. A proof is *valid* if the proof rules are applied properly, meaning that the premise of the previous rule is the conclusion of the next rule. The proof rules are designed to be sound and complete, meaning: $(l, \nu) \in [\![\phi]\!]_{TA,\theta}$ if and only if there is a valid proof for $(l, cc_\nu) \vdash \phi$. Notice that a set of sound and complete proof rules can form a specification similar to an operational semantics definition. For performance reasons, the proof rules involve small steps, similar to a small-step operational semantic definition. Because the proof rules involve small steps, this involves the prover to explore the state machine locally and "on-the-fly" as the prover verifies the formula.

Sound and complete PES proof rules for the alternation-free $L_{\nu,\mu}$ calculus are developed in Zhang and Cleaveland [167, 168]. However, only those for safety properties were implemented. Furthermore, there are currently no proof rules for the relativized operators of $L^{rel}_{\nu,\mu}$. We take the proof-rule framework of Zhang and

Cleaveland [167, 168] and adapt it for timed automata and $L^{rel,af}_{\nu,\mu}$ MES.

## 5.4 $L^{af}_{\nu,\mu}$ Proof Rules

This section contains the proof rules for $L^{af}_{\nu,\mu}$, which are adapted from Zhang and Cleaveland [167, 168]; the proof rules for the relativization operators are in Section 5.5.1.

Several comments are in order.

1. Each rule is intended to relate a conclusion sequent involving a formula with a specific outermost operator to premise sequents involving the maximal subformula(e) of this formula. The name of the rule is based on this operator.

2. The premises also involve the use of functions *succ* and *pred*. Intuitively, $succ((l, cc))$ represents all states that are time successors of any state whose location component is $l$ and whose clock valuation satisfies $cc$; $pred((l, cc))$ is the time predecessors of these same states. These operators may be computed symbolically; that is, for any $(l, cc)$ there is a $cc'$ such that $(l, cc')$ is equivalent to $succ((l, cc))$.

3. Some of the rules involve *placeholders*, which are (potentially) unions of clock constraints, given as (subscripted versions of) $\rho$. Given a specific placeholder, the premise sequent $(l, cc), \phi$ is semantically equivalent to $(l, cc \wedge \rho)$; however, for notational and implementation ease, the placeholder $\rho$ is tracked separately from the clock constraint $cc$. Rules such as $\vee_c$ involve new clock constraints in the premises that are not present in the conclusion. Placeholders represent these new clock constraints.

The complete set of $L^{af}_{\nu,\mu}$ proof rules is given in Figures 5.1 and 5.2. Figure 5.1 contains the adaptations of the rules without placeholders, and Figure 5.2 contains

$$\frac{\text{Premise } 1 \quad \ldots \quad \text{Premise } n}{\text{Conclusion}}\ (\textit{Rule Template}) \qquad \frac{}{(l,cc) \vdash p}\ (p),\ p \in Lab(l)$$

$$\frac{}{(l,cc) \vdash cc'}\ (cc'),\ cc \models cc' \qquad \frac{(l,cc) \vdash \phi_i}{(l,cc) \vdash X_i}\ (p),\ X_i \overset{\nu/\mu}{=} \phi_i \qquad \frac{}{(l,\texttt{ff}) \vdash \phi}\ (\text{Empty})$$

$$\frac{(l,cc) \vdash \phi_1}{(l,cc) \vdash \phi_1 \vee \phi_2}\ (\vee_l) \qquad \frac{(l,cc) \vdash \phi_2}{(l,cc) \vdash \phi_1 \vee \phi_2}\ (\vee_r) \qquad \frac{(l,cc) \vdash \phi_1 \qquad (l,cc) \vdash \phi_2}{(l,cc) \vdash \phi_1 \wedge \phi_2}\ (\wedge)$$

$$\frac{(l,cc), \rho_s \vdash \phi_1 \qquad (l,cc), \neg\rho_s \vdash \phi_2}{(l,cc) \vdash \phi_1 \vee \phi_2}\ (\vee_c) \qquad \frac{succ((l,cc)) \vdash \phi}{(l,cc) \vdash \forall(\phi)}\ (\forall_{t1})$$

$$\frac{succ((l,cc)), \rho_s \vdash \phi \qquad (l,cc) \vdash pred(\rho_s)}{(l,cc) \vdash \exists(\phi)}\ (\exists_{t1})$$

$$\frac{(l, post(cc, \lambda := 0)) \vdash \phi}{(l,cc) \vdash \phi[\lambda := 0]}\ ([])$$

$$\frac{(l_1, cc \wedge g_1) \vdash \phi[\lambda_1 := 0] \qquad \ldots \qquad (l_n, cc \wedge g_n) \vdash \phi[\lambda_n := 0]}{(l,cc) \vdash [a](\phi)}\ ([a]_{Act}),\ \text{cond}[a]$$

$\text{cond}[a]$: $\bigcup_i \{(g_i, \lambda_i, l_i)\} = \{(l', g', \lambda') \mid (l, a, g', \lambda', l') \in E\}$

$$\frac{(l,cc) \vdash [a_1](\phi) \qquad \ldots \qquad ((l,cc) \vdash [a_n](\phi)}{(l,cc) \vdash [-](\phi)}\ ([-]_{Act}),\ \Sigma = \{a_1, \ldots, a_n\}$$

$$\frac{(l_n, cc \wedge g) \vdash \phi[\lambda := 0]}{(l,cc) \vdash \langle a \rangle(\phi)}\ (\langle a \rangle_{Act}),\ (l, a, g, \lambda, l') \in E,\ cc \wedge g \text{ is satisfiable}$$

$$\frac{((l,cc) \vdash \langle a_i \rangle(\phi)}{(l,cc) \vdash \langle - \rangle(\phi)}\ (\langle - \rangle_{Act}),\ a_i \in \Sigma$$

**Figure 5.1:** Proof rules (without placeholders) adapted for timed automata and MES.

the adaptations of the rules involving placeholders. Conditions on the proof rules are given after the rule label; the rule labels are in (), and the conditions are outside parentheses.

A note about clock resets (substations of clocks to 0 in $\phi$ or $\rho$). The formal

$$\frac{succ((l,cc)),\rho_s \vdash \phi \qquad succ((l,cc),\rho_\forall) \vdash succ((l,cc)) \wedge \rho_s}{(l,cc),\rho_\forall \vdash \forall(\phi)}\,(\forall_{t2})$$

$$\frac{succ((l,cc)),\rho_s \vdash \phi \qquad \rho_\exists \vdash pred(\rho_s)}{(l,cc),\rho_\exists \vdash \exists(\phi)}\,(\exists_{t2})$$

$$\frac{(l,post(cc,\lambda := 0)),\rho_s \vdash \phi \qquad \rho_{[]} \vdash \rho_s[\lambda := 0]}{(l,cc),\rho_{[]} \vdash \phi[\lambda := 0]}\,([]_p)$$

**Figure 5.2:** Proof rules (involving placeholders) adapted for timed automata and MES.

definition uses the *post* operator from Zhang and Cleaveland [167, 168], defined as:

$$post(cc,\lambda := e) \stackrel{def}{=} \exists v\colon \big(\lambda = (e[\lambda := v]) \wedge cc[\lambda := v]\big) \tag{5.1}$$

In the special case of resetting clocks to 0, computing post results in one of two cases:

1. If the original clock constraint $cc$ is unsatisfiable, $post((l,cc),\lambda := 0)$ produces $(l,cc')$ where $cc'$ is logically equivalent to `ff`.

2. Otherwise, $cc$ is a satisfiable clock constraint, and $post((l,cc),\lambda := 0)$ becomes $(l,reset(cc,\lambda := 0))$, where $reset(cc,\lambda := 0)$ is the clock zone reset operator given in Bengtsson and Yi [27].

More comment on placeholders ($\rho$) is in order. Placeholders encode a set of clock valuations that will make a sequent valid; in practice, we are interested in computing the largest such set. To understand their use in practice, consider the operator $\exists$. To check $\exists$, we need to find some time advance $\delta$ such that $\psi$ is satisfied after $\delta$ time units. Rather than non-deterministically guessing $\delta$, we use a placeholder $\phi$ in the left premise in rule $\exists_{t1}$ to encode all the time valuations

that ensure satisfaction of $\phi$. The right premise then checks that the placeholder $\rho_s$ is some $\delta$-unit time elapse from $(l, cc)$. The placeholder allows us to delay the non-deterministic guess of the value of $\rho_s$ until it is no longer required to guess. Additionally, for performance reasons, we use *new placeholders* to handle time advance operators for sequents with placeholders. An example may be found in Rule $\exists_{t2}$, where a new placeholder $\rho_\exists$ is introduced in the right premise. While useful for performance, this choice results in subtle implementation complexities, which we discuss in Section 5.6.2.

Given judgments and proof rules, proofs now may be constructed in a goal-directed fashion. For a given judgment to be proved, rules whose conclusion matches the form of the judgment, yielding as subgoals the corresponding premises of the rule. These subgoals may then recursively be proved, with subgoals involving given placeholders selected first for proof so that they may be solved for as the proof progresses. If a sequent may be proved using a rule with no premises, then the proof is complete; similarly, if a sequent is encountered a second time (because of loops in the timed automaton and recursion in an MES), then the second occurrence is also a leaf. Details may be found in Zhang and Cleaveland [167, 168]. If the recurrent leaf involves an MES variable with parity $\mu$, then the leaf is unsuccessful; if it involves a variable with parity $\nu$, it is successful. A proof is valid if all its leaves are successful.

**Example 5.4.1 (A simple proof).** To illustrate the proof rules, consider the timed automaton in Figure 5.3. Suppose we wish to prove the sequent $(2\colon \text{in}, x_1 \leq 3) \vdash [exit](0\colon \text{far})$. Utilizing the proof rule $[a]_{Act}$ in Figure 5.1, we get the proof:

$$\frac{\overline{(0\colon \text{far}, 1 \leq x_1 \leq 3) \vdash 0\colon \text{far}}}{(2\colon \text{in}, x_1 \leq 3) \vdash [exit](0\colon \text{far})}$$

Following this proof rule, we intersect the clock constraint with the guard $x_1 \geq 1$

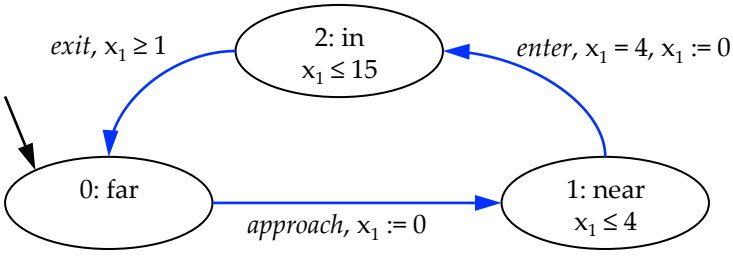


**Figure 5.3:** A timed automaton of the Alur-Dill model. This is the same figure as Figure 1.1 in Chapter 1.

(if $x_1 < 1$, then there are no possible actions so the formula is true) make the destination location the new sequent, and ask if the destination state satisfies the formula. Since the location is 0: far, we have a terminal proof rule and have constructed a proof. ■

## 5.5 Extended Tool: Verifying $L^{rel,af}_{\nu,\mu}$

Our first enhancement of the tool CWB-RT CWB-RT, from Zhang and Cleaveland [167] is the implementation of the placeholder proof rules for $L^{af}_{\nu,\mu}$. The provided tool converted timed automaton and safety properties of $L_{\nu,\mu}$ into a PES and model checked the PES. Proof Rules for the $\exists(\phi)$ operator was not implemented, as well as any placeholder rule.

Additionally, we develop the theoretical proof rules for the relativization operators $\exists_{\phi_1}(\phi_2)$ and $\forall_{\phi_1}(\phi_2)$ as well as implement them. This results in an implementation for a timed automata model checker that can verify all of $L^{rel,af}_{\nu,\mu}$.

### 5.5.1 New $L^{rel,af}_{\nu,\mu}$ Proof Rules

We now introduce the new rules for handling the relativized time-passage modalities in $L^{rel,af}_{\nu,\mu}$. We provide the proof rules for the relativized operator $\exists_{\phi_1}(\phi_2)$ in

$$\frac{(l,cc),\rho_{s_1} \vdash \phi_1 \qquad (l,cc),\rho_{s_2} \vdash \phi_2 \qquad (l,cc) \vdash \rho_{s_1} \vee \rho_{s_2}}{(l,cc) \vdash \phi_1 \vee \phi_2}\ (\vee_s)$$

$$\frac{(l,cc),\rho_{s_1} \vdash \phi_1 \qquad (l,cc),\rho_{s_2} \vdash \phi_2 \qquad (l,cc),\rho_{\vee} \vdash \rho_{s_1} \vee \rho_{s_2}}{(l,cc),\rho_{\vee} \vdash \phi_1 \vee \phi_2}\ (\vee_{s2})$$

$$\frac{succ((l,cc)),\rho_s \vdash \phi_2 \qquad succ((l,cc)),pred_<(\rho_s) \vdash \phi_1 \qquad (l,cc) \vdash pred(\rho_s)}{(l,cc) \vdash \exists_{\phi_1}(\phi_2)}\ (\exists_{r1})$$

$$\frac{\begin{array}{c} \\ succ((l,cc)),\rho_{s'} \vdash \phi_2 \end{array} \qquad \begin{array}{c} succ((l,cc)),pred_<(\rho_{s'}) \vdash \phi_1 \\ (l,cc),\rho_s \vdash pred(\rho_{s'}) \end{array}}{(l,cc),\rho_s \vdash \exists_{\phi_1}(\phi_2)}\ (\exists_{r2})$$

**Figure 5.4:** Proof Rules for $\vee$ and $\exists_{\phi_1}(\phi_2)$.

Figure 5.4. To model check the $\forall_{\phi_1}(\phi_2)$ operator, we use the derivation given in Lemma 5.5.1. We discuss optimized proof rules for $\forall_{\phi_1}(\phi_2)$ in Section 5.5.4. Figure 5.4 also contains an alternative proof rule for $\vee$: rule $\vee_s$. In order to format the proof rules, we sometimes stack premises without any proof lines, such as for rule $\exists_{r2}$. The consequent has three premises, two of which are stacked instead of placed horizontally in order to diagram the rule within the page margins.

Here is an explanation of the proof rule $\exists_{r1}$; the proof rule $\exists_{r2}$ is similar. The idea is for the placeholder $\rho_s$ to encode the $\delta$ time advance needed for $\phi_1$ to be true. The proof-rule premises enforce that this placeholder has three properties:

1. *Left premise:* This premise checks that after the time advance taken by $\rho_s$, $\phi_2$ is satisfied.

2. *Middle premise:* This premise checks that until all $\delta$ time-units have elapsed, that $\phi_1$ is indeed true. The $pred_<(\rho_s)$ encodes the times before $\rho_s$.

3. *Right premise:* This premise checks that $\rho_s$ encodes some range of time elapses $\delta$, ensuring that the state can elapse to valuations in $\rho_s$.

To implement this rule, we implement the premises in left-to-right order. Some

subtleties involving the middle premise are discussed in Section 5.6.2.

By proving some lemmas, we prove the correctness of these new proof rules. This first lemma is a corrected version of a similar lemma in Bouyer et al. [45].

**Lemma 5.5.1.** $\forall_{\phi_1}(\phi_2)$ is logically equivalent to $\forall(\phi_2) \vee \exists_{\phi_2}(\phi_1 \wedge \phi_2)$.

***Proof of Lemma 5.5.1.*** We prove both directions of the lemma. Let $TA$ be an arbitrary timed automaton and let $(l, \nu, \nu_f)$ be an arbitrary (extended) state in $TA$.

First, suppose that $(l, \nu, \nu_f) \models \forall_{\phi_1}(\phi_2)$. By definition, $\forall \delta \geq 0 : \big((l, \nu + \delta, \nu_f + \delta) \models \phi_2 \vee \exists \delta', 0 \leq \delta' < \delta : (l, \nu + \delta', \nu_f + \delta') \models \phi_1\big)$. Notice that the entire quantification is inside the $\forall\delta$. Now for each time instance, one of the two disjunctions inside the $\forall$ is true. We split the cases into two cases:

**Case 1:** The $\phi_2$ disjunct is always true.

Then the formula reduces to $\forall \delta \geq 0 : \big((l, \nu + \delta, \nu_f + \delta) \models \phi_2$. By definition, this means that $(l, \nu, \nu_f) \models \forall(\phi_2)$.

**Case 2:** The $\exists$ disjunct is satisfied for at least one such $\delta$.

While $\mathbb{R}^{\geq 0}$ is not well-ordered with respect to $\leq$, we utilize that there are a finite number of constraints involving a finite number of (possibly not integer) constants. We can then group all $\delta$ values into groups of consecutive values based on whether the group of $\delta$ values satisfy $\phi_1$, $\phi_2$, both, or neither. By construction of $\phi_1$ and $\phi_2$, the finest-grained groups are clock regions (the same regions used in region equivalence) and each time advance $\delta$ appears in exactly one group. Hence, we have a finite number of groups of $\delta$, which are well ordered. (We can well order the groups by the largest $\delta$ in each group. Also note that $\delta \in \mathbb{R}^{\geq 0}$.)

Let $\delta_s$ be a time advance in the smallest such group. Since the disjunct is satisfied at time $\delta_s$, there is some time $\delta_p < \delta_s, (l, \nu + \delta_p, \nu_f + \delta_p) \models \phi_1$. Let $\delta_p$ be

the time when $\phi_1$ is satisfied (there may be many of these, but for this definition, we do not care.) Likewise, since $\delta_s$ is the in smallest such $\delta$ group, we know that all smaller $\delta$ (smaller than the group of $\delta$ values that $\delta_s$ is in) satisfy the left disjunct, meaning that $\forall\delta \geq 0\colon (\delta'' < \delta_s)(l, \nu + \delta'', \nu_f + \delta'') \models \phi_2$. Since $\delta_p < \delta_s$ (and $\delta_p$ is smaller than any delta in the group of $\delta_s$), we have $(l, \nu + \delta_p, \nu_f + \delta_p) \models \phi_1 \wedge \phi_2 \wedge \forall\delta \geq 0\colon (\delta' < \delta_p)(l, \nu + \delta', \nu_f + \delta') \models \phi_2$. Using $\delta_p$ as the chosen delta for $\exists$, we have that $(l, \nu, \nu_f) \models \exists_{\phi_2}(\phi_1 \wedge \phi_2)$.

Now we suppose that $(l, \nu, \nu_f) \models \forall\delta \geq 0\colon (\phi_2) \vee \exists_{\phi_2}(\phi_1 \wedge \phi_2)$. This direction is similar to the previous direction. We break the case on which disjunct is satisfied.

**Case 1:** $(l, \nu, \nu_f) \models \forall(\phi_2)$.

By definition, $(l, \nu, \nu_f) \models \forall_{\phi_1}(\phi_2)$. (The above case is the special case with $\phi_1 = \texttt{ff}$, which is harder to satisfy.)

**Case 2:** $(l, \nu, \nu_f) \models \exists_{\phi_2}(\phi_1 \wedge \phi_2)$.

Let $\delta_e$ be the chosen time when $\phi_1 \wedge \phi_2$ is true. Now we handle all time advances $\delta$. For all time advances $\delta > \delta_e$. Since $\phi_1$ is true, $\forall_{\phi_1}(\phi_2)$ is satisfied for those times. For all times $\delta < \delta_e$, since $\phi_2$ is true, $\forall_{\phi_1}(\phi_2)$ is satisfied for those times. When $\delta = \delta_e$, we need $\phi_2$ to be true, which it is.

Hence, for all time advances $\delta$, the definition of $\forall_{\phi_1}(\phi_2)$ is satisfied. Hence $(l, \nu, \nu_f) \models \forall_{\phi_1}(\phi_2)$. □

When concerning a system of proof rules, we wish to prove them sound and complete. A proof system is *sound* if every proof is correct (this is the if direction of the theorem). A proof system is *complete* if every true formula has some proof of correctness (this is the only if direction of the theorem).

**Theorem 5.5.2 (Soundness and Completeness).** The additional $L^{rel,af}_{\nu,\mu}$ proof rules are sound and complete. I.e. for any timed automaton state $(l, \nu)$ and any $L^{rel,af}_{\nu,\mu}$ $\phi$, it is the case that $(l, \nu) \models \phi$ if and only if $(l, cc_\nu) \vdash \phi$.

***Proof of Theorem 5.5.2.*** We use the soundness and completeness proofs of rules in Zhang and Cleaveland [167, 168]. Hence, we only need to prove the correctness of the proof rules we provided in this paper.We now prove the soundness and completeness of the $\exists_{\phi_1}(\phi_2)$ and the $\vee_s$. proof rules.

First, we start with the proof rule for $\vee_s$.

**Soundness:** Suppose this rule is true. Then $\phi_{s_1}$ acts as the placeholder $\phi_s$. Given that $z_\infty \subseteq (\phi_{s_1} \vee \phi_{s_2})$, we know that $\neg\phi_s \subseteq \phi_{s_2}$, since $\neg\phi_s = z_\infty - \phi_s$ by definition of complement.

**Completeness:** Suppose the conclusion is indeed true. Then by the completeness of $\vee_s$, we can use that rule. Choose $\phi_{s_1} = \phi_s$ and $\phi_{s_2} = \neg\phi_s$ By definition of $\neg$, $z_\infty \subseteq (\phi_{s_1} \vee \phi_{s_2}) = (\phi_s \vee \neg\phi_s) = z_\infty$.

Now we prove the correctness of the remaining $\exists_{\phi_1}(\phi_2)$ proof rules. We start with Rule $\exists_{r1}$.

**Soundness:** Suppose this rule is true. By the correctness of $\exists_{t_1}$ in Zhang and Cleaveland [167, 168], we know that $\Gamma \vdash \exists(\phi_2)$. We now need to argue that $\phi_1$ is true until $\phi_2$ is true. Examine the valuation set $succ(\Gamma) \cap \rho_s$. By construction, $succ(\Gamma)$ gives all possible valuations after some time advance from $\Gamma$. By the constraint $\Gamma \vdash pred(\rho_s)$, we know that $\Gamma$ must be able to time-lapse to each valuation in the clock set $\rho_s$. Hence, $succ(\Gamma) \cap pred(\rho_s)$ is the set of valuations that elapses to $succ(\Gamma) \cap \rho_s$, and $succ(\Gamma) \cap pred_<(\rho_s)$ is that set of valuations requiring some non-zero time elapse to $\rho_s$. From the truth of this second premise, $\phi_1$ is must be

true for all of those times.

**Completeness:** Suppose the conclusion is indeed true; suppose $\Gamma \vdash \exists_{\phi_1}(\phi_2)$. Therefore, there is a time advance $t$ such that after elapsing $t$ units, $\phi_2$ is true, and for all times until (and not including $t$), $t$, $\phi_1$ is true. Choose $\rho_s$ to be a placeholder such that $succ(\Gamma) \wedge \rho_s$ is the sequent $\Gamma$ after $t$ units has elapsed. By the correctness of the proof for the rule $\exists_{t1}$ in Zhang and Cleaveland [167, 168], we know that $succ(\Gamma), \rho_s \vdash \phi_2$. Because for all times until the times $\rho_s$, $\phi_1$ is true, this set of times by definition is $succ(\Gamma) \wedge pred_<(\rho_s)$. Therefore, $succ(\Gamma) \wedge pred_<(\rho_s) \vdash \phi_1$ By definition of the time elapse, since this is true we know that $\Gamma \vdash pred(\rho_s)$.

We now examine rule $\exists_{r2}$. Its proof of soundness and completeness is similar to $\exists_{r1}$. The difference is that we are elapsing from $\Gamma \cap \rho_s$. In $\exists_{t2}$, the third clause shrinks $\rho_s$ to ensure that the time-elapse relation holds.

**Soundness:** Suppose this rule is true. By the correctness of $\exists_{t2}$, the first and third premise show that $\exists(\phi_2)$ is true. Let that time advance be $\delta$ units. We now need to argue that $\phi_1$ is true until $\phi_2$ is true. Examine the valuation set $succ(\Gamma) \cap \rho_{s'}$. By construction, $succ(\Gamma)$ gives all possible valuations after some time advance from $\Gamma$. By the constraint $\rho_s \vdash pred(\rho_{s'})$, we know that $\Gamma, \rho_s$ must be able to time-lapse to each valuation in the clock zone $\rho_{s'}$. Hence, $succ(\Gamma) \cap pred(\rho_{s'})$ is the set of valuations that elapses to $succ(\Gamma) \cap \rho_{s'}$, and $succ(\Gamma) \cap pred_<(\rho_{s'})$ is that set of valuations requiring some non-zero time elapse to $\rho_{s'}$. From the truth of this second premise, $\phi_1$ is must be true for all of those times.

**Completeness:** Suppose the conclusion is indeed true; suppose $\Gamma, \rho_s \vdash \exists_{\phi_1}(\phi_2)$. Therefore, there is a time advance $t$ such that after elapsing $t$ units, $\phi_2$ is true, and for all times until (and not including $t$) $t$, $\phi_1$ is true. Choose $\rho_{s'}$ to be a placeholder such that $succ(\Gamma) \wedge \rho_{s'}$ is the sequent $\Gamma, \rho_s$ after $t$ units has elapsed. By the correctness of the proof for the rule $\exists_{t2}$ in Zhang and Cleaveland [167, 168], we know

that $succ(\Gamma), \rho_{s'} \vdash \phi_2$. Because for all times until the times $\rho_{s'}$, $\phi_1$ is true, this set of times by definition is $succ(\Gamma) \wedge pred_<(\rho_{s'})$. Therefore, $succ(\Gamma) \wedge pred_<(\rho_{s'}) \vdash \phi_1$ By definition of the time elapse, since this is true we know that $\rho_s \vdash pred(\rho_{s'})$. □

*Remark* 5.5.1 (Establishing invalidity). Consider timed automaton $TA$ with state $(l, \nu)$, and consider logical formula $\phi$. Concerning soundness and completeness, we establish validity (showing that $(l, \nu) \models \phi$) by providing a proof that $(l, cc_\nu) \vdash \phi$. Because the proof system is sound, we know that $(l, \nu) \models \phi$. Showing invalidity is different. To show invalidity, we enumerate over all our options and show that no such proof is possible. (At each step, we address the outermost logical operator of $\phi$; hence, there are only a small number of proof rules to consider) Then, by completeness, if we have no proof that $(l, cc_\nu) \vdash \phi$, then we know that $(l, \nu) \not\models \phi$.

### 5.5.2 Performance Optimization: Derived Proof Rules

Typically, computers reason best with small baseline proof rules. However, we can improve the performance by having the computer work with *derived* proof rules. We describe two such situations where we use derived proof rules.

### 5.5.3 Optimizing $\vee$

For performance reasons we replace a rule for $\vee$ in Zhang and Cleaveland [167, 168]. Those papers use the proof rule $\vee_c$ given in Figure 5.1. We instead use the proof rule $\vee_s$, which we give in Figure 5.4. By pushing fresh placeholders for both branches, we avoid computing the complementation operator, which often results in forming a placeholder involving a union of clock constraints. This rule's soundness and completeness is proven in Theorem 5.5.2.

$$\frac{(l,cc) \vdash \forall(\phi_2)}{(l,cc) \vdash \forall_{\phi_1}(\phi_2)}\,(\forall_{ro1}) \qquad \frac{(l,cc) \vdash \phi_1 \wedge \phi_2}{(l,cc) \vdash \forall_{\phi_1}(\phi_2)}\,(\forall_{ro2})$$

$$\frac{\begin{array}{cc} succ((l,cc)), \rho_{s_1} \vdash \phi_1 & \rho_\exists \vdash pred(\rho_{s_1}) \\ succ((l,cc)), \rho_{s_2} \vdash \phi_2 & succ((l,cc), \rho_\forall) \vdash succ((l,cc)) \wedge \rho_{s_2} \\ succ((l,cc)), pred(\rho_{s_1}) \vdash succ((l,cc)), \rho_{s_2} & (l,cc) \vdash \rho_\exists \vee \rho_\forall \end{array}}{(l,cc) \vdash \forall_{\phi_1}(\phi_2)}\,(\forall_{ro3})$$

**Figure 5.5:** Derived proof rules for $\forall_{\phi_1}(\phi_2)$.

### 5.5.4 Optimizing $\forall_{\phi_1}(\phi_2)$

Recall the derived formula for $\forall_{\phi_1}(\phi_2)$ from Lemma 5.5.1: $\forall_{\phi_1}(\phi_2)$ is equivalent to $\forall(\phi_2) \vee \exists_{\phi_2}(\phi_1 \wedge \phi_2)$. This formula requires $\phi_2$ to be checked twice. However, by deriving the proof rule, we notice that we can perform the checking of $\phi_2$ *only once*. The key is to compute the largest placeholder that satisfies $\phi_2$, and then to reason with this placeholder (and its time predecessor) to find the placeholders needed to satisfy the two branches of the derived formula. This reasoning allows the tool to reason with the sub formula $\phi_2$ only once, reusing the obtained information. This technique is **memoization**, the same technique used in dynamic programming. By making a memo of $\phi_2$, we need only compute it once rather than multiple times. The derived proof rules are in Figure 5.5; the proof and the derivation are given in Section 5.5.4. The first two handle the simpler cases when either $\phi_2$ is always true (or when $\phi_1$ is always false) or $\phi_1$ is immediately true (such as when $\phi_1$ is an atomic proposition), and the third rule ($\forall_{ro3}$) is the more complex case. The proof rules involving placeholders are similar.

#### Derivation of Derived Proof Rules

To derive an optimized proof for $\forall_{\phi_1}(\phi_2)$, we first will derive a slightly different version for $\exists_{\phi_2}(\phi_1 \wedge \phi_2)$, rewriting the proof rule for this special case slightly. This version will allow us to get the same premise to appear twice in the proof.

The $\exists_{\phi_1}(\phi_1 \wedge \phi_2)$ can be slightly rewritten as follows:

$$\dfrac{\dfrac{\boxed{succ((l,cc)), \rho_{s_1} \vdash \phi_1} \qquad \boxed{succ((l,cc)), pred(\rho_{s_1}) \vdash \phi_2} \qquad \boxed{\rho_\exists \vdash pred(\rho_{s_1})}}{(l,cc), \rho_\exists \vdash \exists_{\phi_2}(\phi_1 \wedge \phi_2)}\ (\exists_{r2}\text{ rewrite})}{\exists\text{ ph rewrite}}$$

Hence, we look more closely at the rewrite rule, comparing the derivation that is obtained, we get the following proof rule, $\exists_{rw}$:

$$\dfrac{succ((l,cc)), \rho_{s_1} \vdash \phi_1; succ((l,cc)), pred(\rho_{s_1}) \vdash \phi_2; \rho_\exists \vdash pred(\rho_{s_1})}{succ((l,cc)), \rho_{s_1} \vdash \phi_1 \wedge \phi_2; succ((l,cc)), pred_<(\rho_{s_1}) \vdash \phi_2; \rho_\exists \vdash pred(\rho_{s_1})}\ (\exists_{rw})$$

This rule has three conclusions that are written as three premises.

**Lemma 5.5.3.** The $\exists_{rw}$ rule is sound and complete.

***Proof of Lemma 5.5.3.*** **Soundness:** Assume that the three top premises are true. Since the third conclusion is the same as the third premise, that is true. Now we must argue that $succ((l,cc)), \rho_{s_1} \vdash \phi_1 \wedge \phi_2$ and $succ((l,cc)), pred_<(\rho_{s_1}) \vdash \phi_2$. Since we have $succ((l,cc)), pred(\rho_{s_1}) \vdash \phi_{s_2}$ and $pred_<(phi_{s_1}) \subseteq pred_<(phi_{s_1})$, we know $succ((l,cc)), pred(\rho_{s_1}) \vdash \phi_{s_2}$. Furthermore, since $\rho_{s_1} \subseteq pred(\rho_{s_1})$, $succ((l,cc)), \rho_{s_1} \vdash \phi_2$. Therefore, $succ((l,cc)), \rho_{s_1} \vdash \phi_1 \wedge \phi_2$.

**Completeness:** Assume that the three bottom conclusions are true. Since the third premise is the same as the third conclusion, that is true. Now we must argue that $succ((l,cc)), \rho_{s_1} \vdash \phi_1$ and $succ((l,cc)), pred(\rho_{s_1}) \vdash \phi_2$. Since we have $succ((l,cc)), \rho_{s_1} \vdash \phi_1 \wedge \phi_2$, we have $succ((l,cc)), \rho_{s_1} \vdash \phi_1$. Furthermore, since $\rho_{s_1} \cup pred_<(\rho_{s_1}) = pred(\rho_{s_1})$, we know that $succ((l,cc)), pred(\rho_{s_1}) \vdash \phi_{s_2}$. □

Now, with the rule $\exists_{rw}$, we utilize the formulation in Lemma 5.5.1 to derive a proof for $\forall_{\phi_1}(\phi_2)$. Here is the derivation for $\forall_{\phi_1}(\phi_2)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\boxed{succ((l,cc)), \rho_{s_1} \vdash \phi_1} \quad \boxed{succ((l,cc)), pred(\rho_{s_1}) \vdash \phi_2} \quad \boxed{\rho_\exists \vdash pred(\rho_{s_1})}}{succ((l,cc)), \rho_{s_1} \vdash \phi_1 \wedge \phi_2; succ((l,cc)), pred_<(\rho_{s_1}) \vdash \phi_2; \rho_\exists \vdash pred(\rho_{s_1})}\,(\exists_{rw})}{(l,cc), \rho_\exists \vdash \exists_{\phi_2}(\phi_1 \wedge \phi_2)}\,(\exists_{r2})}{\exists \text{ ph}}$$

$$\dfrac{\dfrac{\boxed{succ((l,cc)), \rho_{s_2} \vdash \phi_2} \quad \boxed{succ((l,cc), \rho_\forall) \vdash succ((l,cc)) \wedge \rho_{s_2}}}{(l,cc), \rho_\forall \vdash \forall(\phi_2)}\,(\forall_{t2})}{\forall \text{ ph}}$$

$$\dfrac{\dfrac{\dfrac{\text{See } \exists \text{ ph}}{(l,cc), \rho_\exists \vdash \exists_{\phi_2}(\phi_1 \wedge \phi_2)} \quad \dfrac{\text{See } \forall \text{ ph}}{(l,cc), \rho_\forall \vdash \forall(\phi_2)} \quad \boxed{(l,cc) \vdash \rho_\exists \vee \rho_\forall}}{(l,cc) \vdash \exists_{\phi_2}(\phi_1 \wedge \phi_2) \vee \forall(\phi_2)}\,(\vee_{s_r})}{(l,cc) \vdash \forall_{\phi_1}(\phi_2)}\,\text{Lemma 5.1}$$

We stop the derivation here and examine the derived sequents. Notice that we are computing a placeholder for $\phi_2$ twice. We can perform this computation **once** and save ourselves a good amount of computation time. We can compute things in this order:

1. Find the placeholder for $\phi_1$. Utilize simpler proof rules if $\phi_1$ is one of the easier cases.

2. Find the placeholder for $\phi_2$. (Copy this value for the other branch.) Now using this, solve the $\forall_{t2}$ rule to obtain a placeholder $\rho_\forall$.

3. After solving $\rho_\forall$, use the solved $\phi_2$ placeholder to solve for $\rho_\exists$. (This is the hard step that yields the optimization.)

4. Now solve the $\vee_s$ placeholder rule.

Using this insight, we now give the optimized proof rules. These rules are:

$$\frac{(l, cc) \vdash \forall(\phi_2)}{(l, cc) \vdash \forall_{\phi_1}(\phi_2)} \, (\forall_{ro1})$$

$$\frac{(l, cc) \vdash \phi_1 \wedge \phi_2}{(l, cc) \vdash \forall_{\phi_1}(\phi_2)} \, (\forall_{ro2})$$

$$\frac{\begin{array}{cc} succ((l, cc)), \rho_{s_1} \vdash \phi_1 & \rho_\exists \vdash pred(\rho_{s_1}) \\ succ((l, cc)), \rho_{s_2} \vdash \phi_2 & succ((l, cc), \rho_\forall) \vdash succ((l, cc)) \wedge \rho_{s_2} \\ succ((l, cc)), pred(\rho_{s_1}) \vdash succ((l, cc)), \rho_{s_2} & (l, cc) \vdash \rho_\exists \vee \rho_\forall \end{array}}{(l, cc) \vdash \forall_{\phi_1}(\phi_2)} \, (\forall_{ro3})$$

The first proof rule is used when $\phi_1$ is never satisfied for any time advance. Such as case is when $\phi_1$ is a false atomic proposition. The second proof rule is when $\phi_1$ is immediately true without a time advance, such as a true atomic proposition. The third proof rule uses the expanded derived proof rule to enforce that $\phi_2$ is only computed once. The premise $pred(\rho_{s_1}) \subseteq \rho_{s_2}$ ensures that all of the predecessor of $\rho_1$ satisfies $\phi_2$. The last premise of ($\forall$ ph1) is mostly solved by checking that $pred(\rho_{s_1}) \subseteq \rho_{s_2}$; we require the intersection with the successor for completeness, since we need not require times before $(l, cc)$ to satisfy $\phi_2$.

If a placeholder is involved, we use the following analogous optimized proof rules:

$$\frac{(l, cc), \rho_s \vdash \forall(\phi_2)}{(l, cc), \rho_s \vdash \forall_{\phi_1}(\phi_2)} \, (\forall_{rop1})$$

$$\frac{(l, cc), \rho_s \vdash \phi_1 \wedge \phi_2}{(l, cc), \rho_s \vdash \forall_{\phi_1}(\phi_2)} \, (\forall_{rop2})$$

$$\frac{\begin{array}{cc} succ((l, cc)), \rho_{s_1} \vdash \phi_1 & \rho_\exists \vdash pred(\rho_{s_1}) \\ succ((l, cc)), \rho_{s_2} \vdash \phi_2 & succ((l, cc), \rho_\forall) \vdash succ((l, cc)) \wedge \rho_{s_2} \\ succ((l, cc)), pred(\rho_{s_1}) \vdash succ((l, cc)), \rho_{s_2} & (l, cc), \rho_s \vdash \rho_\exists \vee \rho_\forall \end{array}}{(l, cc), \rho_s \vdash \forall_{\phi_1}(\phi_2)} \, (\forall_{rop3})$$

Notice that due to the fresh placeholder generated by $\vee_{s_r}$, that the placeholder rule is similar to the rule without the placeholder.

Also notice that the implementation complexity of $\exists_{r2}$ is placed in the third premise of ($\forall$ ph1): $succ((l,cc)), pred(\rho_{s_1}) \vdash succ((l,cc)), \rho_{s_2}$. The catch is to make $\rho_{s_1}$ as large as possible such that all the proof rules go through. yet ensuring that all of $pred(\rho_{s_1})$ satisfies $\phi_2$.

**Lemma 5.5.4.** The optimized proof rules for $\forall_{\phi_1}(\phi_2)$ are sound and complete.

***Proof of Lemma 5.5.4.*** Given the correctness of $\vee_{s_r}$, the soundness and completeness of the proof rules with and without placeholders are the same. Hence, we only give the soundness and completeness proof rules when placeholders are not used. Note that the soundness and completeness heavily depends on the derivation and the correctness of $\exists_{rw}$; we only need argue the changes to the derivation.

**Soundness:** The soundness of $\forall_{ro1}$ and $\forall_{ro2}$ follow from the definition of $\forall_{\phi_1}(\phi_2)$. In the first case, $\phi_2$ is always true (relativization not needed) and in the second rule, $\phi_1$ is immediately satisfied. Hence we examine rule $\forall_{ro3}$. Suppose that the premises are true. Given that the derived proof rule is correct, we only need show that the proof rule $succ((l,cc), \rho_\forall) \vdash succ((l,cc)) \wedge \rho_{s_2}$ results in the correct placeholders $\phi_{s_1}$ and $\rho_{s_2}$. We then invoke the soundness of the derived proof. Since we have $succ((l,cc)), pred(\rho_{s_1}) \vdash succ((l,cc)), \rho_{s_2}$, we know that by definition of $\subseteq$ and $\vdash$ and the premise $succ((l,cc)), \rho_{s_2} \vdash \phi_{s_2}$ that $succ((l,cc)), pred(\rho_{s_1}) \vdash \phi_2$. Now we have all the premises from the derived rule (including the rule $\exists_{rw}$). Hence, the proof is sound.

**Completeness:** The proof rules $\forall_{ro1}$ and $\forall_{ro2}$ cover the corner cases when $\rho_{s_1}$ is either empty or all possible clock values. We now address the completeness using rule $\forall_{ro3}$. Suppose that the conclusion is true; that $(l,cc) \vdash \forall_{\phi_1}(\phi_2)$. From

the soundness and completeness of the derived proof rules, the only premise that is different is $succ((l,cc),\rho_\forall) \vdash succ((l,cc)) \wedge \rho_{s_2}$. Using the derived proof rules (including rule $\exists_{rw}$), we get two placeholders based on $\phi_{s_2}$: $succ((l,cc)), pred(\rho_{s_1}) \vdash \phi_{s_2}$ and $succ((l,cc)), \rho_{s_2} \vdash \phi_{s_2}$. Using soundness and completeness, choose $\rho_{s_2}$ to be the largest such placeholder (since the proof rules can be solved exactly with unions of clock zones, one such largest placeholder exists). Since $\rho_{s_1}$ and $\rho_{s_2}$ are clock constraints (unions of clock zones independent of location), The choices of $pred(\rho_{s_1})$ and $\rho_{s_2}$ do not change the discrete state, and only the clock state. Since both clock states are contained in the set of clock states that satisfy $\phi_2$, both depend on clock constraints. Since $\rho_{s_2}$ was chosen to be the largest possible such placeholder when intersection with $succ((l,cc))$, we know $succ((l,cc)), pred(\rho_{s_1}) \vdash succ((l,cc)), \rho_{s_2}$. (Note that the proof requires the intersection with $succ((l,cc))$ on both sides.) Hence, we have found placeholders that satisfy all the premises. □

### 5.5.5 Optimizing the Handling of Invariants

In order to prove properties of timed automata with invariants, whenever a time advance or an action execution occurs, the prover must account for the invariant. Formally, the prover must utilize the definition of the invariant and include it as a logical subformula of the right hand sequent (invariants are clock constraints, which are valid $L^{rel,af}_{\nu,\mu}$ formulas) and then execute proof rules to handle the expanded formula. However, by taking the sequence of proof rules needed to handle invariants, we can form derived proof rules that reduce handling an invariant to some computations. As a result, we can handle the proofs computationally rather than including invariants as part of the formulas in the right-hands of the sequents of the proofs.

We represent an invariant with $Inv$. To handle invariants, we add them to the formula (similar to how invariants are handled when they are converted to a PES

in Zhang and Cleaveland [167, 168]). We incorporate invariants into MES as follows: $\exists(\phi)$ becomes $\exists(Inv \wedge \phi)$ and $\forall(\phi)$ becomes $\forall(\neg Inv \vee \phi)$. These encodings require that the invariants are *past closed*: if the invariant is true at some time, then the invariant must be true at all previous times. Using these derived proof rules, we can reduce computation by specializing the proof tree by substituting in $Inv$ (or $\neg Inv$) for the relevant placeholders. Since we know the value of $Inv$, which is a specific clock constraint, rather then using the general-purpose rules to solve for the placeholders, we input in these values and *specialize* the rules. Invariants in action operators $\langle - \rangle(\phi)$ and $[-](\phi)$ are handle as guards are handled in Zhang and Cleaveland [167, 168].

To optimize the implementation of invariants, we derive the rules that are formed when using invariants. Here, we let $Inv$ represent the invariant. Now using the invariant, we derive the proof rules:

$$\dfrac{\dfrac{\boxed{succ((l,cc)),\rho_s \vdash Inv} \qquad succ((l,cc)),\rho_s \vdash \phi}{succ((l,cc)),\rho_s \vdash Inv \wedge \phi}(\wedge) \qquad (l,cc) \vdash \rho_s}{(l,cc) \vdash \exists(Inv \wedge \phi)}(\exists_{t1})$$

From this derivation, we know that for $\exists$, the proper thing to do concerning invariants is to intersect the invariants with the placeholder $\rho_s$ and not with $succ((l,cc))$.

$$\dfrac{\dfrac{\boxed{succ((l,cc)),\rho_s \vdash Inv} \qquad succ((l,cc)),\rho_s \vdash \phi}{succ((l,cc)),\rho_s \vdash Inv \wedge \phi}(\wedge) \qquad (l,cc),\rho_\exists \vdash \rho_s}{(l,cc),\rho_\exists \vdash \exists(Inv \wedge \phi)}(\exists_{t2})$$

This above derivation for $\exists$ is similar to the one without the placeholder. Hence, we know to intersect the invariants with the placeholder $\rho_s$ and not with $succ((l,cc))$.

Now we derive the invariant for $\forall$:

$$\dfrac{\dfrac{succ((l,cc)),\rho_s \vdash \neg Inv \qquad succ((l,cc)),\neg\rho_s \vdash \phi}{succ((l,cc)) \vdash \neg Inv \vee \phi}\,(\vee_c)}{(l,cc) \vdash \forall(\neg Inv \vee \phi)}\,(\forall_{t1})$$

Since $\rho_s = \neg Inv$, this rule reduces to:

$$\dfrac{\dfrac{succ((l,cc)), Inv \vdash \phi}{succ((l,cc)) \vdash \neg Inv \vee \phi}\,(\vee_c \text{ d})}{(l,cc) \vdash \forall(\neg Inv \vee \phi)}\,(\forall_{t1})$$

From this derivation, since there is no placeholder, we intersect the invariant $Inv$ with $succ((l,cc))$. In the proof rules, a "d" in the label indicates a derived rule. Now we consider $\forall$ with a placeholder:

$$\dfrac{\dfrac{succ((l,cc)),\rho_s \vdash \neg Inv \qquad succ((l,cc)),\rho_\vee - \rho_s \vdash \phi}{succ((l,cc)),\rho_\vee \vdash \neg Inv \vee \phi}\,(\vee_c)}{\forall_{t2}\ \text{inv1}}$$

$$\dfrac{\dfrac{\text{see } \forall_{t2}\ \text{inv1}}{succ((l,cc)),\rho_\vee \vdash \neg Inv \vee \phi} \qquad succ((l,cc),\rho_\forall) \vdash succ((l,cc)) \wedge \rho_\vee}{(l,cc),\rho_\forall \vdash \forall(\neg Inv \vee \phi)}\,(\forall_{t2})$$

This derivation is more complex, due to the placeholder in the $\forall$. Here we use the version of $\vee_c$ that uses the complement of the placeholder. After doing some set operations and solving the left ($\rho_s = \neg Inv$), we get the cleaned up version of $\forall$ with a placeholder:

$$\dfrac{\dfrac{\boxed{succ((l,cc)), Inv, \rho_\vee \vdash \phi}}{succ((l,cc)),\rho_\vee \vdash \neg Inv \vee \phi}\,(\vee_c \text{d}) \qquad succ((l,cc),\rho_\forall) \vdash succ((l,cc)) \wedge \rho_\vee}{(l,cc),\rho_\forall \vdash \forall(\neg Inv \vee \phi)}\,(\forall_{t2})$$

This means that the Invariant is not part of the placeholder, $\rho_\vee$. Hence, we still intersect $Inv$ with $succ(\Gamma)$, and not the placeholder, and we might to allow valuations that do not satisfy the invariant in the placeholder $\rho_\vee$. Furthermore, to get the largest placeholder $\rho_\vee$, we have to include all of $\neg Inv$ in $\rho_v$. This means

that to get all possible valuations for the placeholders, we union the complement of the invariant ($\neg Inv$) with the placeholder $\rho_\vee$ and then use $\rho_v$ to find $\rho_\forall$.

To illustrate this point, we derive the $\forall$ rule with a placeholder using the $\vee_s$ proof rule from this paper. The alternative derivation is:

$$\dfrac{\dfrac{succ((l,cc)),\rho_{s_1} \vdash \neg Inv \qquad \begin{array}{c} succ((l,cc)),\rho_{s_2} \vdash \phi \\ succ((l,cc)),\rho_\vee \vdash \rho_{s_1} \vee \rho_{s_2} \end{array}}{succ((l,cc)),\rho_\vee \vdash \neg Inv \vee \phi}\,(\vee_s)}{\text{Inv2}}$$

$$\dfrac{\dfrac{\text{see Inv2}}{succ((l,cc)),\rho_\vee \vdash \neg Inv \vee \phi} \qquad succ(((l,cc),\rho_\forall)) \vdash succ((l,cc)) \wedge \rho_\vee}{(l,cc),\rho_\forall \vdash \forall(\neg Inv \vee \phi)}\,(\forall_{t2})$$

Using that $\rho_{s_1} = \neg Inv$, we get the cleaned up rule for $\forall$ with a placeholder:

$$\dfrac{\dfrac{\boxed{succ((l,cc)),\rho_s \vdash \phi} \qquad \boxed{succ((l,cc)),\rho_\vee \vdash \rho_s \vee \neg Inv}}{succ((l,cc)),\rho_\vee \vdash \neg Inv \vee \phi}\,(\vee_s \text{ derived})}{\text{Inv3}}$$

$$\dfrac{\dfrac{\text{see Inv3}}{succ((l,cc)),\rho_\vee \vdash \neg Inv \vee \phi} \qquad succ(((l,cc),\rho_\forall)) \vdash succ((l,cc)) \wedge \rho_\vee}{(l,cc),\rho_\forall \vdash \forall(\neg Inv \vee \phi)}\,(\forall_{t2})$$

which is the same as the previous derivation using the rule $\vee_c$.

How we included the invariant $Inv$ in the proof rules depends on the definition of the time advance operators. Also note that these uses of $Inv$ require that $Inv$ is **past closed**. Note that $Inv \rightarrow \phi$ is equivalent to $\neg Inv \vee \phi$; since $\rightarrow$ is not fully supported, we encode the invariant with $(\neg Inv) \vee \phi$ and use the derived results to handle the negation over the invariant.

## 5.6 Additional Implementation Details

The proof rules encode the algorithm: the implementation checks the consequent by proving the premises and uses leafs (rules with no premises) to terminate the

proof. As a result, the bulk of the algorithm follows from the construction of the proof rules.

However, there are some additional lower-level details that can result in better performance. These vary from using simpler formulas when possible to handling the placeholder implementation in subtle cases. The implementation details of the data structures is discussed in Section 5.8.

### 5.6.1 Addressing Performance: Simpler PES Formulas

When writing safety and liveness properties, we can use the formulas from Section 4.7.3 of this dissertation. However, in the common case where there are no nested temporal operators and the formula does not involve clock constraints, we can simplify the formulations considerably. In these cases, the subformula is a conjunction and disjunction of atomic propositions, and is represented by $p$ or $q$. Here are some simplifications:

$$AG\,[p] \equiv Y \stackrel{\nu}{=} \; p \wedge \forall([-](Y)) \tag{5.2}$$

$$AF\,[p] \equiv Y \stackrel{\mu}{=} \; p \vee \Big(\forall([-](Y)) \wedge \exists(z.(\forall(z < 1)))\Big) \tag{5.3}$$

$$EF\,[p] \equiv Y \stackrel{\mu}{=} \; p \vee \exists(\langle - \rangle(Y)) \tag{5.4}$$

$$EG\,[p] \equiv Y \stackrel{\nu}{=} \; p \wedge \Big(\exists(\langle - \rangle(Y)) \vee \forall(z.(\exists(z \geq 1)))\Big) \tag{5.5}$$

The TCTL operators here are: $AG\,[p]$ (always $p$), $AF\,[p]$ (inevitably $p$), $EG\,[p]$ (there exists a path where always $p$), and $EF\,[p]$ (possibly $p$). One noticeable feature is that these simplified liveness properties do not require relativization. Another noticeable feature is that the $\vee$ can be simplified to not use placeholders if one side is an atomic proposition; consequently, $AG\,[p]$ and $AF\,[p]$ do not require placeholders. This feature is a reason why the correct formulations were hard to get: easier formulas can be used in tools. Additionally, our tool directly

implements $\exists(z.(\forall(z < 1)))$ and $\forall(z.(\exists(z \geq 1)))$.

**Correctness of Simplified PES Formulas**

If $\phi_1$ is an atomic proposition, conjunction, or disjunction of them (it has no fixpoint variables, transitions, time advances or clock constraints), the the relativized formulas can be simplified. Let the conjunction and disjunctive constraint (normal form not required) of atomic propositions be $p_p$. We can construct $p_p$ with the following grammar:

$$p_p ::= p \mid \neg p \mid \texttt{tt} \mid \texttt{ff} \mid p_p \wedge p_p \mid p_p \vee p_p \tag{5.6}$$

where $p \in 2^L$ is an atomic proposition.

We represent such atomic literals with $p$ and $q$. If we only consider subformulas with this specified grammar, we also give simplified formulas for common TCTL operators.

**Theorem 5.6.1.** Let $p$ and $q$ be a combinations of conjunctions and disjunctions of atomic propositions constructed using Equation 5.6. Then we have the following simplified TCTL formulas:

$$AG\,[p] \equiv Y \stackrel{\nu}{=} p \wedge \forall([-](Y)) \tag{5.7}$$

$$AF\,[p] \equiv Y \stackrel{\mu}{=} p \vee \Big(\forall([-](Y)) \wedge \exists(z.(\forall(z < 1)))\Big) \tag{5.8}$$

$$EF\,[p] \equiv Y \stackrel{\mu}{=} p \vee \exists(\langle - \rangle(Y)) \tag{5.9}$$

$$EG\,[p] \equiv Y \stackrel{\nu}{=} p \wedge \Big(\exists(\langle - \rangle(Y)) \vee \forall(z.(\exists(z \geq 1)))\Big) \tag{5.10}$$

$$AG\,[p \rightarrow AF\,[q]] \equiv Y \stackrel{\nu}{=} (\neg p) \vee \forall(Y_2 \wedge [-](Y))$$

$$Y_2 \stackrel{\mu}{=} q \vee (\forall([-](Y_2)) \wedge \exists(z.(\forall(z < 1)))) \tag{5.11}$$

The last operator is the "leads to" operator. Here we use the simplified $AF\,[q]$ but use the regular $AG\,[p]$ formula. Also recall that the tool has operators to handle the subpaths with the freeze quantifiers.

To prove this operators, we will rely on some of the properties of formulas involving only atomic propositions. The proofs rely on the following property: if $p$ is true, then $\forall(p)$ is true. Also, for all previous times, $p$ is true. Hence, the semantics of the formula is equivalent regardless of whether a continuous or a pointwise semantics is used for $p$. As a result, we have the equivalences in the following lemma:

**Lemma 5.6.2** (Properties of atomic proposition formulas)**.** Let $p$ be a combination

of conjunctions and disjunctions of atomic propositions. Then:

$$p \equiv \exists(p) \equiv \forall(p) \tag{5.12}$$

$$p \vee \forall(\phi) \equiv \forall_p(p \vee \phi) \text{ for any formula } \phi \tag{5.13}$$

$$p \wedge \exists(\phi) \equiv \exists_p(p \wedge \phi) \text{ for any formula } \phi \tag{5.14}$$

***Proof of Lemma 5.6.2.*** From the definitions of the $L^{rel}_{\nu,\mu}$ operators. For some insight into the second and third equivalences, try using $\phi = \texttt{ff}$ and $\phi = \texttt{tt}$. □

***Proof of Theorem 5.6.1.*** Here we show $AG\,[p]$ and $AF\,[p]$. The proofs for $EG\,[p]$ and $EF\,[p]$ are similar, and the proof of the last equivalence follows from the proofs for $AG\,[p]$ and $AF\,[p]$.

Proof of $AG\,[p]$:

$$\begin{aligned} AG\,[p] \equiv & Y \overset{\nu}{=} \forall(p \wedge [-](Y)) \text{ (Original Formula)} \\ & Y \overset{\nu}{=} \forall(p) \wedge \forall([-](Y)) \text{ (Distributivity } \forall, \wedge) \\ & Y \overset{\nu}{=} p \wedge \forall([-](Y)) \ (p \equiv \forall(p)) \end{aligned}$$

Proof of $AF\,[p]$:

$$AF\,[p] \equiv Y \stackrel{\mu}{=} \forall_p\Big(p \vee [-](Y)\Big) \wedge \Big(\exists(z.(\forall(z<1))) \vee \exists(p)\Big)$$

(Original Formula)

$$Y \stackrel{\mu}{=} (p \vee \forall([-](Y))) \wedge \Big(\exists(z.(\forall(z<1))) \vee \exists(p)\Big)$$

$(p \vee \forall(\phi) \equiv \forall_p(p \vee \phi))$

$$Y \stackrel{\mu}{=} (p \vee \forall([-](Y))) \wedge \Big(\exists(z.(\forall(z<1))) \vee p\Big)\ (p \equiv \exists(p))$$

$$Y \stackrel{\mu}{=} (p \vee \forall([-](Y))) \wedge (p \vee \exists(z.(\forall(z<1))))$$

(Commutativity $\vee$)

$$Y \stackrel{\mu}{=} p \vee \Big(\forall([-](Y)) \wedge \exists(z.(\forall(z<1)))\Big)\ \text{(Distributivity } \wedge, \vee)$$

□

### 5.6.2 Placeholder Implementation Complexities

Consider the two placeholder premises in the $\forall(\phi)$ and $\exists_{\phi_1}(\phi_2)$ proof rules in Figures 5.2 and 5.4. The placeholder sequents are given here:

$$succ((l,cc),\rho_\forall) \vdash succ((l,cc)) \wedge \rho_s \text{ and } succ((l,cc)), pred_<(\rho_s) \vdash \phi_1 \qquad (5.15)$$

In soundness and completeness proofs, we use soundness to give us a placeholder to show that the formula holds, and with completeness, we argue that some placeholder exists. Given the complexities of the formulas, the tool needs to find the *largest* such placeholder. The rules are designed for the tool to implement them in a left-to-right fashion, where placeholders are tightened by right-hand rules. However, as the placeholders are tightened, we need to make sure that the tightened placeholder still satisfies the left-hand premise. For instance, consider

the second of the above placeholders. As we tighten the placeholder to satisfy $\phi_1$, we need to check that this placeholder is the predecessor$_<$ of the placeholder that satisfies $\phi_2$. These checks take extra algorithmic work.

## 5.7 Clock Zones

Before discussing the implementation of the clock data structures, we discuss the conceptual data structure: clock zones.

For model checking, we have to deal with an infinite number of states (given an infinite number of valuations) and terminate in a finite amount of time. To do this, we need an abstraction or a way of grouping states together to form a finite number of state collections or abstract states. One commonly used abstraction is to group sets of valuations into *clock zones*.

For model checking, one abstraction to aid in reachability and model checking is to model check a convex set of clock valuations at once. Thus instead of checking states $(l, \nu)$ individually, we can group valuations together into *clock zones* and then reason on clock zones. Clock zones have the property of being *convex* sets of valuations. This definition of a clock zone is taken from Alur [5], Clarke et al. [55].

**Definition 5.7.1 (Clock zone).** A *clock zone* is a convex combination of single-clock inequalities. Each clock zone can be constructed using the following grammar, where $x_i$ and $x_j$ are arbitrary clocks and $c \in \mathbb{Z}$:

$$Z ::= x_i < c \mid x_i \leq c \mid x_i > c \mid x_i \geq c \mid x_i - x_j < c \mid x_i - x_j \leq c \mid Z \wedge Z \quad (5.16)$$

■

**Example 5.7.1.** The following are examples of clock zones:

$$z_1 = x_1 \geq 3 \wedge 5 \leq x_2 - x_1 \leq 7$$

$$z_2 = x_1 < 6 \wedge x_1 - x_2 \leq 3 \text{ (notice it is } x_1 - x_2 \text{ here)}$$

$$z_3 = x_2 > 1 \wedge 5 < x_2 - x_1 < 8$$

$$z_4 = 1 \leq x_1 \leq 2 \wedge 1 \leq x_2 \leq 2 \wedge x_2 - x_1 \geq 0$$

For these clock zones, $CX = \{x_1, x_2\}$. ■

Clock zones extend clock constraints with inequalities of clock differences. These inequalities are used for model checking even though clock difference inequalities are not used in timed automata. However, in general, clock zones are not unique. To make model checking easier, we use a standardized, or **canonical**, form for clock zone representations. We use shortest path closure [27]. This form makes every implicit constraint explicit. This can be implemented in $O(n^3)$ time using Floyd-Warshall all-pairs shortest path algorithm, described in Ahuja et al. [3], Cormen et al. [58]. Other standard forms exist [112, 154].

While converting to a canonical form takes a considerable amount of time, it is needed to simplify and standardize the algorithms for the zone operations including time successor ($succ(z)$) computations and subset checks. For time successor, having the zone in canonical form allows the time elapse operation to simply set all single-clock upper bound constraints to $< \infty$. Different standard forms require different algorithms for clock zone operators.

### 5.7.1 Clock Zone Operations

There are a variety of clock zone operations that are desirable for timed automata model checking. Here we proved the operation as well as a sketch of the imple-

mentation for the DBM data structure. Given clock zones, we define some commonly used operators on clock zones. We will want any representation of a clock zone to implement these.

The following operators work on clock zones and ouptput clock zones:

**Canonical Form—**`cf(`$z$`)`**:** Given a clock zone $z \in \mathcal{Z}$, `cf(`$z$`)` produces the *canonical form* of the zone $z$, which gives a representation where any implicit constraints are made explicit, including making constraints that can be more strict strict.

**Valuation Search—**`search(`$z$`, `$\nu$`)`**:** Given clock zone $z \in \mathcal{Z}$ and valuation $\nu \in \mathcal{V}$, `search(`$z$`, `$\nu$`)` tells you if $\nu \in z$.

**Get Clock Constraint—**`getConstraint(`$z$`, `$x_i$`, `$x_j$`)`**:**

`getConstraint(`$z$`, `$x_i$`, `$x_j$`)` returns the exact lower bound constraint ($<$ or $\leq$) of $x_i - x_j$ within clock zone $z$.

Giving $x_0$ is equivalent to giving o as a clock value, and will thus give you the single-clock constraint (which could be a lower bound or upper bound).

**Emptiness—**`isEmpty(`$z$`)`**:** Given clock zone $z \in \mathcal{Z}$, `isEmpty(`$z$`)` returns true if the zone is empty (has no valuations in it) and false otherwise.

**Intersection, $\wedge$—**`intersect(`$z_1$`,`$z_2$`)`**:** Let $z_1, z_2 \in \mathcal{Z}$. Then the *intersection* of the two clock zones, `intersect(`$z_1$`,`$z_2$`)` denoted $z_1 \wedge z_2$ is the clock zone equivalent to the clock zone $z' = z_1 \wedge z_2$ formed by the clock zone grammar which denotes the set of clock valuations in both of those clock zones.

**Contains, $z_1 \subseteq z_2$—**`contains(`$z_1$`, `$z_2$`)`**:** Given two clock zones $z_1, z_2$, `contains(`$z_1$`, `$z_2$`)` returns `tt` if $z_1$ is a subset (or equal to) the zone $z_2$ and `ff` otherwiwse.

**Equality,** $z_1 == z_2$**—**`equals(`$z_1$`, `$z_2$`)`**:** Given two clock zones $z_1$, $z_2$, `contains(`$z_1$`, `$z_2$`)` returns `tt` if $z_1$ is equal to the zone $z_2$ and `ff` otherwiwse.

**Time Successor,** $z^{\Uparrow}$**—**`succ(`$z$`)`**:** Given a clock zone $z$, `succ(`$z$`)` returns the set of valuations (clock zone) formed when any amount of time $\delta$ is allowed to elapse from $z$. I.e.

$z^{\Uparrow} = \{\nu \mid \nu - \delta \in z, \exists\ \delta \geq 0\} \equiv \{\nu \mid \nu \in z + \delta, \exists \delta \geq 0\} \equiv \{\nu \mid \nu \models \exists \delta \geq 0 \colon \nu \in z + \delta\}$.

**Time Predecessor,** $z^{\Downarrow}$**—**`pred(`$z$`)`**:** Given a clock zone $z$, this returns all the possible valuations that are predecessors, which are valuations that would be in this clock zone after some amount $\delta$ of time-elapse. I.e

$z^{\Downarrow} = \{\nu \mid \nu + \delta \in z, \exists\ \delta \geq 0\}$

**Reset Successor,** $z[Y := 0]$**—**`resetSuc(`$z$`, `$Y$`)`**:** Given a clock zone $z$ and a set of clocks $Y$ to reset to 0 `resetSuc(`$z$`, `$Y$`)` gives the new clock zone (of all valuations) where the set of clocks $Y$ are all reset to o. This is notated as $z[Y := 0]$.

**Assignment Successor,** $z[x_i := x_j]$**—**`assignSuc(`$z$`, `$x_i$`, `$x_j$`)`**:** Given a clock zone $z$, a clock $x_i$ and a clock $x_j$, `assignSuc(`$z$`, `$x_i$`, `$x_j$`)` gives the new clock zone (of all valuations) where the clock $x_i$ is reset (or assigned to) the current value of the clock $x_j$.

**Reset Predecessor—**`resetPred(`$z$`, `$Y$`)`**:** Given a clock zone $z$ and a set of clocks $Y$ to reset to 0 `resetPred(`$z$`, `$Y$`)` gives the clock zone (of all valuations) that would become $z$ after the clocks $Y$ are reset to 0.

This operator assumes that all the clocks $Y$ are exactly 0 in the zone $z$, i.e. $z[Y := 0] = z$ or that a reset immediately happens. This operator may not make sense if the zone $z$ does not meet this assumption.

**Assignment Predecessor—**`assignPred(`$z$`,` $x_i$`,` $x_j$`)`**:** Given a clock zone $z$, a clock $x_i$ and a clock $x_j$, `assignPred(`$z$`,` $x_i$`,` $x_j$`)` gives the clock zone $z'$ (of all valuations) that becomes $z$ when the clock $x_i$ in $z'$ is reset (or assigned to) the current value of the clock $x_j$ in $z'$.

This operator assumes that in $z$, $x_i = x_j$, which is equivalent to the assignment $x_i = x_j$ just happening in $z$. This operator may not make sense if the zone $z$ does not meet this assumption.

**Clock Constraint Normalization—**`normalize(`$z$`,` $c$`)`**:** Given a clock zone $z \in \mathcal{Z}$ and a function $c\colon CX \longrightarrow \mathbb{N}$ where $c(x)$ denotes the largest constant in a constraint for clock $x$, `normalize(`$z$`,` $c$`)` normalizes the clock zone by relaxing constraints so all constraints are treated as if the constants from $c$ are the largest constants.

There are other operators that are desirable but do not always return clcok zones. Some may even operate on other sets of clock valuations (or specific representations of them). Those specific to certain representations will be described when that valuation data structure is described. Some of these are:

**Union—**`union(`$z_1$`,` $z_2$`)`**:** Given two clock zones $z_1, z_2 \in \mathcal{Z}$, `union(`$z_1$`,` $z_2$`)` gives the *disjunction* of the two clock zones.

**Complementation ($\overline{z}$)—**`comp(`$z_1$`)`**:** Given a clock zone $z \in \mathcal{Z}$, `comp(`$z$`)` gives the *complementation* of the clock zone, $\overline{z}$ which is the set of all valuations not in $z$.

Some of the operators (including the reset predecessor and the time predecessor operators are described in Yovine [164]) (Though we deviate some from the versions in that paper). The normalization operator is described in Bengtsson and Yi [27].

### 5.7.2 Clock Zone Operation Details

We now describe the specifics for some of the operations in more detail. These specifics decribe the meaning of the operator and which constraints are changed; these apply to all implementations of a clock zone.

**Definition 5.7.2** (**Clock Zone Reset,** $z[Y := 0]$)**.** Let $z$ be a clock zone. Then the clock zone after the clocks $Y \subseteq CX$ are reset is denoted $z[Y := 0]$, and denotes the set of clock valuations that arise after the clocks $Y$ are reset in $z$.

Here is how the constraints are changed:

- Any single clock-constraint $y \in Y$ is deleted and replaced by $y \leq 0 \wedge 0 \leq y$. (Since $y$ has to be 0)
- Any clock-difference constraint involving $y_1, y_2 \in Y$ is deleted and replaced by $y_1 - y_2 \leq 0 \wedge y_2 - y_1 \leq 0$ (Since both $y_1$ and $y_2$ are 0, their difference must be exactly 0).
- Any clock-difference constraint involving $y \in Y$, $x \in CX - Y$ is deleted. (Note that the difference constraints will then be implicitly limited by the single-bound constraints involving $x$).

■

An insightful visualization, taken from Sokolsky and Smolka [147] is to imagine that the constraints (or set of valuations) is projected to the hyperplane $Y = 0$.

**Definition 5.7.3** (**Time-Elapsed Clock Zones (Time Successors),** $z^{\Uparrow}$)**.** Let $z$ be a clock zone. Then the *time-elapsed clock zone* or *Time Successor* zone, $z^{\Uparrow}$ denotes the set of valuations formed when any amount of time is allowed to elapse from $z$. I.e.

$z^{\Uparrow} = \{v \mid v - \delta \in z, \exists\, \delta \geq 0\}$.

In other words (from Clarke et al. [55]), any clock valuation $v$ that satisfies $\exists \delta \geq 0 \colon v \in z + \delta$ will be in $z^{\Uparrow}$.

A clock zone $z$ after specific time elapse $\delta$ is a *specific time-elapsed clock zone* denoted $z + \delta$.

Here is how the constraints are changed for a specific time-elapsed clock zone of $\delta$ units (the $^{\Uparrow}$ operator would make a different zone for each time elapse):

- For any clock $x \in CX$, any constraint of $x < c$ or $x \leq c$ becomes $x < c + \delta$ or $x \leq c + \delta$.
- Likewise, for any clock $x \in CX$, any constraint of $c < x$ or $c \leq x$ becomes $c + \delta < x$ or $c + \delta \leq x$.
- All constraints involving the difference of clocks are unchanged.

For the set of all possible time elapses, this means that every single-clock upper bound is set to be $< \infty$ and all other constraints remain unchanged. ■

As a visualization, take each point in the current zone and draw a line with slope 1 (in all dimensions). All points in these lines are possible valuations after some arbitrary time advance.

For elapsing of time, here we are not concerned with invariants or guards. When we want to be concerned, we can intersect this with clock zones representing the invariants or guards when desired.

**Definition 5.7.4 (Clock Constraint Normalization—`normalize(z, c)`:).** Given a clock zone $z \in \mathcal{Z}$ and a function $c \colon CX \longrightarrow \mathbb{N}$ where $c(x)$ denotes the largest

constant in a constraint for clock $x$, `normalize(z, c)` normalizes the clock zone by relaxing constraints so all constraints are treated as if the constants from $c$ are the largest constants.

This means that all upper bounds higher than $c(x)$ are relaxed to be $(\infty, <)$ and all lower bounds higher than $c(x)$ are lowered (relaxed) to be $> c(x)$. For clock difference constraints $x_i - x_j$, they are normalized so the upper bound is loosened to $(\infty, <)$ if it exceeds $c(x_i)$ and the lower bound of $x_i - x_j$ is lowered to be $> -c(x_j)$ (Which means that the upper bound on $x_j - x_i > c(x_j)$).

*Note.* As discussed in previous sections, this algorithm only works if clock difference constraints are not allowed in the timed automata. While the clock zones can encode such constraints, the automata cannot for this algorithm to work.

■

The reason that clock zones are normalized is in order to make sure that a timed automaton can be represented by a finite number of clock zones. While normalization does add valuations to the set of clock zone (information is lost), we do not care about the information that is lost, since all the properties of region equivalence still hold because of the following condition in the definition of *region equivalence* (See Definition 2.5.3):

- For all $x \in CX$, either $\lfloor \nu_1(x) \rfloor = \lfloor \nu_2(x) \rfloor$ or $\nu_1(x), \nu_2(x) > c(x)$.

Thus, if any value exceeds $c(x)$ for that specific clock, it does not matter by how much. In differences $x_i - x_j$, if the upper bound exceeds $c(x_i)$, that can only be possible if $x_i > c(x_i)$ since all clocks are at least 0. For lower bounds, if the lower bound exceeds $-c(x_j)$, then it is only possible when $x_j > c(x_j)$.

Thus, by eliminating the distinguishing between $\infty$ and values of clocks larger than their largest constants, we can compress our representation into a finite number of zones and still have those zones preserve the Reachability and Model-Checking properties we want them to preserve.

**Example 5.7.2.** Suppose $CX = \{x_1, x_2\}$. If we consider the following clock zones:

$$z_1 = x_1 \geq 3 \wedge 5 \leq x_2 - x_1 \leq 7$$

$$z_2 = x_1 < 6 \wedge x_1 - x_2 \leq 3 \text{ (notice it is } x_1 - x_2 \text{ here)}$$

$$z_3 = x_2 > 1 \wedge 5 < x_2 - x_1 < 8$$

$$z_4 = 1 \leq x_1 \leq 2 \wedge 1 \leq x_2 \leq 2 \wedge x_2 - x_1 \geq 0$$

Here are some of the intersections of the clock zones:

$$z_1 \wedge z_2 = 3 \leq x_1 < 6 \wedge 5 \leq x_2 - x_1 \leq 7$$

$$z_1 \wedge z_3 = x_1 \geq 3 \wedge x_2 \geq 1 \wedge 5 < x_2 - x_1 \leq 7$$

$$z_1 \wedge z_2 \wedge z_3 = 3 \leq x_1 < 6 \wedge 1 \leq x_2 \wedge 5 < x_2 - x_1 \leq 7$$

Note that in $z_1$, $x_1 - x_2 \leq 3 \equiv x_2 - x_1 \geq -3$.

Here are some of the zones after some clock resets.

$$z_1[x_1 := 0] = 0 \leq x_1 \leq 0 \wedge 8 \leq x_2$$

$$z_4[x_1 := 0] = 0 \leq x_1 \leq 0 \wedge 1 \leq x_2 \leq 2$$

$$z_4[x_2 := 0] = 1 \leq x_1 \leq 2 \wedge 0 \leq x_2 \leq 0 \wedge$$

$$z_4[\{x_1, x_2\} := 0] = 0 \leq x_1 \leq 0 \wedge 0 \leq x_2 \leq 0$$

and here are some clock zones after some specific time elapses:

$$z_1 + 0 = z_1$$

$$z_1 + 2 = x_1 \geq 5 \wedge 5 \leq x_2 - x_1 \leq 7$$

$$z_4 + 3 = 4 \leq x_1 \leq 5 \wedge 4 \leq x_2 \leq 5 \wedge x_2 - x_1 \geq 0$$

■

Implementations for these operators will be discussed in more detail when the various representations of clock zones are discussed, so the implementations can be representation-specific.

Sometimes in a clock zone, given constraints on one clock and a difference of two clocks, there will be *implicit* constraints on the other clock (or given constraints on two clocks there will be implicit constraints on their differences). These constraints must be persevered even if they are not explicitly written.

Intersection is closed by the definition of the grammar for a clock zone. The closure for the other operators, except for union, is a corollary to the following Claim, proved in Clarke et al. [55].

**Claim 5.7.1** (Lemma 46 of Clarke et al. [55])**.** If $z$ is a clock zone with free clock variable $x$, then $\exists x\colon z$ is also a clock zone.

The above claim can be used on any clock zone $z$. We illustrate it for the reset operator for resetting one clock $x$. By definition, the reset operator converts $z$ to $z[x \mapsto 0]$, which by definition, is the weakest precondition, or $\exists x\colon z[x := 0]$. The above lemma establishes that it is a clock zone.

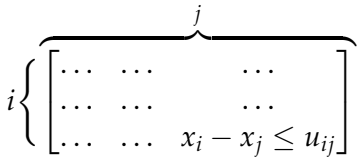


**Figure 5.6:** DBM: a matrix with constraint $x_i - x_j \leq u_{ij}$ in entry $(i, j)$.

## 5.8 Clock Zone Implementations

Clock zones are an abstract data structure. The standard implementation is the **difference bound matrix (DBM)**, a matrix form that stores every clock constraint. To improve performance, we designed two sparser implementations (based off of ideas of others), the sparse linked-list **CRDZone** and the sparse array-list **CRDArray**. Their name comes from their inspiration, the **clock restriction diagram (CRD)**, which is from Wang [154, 155].

### 5.8.1 Difference Bound Matrix (DBM)

The basic way to implement a clock zone is a *difference bound matrix (DBM)*, described in Definition 5.8.1. See Bengtsson and Yi [27], Dill [64] for a more thorough description.

**Definition 5.8.1** (**Difference bound matrix (DBM)**)**.** Let $n - 1$ be the number of clocks. A *DBM* is an $n \times n$ matrix where entry $(i, j)$ is the upper bound of the clock constraint $x_i - x_j$, represented as $x_i - x_j \leq u_{ij}$ or $x_i - x_j < u_{ij}$. The $0^{\text{th}}$ index is reserved for a dummy clock $x_0$, which is always 0, allowing bounds on single clocks to be represented by the clock differences $x_i - x_0$ and $x_0 - x_j$. See Figure 5.6 for a picture of the DBM structure and Example 5.8.1 for a concrete example. ■

### 5.8.2 Alternative Implementations, CRDZone and CRDArray

When implementing clock zones, we can implement them densely, where we store the value of every constraint, or we can implement them sparsely, where we can implicitly store vacuous clock inequalities. One advantage of the dense implementation is that determining the value of any constraint on a single clock difference can be found in O(1) time and that we do not need to store clock indices, since they are implicitly encoded in their location. The advantage of a sparser implementation is that we can store fewer clock constraints and thus traverse through all clock constraints faster. The sparser the clock zone is, the more value there is likely to a sparse implementation.

However, we consider a sparser implementations. Rather than storing the value for every $x_i - x_j$, we can omit storing vacuous constraints (like $x_i - x_i \leq 0$ and $x_i - x_j < \infty$) and thus save space. To do this, we use a linked-list implementation of a clock zone, called a **CRDZone**. The constraints are stored in lexicographical order on $(i, j)$ and vacuous constraints such as $x_i - x_i \leq 0$ and $x_i - x_j < \infty$ are omitted, since they are considered implicit. We can also implement this as a statically allocated array list. This version is the **CRDArray**. Using a dynamic allocation instead of our static allocation for the CRDArray array list is conjectured to save space at the expense of time. An example of the CRDZone and CRDArray is given in Example 5.8.1.

For more details on the DBM, CRDZone and CRDArray, see Fontana and Cleaveland [73].

**Example 5.8.1 (Clock zone in various representations).** Consider the clock zone $z = 1 \leq x_1 < 3 \wedge 0 \leq x_2 \leq 5 \wedge x_2 - x_1 \leq 4$.

**DBM representation** of $z$:

$$\begin{bmatrix} x_0 - x_0 \leq 0 & x_0 - x_1 \leq -1 & x_0 - x_2 \leq 0 \\ x_1 - x_0 < 3 & x_1 - x_1 \leq 0 & x_1 - x_2 < \infty \\ x_2 - x_0 \leq 5 & x_2 - x_1 \leq 4 & x_2 - x_2 \leq 0 \end{bmatrix}$$

**CRDZone representation** of $z$:

$$\boxed{x_0 - x_0 \leq 0} \longrightarrow \boxed{x_0 - x_1 \leq -1} \longrightarrow \boxed{x_0 - x_2 \leq 0}$$

$$\longrightarrow \boxed{x_1 - x_0 < 3} \longrightarrow \boxed{x_2 - x_0 \leq 5} \longrightarrow \boxed{x_2 - x_1 \leq 4}$$

**CRDArray representation** of $z$:

$$[x_0 - x_0 \leq 0 | x_0 - x_1 \leq -1 | x_0 - x_2 \leq 0$$
$$| x_1 - x_0 < 3 | x_2 - x_0 \leq 5 | x_2 - x_1 \leq 4]$$

■

*Remark* 5.8.1 (On DBM vs. CRDZone and CRDArray methods). Due to the sparse implementation and removal of implicit nodes, the CRDZone and CRDArray can improve time by reducing the number of nodes, and thus the number of nodes looked at during a full traversal. This can speed up traversal-based algorithms such as intersect and subset check. However, algorithms like clock reset, emptiness check and canonical form use $O(1)$ access of middle nodes in DBMs (the CRDZone and CRDArray do not have $O(1)$ access for all nodes), resulting in a performance slowdown for those CRDZone and CRDArray methods. For space, the CRDZone and CRDArray can store fewer nodes but must store the explicit indices, resulting in more space per node.

## 5.9 Unions of Clock Zones and More Complex Data Structures

For reachability and many properties, clock zones provide a sufficient abstraction. However, for certain properties including $\vee$, clock zones are not sufficient. This is because clock zones are *convex*, and properties like $x_1 < 2 \vee x_1 \geq 4$ is not convex. To solve this problem, we use a *union of clock zones*. There are some implementations and different data structures out there; three potential implementations are:

1. **Lists of Clock Zones (typically DBMs).** One take represent a clock zone directly as a DBM and represent a union of clock zones as a list of DBMs (or other clock zone implementation). If we have a list of zones $z_1, z_2 \ldots z_n$, it represents the union of zones $z_1 \vee z_2 \vee \ldots \vee z_n$.

2. **Clock Difference Diagrams (CDD) [21, 114]**. The clock difference diagram stores the union of zones as a tree, inspired by a binary decision diagram (BDD) (see Clarke et al. [55]), where each path is a clock zone. This structure is labeled with `tt` and `ff` leaves, where if a valuation reaches a `tt` node then it is in the union of zones. Each node has edges representing an upper bound and a lower bound on some clock difference $x_i - x_j$.

3. **Clock Restriction Diagrams (CRD) [154, 155]**. Similar to a CDD, this structure only stores paths that are in the zone. Also, it only uses upper bound constraints and represents lower bound constraints with a different node.

Behrmann et al. [21], Wang [155] show that the BDD-like tree-like representation of a union of clock zones can result in time and space savings for unions of zones. However, these are quite complex to implement. Thus, for simplicity, we will begin using a list of clock zones for our implementation.

## 5.10 Preliminary Evaluation I: Clock Zone Implementation Performance

We did a preliminary analysis [73] analyzing the time and space performances of the three clock zone implementations.

### 5.10.1 Experimental Setup

We compare the DBM implementation to the CRDZone and CRDArray implementations. Each implementation uses shortest path closure to compute canonical form. The only difference in the DBM, CRDZone and CRDArray versions is the data structure implementation. The benchmark choice was modeled off of Zhang and Cleaveland [167], with the addition of a model of the generalized railroad crossing (GRC) protocol [85]. We also used all the protocols in Zhang and Cleaveland [167], which are the Carrier Sense, Multiple Access with Collision Detection (CSMA/CD), the Fiber Distributed Data Interface (FDDI), Fischer's Mutual Exclusion (FISCHER), the Leader Election protocol (LEADER and LBOUND) and the PATHO Operating System (PATHOS) protocol, where each of these protocols is described some in Zhang and Cleaveland [167]. There are 53 benchmarks that ran on each implementation.

Experiments were run on a Linux machine with a 3.4 GHz Intel Pentium 4 Dual Processor (each a single core) with 4 GB RAM. Time and space measurements (maximum space used) were made using the `memtime` (`http://www.update.uu.se/~johanb/memtime/`) tool [26] (using time `elapsed` and `Max VSize`). The data tables are in Section 5.10.2.

For the experiments, we use three kinds of examples:

- **Valid $A$ Examples (in Tables 5.1 and 5.2):** Correct system implementations with valid safety specifications.

**Table 5.1:** Experiment Results—*A* Examples—Time (s): correct system, correct specification.

| Example | DBM | CRDZone | CRDArray |
|---|---|---|---|
| CSMACD-3-a | **0.10** | 0.20 (200%) | 0.20 (200%) |
| CSMACD-4-a | **3.16** | 4.48 (142%) | 6.50 (206%) |
| FDDI-20-a | **2.04** | 3.03 (149%) | 4.66 (228%) |
| FDDI-40-a | **58.49** | 79.2 (135%) | 126.82 (217%) |
| FDDI-50-a | **169.66** | 230.7 (136%) | 370.71 (219%) |
| MUX-5-a | **1.22** | 2.14 (175%) | 2.75 (225%) |
| MUX-6-a | **35.49** | 74.44 (210%) | 98.08 (276%) |
| MUX-7-a | **2623.61** | 5742.55 (219%) | 7383.73 (281%) |
| LEADER-6-a | **0.41** | 0.71 (173%) | 0.92 (224%) |
| LEADER-7-a | **12.99** | 25.89 (199%) | 34.22 (263%) |
| LBOUND-6-a | **0.51** | 1.02 (200%) | 1.32 (259%) |
| LBOUND-7-a | **17.36** | 37.07 (214%) | 49.64 (286%) |
| PATHOS-4-a | **13.7** | 35.23 (257%) | 50.58 (369%) |
| GRC-3-a | **0.92** | 1.63 (177%) | 2.12 (230%) |
| GRC-4-a | **252.05** | 431.63 (171%) | 748.01 (297%) |

- **Invalid *B* Examples (in Tables 5.3 and 5.4):** *A* examples with invalid specifications.

- **Invalid *C* Examples (in Tables 5.5 and 5.6):** *A* examples with buggy implementations of the systems that do not satisfy the *A* specifications.

### 5.10.2 Experimental Data

The experimental data for the 53 example benchmarks is provided in Tables 5.1, 5.2, 5.3, 5.4, 5.5 and 5.6, with the best entry(ies) in each row **bolded** and percentage change relative to the DBM, to the nearest %, in parenthesis. Time data is given to the nearest 0.01s (second) and space data is given to the nearest 0.01MB (Megabyte). Given the percentage rounding, sometimes an example with slightly different performance may still have a 100% value.

**Table 5.2:** Experiment Results—*A* Examples—Space (MB): correct system, correct specification.

| Example | DBM | CRDZone | CRDArray |
|---|---|---|---|
| CSMACD-3-a | **2.88** | 7.55 (262%) | 11.02 (382%) |
| CSMACD-4-a | 209.97 | **104.47** (50%) | 179.53 (86%) |
| FDDI-20-a | **5.96** | 9.00 (151%) | 13.57 (227%) |
| FDDI-40-a | **27.55** | 57.24 (208%) | 100.30 (364%) |
| FDDI-50-a | **53.91** | 116.79 (217%) | 209.29 (388%) |
| MUX-5-a | 14.57 | **12.73** (87%) | 18.55 (127%) |
| MUX-6-a | **84.05** | 116.35 (138%) | 168.38 (200%) |
| MUX-7-a | **625.42** | 1667.94 (267%) | 2302.39 (368%) |
| LEADER-6-a | **3.57** | 6.59 (185%) | 7.82 (219%) |
| LEADER-7-a | **20.98** | 104.02 (496%) | 133.39 (636%) |
| LBOUND-6-a | **3.93** | 8.66 (220%) | 10.39 (264%) |
| LBOUND-7-a | **27.89** | 157.54 (565%) | 199.99 (717%) |
| PATHOS-4-a | 40.73 | **38.11** (94%) | 57.45 (141%) |
| GRC-3-a | 10.48 | **7.87** (75%) | 11.23 (107%) |
| GRC-4-a | 318.22 | **220.64** (69%) | 355.02 (112%) |

### 5.10.3 Histograms and Descriptive Statistics

Running the different data structure implementations with the same examples yields *paired data*. Hence, we can take the two implementations and pair them example-by-example on their time and space differences to analyze their performance. When we pair the DBM − CRDZone samples, we take the DBM measurement and subtract the CRDZone measurement for the same example to get a DBM − CRDZone paired data point. For instance, the MUX-5-a paired point is -0.92s, 1.94MB, since the DBM point is 1.22s, 14.67MB, and the CRDZone point is 2.14s, 12.73MB. Pairings are likewise done to obtain the paired samples for DBM − CRDArray and CRDZone − CRDArray. For more information, see a Statistics text such as Devore [62].

Tables 5.7, 5.8 and 5.9 contain descriptive statistics on the paired difference in example-by-example performance of the DBM, CRDZone and CRDArray. Figures

**Table 5.3:** Experiment Results—*B* Examples—Time (s): correct system, invalid specification.

| Example | DBM | CRDZone | CRDArray |
|---|---|---|---|
| CSMACD-4-b | **0.10** | **0.10** (100%) | 0.20 (200%) |
| CSMACD-5-b | **0.51** | **0.51** (100%) | 0.71 (139%) |
| CSMACD-6-b | 3.35 | **2.73** (81%) | 3.97 (119%) |
| FDDI-30-b | **1.53** | **1.53** (100%) | 2.23 (146%) |
| FDDI-40-b | 4.66 | **4.58** (98%) | 6.60 (142%) |
| FDDI-60-b | 8.64 | **5.07** (59%) | 5.48 (63%) |
| MUX-20-b | **0.41** | **0.41** (100%) | 0.51 (124%) |
| MUX-30-b | 0.92 | **0.91** (99%) | 1.21 (132%) |
| MUX-40-b | 1.93 | **1.73** (90%) | 2.23 (116%) |
| LEADER-10-b | **0.10** | **0.10** (100%) | **0.10** (100%) |
| LEADER-20-b | **0.10** | 0.20 (200%) | 0.20 (200%) |
| LBOUND-10-b | **0.10** | **0.10** (100%) | 0.20 (200%) |
| LBOUND-40-b | 6.82 | 17.46 (256%) | 29.54 (433%) |
| PATHOS-7-b | **0.10** | **0.10** (100%) | **0.10** (100%) |
| PATHOS-8-b | **0.10** | **0.10** (100%) | **0.10** (100%) |
| PATHOS-9-b | **0.10** | **0.10** (100%) | **0.10** (100%) |
| GRC-3-b | **0.10** | **0.10** (100%) | **0.10** (100%) |
| GRC-4-b | **0.51** | 0.61 (120%) | 0.82 (161%) |
| GRC-5-b | **9.75** | 13.4 (137%) | 19 (195%) |

5.7, 5.8 and 5.9 have histograms that plot the overall time and space differences between the DBM, CRDZone and CRDArray implementations. Bin colors and are changed to help more easily find the -0.001 to 0.001 (equal performance, since our measurement precision is 0.01 units), and -0.25 to -0.001 and 0.001 to 0.25 bins (slight differences).

We do not use 95% confidence intervals, paired two-sample hypothesis ($z$) tests or ANOVA (Analysis of Variance) because the independence assumption of the samples (the example benchmarks) does not hold. Furthermore, we do not use a Wilcoxon signed-rank test for the median because the symmetry assumption of the distribution is not believed to hold, and thus we cannot analyze the hypo-

**Table 5.4:** Experiment Results—*B* Examples—Space (MB): correct system, invalid specification.

| Example | DBM | CRDZone | CRDArray |
|---|---|---|---|
| CSMACD-4-b | **2.88** | 2.89 (100%) | 13.67 (474%) |
| CSMACD-5-b | 144.14 | **72.38** (50%) | 123.52 (86%) |
| CSMACD-6-b | 1134.30 | **553.21** (49%) | 961.90 (85%) |
| FDDI-30-b | 9.60 | **9.06** (94%) | 19.08 (199%) |
| FDDI-40-b | 17.19 | **16.03** (93%) | 39.00 (227%) |
| FDDI-60-b | 27.85 | **14.53** (52%) | 63.52 (228%) |
| MUX-20-b | 19.37 | **11.15** (58%) | 16.96 (88%) |
| MUX-30-b | 28.28 | **16.87** (60%) | 28.58 (101%) |
| MUX-40-b | 43.01 | **21.85** (51%) | 42.81 (100%) |
| LEADER-10-b | **2.88** | 2.89 (100%) | 2.89 (100%) |
| LEADER-20-b | **2.88** | 4.59 (159%) | 5.69 (197%) |
| LBOUND-10-b | **2.88** | 2.89 (100%) | 3.38 (117%) |
| LBOUND-40-b | 18.29 | **15.23** (83%) | 30.73 (168%) |
| PATHOS-7-b | **2.88** | 2.89 (100%) | 2.89 (100%) |
| PATHOS-8-b | **2.88** | 2.89 (100%) | 2.89 (100%) |
| PATHOS-9-b | **2.88** | 2.89 (100%) | 2.89 (100%) |
| GRC-3-b | **2.88** | 2.89 (100%) | 2.89 (100%) |
| GRC-4-b | 58.74 | **32.08** (55%) | 53.42 (91%) |
| GRC-5-b | 717.21 | **379.44** (53%) | 648.00 (90%) |

thetical benchmark distribution referred to in Remark 5.10.1. We do use paired sampling since we have its only requirement—perfect correlation of the samples. More information is in Devore [62].

To get an better idea of the distribution of the benchmark examples themselves and to put some light on the differences, histograms of the time and space used for the DBM implementation to check the example benchmarks are given in Figure 5.10.

### 5.10.4 Analysis of Results

*Remark* 5.10.1 (On our analysis approach). We ask: *what does it mean for an implementation to perform better than another?* **We consider consider *better* to be mea-**

**Table 5.5:** Experiment Results—*C* Examples—Time (s): buggy system, correct specification.

| Example | DBM | CRDZone | CRDArray |
|---|---|---|---|
| CSMACD-6-c | 0.51 | **0.41** (80%) | 0.51 (100%) |
| CSMACD-7-c | 2.03 | **1.82** (90%) | 2.03 (100%) |
| CSMACD-8-c | 9.55 | **8.42** (88%) | 9.55 (100%) |
| FDDI-30-c | 0.51 | **0.41** (80%) | 0.41 (80%) |
| FDDI-40-c | 1.52 | **0.92** (61%) | 1.01 (66%) |
| FDDI-60-c | 6.71 | **3.98** (59%) | 4.17 (62%) |
| MUX-6-c | **139.02** | 258.32 (186%) | 401.84 (289%) |
| LEADER-60-c | 6.81 | **3.96** (58%) | 4.06 (60%) |
| LEADER-70-c | 14.42 | **8.12** (56%) | 8.13 (56%) |
| LEADER-100-c | 82.94 | **45.78** (55%) | 45.88 (55%) |
| LBOUND-6-c | **0.10** | **0.10** (100%) | 0.20 (200%) |
| LBOUND-7-c | **0.61** | 0.81 (133%) | 1.12 (184%) |
| LBOUND-8-c | **12.48** | 32.00 (256%) | 52.9 (424%) |
| PATHOS-5-c | **0.10** | **0.10** (100%) | **0.10** (100%) |
| PATHOS-6-c | **0.10** | **0.10** (100%) | **0.10** (100%) |
| PATHOS-7-c | **0.10** | **0.10** (100%) | **0.10** (100%) |
| GRC-3-c | **0.10** | **0.10** (100%) | **0.10** (100%) |
| GRC-4-c | **0.51** | 0.81 (159%) | 1.02 (200%) |
| GRC-5-c | **9.65** | 13.31 (138%) | 18.88 (196%) |

**sured in the number (or percentage) of examples that one system outperforms another in.** The larger aim is for any implementation, if we were to know all the examples that it would run (including and beyond the experiment examples), we would *like* one implementation to perform (strictly) better for at least 51% of this hypothetical set. This influences our analysis.

Given our meaning of *better* in Remark 5.10.1, we consider the median, 25% and 75% percentile values as *insights* into typical examples and use the histograms to get a bigger picture of the sample distribution of the performance differences for the experiment, and weigh these more heavily than the mean and standard deviation values. The mean and the standard deviation provide us with an alter-

**Table 5.6:** Experiment Results—*C* Examples—Space (MB): buggy system, correct specification.

| Example | DBM | CRDZone | CRDArray |
|---|---|---|---|
| CSMACD-6-c | 85.32 | **18.53** (22%) | 79.30 (93%) |
| CSMACD-7-c | 337.43 | **191.84** (57%) | 320.89 (95%) |
| CSMACD-8-c | 1369.07 | **787.75** (58%) | 1338.62 (98%) |
| FDDI-30-c | 4.88 | **4.60** (94%) | 9.93 (203%) |
| FDDI-40-c | 9.55 | **6.41** (67%) | 19.63 (206%) |
| FDDI-60-c | 24.07 | **14.24** (59%) | 55.57 (231%) |
| MUX-6-c | 1607.73 | **1047.64** (65%) | 1723.25 (107%) |
| LEADER-60-c | 29.24 | **10.79** (37%) | 63.66 (218%) |
| LEADER-70-c | 51.18 | **15.30** (30%) | 108.80 (213%) |
| LEADER-100-c | 203.89 | **40.65** (20%) | 431.71 (212%) |
| LBOUND-6-c | **2.88** | 2.89 (100%) | 4.48 (155%) |
| LBOUND-7-c | 12.52 | **10.34** (83%) | 14.42 (115%) |
| LBOUND-8-c | 75.66 | **59.23** (78%) | 90.48 (120%) |
| PATHOS-5-c | **2.88** | 2.89 (100%) | 2.89 (100%) |
| PATHOS-6-c | **2.88** | 2.89 (100%) | 2.89 (100%) |
| PATHOS-7-c | **2.88** | 2.89 (100%) | 2.89 (100%) |
| GRC-3-c | **2.88** | 2.89 (100%) | 2.89 (100%) |
| GRC-4-c | 58.74 | **35.94** (61%) | 59.47 (101%) |
| GRC-5-c | 717.29 | **379.35** (53%) | 647.94 (90%) |

native picture of the overall performance and give hints of either a unusual sample distribution (since in a symmetric distribution the mean equals the median) or the presence of potential outliers.

**DBM vs. CRDZone**

The CRDZone performs slower for 45% of the tested examples (at least as slow for 74%) with a median difference of 0.00s slower, while the CRDZone has a mean difference of 67.55s slower. Thus, we infer the CRDZone is either slightly slower or competitive to the DBM for this benchmark, but due to insufficient evidence (45% of the examples is not enough) do not infer that the DBM performs strictly faster than the CRDZone.

**Table 5.7:** Descriptive Statistics for paired DBM − (minus) CRDZone examples, for time (s) and space (MB).

| **Statistic** | **DBM − CRDZone (Time)** | **DBM − CRDZone (Space)** |
|---|---|---|
| Mean | -67.55 | 34.96 |
| Standard Deviation | 428.35 | 212.65 |
| 25% Percentile | -1.24 | 0.00 |
| Median | 0.00 | 1.85 |
| 75% Percentile | 0.06 | 25.70 |

**Table 5.8:** Descriptive Statistics for paired DBM − (minus) CRDArray examples, for time (s) and space (MB).

| **Statistic** | **DBM − CRDArray (Time)** | **DBM − CRDArray (Space)** |
|---|---|---|
| Mean | -112.95 | -47.75 |
| Standard Deviation | 655.65 | 235.63 |
| 25% Percentile | -3.16 | -20.54 |
| Median | -0.29 | -2.81 |
| 75% Percentile | 0.00 | -0.01 |

The CRDZone takes less space for 57% of the tested examples (at most as much space for 57%) with a median amount of 1.85MB less space and a mean amount of 34.96MB less space. The CRDZone takes at least 0.25MB less space for 28 such examples and more than 0.25MB space for only 11 examples. Thus (even though 57% is not a large majority), we infer the CRDZone takes less space overall for this benchmark.

#### DBM vs. CRDArray

The CRDArray performs slower for 64% of the tested examples (at least as slow for 87%) with a median difference of 0.29s slower and a mean difference of 112.95s slower. Thus we infer the CRDArray performs slower overall for this benchmark.

The CRDArray takes more space for 79% (at least as much space for 79%) of

**Table 5.9:** Descriptive Statistics for paired CRDZone − (minus) CRDArray examples, for time (s) and space (MB).

| **Statistic** | **CRDZone − CRDArray (Time)** | **CRDZone − CRDArray (Space)** |
|---|---|---|
| Mean | -45.40 | -82.71 |
| Standard Deviation | 229.06 | 160.91 |
| 25% Percentile | -2.02 | -52.67 |
| Median | -0.21 | -19.35 |
| 75% Percentile | -0.03 | -1.63 |

the examples with a median amount of 2.81MB more space and mean amount of 47.75MB more. Thus we infer the CRDArray takes more space overall for this benchmark.

#### CRDZone vs. CRDArray

The CRDArray performs slower for 77% of the tested examples (at least as slow for 100%) with a median difference of 0.21s slower and a mean difference of 45.40s slower. Thus we infer the CRDArray is slower overall for this benchmark.

The CRDArray takes more space for 100% of the examples with a median amount of 19.35MB more space and a mean amount of 82.71MB more. Thus we infer the CRDArray takes more space overall for this benchmark.

### 5.10.5 Conclusions

Here are the conclusions:

1. **Time:** (DBM $\leq_t$ CRDZone) $<_t$ CRDArray). For this benchmark, we infer that the DBM is either competitive with or slightly faster than the CRDZone and both perform faster than the CRDArray. There is insufficient evidence to conclude that the DBM is strictly faster.

2. **Space:** (CRDZone $<_s$ DBM) $<_s$ CRDArray. For this benchmark, we infer

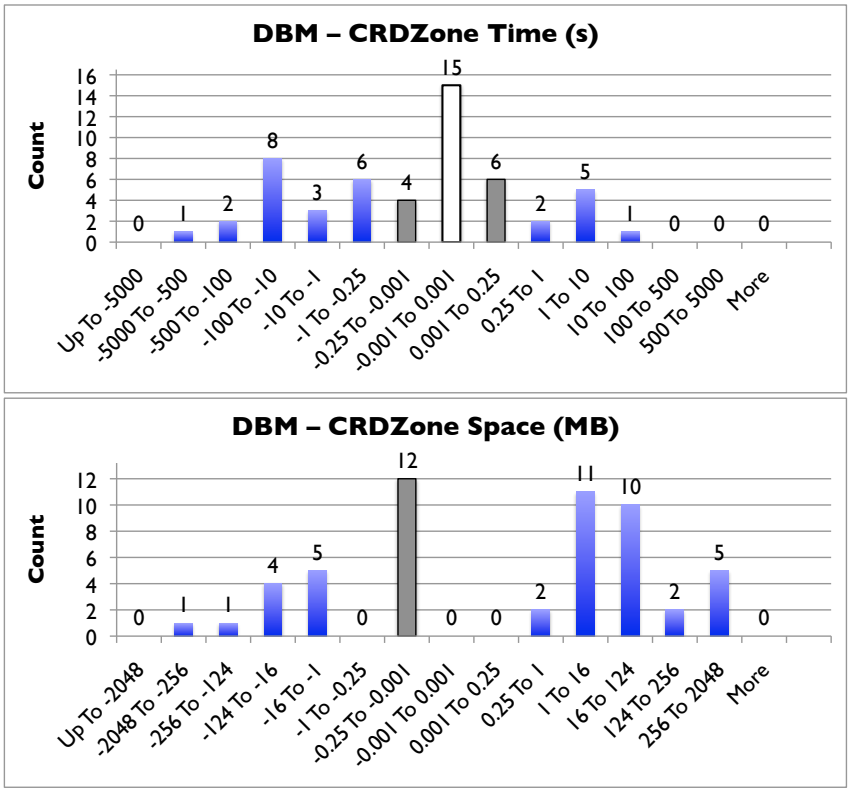


**Figure 5.7:** Histograms comparing the DBM − (minus) CRDZone time (s) (top) and space (MB) (bottom) differences.

that the CRDZone takes the least amount of space and the DBM takes less space than the CRDArray for this experiment.

*Remark* 5.10.2 (On DBM vs. CRDZone and CRDArray methods). Due to the sparse implementation and removal of implicit nodes, the CRDZone and CRDArray can improve time by reducing the number of nodes, and thus the number of nodes looked at during a full traversal. This can speed up traversal-based algorithms such as intersect and subset check. However, algorithms like clock reset, emptiness check and canonical form use $O(1)$ access of middle nodes in DBMs (the CRDZone and CRDArray do not have $O(1)$ access for all nodes), resulting in a performance slowdown for those CRDZone and CRDArray methods. For space, the CRDZone

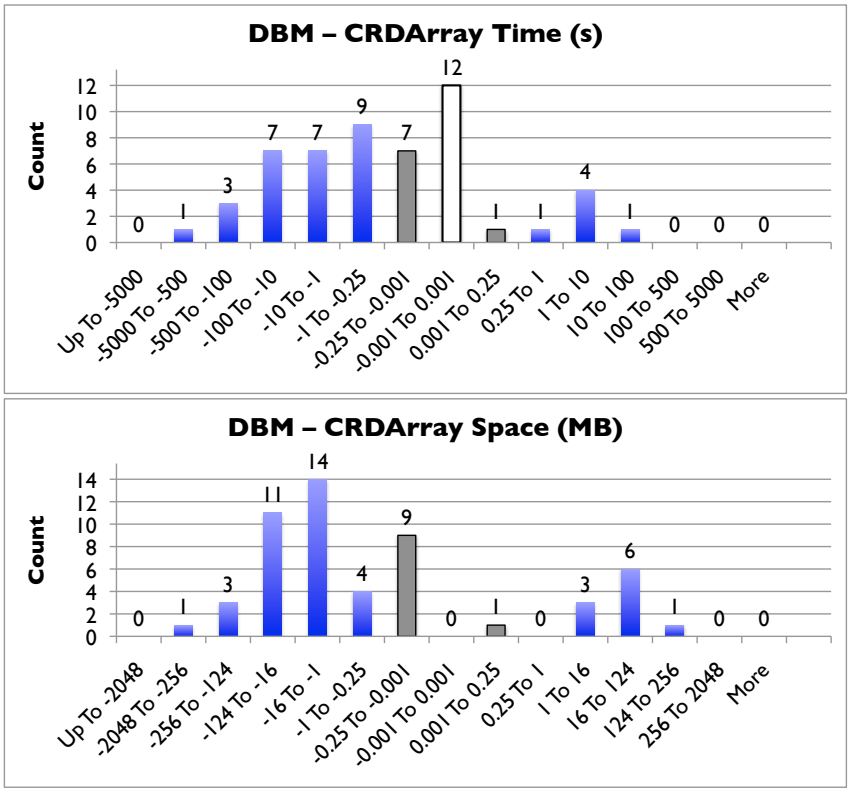


**Figure 5.8:** Histograms comparing the DBM − (minus) CRDArray time (s) (top) and space (MB) (bottom) differences.

and CRDArray can store fewer nodes but must store the explicit indices, resulting in more space per node.

See Fontana and Cleaveland [73] for more information.

## 5.11 Preliminary Evaluation II: PES Tool Implementation

We present the results of a experimental evaluation of our method that demonstrates the types of timed automata and specifications the system can model check. Furthermore, on the subset of specifications that UPPAAL supports, we compare our tool's time performance to their tools's time performance.

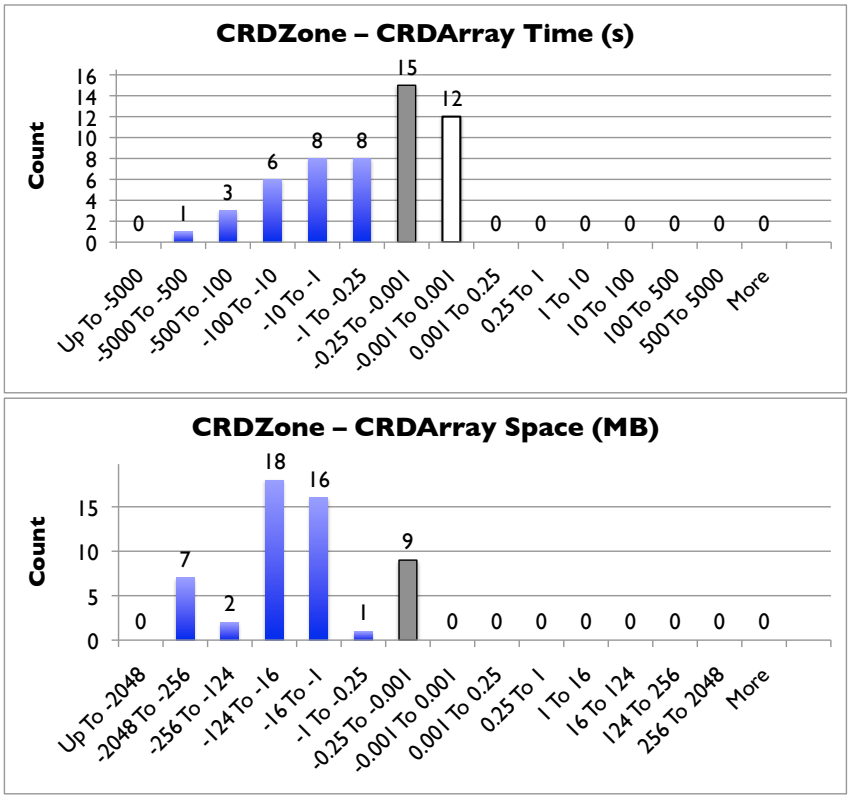


**Figure 5.9:** Histograms comparing the CRDZone − (minus) CRDArray time (s) (top) and space (MB) (bottom) differences.

Experiments were run on a Mac OS 10.7 machine with a single 2.0 GHz Intel Core i7 (quad core) processor with 8 GB RAM. Time and space measurements (maximum space used) were made using the UNIX `time` command (using the *real time* as the output time).

### 5.11.1 Methods: Evaluation Design

In our case study, we use four different models: Carrier Sense, Multiple Access with Collision Detection (CSMA); Fischer's Mutual Exclusion (FISCHER); Generalized Railroad Crossing (GRC); and Leader election (LEADER). Here is a brief description of them:

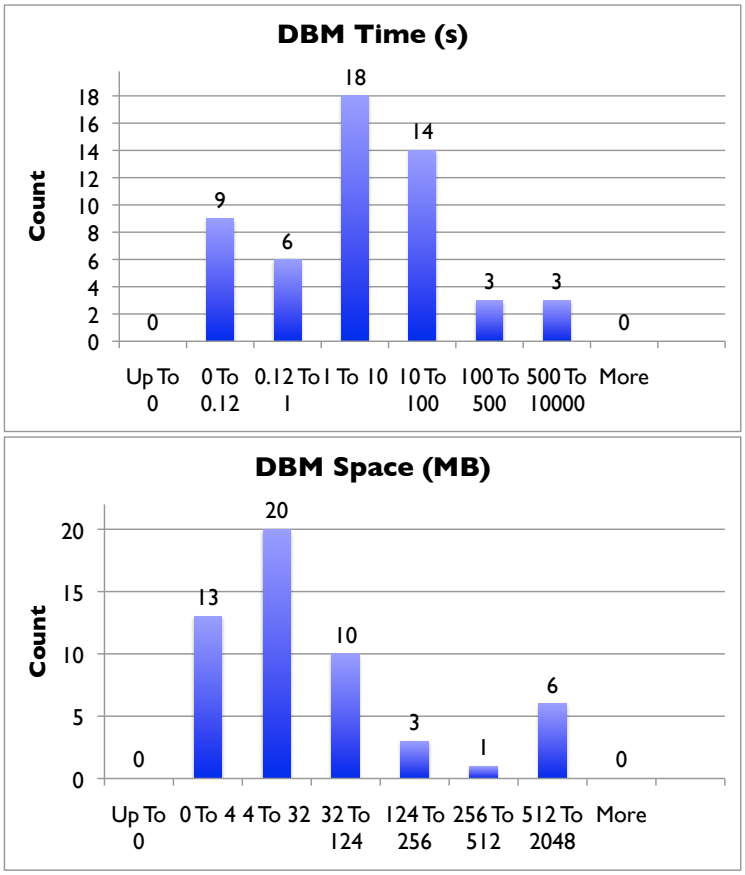


**Figure 5.10:** Histograms illustrating the DBM Time (s) (top) and Space (MB) (bottom) distributions.

1. *Carrier Sense, Multiple Access with Collision Detection (CSMA).* There are $n$ processes sharing who one bus. The bus can only send one message at a time. At various times processes will try to transmit a message. If the process detects that the bus is busy, then the process will wait a random amount of time before retrying.

2. *Fischer's Mutual Exclusion (FISCHER)* This protocol involves $n$ processes vying for access to a critical section. Each process asks for the critical section and then waits until it gets it, re-requesting for access if it is not granted it for a period of time. The critical section identifies which process currently

has access to it.

3. *Generalized Railroad Crossing (GRC).* This protocol has $n$ trains, a gate and a controller. The trains cross a region that intersects a road, and the gate goes down to prevent cars from driving on the road when a train is passing through. When no train is nearby, the gate raises or remains up to allow cars to safely drive through.

4. *Leader election (LEADER).* This protocol involves involves $n$ processes that are electing a leader amongst themselves. To elect a leader, at each step one process asks another process to be its parent. In our model, the smaller-numbered process always becomes the parent. When finished, the process with no parent is the leader.

For more information on these models, see Heitmeyer and Lynch [84], Zhang and Cleaveland [167, 168].

For each model, we start at 4 processes and scale the model up by adding more processes (up to 8 processes). For each model we model-checked one valid safety specification (*as*), one invalid safety specification (*bs*), one valid liveness specification (*al*), and one invalid liveness specification (*bl*). Each of these cases involves only one temporal operator: $\phi_1$ involves conjunctions and disjunctions of atomic propositions and clock constraints. In addition we tested 4 additional specifications on each property ($M1$, $M2$, $M3$, and $M4$). Out of these specifications, at least one (usually $M4$) is a property with no known equivalent TCTL formula. The specifications checked are listed below. The specifications that are not supported by UPPAAL are in *italics* and are marked with a $^*$.

The specifications checked on the CSMA protocol are:

- ***AS***:* *At most one process is in a transmission state for less than 52 ($2\sigma$) units.*

*(Valid)*

- **BS:** At any time, a third process can retry while two are already in transmission status. (Invalid)
- **AL:** It is inevitable that all processes are waiting. (Valid)
- **BL:** It is inevitable that some process needs to retry transmitting a message. (Invalid)
- **M1:** It is always the case that if the first process needs to retry that it will inevitably transmit. (Invalid)
- **M2:** It is always the case that if a bus experiences a collision that it will inevitably become idle. (Valid)
- ***M3***:* *The bus is always idle until a process is active. (Invalid)*
- ***M4***:* *For all paths with an infinite number of actions, the bus is always idle until a process is active (Valid)*

The specifications checked on the FISCHER protocol are:

- **AS:** At any time, at most one process is in the critical section. (Valid)
- **BS:** At any moment, at most four processes in their waiting state at the same time. (Valid for four processes, Invalid for five or more processes)
- **AL:** It is inevitable that all processes are idle. (Valid)
- **BL:** It is inevitable that some process accesses the critical section. (Invalid)
- **M1:** It is always the case that if the first process is not idle, it will eventually access the critical section. (Invalid)

- **M2:** It is always the case that if the third process is not idle, it will eventually access the critical section. (Invalid)
- ***M3***:* *It is possible for the first process to enter the critical section without waiting. (Invalid)*
- ***M4***:* *After at most five action transitions, some process will enter the critical section. (Invalid)*

The specifications checked on the GRC protocol are:

- **AS:** It is always the case that if at least one train (process) is in the track region, the gate is always down. (Valid)
- **BS:** It is always the case that if the gate is raising then the controller (when one train is approaching or in) will not want to lower the gate. (Invalid)
- **AL:** It is inevitable that the gate is up. (Valid)
- **BL:** It is inevitable that the train is near the gate. (Invalid)
- **M1:** It is always the case that if the gate is down, then it will inevitably come up (Invalid).
- ***M2***:* *It is always the case that if the gate is down, then it will inevitably come up after 30 seconds (Invalid).*
- **M3:** It is always the case that at most one train is in the region at one time (Invalid).
- ***M4***:* *For all paths with an infinite number of actions, the gate is up until a train approaches (Valid).*
- ***M4ap***:* *For all paths, the gate is up until a train approaches (Invalid).*

The specifications checked on the LEADER protocol are:

- **AS:** At any time, each process either has no parent or has a parent with a smaller process id (and thus the first process has no parent at all times). (Valid)
- **BS:** At any moment, at least three processes do not have parents. (Invalid)
- **AL:** It is inevitable that the first process is elected the leader. (Valid)
- **BL:** It is inevitable that the third processes' parent is the second process. (Invalid)
- ***M1***:* *For all paths, a the second process cannot have a child until it has a parent. (Invalid)*
- **M2:** It is always the case that if the third process is assigned a parent (chosen to not be leader), then it will not be the leader. (Valid)
- ***M3***:* *It is possible that it takes longer than 3 time units to elect a leader. (Valid)*
- ***M4***:* *For all paths, in at most three votes, a leader is elected. (Valid for four or fewer processes, invalid for five or more processes.)*

The experiments were run on an Intel Mac with 8GB ram and a quad-core 2 GHz Intel Core i7 processor running OS 10.7. Times were measured with the UNIX utility `time`.

### 5.11.2 Data and Results

The data is provided in Tables 5.10, 5.11, and 5.12. Table 5.10 contains the remaining specifications that are not supported by UPPAAL. Tables 5.11 and 5.12 (split due to horizontal space constraints) contain the examples that are supported both

**Table 5.10:** Examples that UPPAAL does not support. All times are in seconds (s).

| File | PES4 | PES5 | PES6 | PES7 | PES8 |
|---|---|---|---|---|---|
| CSMA-as | 0.29 | 4.62 | 139.16 | 6696.08 | TO |
| CSMA-M3 | 0.01 | 0.03 | 0.14 | 0.80 | 3.99 |
| CSMA-M4 | 0.01 | 0.03 | 0.14 | 0.71 | 3.66 |
| FISCHER-M3 | 0.14 | 2.51 | 79.17 | TO | TOsm |
| FISCHER-M4 | 0.00 | 0.00 | 0.00 | 2.04 | 2.42 |
| GRC-M2 | 0.01 | 0.01 | 0.01 | 0.02 | 0.03 |
| GRC-M4 | 0.00 | 0.00 | 0.01 | 0.02 | 0.01 |
| GRC-M4ap | 0.00 | 0.00 | 0.01 | 0.01 | 0.01 |
| LEADER-M1 | 0.00 | 0.00 | 0.00 | 0.01 | 0.01 |
| LEADER-M3 | 0.01 | 0.08 | 2.12 | 79.05 | 4242.97 |
| LEADER-M4 | 0.00 | 0.00 | 0.04 | 0.03 | 0.01 |

by our tool (PES) and by UPPAAL (UPP). In these three tables, we use the following abbreviations: TO (timeout: the example took longer than 2 hours), TOsm (the example timed out with fewer process), and O/M (out of memory). Since our tool supports a superset of the specifications that UPPAAL can support, there are specifications that UPPAAL supports that our tool does not. A scatter plot of the data in Tables 5.11 and 5.12 is given in Figure 5.11. In that figure, any example with O/M, TO or TOsm had its time set to 7200s (2 hours).

### 5.11.3 Analysis and Discussion

After analyzing the data, we conclude three points. First, on the examples that both our PES tool and UPPAAL support, we acknowledge that UPPAAL's performance is faster than ours; however, our tool performs faster on some examples. Additionally, while our tool does time out more often than UPPAAL does, most examples are verified quickly by both tools.

Second, our tool can quickly verify specifications that UPPAL cannot. We suggest this after noticing that our tool verifies most of the examples in Table 5.10 quickly, and that every specification in Table 5.10 is not supported by UPPAAL.

**Table 5.11:** Time performance in seconds (s) on examples comparing PES and UPPAAL (Table 1 of 2).

| File | PES4 | UPP4 | PES5 | UPP5 | PES6 | UPP6 |
|---|---|---|---|---|---|---|
| CSMA-al | 0.01 | 1.45 | 0.03 | 0.24 | 0.13 | 0.25 |
| CSMA-bl | 0.01 | 0.26 | 0.03 | 0.27 | 0.13 | 0.27 |
| CSMA-bs | 0.01 | 0.33 | 0.05 | 0.27 | 0.22 | 0.27 |
| CSMA-M1 | 0.01 | 0.29 | 0.03 | 0.27 | 0.14 | 0.28 |
| CSMA-M2 | 0.33 | 0.35 | 5.21 | 7.00 | 154.56 | 1194.74 |
| FISCHER-al | 0.00 | 0.51 | 0.00 | 0.27 | 0.00 | 0.28 |
| FISCHER-as | 0.07 | 0.27 | 0.51 | 0.28 | 13.44 | 0.67 |
| FISCHER-bl | 0.00 | 0.26 | 0.00 | 0.26 | 0.00 | 0.28 |
| FISCHER-bs | 0.04 | 0.28 | 0.01 | 0.27 | 0.02 | 0.32 |
| FISCHER-M1 | 0.00 | 0.26 | 0.00 | 0.26 | 0.00 | 0.28 |
| FISCHER-M2 | 0.00 | 0.26 | 0.00 | 0.26 | 0.00 | 0.27 |
| GRC-al | 0.00 | 0.27 | 0.01 | 0.28 | 0.47 | 0.59 |
| GRC-as | 53.09 | 0.36 | TO | 7.11 | TOsm | 940.51 |
| GRC-bl | 0.00 | 0.27 | 0.00 | 0.27 | 0.01 | 0.27 |
| GRC-bs | 0.11 | 0.41 | 1.91 | 0.41 | 433.59 | 1.76 |
| GRC-M1 | 0.01 | 0.27 | 0.04 | 0.27 | 0.01 | 0.29 |
| GRC-M3 | 0.00 | 0.27 | 0.00 | 0.31 | 0.01 | 0.56 |
| LEADER-al | 0.00 | 0.28 | 0.01 | 0.33 | 0.17 | 4.30 |
| LEADER-as | 0.00 | 0.27 | 0.01 | 0.27 | 0.22 | 0.33 |
| LEADER-bl | 0.00 | 0.28 | 0.00 | 0.27 | 0.01 | 0.28 |
| LEADER-bs | 0.00 | 0.27 | 0.00 | 0.28 | 0.01 | 0.28 |
| LEADER-M2 | 0.00 | 0.28 | 0.02 | 0.31 | 0.38 | 3.05 |

Third, we noticed that the for these examples, the performance bottleneck seems be safety properties. Even with the additional complexity of supporting the more complicated specifications (in both tables), they were often verified more quickly than safety properties. Here is one possible explanation: while a verifier must check the entire state space for a valid safety property, for a liveness property, often only a subset of the state space must be checked. When a liveness property is true, once the desired state is found in each path, the remainder of the path need not be explored. Conversely, when a liveness property is invalid, only one path needs to be explored.

**Table 5.12:** Time performance in seconds (s) on examples comparing PES and UPPAAL (Table 2 of 2).

| File | PES7 | UPP7 | PES8 | UPP8 |
|---|---|---|---|---|
| CSMA-al | 0.72 | 0.26 | 3.65 | 0.26 |
| CSMA-bl | 0.73 | 0.28 | 3.53 | 0.33 |
| CSMA-bs | 1.14 | 1.33 | 5.09 | 4.66 |
| CSMA-M1 | 0.73 | 0.27 | 3.69 | 0.27 |
| CSMA-M2 | TO | TO | TOsm | TOsm |
| FISCHER-al | 0.00 | 0.40 | 0.00 | 0.27 |
| FISCHER-as | 864.04 | 0.96 | TO | 4.26 |
| FISCHER-bl | 0.00 | 0.34 | 0.00 | 0.26 |
| FISCHER-bs | 0.39 | 0.47 | 0.39 | 0.90 |
| FISCHER-M1 | 0.00 | 0.28 | 0.00 | 0.25 |
| FISCHER-M2 | 0.00 | 0.30 | 0.03 | 0.28 |
| GRC-al | 0.07 | 0.44 | 0.08 | 5.45 |
| GRC-as | TOsm | 3433.14 | TOsm | TO |
| GRC-bl | 0.01 | 0.61 | 0.01 | 0.66 |
| GRC-bs | O/M | 16.19 | O/M | 52.03 |
| GRC-M1 | 0.05 | 0.35 | 0.03 | 0.32 |
| GRC-M3 | 0.04 | 1.23 | 0.01 | 3.85 |
| LEADER-al | 5.80 | 747.82 | 573.84 | TO |
| LEADER-as | 6.23 | 0.86 | 649.52 | 8.21 |
| LEADER-bl | 0.17 | 0.32 | 4.25 | 0.29 |
| LEADER-bs | 0.03 | 4.99 | 0.40 | 1.57 |
| LEADER-M2 | 13.53 | 504.89 | 1570.37 | TO |

## 5.12 Dissertation Contributions

### 5.12.1 Contributions

These are my contributions discussed in this chapter:

- Fine-tuned the PES Model Checker, and specialized it to model check $L^{af}_{\nu,\mu}$ formulas over timed automata
- Created the clock zone implementations CRDZone and CRDArray, which can be seen as alternative sparse DBM implementations.

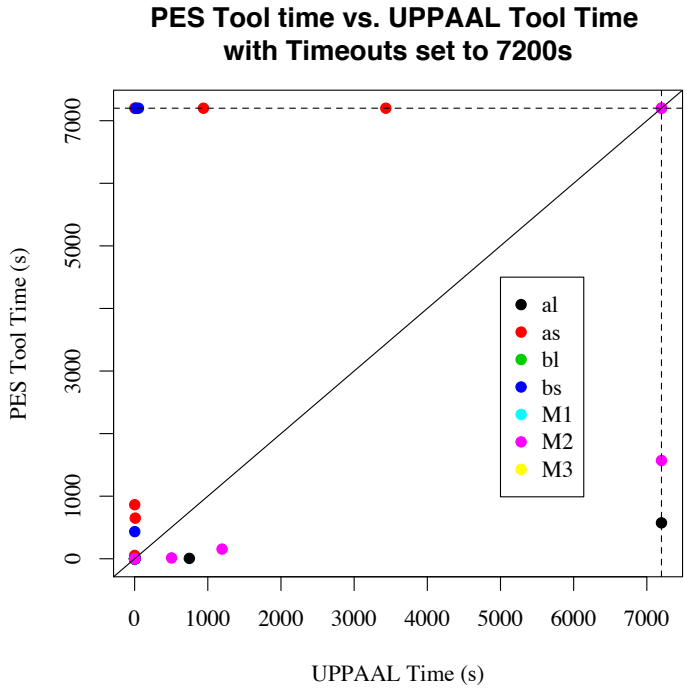


**Figure 5.11:** Figure comparing the PES tool time performance with UPPAAL time performance. Points are colored by the specification category. All timed out (TO) examples or examples that ran out of memory (O/M) have their time set to 7200s, the value of the dashed lines.

- Implemented the CRDZone and CRDArray as well as fined-tuned the DBM implementation, and ran an experiment comparing the performance of the current PES tool on all three data structure implementation.
- Implemented the previously-developed proof rules for $L^{af}_{\nu,\mu}$.
- Design sound, complete and implementable additional proof rules to give a full set of proof rules for $L^{rel,af}_{\nu,\mu}$, as well as proved those rules to be sound and complete.
- Implemented the timed automata model checker to model check any $L^{rel,af}_{\nu,\mu}$

formula over any timed automata.

- Utilized derived proof rules to optimize the performance of this model checker.

### 5.12.2 Future Work

Future work is to further optimize the performance. One such future work is to utilize different standard forms (which other sources have designed considered) and to implement a more modern all-pairs shortest path algorithm; these should help the performance of the tool.

# Chapter 6

# Timed Vacuity in Model Checking

The typical model checking tool gives a yes or no answer when asked if a program satisfies a property. This is the current state with timed automata model checking (some tools generate counterexamples), but we would like get more information from the model checker.

Once such type of information is the identification of formulas that are satisfied vacuously: examples include formulas containing an if-then statement that the model satisfies but satisfies by always having the "if" premise false. For instance, consider the formula $AG[p \rightarrow AF[q]]$; this formula is vacuously satisfied if it is always true but $p$ is always false. This vacuity becomes useful, since the formula $AG[p \rightarrow AF[q]]$ is asked with the intent that $p$ will be sometimes true. If this formula is satisfied such that $AF[q]$ is vacuous, we know that $p$ is never true and have found a bug in our model. Previous work has been done to identify vacuity for untimed systems over untimed logics, including the untimed modal mu-calculus [20, 66]. We extend this research to support timed vacuity over timed automata, providing both the theory and a preliminary implementation. We also leverage Namjoshi [128, 129], which uses a proof to gain understanding of vacuity. By extending the work of Beer et al. [20], Dong et al. [66], Namjoshi [128, 129] to timed automata, we are able to identify some vacuous formulas without any increase in the amount of time or space needed to model check $L_{\nu,\mu}^{rel,af}$ formulas. If we allow for some performance delay, we can identify all vacuous subformulas within an $L_{\nu,\mu}^{rel,af}$ formula.

## 6.1 Vacuity: Definitions

We take many of our definitions from Beer et al. [20]. These definitions involve a formula $\phi$ in some logic and over a model $M$. In this dissertation, we will often use that $\phi$ is a logical formula in some logic and that the model $M$ is some state $q$. In the untimed setting, $q$ will be a state of a transition system $TS$, and in the timed setting, $q = (l, \nu)$ will be a state in a timed automaton $TA$. Also, when examining the satisfaction of logical formulas in the timed setting, we might extend a state with freeze quantification $\nu_f$ to form an extended state $q = (l, \nu, \nu_f)$. When clear from context, we will omit the $\nu_f$. This concept of an extended state is in Section 4.2. Furthermore, recall that a timed automaton satisfies a formula if and only if its initial state satisfies a formula. Although we often use states as our models, because some important results concerning vacuity are model independent, we give most definitions with respect to an arbitrary model $M$.

### 6.1.1 Vacuous Formulas

**Definition 6.1.1** (**Affect [20]**). A subformula $\psi$ *affects* logic formula $\phi$ with respect to a model $M$ iff there exists a formula $\psi'$ such that $M \models \phi$ if and only if $M \models \phi[\psi \mapsto \psi']$. ■

This means that if we replace a subformula $\psi$ with some other formula $\psi'$ (including `tt` or `ff`) and $\psi'$ changes the truth of $\phi$, then that subformula $\psi$ is relevant to the satisfaction of $\phi$. If that is the case, then we say that $\psi$ affects $\phi$.

In this definition, $\psi$ is an instance of a sub formula. When one asks if $\psi$ affects $\phi$, the definition determines whether a specific instance of $\psi$ affects $\phi$.

**Definition 6.1.2** (**Vacuous formula [20]**). A formula $\phi$ is *vacuous* for model $M$. iff there is a subformula $\psi$ such that $\psi$ does not affect $\phi$ in $M$.

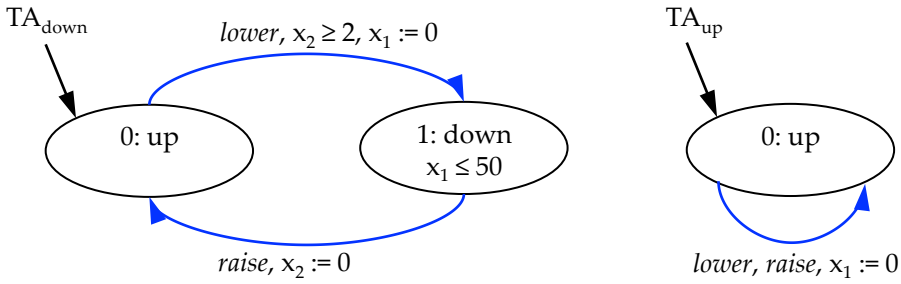


**Figure 6.1:** Two models of a gate, $TA_{down}$ and $TA_{up}$, illustrate that some properties can be satisfied in different ways.

If the formula $\phi$ is true in $M$ and is vacuous, we say that $\phi$ is *vacuously satisfied by* $M$. Likewise, if $\phi$ is false in $M$ and is vacuous, we say that $\phi$ is *vacuously unsatisfied by* $M$. If $\psi$ is a sub formula of $\phi$ and $\psi$ does not affect $\phi$ with respect to $M$, then we say that $\psi$ is a *vacuous subformula* of $\phi$. ■

**Example 6.1.1 (Illustrating vacuous satisfaction with gates).** Consider the two models for an up-down in Figure 6.1, $TA_{down}$ and $TA_{up}$. Notice that the second model has a major glitch that the first does not: the gate is never down in $TA_{up}$ (the first model is also imperfect but does not have this bug).

Consider the property: "It is always the case that if the gate is down then it will inevitably be raised up," written as $AG\,[down \rightarrow AF\,[up]]$ in TCTL (or CTL). *Both* $TA_{down}$ and $TA_{up}$ satisfy this property. However, $TA_{down}$ gives a much more "satisfying" solution. For $TA_{down}$, there is a path where the gate is lowered down, then raised up (by force due to the invariant), and the switching repeats over time. This path illustrates the desired property. However, in $TA_{up}$ the property is still true because the gate is never down. Here, $TA_{up}$ vacuously satisfies the formula, and the subformula $AF\,[up]$ is vacuous because it is never checked. ■

### 6.1.2 Polarity

Given a transition system $TS$ or timed automaton $TA$, consider the complete lattice whose elements are subsets of states of the transition system or of the timed automaton. If we consider the logical operators to be functions on this lattice, we can formalize the notion of *polarity*. If we consider timed automata states as models, we would denote the set of satisfying states as $[\![\phi]\!]_{TA}$. Using a generic model $M$, we will denote the satisfaction set as $[\![\phi]\!]_M$.

**Definition 6.1.3 (Polarity [20]).** Let $\sigma$ be an n-ary operator in some logic, and let $\psi_1$, $\psi_2$ be two formulas. The *$i^{th}$ operand of $\sigma$ has positive polarity* iff for every formula $\phi_1, \ldots \phi_{i-1}$, $\phi_{i+1}, \ldots \phi_n$, if $[\![\psi_1]\!]_M \subseteq [\![\psi_2]\!]_M$, then:

$$[\![\sigma(\phi_1, \ldots \phi_{i-1}, \psi_1, \phi_{i+1}, \ldots \phi_n)]\!]_M \subseteq [\![\sigma(\phi_1, \ldots \phi_{i-1}, \psi_2, \phi_{i+1}, \ldots \phi_n)]\!]_M$$

Likewise, the *$i^{th}$ operand of $\sigma$ has negative polarity* iff for every formula $\phi_1, \ldots \phi_{i-1}$, $\phi_{i+1}, \ldots \phi_n$, if $[\![\psi_2]\!]_M \subseteq [\![\psi_1]\!]_M$, then:

$$[\![\sigma(\phi_1, \ldots \phi_{i-1}, \psi_1, \phi_{i+1}, \ldots \phi_n)]\!]_M \subseteq [\![\sigma(\phi_1, \ldots \phi_{i-1}, \psi_2, \phi_{i+1}, \ldots \phi_n)]\!]_M$$

An operator has *positive polarity* if every one of its operands has positive polarity. Likewise, an operator has *negative* polarity iff ever one of its operands has negative polarity. ■

The idea of positive polarity is that for each of the $i$ subformulas within a logical operator, if we fix all but one of those subformulas and if the remaining subformula is enlargened to satisfy additional states, then the superformula satisfies additional states. In this definition, each logical operation is denoted as a generic logcial operator $\sigma$. One such logical operator is conjunction; $\phi_1 \wedge \phi_2$

would notated as $\wedge(\phi_1, \phi_2)$.

**Definition 6.1.4** (**Logic with polarity**). We say that a *logic has polarity* if every operator in that logic has either positive or negative polarity. ■

For a logic to have polarity, it is allowable that some operators have positive polarity and other operators have negative polarity. Typically most operators have positive polarity, while negation has negative polarity

We can also define the polarity of a subformula.

**Definition 6.1.5** (**Polarity of a subformula**). The *polarity of a subformula $\psi$ of $\phi$* is defined recursively is follows:

- $\psi = \phi$ has positive polarity.
- If $\psi = \sigma(\psi_1, \ldots \psi_n)$ and $\psi$ is of positive (negative) polarity, then $\psi_i$ has positive polarity if the $i^{th}$ operand of $\sigma$ has a positive (negative) polarity, and $\psi_i$ has negative polarity otherwise.

■

Note that if a logic always has positive polarity, then by definition, every subformula has positive polarity.

In Beer et al. [20], the polarity of a logic was used to prove a useful claim, which we will use to show monotonicity of $L^{rel}_{\nu,\mu}$ over the lattice of sets of timed automata states.

**Claim 6.1.1** (Lemma 12 of Beer et al. [20]). In a logic with polarity, if $\psi$ is a subformula of $\phi$ and $\psi$ has a positive (negative) polarity and if $[[\psi]]_M \subseteq [[\psi']]_M$ ($[[\psi']]_M \subseteq [[\psi]]_M$), then $[[\phi]]_M \subseteq [[\phi[\psi \mapsto \psi']]]_M$.

### 6.1.3 Mutual Vacuity

When one examines a formula, one may find many subformulas on their own that are vacous within a larger formula; however, a combination of them may not be vacuous. We formalize this concept with the definition of mutual vacuity, taken from Gurfinkel and Chechik [80].

**Definition 6.1.6** (**Mutual vacuity**). A formula $\phi$ is mutually vacuously in a model $M$ in subformulas $\psi_1, \ldots \psi_n$ iff for all formulas $\gamma_1, \ldots, \gamma_n$, $M \models \phi$ if and only if $M \models \phi[\psi_1 \mapsto \gamma_1, \ldots, \psi_n \mapsto \gamma_n]$. ■

Knowing subformulas that are vacuous is useful; knowing when subformulas are mutually vacuous (or not mutually vacuous) is even more useful.

**Example 6.1.2.** Again consider $TA_{up}$ in Figure 6.1, and consider the subformula $AG\,[down \rightarrow AF\,[up]]$. Since the gate is always up in $TA_{up}$, we have two vacuous subformulas: $down$, and $AF\,[up]$. However, these are not mutually vacuous: for the formula to be true, we either need the subformula $\neg down$ or the subformula $AF\,[up]$. ■

**Example 6.1.3.** We present an example from Namjoshi [129] which is used to illustrate proof vacuity and use it to illustrate both proof vacuity and mutual vacuity. We discuss the mutual vacuity now, and discuss the implications on proof vacuity in Example 6.4.2. Consider the automaton in Figure 6.2, and consider the formula $AX\,(AF\,[p] \vee AF\,[q])$.

In this formula, $AF\,[p]$ is a vacuous subformula, and $AF\,[q]$ is a vacuous subformula. However, they are not mutually vacuous. Because at each branch, either $p$ is eventually true and $q$ is eventually true, either one can be chosen.

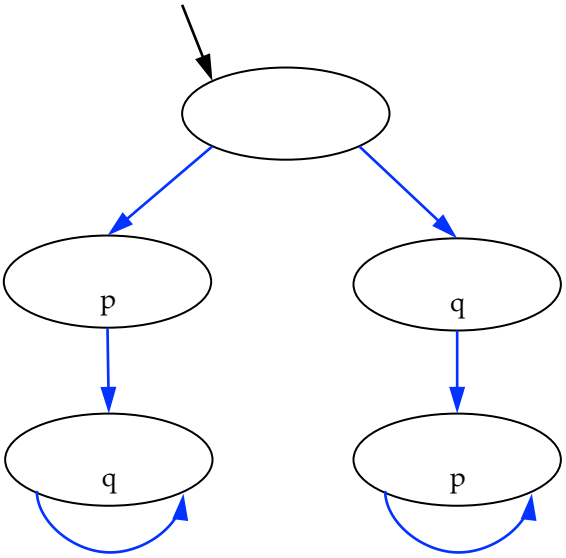


**Figure 6.2:** Timed (or untimed) automaton illustrating that formula vacuity can be subtle and complex.

■

## 6.2 Vacuity and Untimed Temporal Logics

In order to detect vacuous formulas in the timed mu-calculus, we leverage and extend results concerning vacuity for the untimed mu-calculus. Furthermore, results for the untimed mu-calculus leverage results of other untimed temporal logics, including CTL*. One important result is the following claim, proven in Beer et al. [20].

**Claim 6.2.1** (Polarity of CTL*)**.** The untimed logic CTL* has polarity.

Since CTL is a sublogic of CTL*, the untimed logic CTL has polarity.

From the claim: $\neg$ has negative polarity; $\wedge$, $X(\phi)$, $[[\phi_1]\, U\, [\phi_2]]$, and $E\phi$ all have positive polarity. The operator $\vee$ also has positive polarity.

There is a similar claim for the untimed modal mu-calculus, which is proven in Dong et al. [66].

**Claim 6.2.2** (Polarity of the untimed mu-calculus [66])**.** The modal mu-calculus is a logic with polarity.

From the proof of the claim in Dong et al. [66]: $\wedge, [a](\phi)$, and $\nu Y.[\phi]$ all have positive polarity. We use this claim and the equivalents of the derived operators to prove a slightly stronger claim:

**Claim 6.2.3** (The untimed mu-calculus has positive polarity)**.** Given that the untimed modal-mu calculus is in positive normal form (all operators dualized), then every operator in the untimed mu-calculus has positive polarity.

As a reminder, a formula is in positive normal form *positive normal form* iff a all the negations are pushed as far inwards as possible. This means that all negation operators appear immediately next to propositions (or states).

***Proof of Claim 6.2.3.*** From Claim 6.2.2 and Claim 6.2.1, we know that the operators $\wedge, \vee, [\phi]()$, and $\nu Y.[\phi]$ all have positive polarity. We show the proof for the other two operators: $\langle a \rangle(\phi)$ and $\mu Y.[\phi]$.

**Proof of $\langle a \rangle(\phi)$:** Let $\phi_1$ and $\phi_2$ be formulas such that $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$. One can either proof positive polarity directly from the formula definitions or through using the derivations. We will show this way using the definition of $\langle a \rangle(\phi)$. By definition, $[\![\langle a \rangle(\phi_1)]\!] = \{l \mid l \xrightarrow{a} l' \text{ and } l \in [\![\phi_1]\!]\}$ and $[\![\langle a \rangle(\phi_2)]\!] = \{l \mid l \xrightarrow{a} l' \text{ and } l \in [\![\phi_2]\!]\}$. Since $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$, at least as many states can transition via some action $a$ to a state satisfying $\phi_2$ as to a state satisfying $\phi_1$.

**Proof of $\mu Y.[\phi]$:** We prove this using the definition of the derivation. Let

$[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$. By the derivation, $\mu Y.[\phi] \equiv \neg\nu Y.[\neg\phi]$. Since $[\![\phi_1]\!] \subseteq [\![\phi_2]\!]$ and $\neg$ has negative polarity, we know that $[\![\neg\phi_2]\!] \subseteq [\![\neg\phi_1]\!]$. Since $\nu Y.[\phi_1]$ has positive polarity, $[\![\nu Y.[\neg\phi_2]]\!] \subseteq [\![\nu Y.[\neg\phi_1]]\!]$. Because $\neg$ has negative polarity, we have $[\![\neg\nu Y.[\phi_1]]\!] \subseteq [\![\neg\nu Y.[\phi_2]]\!]$. Therefore, $[\![\mu Y.[\phi_1]]\!] \subseteq [\![\mu Y.[\phi_2]]\!]$. □

Utilizing the definitions of derived operators, one can show that the operators all have positive polarity.

*Remark* 6.2.1 (Mu-calculus identifies different vacuous subformuals). Compared to the branching time logic CTL the modal mu-calculus formula writes formulas differently. For instance, $AF\,[p]$ (assuming that $p$ is an atomic proposition), is written as:

$$X_1 \stackrel{\mu}{=} p \vee [-](X_1)$$

Here the "eventually" is written out over a disjunction. Because vacuity involves identifying disjunctions whose truths are irrelevant (when the other disjunct is always true), here the modal mu-calculus may identify a *modality* as vacuous rather than a proposition. This phenomenon is similar in the timed setting when comparing TCTL to the timed mu-calculus. We illustrate this remark in the following example.

**Example 6.2.1 (Vacuity in the mu-calculus subtleties).** Again consider Example 6.1.3 and Figure 6.2, but consider the slightly different formula $AX\,(AF\,[p \vee q])$. In the untimed modal mu-calculus, this is written as:

$$X_1 \stackrel{\mu}{=} [-](X_2)$$
$$X_2 \stackrel{\mu}{=} (p \vee q) \vee [-](X_2)$$

From this formula we can say that $p$ is a vacuous subformula, and $q$ is a vacuous subformula. However, using the vacuity of disjuncts, the entire right disjunct of $X_2$ is vacuous. Translated to the formula, this means that the $AF$ modality is vacuous, and that the formula could also be simplified to $AX(p \vee q)$. In this case, the calculus uses a vacuous disjunct to simplify the translated formula. ■

## 6.3 Detecting Vacuity in Untimed Systems

With these definitions, one can utilize the polarity of the logic to detect vacuous formulas with respect to a Model $M$. First, to detect vacuity, we wish to minimize the number of possible subformulas that we examine, and the number of formulas we need to substitute each formula with. To help with these, we will present more definitions and results from Beer et al. [20], Dong et al. [66].

**Definition 6.3.1** (**Vacuity with respect to a set of subformulas**). Let $S$ be a set of subformulas of $\phi$. We say that $\phi$ is *S-vacuous* in model $M$ if there exists a $\psi \in S$ such that $\psi$ is a vacuous subformula. ■

**Definition 6.3.2** (**Minimal subformulas**). Let $S$ be the set of subformulas. Then the *minimal subformulas of S*, denoted $min(S)$, are:

$$min(S) = \{\psi \in S \mid \text{there is no } \psi' \in S \text{ such that } \psi' \text{ is a subformula of } \psi\} \quad (6.1)$$

■

With the result below from Beer et al. [20], we can determine $S$ vacuity by examining only the minimal subformulas in $S$.

**Claim 6.3.1** (From Beer et al. [20]). $\phi$ is $S$-vacuous if and only if $\phi$ is $min(S)$-

vacuous.

The lemma below also allows us, in certain cases, to only examine larger subformulas.

**Lemma 6.3.2** (From Beer et al. [20])**.** Let $\phi_s$ be a subformula of $\phi_l$, and let $\phi_l$ be a subformula of $\phi$. If $\phi_l$ does not affect $\phi$ in $M$, then $\phi_s$ does not affect $\phi$ in $M$.

With more work, we can reduce the formulas needed to substitute into vacuous subformulas to either `tt` or `ff`. This follows from the following results from Beer et al. [20].

**Claim 6.3.3** (From Beer et al. [20])**.** Let $\psi$ be a subformula of $\phi$ in a logic with polarity. Then for every model $M$, the following are equivalent:

1. $\psi$ does not affect $\phi$ in $M$.
2. $M \models \phi \Leftrightarrow M \models \phi[\psi \mapsto X]$ where $X = \texttt{ff}$ if $M \models \phi$ and $\psi$ is of positive polarity, or $M \not\models \phi$ and $\psi$ is of negative polarity. Otherwise, $X = \texttt{tt}$.

Combining these together, Beer et al. [20] get a usefulcorollary:

**Corollary 6.3.1** (Corollary of Claim 6.3.3 from Beer et al. [20])**.** In a logic with polarity, for a formula $\phi$ and a set $S$ of subformulas of $\phi$, for every model $M$, the following are equivalent:

1. $\phi$ is $S$-vacuous in $M$
2. There is a $\psi \in min(S)$ such that $M \models \phi \Leftrightarrow \phi[\psi \mapsto X]$ where $X = \texttt{ff}$ if $M \models \phi$ and $\psi$ is of positive polarity, or $M \not\models \phi$ and $\psi$ is of negative polarity. Otherwise, $X = \texttt{tt}$.

## 6.4 Vacuity and Proofs

Some model checkers verify formulas by constructing proofs that the model satisfies the formula. Utilizing these proofs, additional vacuity information can be obtained. Recall that a proof is undeniable evidence that a model satisfies (or does not satisfy) a formula, and that it is possible to have multiple proofs. Given these proofs, there are various notions of vacuity, two of which are:

1. Given one proof that $M \models \phi$ (or $M \not\models \phi$), some subformula $\psi$ is unused. In this case, $\psi$ is vacuous in some proof.

2. Considering all proofs of $M \models \phi$ (or $M \not\models \phi$), some subformula $\psi$ is not included in any of the proofs. In this case, $\psi$ is vacuous in all proofs.

These two notions were taken from Namjoshi [129], which discusses these notions for untimed systems. We formalize these two notions with the definitions below.

**Definition 6.4.1** (**Vacuous within a proof**)**.** A subformula $\phi$ is *vacuous within a proof* if and only if for the given proof, replacing $\phi$ with any formula $\psi$ does not invalidate the proof. ■

**Definition 6.4.2** (**Vacuous for all proofs**)**.** A subformula $\phi$ is *vacuous for all proofs* if regardless of the proof generated, replacing $\phi$ with any formula $\psi$ does not invalidate the proof. ■

The first notion, vacuity within a proof, allows us to detect vacuous subformulas. This follows from the following Lemma.

**Lemma 6.4.1** (Relating proof vacuity to formula vacuity.)**.** Let $M$ be a model and $\phi$ be a logical formula with subformula $\psi$. If we have a sound proof $P$ such that $M \models$

$\phi$ and $\psi$ is vacuous within proof $P$, then $\psi$ is a vacuous subformula. Likewise, if we have a sound proof $P$ such that $M \not\models \phi$ and $\psi$ is vacuous within proof $P$, then $\psi$ is a vacuous subformula.

***Proof of Lemma 6.4.1.*** We prove this lemma when $M \models \phi$. The proof is similar when $M \not\models \phi$. Suppose we have a proof $P$ such that $\psi$ is vacuous within $P$. Because we have a sound proof $P$ for $M \models \phi$ and because $\psi$ does not influence the validity of the proof, we know that $\psi$ does not affect $\phi$; regardless of the truth of $\psi$, we still have the sound proof $P$ that $M \models \phi$. □

This Lemma is powerful; it states that any subformula $\psi$ is vacuous if and only if we can show that $\psi$ is vacuous for a single proof. So we can find is some proof where it does not matter what $\psi$ is, we can tell that $\psi$ is vacuous. Likewise, if $\psi$ is not a vacuous subformula, then every proof utilizes $\psi$ in some manner.

Recall that our tool is a proof-search tool: in order to determine if a timed automaton satisfies a formula, it constructs a proof. If a formula is valid (or invalid), there may be multiple proofs for the satisfaction of that formula, and different proofs might find different subformulas vacuous. As a result, the two above notions are useful to distinguish vacuity.

To illustrate these different notions of vacuity, consider the following example.

**Example 6.4.1 (The chosen proof influences vacuity).** Consider the timed automaton $TA_{down}$ in Figure 6.1 and the formula $EG\,[down \rightarrow AF\,[up]]$, where the first allows the verifier to choose the path. $TA_{down}$ satisfies this formula. However, there are two different proofs for this formula:

1. The prover chooses a path where the gate is lowered down, and then due to the invariant, is raised up.

2. The prover waits in the location $up$ forever. (Note: This waiting is a time-divergent path of the automaton).

In the first proof, no subformula is vacuous, since every subformula is needed to establish truth. However, in the second proof, the automaton chooses the path where the light is never turned on. In this path, *down* is never true, making the subformula $AF[up]$ vacuous.

In this case, we know that the formula is vacuously satisfied by $TA_{down}$, and that there *exists a proof* where $AF[up]$ is vacuous, but $AF[up]$ is not vacuous for all proofs. ■

In addition, the length of the proof can make different subformulas vacuous; we present an example from Namjoshi [129].

**Example 6.4.2 (Example 6.1.3 continued).** Again consider the automaton in Figure 6.2 and the formula $AX(AF[p] \vee AF[q])$. The automaton satisfies this formula, but there are a variety of proofs, different formulas are vacuous in each. Here are some of the proofs:

1. **Proof 1: fewest transitions.** Because of the $AX$, the prover must take both the left and right transitions and $AF[p] \vee AF[q]$ for both. The prover takes the left branch, notices that $p$, and hence $AF[p]$ is true, and produces a proof for that branch. The prover then takes the right branch, notices that $q$, and hence, $AF[q]$ is true, and produces a proof. No subformula is vacuous (although this proof is identical for the proof of the simpler formula $AX(p \vee q)$.

2. **Proof 2: find $p$.** In the left branch, $p$ is immedaitely true, and the prover stops for the left branch. For the right branch, the prover prooves that $AF[p]$ is true; it shows that in the next state, $p$ is true. In this case, $AF[q]$ is unused

in the proof.

3. **Proof 3: find $q$.** The prover ignores $AF\,[p]$ and looks to prove $AF\,[q]$ in both branches. This requires taking two transitions in the left branch, and taking one transition in the right branch.

These proofs have different lengths. Depending on the proof strategy and the desired proof (sometimes the shortest proof is sought first), different subformulas are vacuous. Furthermore, a formula can be vauous but require a longer proof if that subformula is removed. See Namjoshi [129] for additional discussion. ■

## 6.5 Timed Vacuity: Theoretical Results

In this section we discuss how we detect vacuity for $L_{\nu,\mu}^{rel,af}$ formulas over timed automata. This extends the previously discussed work for untimed vacuity over transition systems. In order to utilize previous vacuity work, we first show that our timed logic, $L_{\nu,\mu}^{rel}$ is a logic of polarity. We then leverage the previous work and develop two techniques for detecting vacuity over timed automata: one that is fast and sound but incomplete, and one that is slower but incomplete.

The fast technique utilizes the *one* proof produced by the tool and identifies any subformulas that are not used in that proof. Because any unused formula does not influence the proof, we know that any detected subformula is vacuous (vacuous for some proof). However, because this proof may use a subformula that is vacuous for an alternative proof, not all vacuous subformulas are detected.

The complete technique searches all possible proofs, and generates a tree that stores all of the possible proofs. By searching this tree of proofs, the algorithm can detect all vacuous subformulas. This vacuity includes vacuous subformulas, which are subformulas that are vacuous for some proof, and formulas that are vacuous for all proofs.

### 6.5.1 Polarity of $L^{rel}_{\nu,\mu}$

First, we extend the results of Beer et al. [20], Dong et al. [66] to show that $L^{rel}_{\nu,\mu}$ is a logic of polarity.

**Theorem 6.5.1.** $L^{rel}_{\nu,\mu}$ is a logic with polarity. Furthermore, every operator has positive polarity.

***Proof of Theorem 6.5.1.*** We show that each operator has positive polarity.

From Claim 6.2.2 (proven in Dong et al. [66]), the operators $\wedge, \vee, \mu Y.[\phi]$, $\nu Y.[\phi]$, $\langle a \rangle(\phi), [a](\phi)$ all have positive polarity. We only need to show that the timed operators have positive polarity. Let $\phi_1$, and $\phi_2$ be formulas such that $[[\phi_1]]_M \subseteq [[\phi_2]]_M$.

We show the proofs for the operators $\exists(\phi)$ and $\exists_{\phi_1}(\phi_2)$. The proofs for the other operators are similar.

Consider the operator $\exists(\phi)$. Recall that by definition,

$$[[\exists(\phi)]] = \{(l,\nu) \mid \exists \delta \geq 0 s.t. (l,\nu) \xrightarrow{\delta} (l,\nu+\delta) \text{ and } (l,\nu+\delta) \models \phi\}.$$

Now let Hence, $[[\exists(\phi_1)]] = \{(l,\nu) \mid \exists \delta \geq 0 s.t. (l,\nu) \xrightarrow{\delta} (l,\nu+\delta) \text{ and } (l,\nu+\delta) \models \phi_1\}$ Because $[[\phi_1]] \subseteq [[\phi_2]]$, $(l,\nu+\delta) \models \phi_2$. Since $(l,\nu) \xrightarrow{(} l,\nu+\delta)$, we know that $(l,\nu) \models \exists(\phi_2)$.

Now consider the operator $\exists_{\phi_a}(\phi_b)$. To show positive polarity, we fix each operand and show positive polarity.

**Fixing $\phi_a$:** Suppose $(l,\nu) \models \exists_{\phi_a}(\phi_1)$ for some formula $\phi_a$. This means that there is some $\delta$ such that $(l,\nu+\delta) \models \phi_1$ and for all $0 \leq \delta' < \delta, (l,\nu+\delta') \models \phi_a$. Because $[[\phi_1]] \subseteq [[\phi_2]]$, we know that $(l,\nu+\delta) \models \phi_2$. Since $\phi_a$ is fixed, by definition,

$(l, \nu) \models \exists_{\phi_a}(\phi_2)$.

**Fixing $\phi_b$:** The proof is similar to when $\phi_a$ is fixed. Suppose $(l, \nu) \models \exists_{\phi_1}(\phi_b)$ for some formula $\phi_a$. This means that there is some $\delta$ such that $(l, \nu + \delta) \models \phi_b$ and for all $0 \leq \delta' < \delta, (l, \nu + \delta') \models \phi_1$. Because $[[\phi_1]] \subseteq [[\phi_2]]$, we know that $(l, \nu + \delta') \models \phi_2$ for all $0 \leq \delta' < \delta$. Since $\phi_b$ is fixed, by definition, $(l, \nu) \models \exists_{\phi_2}(\phi_b)$. □

Since we have fixed each of the operands and shown positive polarity for each operand of the operator, the operator has positive polarity, as a corollary, we get $L^{rel}_{\nu,\mu}$ formulas are all monotonic. To model check timed automata, we use the lattice over sets of states of timed automata.

**Corollary 6.5.1** (Monotonicity of $L^{rel}_{\nu,\mu}$)**.** Over the lattice of sets of states of timed automata, each $L^{rel}_{\nu,\mu}$ formula is a monotonic function.

***Proof of Corollary 6.5.1.*** By Theorem 6.5.1, each operator has positive polarity.

Therefore, for any two sets of states $Q_1, Q_2$, we know that $Q_1 \subseteq Q_2$. Let the subformula $\psi$ be the set of states $Q_1$ (the predicate variable whose states are the set of states $Q_1$) and let $\psi'$ be the set of states $Q_2$. Let $\phi$ be the formula that is representing the monotonic function. Because $Q_1 \subseteq Q_2$. By Theorem 6.5.1, every operator is of positive polarity, and by definition of a polarity of a subformula $Q_1$ and $Q_2$ are subformulas of positive polarity, as is $\phi$. Therefore, by Claim 6.1.1, $[[\phi(Q_1)]] \subseteq [[\phi(Q_2)]]$. □

Notice that we defined all of the $L^{rel}_{\nu,\mu}$ all of the operators to have positive polarity ($\neg$ is not a valid operator in this version; all formulas are dualized).

### 6.5.2 Using the Proof Paradigm for Fast Vacuity Checking

This subsection describes the theorems required for fast but limited algorithm for vacuity checking. This fast algorithm examining the proof output by the model

checker and determining if there are any vacuous subformulas within that proof. Because any subformula vacuous for a proof is a vacuous subformula, this method is sound; however, it is incomplete since there might be a subformula that is vacuous but required for this particular proof.

Recall that we are given a timed automaton with an initial state and an $L_{\nu,\mu}^{rel,af}$ formula, and we are asked if there are any vacuous subformulas. Note that in order to dualize the logic, any $p \rightarrow q$ formula is converted to $\neg p \vee q$. Our notion of vacuity then, in the simplest cases, reduces to finding branches in $\vee$ or $\wedge$ that are not used.

The first approach utilizes short circuiting; if, for all states, we can prove a property without ever examining that subformula, then that subformula is vacuous.

**Theorem 6.5.2 (Missing subtrees indicate formula substitution).** Let $TA$ be a timed automaton with initial state $(l, \nu, \nu_f)$ and $\phi$ be a $L_{\nu,\mu}^{rel}$ formula. If there exists a proof where $\psi$ never appears as the right-side of a sequent in the proof, the $\psi$ is vacuous for that proof. Likewise, if every appearance of $\psi$ in this proof is the sequent $\emptyset \vdash \psi$, then $\psi$ is vacuous for that proof.

Note that the sequent $\emptyset \vdash \psi$ is a valid leaf; it is one of the proof rules in Figure 5.1. The case concerning this sequent ($\emptyset \vdash \psi$) handles the $[a](\phi)$ formula when some actions cannot be taken.

***Proof of Theorem 6.5.2.*** If $\psi$ never appears in the proof, then regardless of what $\psi$ is substituted with, the proof remains unchanged and $\phi$ is still satisfied. Hence, by definition, $\psi$ is vacuous. If the only time $\psi$ appears is the empty sequent $\emptyset \vdash \psi$, then in those instances, $\psi$ does not affect that proof because $\psi$ can be substituted

with any other formula and those sequents are still true. □

If we combine this Theorem with Lemma 6.4.1, we know that if $\psi$ is vacuous for that proof, then $\psi$ is a vacuous subformula, meaning that $\psi$ does not affect the satisfaction of $\phi$ for our model. The above Theorem extremely useful in certain cases; if we can identify that a subformula was never examined, then we have detected vacuity without any extra work. The downside is that for vacuity to be detected in the above fashion, the tool needs to guess the right branch to check first. In special cases, the user can write the formula to have the tool check the non-vacuous branch first, but this also limits the tool to only detect certain subformula as vacuous. A downside is that this Theorem is dependent on the order that the tool checks the branches; in our tool, all the left branches are examined before the right branch.

### 6.5.3 Using the Proof Paradigm for Additional Vacuity Checking

With the proof paradigm, if we can represent all possible proofs, we can then ask which subformulas are used within each proof. When proving theorems, rather than short-circuiting branches, we will prove all branches, and then store the results of all branches in a tree. This tree will have the results of every possible branch for the formulas. We can use this tree to detect two kinds of vacuity:

1. We can find each vacuous subformula $\phi$, and
2. We can determine if a formula $\phi$ is vacuous over all proofs.

The second item is exactly the definition of a vacuous subformula over all proofs. The first one comes from the following fact: if there is a vacuous subformula, then there must be some proof in which that subformula is vacuous. We show this with the following Lemma.

**Lemma 6.5.3.** Let $TA$ be a timed automaton with initial state $q_0$, $\phi$ be a $L_{\nu,\mu}^{rel,af}$ formula, and $\psi$ be a subformula of $\phi$. If $\psi$ is a vacuous subformula of $\phi$, then there exists a sound proof such that $\psi$ is vacuous within that proof.

This Lemma is the converse of Lemma 6.4.1 when applied to our framework of timed automata and $L_{\nu,\mu}^{rel,af}$ formulas; this argues that any vacuous subformula must be vacuous within some proof, meaning that searching the tree of proofs is a complete way of detecting every vacuous subformula.

***Proof of Lemma 6.5.3.*** We prove the contrapositive: if $\psi$ appears as the right-hand of some sequent within each sound proof, then $\psi$ is not vacuous. We argue the case when $q_0 \models \phi$; the case when $q_0 \not\models \psi$ is similar.

Let $\psi$ be a vacuous subformula. Consider all of the proofs for $q_0 \not\models \phi$. Now, within each proof, replace $\psi$ with $\texttt{ff}$. Since $\psi$ is a vacuous subformula. By Claim 6.3.3, we know that $q_0 \models \phi[\psi \mapsto \texttt{ff}]$. Hence, there must be some proof that $q_0 \models \phi[\psi \mapsto \texttt{ff}]$.

Examine that proof, noting all the sequents where the right-hand side is $\texttt{ff}$ where the formula $\psi$ would have been. If there are no such sequents, then we have found a proof that does not involve $\psi$, and we are done. Else, examine each sequent $(l, cc) \vdash \texttt{ff}$. Since the proof only contains valid sequents, this branch must be valid. The only valid proof rule for a right-hand side of $\texttt{ff}$ is the empty rule $\emptyset \vdash \texttt{ff}$. In this case, since the right-hand side can be anything, replacing $\texttt{ff}$ with $\psi$ does not invalidate the proof. Hence, if we substitute $\psi$ back in to replace each of the $\texttt{ff}$ sequents, we still have a valid proof that does not depend on $\psi$. Therefore, we have a proof that $q_0 \models \psi$ where $\psi$ is vacuous within that proof. □

As a result, if we can construct every single possible proof that the model satisfies $\phi$ (or every proof that the model does not satisfy $\phi$).

When we discuss the implementation in Section 6.6.2, we discuss how we represent the set of all proofs as a tree, and then utilize this tree of proof trees to answer both kinds of vacuity: vacuity within any proof and vacuity for all proofs. We call this structure the **tree of proofs**.

## 6.6 Implementation

With the results in Section 6.5, we now discuss our algorithms and implementations of these results. In our implementation, vacuity is focused on identifying $\wedge$ or $\vee$ branches that are not needed by the prover. This follows because any branch not needed by the prover does not affect some proof, and hence is a vacuous subformula.

We fast implementation checks only one proof and uses boolean flags to identify subformulas that do not affect that proof, and hence are vacuous. While quick, this misses some vacuous subformulas. The complete method tells the prover to examine *additional* proofs. In this case, the tool examines all possible branches to produce a tree of all possible proofs while it is proving the formula true or false. This structure is the *tree of proofs*. With this tree of proofs, it finds all vacuous subformulas. Computing the additional proofs is extra work; however, the vacuity work is not large outside of computing the proofs.

### 6.6.1 Fast Vacuity: Finding Unneeded Subformulas Within One Proof

We run the model checker as usual, but we augment each subformula (each instance of each subformula) with one boolean variable that indicates if that subformula has been checked by the prover by some state. If the subformula has not, then we have found a proof where the formula is not used. By Theorem 6.5.2, we know that this subformula is vacuous.

The tool is implemented in a left-to-right fashion: with every $\wedge$ and $\vee$ branch, the left branch is examined before the right branch. If the left branch of a $\vee$ is `tt` or if the left branch of a $\wedge$ is false, the prover does not examine the other branch. This implementation works well for detecting vacuity of $AF[q]$ in the formula $AG[p \rightarrow AF[q]]$; the prover checks $p$ first, and if $p$ is never true then $AF[q]$ is never checked and hence is vacuous.

However, based on how the formula is written, the tool may not identify vacuity. First, a vacuous subformula may be needed for the proof we chose. See Example 6.4.2 for such an example. As a result, the vacuity checking is sound but incomplete. However, we get this vacuity checking *with little performance overhead*. We give an evaluation of the performance of this implementation in Section 6.7.

### 6.6.2 Complete Vacuity: Building and Searching the Tree of Proofs

To get additional vacuity, first we tell the tool to not short-circuit an $\wedge$ or a $\vee$, making the tool always try to produce proofs for both branches. In some cases, this enumeration can greatly increase the time required for verification because a more complex subformula that could be ignored must now be proven.

As we enumerate all possible branches, we store the tree of sequents. This tree is our *tree of proofs*, since it contains all the sequents for each possible proof. This tree contains the sequents, the logical operations, and the truths of each leaf. We then reason with this tree of proofs to determine vacuity. For implementation ease, as we generate the proof tree, we store the truth of that instance of the subformula in the proof. (Hence, every node, not just a leaf, contains the truth of the proofs).

With this tree of proofs, we can detect both kinds of vacuity: the subformulas that are vacuous for some proof, and the subformulas that are vacuous for every proof. For this discussion of vacuity, we discuss the case when the formula is satisfied. The concept is similar when the formula is false (and vacuity is over $\wedge$

nodes instead of $\vee$ nodes). For this discussion, let $\phi_1 \vee \phi_2$ be the subformula in question.

The easier vacuity to check is vacuity over all proofs. By definition, $\phi_1$ is vacuous over all proofs if and only if there is no proof that uses $\phi_1$. To determine this, we search the tree of proofs only considering nodes that are true. If any of them involve the subformula $\phi$, then $\phi$ is not vacuous over all proofs because we can construct a proof using that instance as part of the proof. If we never encounter $\phi$ in our search, then $\phi$ is a vacuous subformula.

The harder vacuity to check is whether a formula is vacuous (vacuous for some proof). To do this, we examine the tree of proofs and search bottom-up from all the leaves that are true. We then at each node, construct sets of subformulas $A_i$, where each $A_i$ is the set of subformulas that are required for this proof. After we compute the set of subformulas, we mark each subformula $\phi$ that does not appear in some set $A_i$ as vacuously true.

Here is the algorithm for constructing those sets:

1. **Initial Case:** Each leaf has one set whose element is that subformula.

2. **Recursive Case: Unary Operators:** Given a set of sets $S = \{A_1, \ldots A_m\}$, we add the current subformula to each set $A_i$.

3. **Recursive Case: Binary $\vee$:** Given the sets $S_1 = \{A_1, \ldots A_n\}$ and $S_2 = \{B_1, \ldots B_m\}$ of the two children nodes, we construct the set for the parent node by unioning the two sets. This operation produces the set $S = \{A_1, \ldots, A_n, B_1, \ldots B_m\}$. We then add this node into each of the sets.

4. **Recursive Case: Binary $\wedge$:** Given the sets $S_1 = \{A_1, \ldots A_n\}$ and $S_2 = \{B_1, \ldots B_m\}$ of the two children nodes, we construct the set for the parent node by unioning each set $A_i$ with each set $B_j$ to produce a set. This produces

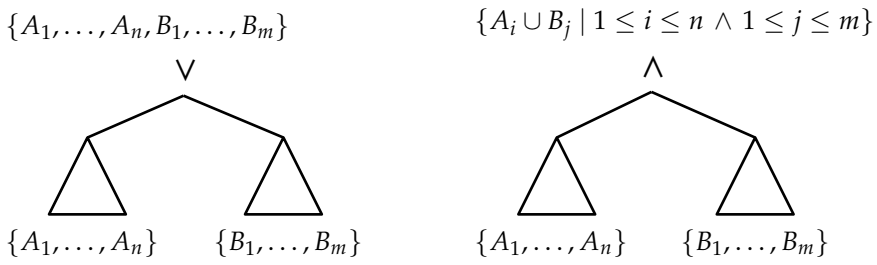


**Figure 6.3:** Diagrams illustrating how to compute the sets of subformulas needed for $\wedge$ and $\vee$ branches of the proof-trees structure.

the set $S = \{A_i \cup B_j \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\}$. We then add this node into each of the sets.

The last two cases are illustrated in Figure 6.3.

Then, given these sets, we examine the set of sets $S = \{A_1, \ldots A_n\}$ for the root node. A formula $\phi$ is vacuous if and only if there is some $A_i \in S$ such that $\phi \notin A_i$. The idea behind the correctness is that each $A_i$ represents a single proof (proof tree) in the tree of proofs, and $A_i$ is the set of all subformulas used in the short-circuiting proof $A_i$. Hence, if $\phi$ is not in $A_i$, then we have a proof that does not use $\phi$. Therefore, by definition, $\phi$ is vacuous. Since we enumerate over all true branches bottom-up, we cover all such proofs.

### 6.6.3 Handling Placeholders and Splitting Rules

Extending these techniques to the timed setting has some complications. One complication is handling the proof rule $\vee_s$, which splits the set of states to involve both branches. With the $\vee_s$ proof rule, neither branch is unnecessary: some states of the sequent satisfy the left branch, and the remainder satisfy the right branch (some states may satisfy both branches). This complicates the checking because we must account for both branches being relevant in such a proof. However, this is fixable: we note when an $\vee$ branch is proven with the split rule instead of choosing a

branch, noting that both branches are required for that proof.

For placeholders, when constructing a proof, since the placeholders only give the states that the proof needs, we can continue the proof as is until we have established a placeholder and need to return a placeholder to the parent rule. Whenever we use a union of placeholders from two branches, this is similar to a $\vee_s$ rule: the subformula is then valid for some states and invalid for others.

Given this complexity, these rules might make our implementation sound but incomplete when we need to handle placeholders. To guarantee completeness in proofs, the proof may need to return up a placeholder that is larger that what may have been needed, requiring it to union two placeholders when a union might not be unnecessary.

## 6.7 Performance Evaluation: One-Proof Vacuity

We give a performance evaluation comparing the PES tool without vacuity to the PES tool with the vacuity implementation for free: the implementation that examines the current proof and determines if a subformula is vacuous. We do two evaluations. The first evaluation evaluates the performace of the additional vacuity implementation, and the second illustrates the power of this implementation to detect vacuity. We evaluate the vacuity that implements with variable flags and does not significantly slow down the system.

### 6.7.1 Evaluation on PES Tool Implementation Examples

This evaluation uses the same models and specifications as the evaluation in Section 5.11. Experiments were run on a Mac OS 10.9 machine with a single 2.0 GHz Intel Core i7 (quad core) processor with 8 GB RAM. Time and space measurements (maximum space used) were made using the UNIX `time` command (using

the *real time* as the output time). Given that the machine is slightly different, the performance for the baseline PES tool is not necessarily the same as it was for the experiments in 5.11.

The data is provided in Tables 6.1 and 6.2 (split due to horizontal space constraints) contain the examples that are supported both by our tool (PES) and by UPPAAL. The original tool is PES, and the tool with the vacuity implementation is PVac.

In these tables, we use the following abbreviations: TO (timeout: the example took longer than 2 hours), TOsm (the example timed out with fewer process), TOp (the example timed out in the evaluation in Section 5.11.2), and O/M (out of memory). A scatter plot of the data in Tables 6.1 and 6.2 is in Figure 6.4. This scatter plot only includes examples that finished. The scatterplot was produced using the R programming language [139]. Furthermore, each example where the tool **detected a vacuous subformula** is in *italics* and is marked with a $^*$.

Additionally, Table 6.1 has the column "Vac," which describes the kind of vacuity with a code. If the entry is blank, then there were no vacuous subformulas. The codes used have the following meaning: NT (No transition), meaning that the modality operators are vacuous because no transitions need to be taken; TS (time simplified), meaning that the encodings of the modality operators did not need all of the subtleties and could have used simpler encodings of them; and UC (unnecessary constraints), meaning that we have a vacuous constraint, either a clock constraint or a constraint involving location variables.

From the scatterplot, the performance of the two implementations are similar. Concerning vacuity, 12 out of the 33 instances (36.36%) have vacuous subformulas detected. Examining the kinds of vacuity, there are 6 formulas with vacuity type NT, 4 formulas with vacuity type TS, and 3 formulas with vacuity type UC.

**Table 6.1:** Table comparing PES tool without vacuity (PES) and PES tool with performance-light vacuity (PVac). Times are reported in seconds (s). (Table 1 of 2.)

| File | Vac | PES4 | PVac4 | PES5 | PVac5 | PES6 | PVac6 |
|---|---|---|---|---|---|---|---|
| *CSMA-al** | NT | 0.01 | 0.03 | 0.03 | 0.03 | 0.18 | 0.17 |
| CSMA-as | | 0.44 | 0.52 | 5.64 | 5.88 | 183.31 | 195.44 |
| *CSMA-bl** | NT | 0.01 | 0.01 | 0.03 | 0.03 | 0.14 | 0.14 |
| *CSMA-bs** | UC | 0.02 | 0.04 | 0.09 | 0.09 | 0.30 | 0.36 |
| CSMA-M1 | | 0.01 | 0.01 | 0.05 | 0.06 | 0.15 | 0.16 |
| CSMA-M2 | | 0.41 | 0.41 | 6.80 | 6.74 | 213.16 | 223.49 |
| *CSMA-M3** | TS | 0.01 | 0.04 | 0.07 | 0.08 | 0.48 | 0.28 |
| CSMA-M4 | | 0.04 | 0.06 | 0.04 | 0.03 | 0.14 | 0.19 |
| *FISCHER-al** | NT | 0.00 | 0.00 | 0.00 | 0.01 | 0.00 | 0.01 |
| FISCHER-as | | 0.09 | 0.09 | 0.72 | 0.76 | 17.95 | 18.41 |
| *FISCHER-bl** | NT | 0.00 | 0.00 | 0.00 | 0.01 | 0.00 | 0.01 |
| FISCHER-bs | | 0.05 | 0.04 | 0.02 | 0.07 | 0.05 | 0.07 |
| FISCHER-M1 | | 0.00 | 0.00 | 0.00 | 0.01 | 0.00 | 0.01 |
| FISCHER-M2 | | 0.00 | 0.00 | 0.00 | 0.01 | 0.00 | 0.01 |
| FISCHER-M3 | | 0.14 | 0.14 | 2.87 | 2.91 | 86.71 | 88.60 |
| FISCHER-M4 | | 0.00 | 0.01 | 0.01 | 0.05 | 0.03 | 0.04 |
| *GRC-al** | NT | 0.01 | 0.01 | 0.01 | 0.04 | 0.11 | 0.06 |
| GRC-as | | 72.64 | 75.47 | TOp | TOp | TOp | TOp |
| *GRC-bl** | NT | 0.00 | 0.00 | 0.01 | 0.01 | 0.01 | 0.04 |
| GRC-bs | | 0.20 | 0.24 | 2.80 | 2.84 | O/M | O/M |
| GRC-M1 | | 0.01 | 0.01 | 0.01 | 0.01 | 0.02 | 0.03 |
| *GRC-M2** | TS | 0.01 | 0.02 | 0.01 | 0.01 | 0.02 | 0.02 |
| GRC-M3 | | 0.00 | 0.04 | 0.01 | 0.01 | 0.02 | 0.01 |
| GRC-M4 | | 0.00 | 0.01 | 0.00 | 0.00 | 0.01 | 0.01 |
| *GRC-M4ap** | UC, TS | 0.00 | 0.01 | 0.01 | 0.01 | 0.01 | 0.05 |
| LEADER-al | | 0.01 | 0.01 | 0.01 | 0.01 | 0.24 | 0.26 |
| LEADER-as | | 0.06 | 0.02 | 0.04 | 0.07 | 0.28 | 0.39 |
| LEADER-bl | | 0.01 | 0.01 | 0.00 | 0.01 | 0.03 | 0.05 |
| *LEADER-bs** | UC | 0.01 | 0.01 | 0.01 | 0.00 | 0.03 | 0.03 |
| *LEADER-M1** | TS | 0.00 | 0.01 | 0.00 | 0.00 | 0.01 | 0.01 |
| LEADER-M2 | | 0.01 | 0.01 | 0.03 | 0.03 | 0.52 | 0.58 |
| LEADER-M3 | | 0.01 | 0.01 | 0.10 | 0.10 | 2.47 | 2.74 |
| LEADER-M4 | | 0.01 | 0.00 | 0.01 | 0.01 | 0.07 | 0.09 |

Because one formula, GRC-M4ap, had two independently vacuous subformulas (UC and TS), the total codes add up to one more than the number of formulas. In

**Table 6.2:** Table comparing PES tool without vacuity (PES) and PES tool with performance-light vacuity (PVac). Times are reported in seconds (s). (Table 2 of 2.)

| File | PES7 | PVac7 | PES8 | PVac8 |
|---|---|---|---|---|
| *CSMA-al** | 0.80 | 0.75 | 3.71 | 3.74 |
| CSMA-as | TO | TO | TOsm | TOsm |
| *CSMA-bl** | 0.80 | 0.74 | 3.72 | 3.71 |
| *CSMA-bs** | 1.50 | 1.25 | 5.56 | 5.58 |
| CSMA-M1 | 0.76 | 0.75 | 3.89 | 3.88 |
| CSMA-M2 | TO | TO | TOsm | TOsm |
| *CSMA-M3** | 0.80 | 0.78 | 3.75 | 3.77 |
| CSMA-M4 | 0.79 | 0.75 | 3.77 | 3.76 |
| *FISCHER-al** | 0.00 | 0.00 | 0.01 | 0.00 |
| FISCHER-as | 1130.02 | 1119.79 | TO | TO |
| FISCHER-bl | 0.00 | 0.00 | 0.00 | 0.00 |
| FISCHER-bs | 0.44 | 0.40 | 0.07 | 0.07 |
| FISCHER-M1 | 0.00 | 0.00 | 0.00 | 0.00 |
| FISCHER-M2 | 0.03 | 0.01 | 0.04 | 0.03 |
| FISCHER-M3 | TOp | TOp | TOsm | TOsm |
| FISCHER-M4 | 0.01 | 0.00 | 0.01 | 0.01 |
| *GRC-al** | 0.06 | 0.03 | 0.02 | 0.03 |
| GRC-as | TOp | TOp | TOp | TOp |
| *GRC-bl** | 0.01 | 0.01 | 0.01 | 0.01 |
| GRC-bs | O/M | O/M | O/M | O/M |
| GRC-M1 | 0.03 | 0.03 | 0.04 | 0.04 |
| *GRC-M2** | 0.03 | 0.03 | 0.05 | 0.05 |
| GRC-M3 | 0.14 | 0.04 | 0.01 | 0.01 |
| GRC-M4 | 0.01 | 0.01 | 0.01 | 0.01 |
| *GRC-M4ap** | 0.02 | 0.01 | 0.01 | 0.01 |
| LEADER-al | 8.57 | 8.65 | 955.83 | 969.97 |
| LEADER-bl | 0.22 | 0.23 | 6.40 | 6.64 |
| *LEADER-bs** | 0.03 | 0.01 | 0.04 | 0.04 |
| *LEADER-M1** | 0.01 | 0.01 | 0.03 | 0.01 |
| LEADER-M2 | 19.81 | 20.16 | 2738.11 | 2805.19 |
| LEADER-M3 | 94.20 | 92.92 | TO | TO |
| LEADER-M4 | 0.09 | 0.34 | 0.03 | 0.14 |

the NT and TS instances, the vacuous subformulas translate to simplifications in detection that could be used. These result from the formal translations requiring many subtleties that do not appear in most models; nevertheless, some of the vac-

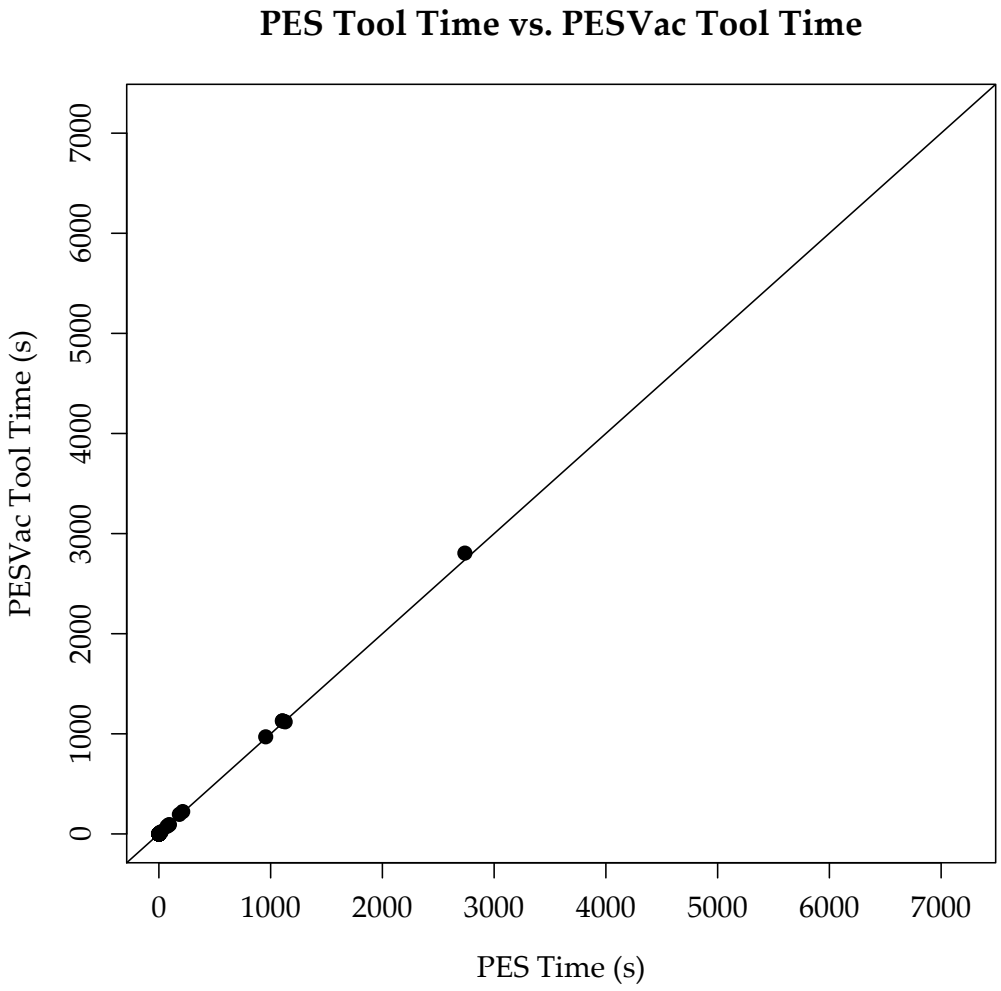


**Figure 6.4:** Figure comparing the PES tool time performance with the PES tool with vacuity time performance. Each example is a point, and the line drawn is the $y = x$ line, or the line where the performance of the PES tool and the PVac tool are the same.

uous subformulas are not these subtleties. Notice that the vacuity checking detects vacuous subformulas for valid and invalid examples.

**Table 6.3:** Table comparing PES tool without vacuity (PES) and PES tool with performance-light vacuity (PVac) on examples to illustrate vacuity. Times are reported in seconds (s). Any example with a vacuous subformula is in *italics* and marked with a *.

| File | PES | PVac |
|---|---|---|
| *LEADER-4-M5** | 0.035 | 0.063 |
| *VacuityTestAXAF1** | 0.005 | 0.018 |
| *VacuityTestAXAF2** | 0.004 | 0.008 |
| *VAcuityTestAXAF3** | 0.004 | 0.022 |
| *SimpleGate1** | 0.003 | 0.001 |
| SimpleGate2 | 0.003 | 0.019 |
| *BrokenGate1** | 0.003 | 0.021 |
| *BrokenGate2** | 0.003 | 0.016 |

### 6.7.2 Evaluation on Additional Vacuity Examples

To further illustate the power of the nearly-free vacuity checking (only using flags to detect unexamined formulas), we run both tools on six small examples. The performance setup is the same as the experiment in the previous subsection, and the performance numbers are in Table 6.3. Since the performance is quick on these examples for both tools, there is no scatterplot. The focus is on the vacuity power of the examples. The eight examples are described here. For purposes of illustration, the example has no vacuous subformulas. Every example that had at least one vacuous subformula had some subformula detected as vacuous, but most examples with many (independently) vacuous subformulas only had one of the subformulas detected as vacuous.

1. **LEADER-4-M5.** This model is the same leader election protocol as for all the LEADER examples in the previous section. For this version, we have four processes. The specification given is: It is always the case that if the first process has a parent, then the third process will inevitably have a parent, ex-

pressed in TCTL as $AG\,[p_1 = 0 \rightarrow AF\,[p_3 \neq 0]]$. In this case, the first process never has a parent (the vacuous subformula), which the tool detects.

2. **VacuityTestAXAF1.** The model is the model in Figure 6.2 (augmented with an initial invariant that does not affect the formula). The specification asked is: for all next actions, it is inevitable that $p$ or $q$ is true, written as the formula $[-](AF\,[p \vee q])$. There are three vacuous subformulas: $p$, $q$, and the inevitably ($AF\,[]$) modality. The tool detects that the inevitably ($AF\,[]$) modality is vacuous.

3. **VacuityTestAXAF2.** The model is in Figure 6.5, which removes the propositions from the two bottom locations. In this case, the only vacuous subformula is the inevitably ($AF\,[]$) modality, which is detected by the tool.

4. **VacuityTestAXAF3.** The model is in Figure 6.2, which is the same model as for VacuityTestAXAF1. However, the specification asked is $[-](AF\,[p] \vee AF\,[q])$. Here there are three choices for vacuity: claiming $AF\,[p]$ is vacuous, claiming $AF\,[q]$ is vacuous, or claiming that both inevitably ($AF\,[]$) modalities are vacuous. The tool detects that $AF\,[q]$ is vacuous.

5. **SimpleGate1.** The model is the left model of Figure 6.1. The specification asked is: the gate is not down or the gate will inevitably be up, written as $down \rightarrow AF\,[up]$. For this model, either $down$ is vacuous, or $AF\,[up]$ is vacuous, and the tool detects that $AF\,[up]$ is vacuous.

6. **SimpleGate2.** The model is the same as the previous example, but the specification asked is: $AG\,[down \rightarrow AF\,[up]]$. In this case no subformula is vacuous, and the tool correctly notes this.

7. **BrokenGate1.** The model is the right model of Figure 6.1, and the specification is the same as for SimpleGate1. In this case, either $down$ is vacuous or

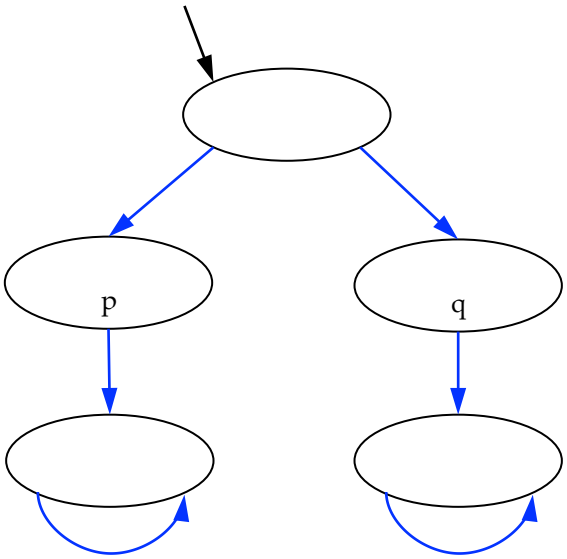


**Figure 6.5:** Timed (or untimed) automaton used as the model for VacuityTestAXAF2.

$AF\,[up]$ is vacuous, and the tool detects that $AF\,[up]$ is vacuous.

8. **BrokenGate2.** The model is the right model of Figure 6.1, and the specification is the same as for SimpleGate2. In this case, both *down* and $AF\,[up]$ are vacuous (independently); the tool detects that $AF\,[up]$ is vacuous.

## 6.8 Dissertation Contributions

### 6.8.1 Contributions

These are my contributions discussed in this chapter:

- Extended the concepts of vacuity for untimed systems to timed automata
- Showed that every operator of $L^{rel}_{\nu,\mu}$ has positive polarity, and hence that $L^{rel}_{\nu,\mu}$ is a logic of polarity.

- Extended the concept of whether a proof is vacuous to the $L_{\nu,\mu}^{rel,af}$ timed automata model checker.
- Implemented the vacuity check that allows the tool to identify any vacuous subformulas within the proof without running slower.

### 6.8.2 Future Work

Future work includes further extending the vacuity. Specifically, to extend the theory and the implementation to generate multiple proofs and examine if a formula is vacuous over all proofs (this is extending this aspect of the theory in Namjoshi [128, 129]).

# Chapter 7

# Conclusions and Future Work

This concludes my dissertation. We hope you enjoyed reading it (or scanning through it), and we hope you got something out of it. This section summarizes the contributions of the dissertation and summarizes potential future work.

## 7.1 Straightforward By Design

An objection that one might raise is, "this research is a straightforward extension of the research in untimed systems," and one might dismiss the work because of its straightforward nature. While some of the extensions are straightforward, there are two points concerning this objection of being straightforward:

1. **Some formulas are subtle.** Like Induction, once one has the right formula, the proof is straightforward, but coming up with the formula may not be. In order to get the correct formulas, many subtleties were identified and addressed.

2. **Some results are straightforward by design.** The research in this dissertation was designed to be straightforward. The formulas were written in order that straightforward proof rules could be designed, the proof rules were designed so that algorithms to implement them would be straightforward, and the algorithm was designed so that the implementation of it is straightforward.

Consequently, while the work may seem to be straightforward, the work should not be dismissed as trivial or irrelevant because of its seemingly-straightforward appearance.

## 7.2 Contributions

Here is a list of all of the current completed contributions from all of the chapters:

1. We gave a formal baseline definition for a timed automata based on definitions of others.

2. We provided formal definitions for the following variants: disjunctive guard

constraints, timed automata with variables and different semantics for unsatisfied invariants.

3. For timed automata with disjunctive guard constraints, timed automata with variables, and guarded-command programs, we show those variants are isomorphic to the baseline formalism and give a conversion translating out each variant.

4. For the different unsatisfied invariant semantics and allowing clock differences in clock constraints, we show that the reachable subsystems of those variants are isomorphic to the reachable subsystem of the baseline formalism and give a conversion translating out each variant.

5. For rational clock constraints, we give a non-label preserving isomorphism to the baseline formalism (uses integer constants only) and give a conversion translating out the rational constants.

6. We showed how the above conversions are composable, not only for timed automata with these features but also for timed automata with even more features. We give a framework, a composable timed automata, that give sufficient conditions describing extensions that still allow the equivalent variants to be converted out. We then showed that these conversions are commutative and associative at the semantic level.

7. With a common assumption regarding atomic propositions, we show that $L^{rel}_{\nu,\mu}$, $L_{\nu,\mu}$ and $T_\mu$ are bisimulation invariant. Additionally, for the region equivalence relation, we show that $L^{rel}_{\nu,\mu}$ is invariant.

8. We show $T_\mu \subseteq L^{rel}_{\nu,\mu}$ Furthermore, we show this result without requiring additional fixpoints, thus keeping the complexity simple.

9. We show $TCTL \subseteq L^{rel}_{\nu,\mu}$ For $E\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]$ we assume the timelock-free assuption and for $E\,[[\phi_1]\,R_{\bowtie c}\,[\phi_2]]$ (and its dual, $A\,[[\phi_1]\,U_{\bowtie c}\,[\phi_2]]$), we assume both a timelock-free assumption and a nonzeno assumption.

   We then show, using a formula from Henzinger et al. [88] that with an alternation, we can both detect timelocks as well as bypass timelocked states, thus removing the need for the timelock-free assumption.

10. We show $L_{\nu,\mu} \not\subseteq TCTL$.

11. We show $TCTL \not\subseteq L_{\nu,\mu}$, showing that expressing all of TCTL requires the additional power of the relativization.

12. We give a way of writing the *set of next states* of $TS(TA)$ in $L_{\nu,\mu}$.

13. Fine-tuned the PES Model Checker, and specialized it to model check $L^{af}_{\nu,\mu}$ formulas over timed automata

14. Created the clock zone implementations CRDZone and CRDArray, which can be seen as alternative sparse DBM implementations.

15. Implemented the CRDZone and CRDArray as well as fined-tuned the DBM implementation, and ran an experiment comparing the performance of the current PES tool on all three data structure implementation.

16. Implemented the previously-developed proof rules for $L^{af}_{\nu,\mu}$.

17. Design sound, complete and implementable additional proof rules to give a full set of proof rules for $L^{rel,af}_{\nu,\mu}$, as well as proved those rules to be sound and complete.

18. Implemented the timed automata model checker to model check any $L^{rel,af}_{\nu,\mu}$ formula over any timed automata.

19. Utilized derived proof rules to optimize the performance of this model checker.

20. Extended the concepts of vacuity for untimed systems to timed automata

21. Showed that every operator of $L^{rel}_{\nu,\mu}$ has positive polarity, and hence that $L^{rel}_{\nu,\mu}$ is a logic of polarity.

22. Extended the concept of whether a proof is vacuous to the $L^{rel,af}_{\nu,\mu}$ timed automata model checker.

23. Implemented the vacuity check that allows the tool to identify any vacuous subformulas within the proof without running slower.

## 7.3 Future Work

Here is the future work from the previous chapters.

Future work includes allowing the initial state to have clock values other than 0, and potentially to allow a set of initial states whose clock values are defined by a clock zone or a union of clock zones. Additionally, future work includes handling disjunctive constraints in invariants. While these constraints cannot be converted in a fashion similar to converting out disjunctive constraints in guards, future work involves determining the expressiveness of this additional feature. Disjunctive constraints in invariants are used to express timed automata with deadlines (see Bornot and Sifakis [32], Bornot et al. [33], Bowman [46], Bowman and Gómez [47], Gómez and Bowman [78]).

Future work includes answering some of the unanswered expressivity questions of various timed logics:

- Can we detect zeno executions in $L^{rel}_{\nu,\mu}$? Can we write formulas to bypass zeno executions?

- Is $L^{rel}_{\nu,\mu} \subseteq T_\mu$? We conjuecture no, but are not sure.
- Is $TPTL \subseteq L^{rel}_{\nu,\mu}$? Since TPTL is MTL with freeze quantification instead of timing intervals, determining if $TPTL \subseteq L^{rel}_{\nu,\mu}$ is similar to determining if $MTL \subseteq L^{rel}_{\nu,\mu}$.
- Is $L^{rel}_{\nu,\mu} \subseteq TPTL$? What about $L_{\nu,\mu}$ and $T_\mu$?

Answers to these items will allow us to better decide if $L^{rel}_{\nu,\mu}$ can verify the formulas we want or if we have to leverage properties of additional logics when verifying $L^{rel}_{\nu,\mu}$ formulas.

Future work is to further optimize the performance. One such future work is to utilize different standard forms (which other sources have designed considered) and to implement a more modern all-pairs shortest path algorithm; these should help the performance of the tool.

Future work includes further extending the vacuity. Specifically, to extend the theory and the implementation to generate multiple proofs and examine if a formula is vacuous over all proofs (this is extending this aspect of the theory in Namjoshi [128, 129]).

# Bibliography

[1] Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. Scheduling with timed automata. *Theoretical Computer Science*, 354(2):272–300, 2006. doi: http://dx.doi.org/10.1016/j.tcs.2005.11.018.

[2] Luca Aceto and François Laroussinie. Is your model checker on time? On the complexity of model checking for timed modal logics. *Journal of Logic and Algebraic Programming*, 52–53(0):7–51, 2002. ISSN 1567-8326. doi: http://dx.doi.org/10.1016/S1567-8326(02)00022-X.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall, Upper Saddle River, NJ, USA, 1993.

[4] Rajeev Alur. Timed automata. NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems, 1998.

[5] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22, Trento, Italy, July 1999. Springer Berlin Heidelberg. ISBN 3-540-66202-2. doi: http://dx.doi.org/10.1007/3-540-48683-6_3.

[6] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In Michael S. Paterson, editor, *Proceedings of the 17th International Colloquium on Automata, Languages and Programming (ICALP '90)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335, Trento, Italy, July 1990. Springer Berlin Heidelberg. ISBN 978-3-540-52826-5. doi: http://dx.doi.org/10.1007/BFb0032042.

[7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(94)90010-8.

[8] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM (JACM)*, 41(1):181–203, January 1994. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/174644.174651.

[9] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings of the Fifth Annual IEEE Symposium on Logic*

*in Computer Science (LICS '90)*, pages 414–425, Philadelphia, PA, USA, June 1990. IEEE Computer Society. doi: http://dx.doi.org/10.1109/LICS.1990.113766.

[10] Rajeev Alur, Costas Courcoubetis, and David Dill. Verifying automata specifications of probabilistic real-time systems. In Jaco W. de Bakker, Cornelis Huizing, Willem-Paul de Roever, and Grzegorz Rozenberg, editors, *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 28–44, Mook, The Netherlands, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-55564-3. doi: http://dx.doi.org/10.1007/BFb0031986.

[11] Rajeev Alur, Costas Courcoubetis, David Dill, Nicolas Halbwachs, and Howard Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the Real-Time Systems Symposium (RTSS '92)*, pages 157–166, Phoenix, AZ, USA, December 1992. IEEE Computer Society. doi: http://dx.doi.org/10.1109/REAL.1992.242667.

[12] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, May 1993. doi: http://dx.doi.org/10.1006/inco.1993.1024.

[13] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing (STOC '93)*, pages 592–601, San Diego, CA, USA, 1993. ACM. ISBN 0-89791-591-7. doi: http://doi.acm.org/10.1145/167088.167242.

[14] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.489079.

[15] Henrik Reif Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 1994. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(94)90266-6.

[16] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 2009.

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

[18] Thomas Ball and Orna Kupferman. Vacuity in testing. In Bernhard Beckert and Reiner Hähnle, editors, *Proceedings of the 2nd International Conference on*

*Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 4–17, Prato, Italy, 2008. Springer Berlin Heidelberg. ISBN 3-540-79123-X, 978-3-540-79123-2. doi: http://dx.doi.org/10.1007/978-3-540-79124-9_2.

[19] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in ACTL formulas. In Orna Grumberg, editor, *Proceedings of the International Conference on Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 279–290, Haifa, Israel, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-63166-8. doi: http://dx.doi.org/10.1007/3-540-63166-6_28.

[20] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, 2001. ISSN 0925-9856. doi: http://dx.doi.org/10.1023/A:1008779610539.

[21] Gerd Behrmann, Kim G. Larsen, Justin Pearson, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In Nicholas Halbwachs and Doron Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-66202-0. doi: http://dx.doi.org/10.1007/3-540-48683-6_30.

[22] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal implementation secrets. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Proceedings of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT '02)*, volume 2469 of *Lecture Notes in Computer Science*, pages 3–22, Oldenburg, Germany, September 2002. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/3-540-45739-9_1.

[23] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio 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 '04)*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236, Bertinoro, Italy, September 2004. Springer Berlin Heidelberg. ISBN 978-3-540-23068-7. doi: http://dx.doi.org/10.1007/b110123.

[24] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing Uppaal over 15 years. *Software Prac-*

*tice and Experience*, 41(2):133–142, February 2011. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.1006.

[25] Ramzi Ben Salah, Marius Dorel Bozga, and Oded Maler. Compositional timing analysis. In *Proceedings of the seventh ACM international conference on Embedded software (EMSOFT '09)*, pages 39–48, Grenoble, France, October 2009. ACM. ISBN 978-1-60558-627-4. doi: http://doi.acm.org/10.1145/1629335.1629342.

[26] Johan Bengtsson. Memtime download page, 2002. URL `http://www.update.uu.se/~johanb/memtime/`.

[27] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin Heidelberg, 2004. doi: http://dx.doi.org/10.1007/978-3-540-27755-2_3.

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

[29] Dirk Beyer and Andreas Noack. Can decision diagrams overcome state space explosion in real-time verification? In Hartmut Köonig, Monika Heiner, and Adam Wolisz, editors, *Proceedings of the 23rd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE '03)*, volume 2767 of *Lecture Notes in Computer Science*, pages 193–208, Berlin, Germany, 2003. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/978-3-540-39979-7_13.

[30] Dirk Beyer, Claus Lewerentz, and Andreas Noack. Rabbit: A tool for BDD-based verification of real-time systems. In Warren A. Hunt Jr and Fabio Somenzi, editors, *Proceedings of the 15th Internation Conference on Computer Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 122–125, Boulder, CO, USA, 2003. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/978-3-540-45069-6_13.

[31] Girish Bhat and Rance Cleaveland. Efficient model checking via the equational $\mu$-calculus. In *Proceedings of the 11th Annual IEEE Symposium on Logic and Computer Science (LICS '96)*, pages 304–312, New Brunswick, NJ, USA, July 1996. IEEE Computer Society. doi: http://dx.doi.org/10.1109/LICS.1996.561358.

[32] Sébastien Bornot and Joseph Sifakis. On the composition of hybrid systems. In Thomas Henzinger and Shankar Sastry, editors, *First International Workshop in Hybrid Systems: Computation and Control (HSCC '98)*, volume 1386 of *Lecture Notes in Computer Science*, pages 49–63. Springer Berlin Heidelberg, 1998. doi: http://dx.doi.org/10.1007/3-540-64358-3_31.

[33] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *International Symposium on Compositionality: The Significant Difference (COMPOS)*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer Berlin Heidelberg, 1998. doi: http://dx.doi.org/10.1007/3-540-49213-5_5.

[34] Ahmed Bouajjani, Stavros Tripakis, and Sergio Yovine. On-the-fly symbolic model checking for real-time systems. In *The 18th IEEE Proceedings on Real-Time Systems Symposium (RTSS '97)*, pages 25–34, San Francisco, CA, USA, December 1997. IEEE Computer Society. doi: http://dx.doi.org/10.1109/REAL.1997.641266.

[35] Patricia Bouyer. Timed automata may cause some troubles. Technical Report RS-02-35, BRICS-Aalborg University, August 2002.

[36] Patricia Bouyer. Untameable timed automata! In Helmut Alt and Michel Habib, editors, *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science (STACS '03)*, volume 2607 of *Lecture Notes in Computer Science*, pages 620–631. Springer Berlin Heidelberg, 2003. doi: http://dx.doi.org/10.1007/3-540-36494-3_54.

[37] Patricia Bouyer. Model-checking timed temporal logics. *Electronic Notes in Theoretical Computer Science*, 231(0):323–341, March 2009. ISSN 1571-0661. doi: http://dx.doi.org/10.1016/j.entcs.2009.02.044. Proceedings of the 5th Workshop on Methods for Modalities (M4M5 2007).

[38] Patricia Bouyer and François Laroussinie. Model checking timed automata. In Stephan Merz and Nicolas Navet, editors, *Modeling and Verification of Real-Time Systems*, chapter 4, pages 111–140. ISTE, London, UK, 2010. doi: http://dx.doi.org/10.1002/9780470611012.ch4.

[39] Patricia Bouyer, Catherine Dufourd, Emmanuel Fleury, and Antoine Petit. Updatable timed automata. *Theoretical Computer Science*, 321(2–3):291–345, 2004. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2004.04.003.

[40] Patricia Bouyer, Franck Cassez, and François Laroussinie. Modal logics for timed control. Technical Report RI-2005-2, IRCCyN/CNRS, Nantes, March 2005. URL `http://www.lsv.ens-cachan.fr/~fl/cmcweb.html`.

[41] Patricia Bouyer, Franck Cassez, and François Laroussinie. Modal logics for timed control. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR '05)*, volume 3653 of *Lecture Notes in Computer Science*, pages 81–94, San Francisco, CA, USA, August 2005. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/11539452_10.

[42] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of TPTL and MTL. In Sundar Sarukkai and Sandeep Sen, editors, *Proceedings of the 25th International Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS '05)*, volume 3821 of *Lecture Notes in Computer Science*, pages 432–443, Hyderabad, India, 2005. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/11590156_35.

[43] Patricia Bouyer, François Laroussinie, and Pierre-Alain Reynier. Diagonal constraints in timed automata: Forward analysis of timed systems. In Paul Pettersson and Wang Yi, editors, *Proceedings of the 3rd International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS '05)*, volume 3829 of *Lecture Notes in Computer Science*, pages 112–126. Springer Berlin Heidelberg, 2005. doi: http://dx.doi.org/10.1007/11603009_10.

[44] Patricia Bouyer, Fabrice Chevalier, and Nicolas Markey. On the expressiveness of TPTL and MTL. *Information and Computation*, 208(2):97–116, 2010. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/j.ic.2009.10.004.

[45] Patricia Bouyer, Franck Cassez, and François Laroussinie. Timed modal logics for real-time systems. *Journal of Logic, Language and Information*, 20(2):169–203, 2011. ISSN 0925-8531. doi: http://dx.doi.org/10.1007/s10849-010-9127-4.

[46] Howard Bowman. Time and action lock freedom properties for timed automata. In Myungchui Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Proceedings of the 21st International Conference on Formal Techniques for Networked and Distributed Systems (FORTE '01)*, volume 69 of *IFIP: International Federation for Information Processing*, pages 119–134, Deventer, The Netherlands, 2001. Springer US. ISBN 0-7923-7470-3. doi: http://dx.doi.org/10.1007/0-306-47003-9_8.

[47] Howard Bowman and Rodolfo Gómez. How to stop time stopping. *Formal Aspects of Computing*, 18(4):459–493, December 2006. doi: http://dx.doi.org/10.1007/s00165-006-0010-7.

[48] Julian Bradfield and Colin Stirling. Modal logics and mu-calculi: An introduction. In Jan A. Bergstra, A. Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier Science, 2001. doi: http://dx.doi.org/10.1016/B978-044482830-9/50022-9.

[49] Julian Bradfield and Colin Stirling. Modal mu-calculi. In Patrick Blackburn, Johan Van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, volume 3, chapter 12, pages 721–756. Elsevier Science, 2007. doi: http://dx.doi.org/10.1016/S1570-2464(07)80015-2.

[50] Hana Chockler and Ofer Strichman. Before and after vacuity. *Formal Methods in System Design*, 34(1):37–58, 2009. ISSN 0925-9856. doi: http://dx.doi.org/10.1007/s10703-008-0060-y.

[51] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. Proving and explaining the unfeasibility of message sequence charts for hybrid systems. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD '11)*, pages 54–62, Austin, TX, USA, 2011. ISBN 978-0-9835678-1-3. URL `http://dl.acm.org/citation.cfm?id=2157654.2157666`.

[52] Edmund Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer Berlin Heidelberg, 1989. ISBN 978-3-540-51080-2. doi: http://dx.doi.org/10.1007/BFb0013029.

[53] Edmund M. Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking: History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69849-4. doi: http://dx.doi.org/10.1007/978-3-540-69850-0_1.

[54] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/5397.5399.

[55] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999.

[56] Rance Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–747, 1990. ISSN 0001-5903. doi: http://dx.doi.org/10.1007/BF00264284.

[57] Rance Cleaveland and Bernhard Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, 1993. ISSN 0925-9856. doi: http://dx.doi.org/10.1007/BF01383878.

[58] Thomas H. Cormen, Clarles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, second edition, 2001.

[59] Patrick Cousot and Radhia Cousot. Constructive versions of tarski's fixed point theorems. *Pacific Journal of Mathematics*, 82(1):43–57, 1979. URL `http://projecteuclid.org/euclid.pjm/1102785059`.

[60] Conrado Daws, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. The tool KRONOS. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer Berlin Heidelberg, 1996. doi: http://dx.doi.org/10.1007/BFb0020947.

[61] Vasileios Deligiannis and Stamatis Manesis. A survey on automata-based methods for modelling and simulation of industrial systems. In *IEEE Conference on Emerging Technologies and Factory Automation (EFTA '07)*, pages 398–405. IEEE, September 2007. doi: http://dx.doi.org/10.1109/EFTA.2007.4416795.

[62] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/Cole—Thomson Learning, Belmont, CA, USA, sixth edition, 2003.

[63] Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli. Vacuity analysis for property qualification by mutation of checkers. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*, pages 478–483. European Design and Automation Association, 2010. ISBN 978-3-9810801-6-2. URL `http://dl.acm.org/citation.cfm?id=1870926.1871041`.

[64] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In Joseph Sifakis, editor, *Proceedings of the International Workshop on*

*Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer Berlin Heidelberg, June 1990. ISBN 3-540-52148-8. doi: http://dx.doi.org/10.1007/3-540-52148-8_17.

[65] Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, November–December 2008. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2008.52.

[66] Yifei Dong, Beata Sarna-Starosta, C. R. Ramakrishnan, and Scott A. Smolka. Vacuity checking in the modal mu-calculus. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology (AMAST '02)*, volume 2422 of *Lecture Notes in Computer Science*, pages 147–162. Springer Berlin Heidelberg, 2002. ISBN 3-540-44144-1. doi: http://dx.doi.org/10.1007/3-540-45719-4_11.

[67] Rüdiger Ehlers, Daniel Fass, Michael Gerke, and Hans-Jörg Peter. Fully symbolic timed model checking using constraint matrix diagrams. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS '10)*, pages 360–371, San Diego, CA, USA, November–December 2010. IEEE. doi: http://dx.doi.org/10.1109/RTSS.2010.36.

[68] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1):151–178, January 1986. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/4904.4999.

[69] E. Allen Emerson and Chin-Laung Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 1st Symposium on Logic in Computer Science (LICS '86)*, pages 267–278. IEEE Computer Society, June 1986.

[70] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Harcourt/Academic Press, San Diego, CA, USA, second edition, 2001.

[71] Susanna S. Epp. *Discrete Mathematics with Applications*. Brooks/Cole—Thomson Learning, Belmont, CA, USA, third edition, 2004.

[72] Uli Fahrenberg, Kim Larsen, and Claus Thrane. Verification, performance analysis and controller synthesis for real-time systems. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, volume 5961 of

*Lecture Notes in Computer Science*, pages 34–61. Springer Berlin Heidelberg, 2010. doi: http://dx.doi.org/10.1007/978-3-642-11623-0_2.

[73] Peter Fontana and Rance Cleaveland. Data structure choices for on-the-fly model checking of real-time systems. In Malay Ganai and Armin Biere, editors, *Proceedings of the First International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS '11)*, volume 832 of *CEUR Workshop Proceedings*, pages 13–21, Austin, TX, USA, November 2011. URL `http://ceur-ws.org/Vol-832/Difts11Proceedings.pdf#page=17`.

[74] Peter Fontana and Rance Cleaveland. A menagerie of timed automata. *ACM Computing Surveys*, 46(3):40:1–40:56, January 2014. doi: http://dx.doi.org/10.1145/2518102.

[75] John B. Fraleigh. *A First Course in Abstract Algebra*. Addison-Wesley, Boston, MA, USA, sixth edition, 1999.

[76] Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. Modeling time in computing: A taxonomy and a comparative survey. *ACM Computing Surveys*, 42(2):6:1–6:59, February 2010. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1667062.1667063.

[77] Mihaela Gheorghiu and Arie Gurfinkel. Vaquot: A tool for vacuity detection. Tool and Poster Track of the 14th International Symposium on Formal Methods, August 2006. URL `http://fm06.mcmaster.ca/VaqUoT.pdf`.

[78] Rodolfo Gómez and Howard Bowman. Efficient detection of zeno runs in timed automata. In Jean-Francois Raskin and P.S. Thiagarajan, editors, *Proceedings of the 5th International Conference on the Formal Modeling and Analysis of Timed Systems (FORMATS '07)*, volume 4763 of *Lecture Notes in Computer Science*, pages 195–210. Springer Berlin Heidelberg, October 2007. doi: http://dx.doi.org/10.1007/978-3-540-75454-1_15.

[79] Jan Friso Groote and Tim A.C. Willemse. Parameterised boolean equation systems. *Theoretical Computer Science*, 343(3):332–369, 2005. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/j.tcs.2005.06.016.

[80] Arie Gurfinkel and Marsha Chechik. How vacuous is vacuous? In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 451–466. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21299-7. doi: http://dx.doi.org/10.1007/978-3-540-24730-2_34.

[81] Arie Gurfinkel and Marsha Chechik. Robust vacuity for branching temporal logic. *ACM Transactions on Computational Logic (TOCL)*, 13(1):1:1–1:32, January 2012. ISSN 1529-3785. doi: http://doi.acm.org/10.1145/2071368.2071369.

[82] Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli. Formal verification of use case maps with real time extensions. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *13th International System Design Languages (SDL) Forum (SDL '07)*, volume 4745 of *Lecture Notes in Computer Science*, pages 225–241, Paris, France, 2007. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/978-3-540-74984-4_14.

[83] Jameleddine Hassine, Juergen Rilling, and Rachida Dssouli. An evaluation of timed scenario notations. *Journal of Systems and Software*, 83(2):326–350, 2010. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2009.09.014.

[84] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: a case study in formal verification of real-time systems. In *Proceedings of the Real-Time Systems Symposium (RTSS '94)*, pages 120–131. IEEE, December 1994. doi: http://dx.doi.org/10.1109/REAL.1994.342724.

[85] Constance L. Heitmeyer, Bruce G. Labaw, and Ralph D. Jeffords. A benchmark for comparing different approaches for specifying and verifying real-time systems. Technical Report ADA462244, Naval Research Laboratory, 1993. URL `http://handle.dtic.mil/100.2/ADA462244`.

[86] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, Los Alamitos, CA, USA, July 1996. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/LICS.1996.561342.

[87] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science (LICS '92)*, pages 394–406. IEEE, June 1992. doi: http://dx.doi.org/10.1109/LICS.1992.185551.

[88] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994. doi: http://dx.doi.org/10.1006/inco.1994.1045.

[89] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing (STOC '95)*, pages

373–382, Las Vegas, Nevada, USA, 1995. ACM. ISBN 0-89791-718-9. doi: http://doi.acm.org/10.1145/225058.225162.

[90] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: a model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):110–122, 1997. ISSN 1433-2779. doi: http://dx.doi.org/10.1007/s100090050008.

[91] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94–124, 1998. ISSN 0022-0000. doi: http://dx.doi.org/10.1006/jcss.1998.1581.

[92] Paula Herber, Joachim Fellmuth, and Sabine Glesner. Model checking SystemC designs using timed automata. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (CODES+ISSS '08)*, pages 131–136, Atlanta, GA, USA, 2008. ACM. ISBN 978-1-60558-470-6. doi: http://doi.acm.org/10.1145/1450135.1450166.

[93] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, MA, USA, second edition, 2001.

[94] Magdalena Kacprzak, Wojciech Nabiałek, Artur Niewiadomski, Wojciech Penczek, Agata Półrola, Maciej Szreter, Bożzena Woźna, and Andrzej Zbrzezny. VerICS 2007 - a model checker for knowledge and real-time. *Fundamenta Informaticae*, 85(1-4):313–328, 2008. ISSN 01692968. URL `http://iospress.metapress.com/content/aju14j810h82w141/`.

[95] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed I/O automata: a mathematical framework for modeling and analyzing real-time systems. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS '03)*, pages 166–177. IEEE, December 2003. doi: http://dx.doi.org/10.1109/REAL.2003.1253264.

[96] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, San Raftel, CA, USA, second edition, 2010. doi: http://dx.doi.org/10.2200/S00310ED1V01Y201011DCT005.

[97] Lina Khatib, Nicola Muscettola, and Klaus Havelund. Mapping temporal planning constraints into timed automata. In *Proceedings of the 8th International Symposium on Temporal Representation and Reasoning (TIME '01)*, pages

21–27, Cividale del Friuli, Italy, June 2001. IEEE Computer Society. doi: http://dx.doi.org/10.1109/TIME.2001.930693.

[98] Ahmet Koltuksuz, Brucu Kulahcioglu, and Murat Ozkan. Utilization of timed automata as a verification tool for security protocols. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement, Companion Volume (SSIRI-C '10)*, pages 86–93. IEEE Computer Society, June 2010. doi: http://dx.doi.org/10.1109/SSIRI-C.2010.27.

[99] Maria Kourkouli and George Hassapis. Application of the timed automata abstraction to the performance evaluation of the architecture of a bank on-line transaction processing system. In George Eleftherakis, editor, *Proceedings of the 2nd South-East European Workshop on Formal Methods (SEEFM)*, pages 142–153. South-East European Research Centre (SEERC), 2006. ISBN 960-87869-8-3. URL `http://www.seefm.info/seefm05/book/11-C5-SEEFM05.pdf`.

[100] Dexter Kozen. Results on the propositional $\mu$-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(82)90125-6.

[101] Orna Kupferman. Sanity checks in formal verification. In Christel Baier and Holger Hermanns, editors, *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR '06)*, volume 4137 of *Lecture Notes in Computer Science*, pages 37–51, Bonn, Germany, August 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37376-6. doi: http://dx.doi.org/10.1007/11817949_3.

[102] Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, 2003. ISSN 1433-2779. doi: http://dx.doi.org/10.1007/s100090100062.

[103] Sebastian Kupferschmid, Martin Wehrle, Bernhard Nebel, and Andreas Podelski. Faster than Uppaal? In Aarti Gupta and Sharad Malik, editors, *Proceedings of the 20th International Conference on Computer Aided Verification (CAV '08)*, volume 5123 of *Lecture Notes in Computer Science*, pages 552–555, Princeton, NJ, USA, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70543-7. doi: http://dx.doi.org/10.1007/978-3-540-70545-1_53.

[104] Leslie Lamport. "Sometime" is sometimes "not never": on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '80)*, pages 174–185, Las

Vegas, Nevada, 1980. ACM. ISBN 0-89791-011-7. doi: http://doi.acm.org/10.1145/567446.567463.

[105] Leslie Lamport. Real-time model checking is really simple. In Dominique Borrione and Wolfgang Paul, editors, *Proceedings of the Advanced Research Working Conference on Correct Hardware-like Design and Verification Methods (CHARME '05)*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175. Springer Berlin Heidelberg, 2005. doi: http://dx.doi.org/10.1007/11560548_14.

[106] Christine Largouët, Marie-Odile Cordier, Yves-Marie Bozec, Yulong Zhao, and Guy Fontenelle. Use of timed automata and model-checking to explore scenarios on ecosystem models. *Environmental Modelling & Software*, 30:123–138, 2012. ISSN 1364-8152. doi: http://dx.doi.org/10.1016/j.envsoft.2011.08.005.

[107] François Laroussinie and Kim Guldstrand Larsen. CMC: A tool for compositional model-checking of real-time systems. In Stan Budkowski, Ana Cavalli, and Elie Najm, editors, *Proceedings of the Joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV '98)*, The International Federation for Information Processing (IFIP), pages 439–456, Paris, France, 1998. Springer US. ISBN 0-412-84760-4. doi: http://dx.doi.org/10.1007/978-0-387-35394-4_27.

[108] François Laroussinie, Kim Larsen, and Carsten Weise. From timed automata to logic — and back. In Jirí Wiedermann and Petr Hájek, editors, *Proceedings of the 20th Annual Symposium on the Mathematical Foundations of Computer Science (MFCS '95)*, volume 969 of *Lecture Notes in Computer Science*, pages 529–539, Prague, Czech Republic, August 1995. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/3-540-60246-1_158.

[109] François Laroussinie, Kim G. Larsen, and Carsten Weise. From timed automata to logic - and back. Technical Report BRICS RS-95-2, BRICS-Aalborg University, 1995. URL `http://www.brics.dk/RS/95/2/`.

[110] Kim Larsen and Wang Yi. Time abstracted bisimulation: Implicit specifications and decidability. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics (MFPS '94)*, volume 802 of *Lecture Notes in Computer Science*, pages 160–176, New Orleans, LA, USA, April 1994. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/3-540-58027-1_8.

[111] Kim G. Larsen and Yi Wang. Time-abstracted bisimulation: Implicit specifications and decidability. *Information and Computation*, 134(2):75–101, 1997. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1997.2623.

[112] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 14–24. IEEE Computer Society, December 1997. doi: http://dx.doi.org/10.1109/REAL.1997.641265.

[113] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):134–152, 1997. doi: http://dx.doi.org/10.1007/s100090050010.

[114] Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999. ISSN 1236-6064. URL `http://www.cs.helsinki.fi/njc/References/larsenpwy1999:271.html`.

[115] Huimin Lin. Symbolic transition graph with assignment. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 50–65. Springer Berlin Heidelberg, 1996. ISBN 3-540-61604-7. doi: http://dx.doi.org/10.1007/3-540-61604-7_47.

[116] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. In Bernhard Steffen, editor, *Proceedings of the International Conference on the Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 281–297. Springer Berlin Heidelberg, 1998. doi: http://dx.doi.org/10.1007/BFb0054178.

[117] Nancy Lynch and Frits Vaandrager. Forward and backward simulations for timing-based systems. In Jaco de Bakker, Cornelis Huizing, Willem-Paul de Roever, and Grzegorz Rozenberg, editors, *Proceedings of the Workshop on Research and Education in Concurrent Systems (REX)*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446. Springer Berlin Heidelberg, 1992. doi: http://dx.doi.org/10.1007/BFb0032002.

[118] Nancy Lynch and Frits Vaandrager. Forward and backward simulations part I: Untimed systems. *Information and Computation*, 121(2):214–233, 1995. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1995.1134.

[119] Nancy Lynch and Frits Vaandrager. Forward and backward simulations part II: Timing systems. Technical Report MIT-LCS-TM-458, MIT Laboratory for Computer Science, 1995. URL `http://groups.csail.mit.edu/tds/papers/Lynch/MIT-LCS-TM-458.pdf`.

[120] Nancy Lynch and Frits Vaandrager. Forward and backward simulations: II. timing-based systems. *Information and Computation*, 128(1):1–25, 1996. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1996.0060.

[121] Angelika Mader. *Verification of Modal Properties Using Boolean Equation Systems*. Edition versal 8. Bertz Verlag, Berlin, Germany, 1997. URL `http://doc.utwente.nl/64253/`.

[122] Oded Maler and Grégory Batt. Approximating continuous systems by timed automata. In Jasmin Fisher, editor, *Proceedings of the First International Workshop on Formal Methods in Systems Biology (FMSB '08)*, volume 5054 of *Lecture Notes in Computer Science*, pages 77–89, Cambridge, UK, June 2008. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/978-3-540-68413-8_6.

[123] Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, 2003. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(02)00094-1.

[124] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0131149849.

[125] Jesper Møller. DDDLIB: a library for solving quantified difference inequalities. In Andrei Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE '02)*, volume 2392 of *Lecture Notes in Computer Science*, pages 221–241, Copenhagen, Denmark, July 2002. Springer Berlin Heidelberg. URL `http://dx.doi.org/10.1007/3-540-45620-1_9`.

[126] Jesper Møller, Jakob Lichtenberg, Henrik Andersen, and Henrik Hulgaard. Difference decision diagrams. In Jörg Flum and Mario Rodriguez-Artalejo, editors, *Proceedings of the 13th Annual Workshop of Computer Science Logic (CSL '99)*, volume 1683 of *Lecture Notes in Computer Science*, pages 826–826. Springer Berlin Heidelberg, September 1999. doi: http://dx.doi.org/10.1007/3-540-48168-0_9.

[127] Georges Morbé, Florian Pigorsch, and Christoph Scholl. Fully symbolic model checking for timed automata. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer*

*Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 616–632. Springer Berlin Heidelberg, 2011. doi: http://dx.doi.org/10.1007/978-3-642-22110-1_50.

[128] Kedar S. Namjoshi. Certifying model checkers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 2–13. Springer Berlin Heidelberg, July 2001. ISBN 978-3-540-42345-4. doi: http://dx.doi.org/10.1007/3-540-44585-4_2.

[129] Kedar S. Namjoshi. An efficiently checkable, proof-based formulation of vacuity in model checking. In Rajeev Alur and DoronA. Peled, editors, *Proceedings of the 16th Annual Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 57–69, Boston, MA, USA, July 2004. Springer Berlin Heidelberg. ISBN 978-3-540-22342-9. doi: http://dx.doi.org/10.1007/978-3-540-27813-9_5.

[130] Peter Niebert, Moez Mahfoudh, Eugene Asarin, Marius Bozga, Oded Maler, and Navendu Jain. Verification of timed automata via satisfiability checking. In Werner Damm and Ernst Olderog, editors, *Proceedings of the 7th Annual Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '02)*, volume 2469 of *Lecture Notes in Computer Science*, pages 225–243. Springer Berlin Heidelberg, 2002. doi: http://dx.doi.org/10.1007/3-540-45739-9_15.

[131] Ernst-Rüdiger Olderog and Henning Dierks. *Real-Time Systems: Formal Specification and Automatic Verification*. Cambridge University Press, New York, NY, USA, 2008.

[132] Miroslav Pajic, Insup Lee, Rahul Mangharam, and Oleg Sokolsky. UPP2SF: Translating UPPAAL models to simulink, technical report. Technical Report, 2011. URL `http://www.seas.upenn.edu/~pajic/TEMP/UPP2SF_report.pdf`.

[133] Paritosh Pandya and Simoni Shah. On expressive powers of timed logics: Comparing boundedness, non-punctuality, and deterministic freezing. In Joost-Pieter Katoen and Barbara König, editors, *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR '11)*, volume 6901 of *Lecture Notes in Computer Science*, pages 60–75. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-23216-9. doi: http://dx.doi.org/10.1007/978-3-642-23217-6_5.

[134] Wojciech Penczek and Agata Półrola. Specification and model checking of temporal properties in time petri nets and timed automata. In Jordi Cor-

tadella and Wolfgang Reisig, editors, *Proceedings of the 25th International Conference on the Applications and Theory of Petri Nets (ICATPN '04)*, volume 3099 of *Lecture Notes in Computer Science*, pages 37–76. Springer Berlin Heidelberg, June 2004. doi: http://dx.doi.org/10.1007/978-3-540-27793-4_4.

[135] Wojciech Penczek and Agata Pólrola. *Advances in Verification of Time Petri Nets and Timed Automata*, volume 20 of *Studies in Computational Intelligence*. Springer Berlin Heidelberg, Secaucus, NJ, USA, 2006. doi: http://dx.doi.org/10.1007/978-3-540-32870-4.

[136] Hans-Jörg Peter, Rüdiger Ehlers, and Robert Mattmüller. Synthia: Verification and synthesis for timed automata. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 649–655. Springer Berlin Heidelberg, 2011. doi: http://dx.doi.org/10.1007/978-3-642-22110-1_52.

[137] André Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, Berlin, Germany, 2010. doi: http://dx.doi.org/10.1007/978-3-642-14509-4.

[138] Amalinda Post, Jochen Hoenicke, and Andreas Podelski. Vacuous real-time requirements. In *Proceedings of the 19th IEEE International Conference on Requirements Engineering (RE '11)*, pages 153–162. IEEE, August–September 2011. doi: http://dx.doi.org/10.1109/RE.2011.6051657.

[139] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. URL `http://www.R-project.org/`.

[140] Anders Ravn, Jiří Srba, and Saleem Vighio. Modelling and verification of web services business activity protocol. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '11)*, volume 6605 of *Lecture Notes in Computer Science*, pages 357–371. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19834-2. doi: http://dx.doi.org/10.1007/978-3-642-19835-9_32.

[141] Marko Samer and Helmut Veith. On the notion of vacuous truth. In Nachum Dershowitz and Andrei Voronkov, editors, *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '07)*, volume 4790 of *Lecture Notes in Computer Science*, pages 2–14. Springer

Berlin Heidelberg, 2007. ISBN 978-3-540-75558-6. doi: http://dx.doi.org/10.1007/978-3-540-75560-9_2.

[142] Edward R. Scheinerman. *Mathematics: A Discrete Introduction*. Brooks/Cole—Thomson Learning, Pacific Grove, CA, USA, first edition, 2000.

[143] Sanjit Seshia and Randal Bryant. Unbounded, fully symbolic model checking of timed automata using boolean methods. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03)*, volume 2742 of *Lecture Notes in Computer Science*, pages 154–166. Springer Berlin Heidelberg, 2003. doi: http://dx.doi.org/10.1007/978-3-540-45069-6_16.

[144] Sanjit A. Seshia and Randal E. Bryant. A boolean approach to unbounded, fully symbolic model checking of timed automata. Technical Report CMS-CS-03-117, Carnegie Mellon University, 2003. URL `http://www.eecs.berkeley.edu/~sseshia/pubdir/tr-03-117.ps`.

[145] Michael Sipser. *Introduction to the Theory of Computation*. Thompson Course Technology, Boston, MA, USA, second edition, 2006.

[146] Christoffer Sloth and Rafael Wisniewski. Verification of continuous dynamical systems by timed automata. *Formal Methods in System Design*, 39 (1):1–36, August 2011. ISSN 0925-9856. doi: http://dx.doi.org/10.1007/s10703-011-0118-0.

[147] Oleg V. Sokolsky and Scott A. Smolka. Local model checking for real-time systems. In Pierre Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided Verification (CAV '95)*, volume 939 of *Lecture Notes in Computer Science*, pages 211–224. Springer Berlin Heidelberg, July 1995. doi: http://dx.doi.org/10.1007/3-540-60045-0_52.

[148] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. URL `http://www.projecteuclid.org/DPubS?verb=Display&version=1.0&service=UI&handle=euclid.pjm/1103044538&page=record`.

[149] Serdar Tasiran, Rajeev Alur, Robert P. Kurshan, and Robert K. Brayton. Verifying abstractions of timed systems. In Ugo Montanari and Vladimiro Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96)*, volume 1119 of *Lecture Notes in Computer Science*,

pages 546–562. Springer Berlin Heidelberg, 1996. ISBN 3-540-61604-7. doi: http://dx.doi.org/10.1007/3-540-61604-7_75.

[150] Stavros Tripakis. Verifying progress in timed systems. In Joost-Pieter Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *Lecture Notes in Computer Science*, pages 299–314. Springer Berlin Heidelberg, 1999. doi: http://dx.doi.org/10.1007/3-540-48778-6_18.

[151] Stavros Tripakis. Checking timed büchi automata emptiness on simulation graphs. *ACM Transactions on Computational Logic*, 10(3):1–19, 2009. ISSN 1529-3785. doi: http://doi.acm.org/10.1145/1507244.1507245.

[152] Stavros Tripakis and Sergio Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, January 2001. ISSN 0925-9856 (Print) 1572-8102 (Online). doi: http://dx.doi.org/10.1023/A:1008734703554.

[153] Farn Wang. Clock restriction diagram: Yet another data-structure for fully symbolic verification of timed automata. Technical Report TR-IIS-01-002, Institute of Information Science, Academia Sinica, 2001.

[154] Farn Wang. Efficient verification of timed automata with BDD-like data-structures. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 189–205. Springer Berlin Heidelberg, 2003. ISBN 3-540-00348-7. doi: http://dx.doi.org/10.1007/3-540-36384-X_17.

[155] Farn Wang. Efficient verification of timed automata with BDD-like data structures. *International Journal on Software Tools for Technology Transfer*, 6(1):77–97, 2004. ISSN 1433-2779. doi: http://dx.doi.org/10.1007/s10009-003-0135-4.

[156] Farn Wang. Formal verification of timed systems: A survey and perspective. *Proceedings of the IEEE*, 92(8):1283–1307, 2004. doi: http://dx.doi.org/10.1109/JPROC.2004.831197.

[157] Farn Wang. Redlib for the formal verification of embedded systems. In *Proceedings of the Second Internation Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '06)*, pages 341–346. IEEE Computer Society, November 2006. doi: http://dx.doi.org/10.1109/ISoLA.2006.68.

[158] Farn Wang. Specification formalisms and models. In Benjamin Wah, editor, *Wiley Encyclopedia of Computer Science and Engineering*, pages 2775–2789. John Wiley & Sons, Inc., 2007. doi: http://dx.doi.org/10.1002/9780470050118.ecse410.

[159] Farn Wang. Time-progress evaluation for dense-time automata with concave path conditions. In Sungdeok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA '08)*, volume 5311 of *Lecture Notes in Computer Science*, pages 258–273, Seoul, Korea, October 2008. Springer Berlin Heidelberg. doi: http://dx.doi.org/10.1007/978-3-540-88387-6_24.

[160] Farn Wang and Pao-Ann Hsiung. Efficient and user-friendly verification. *IEEE Transactions on Computers*, 51(1):61–83, January 2002. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/12.980017.

[161] Farn Wang, Geng-Dian Huang, and Fang Yu. TCTL inevitability analysis of dense-time systems: From theory to engineering. *IEEE Transactions on Software Engineering*, 32(7):510–526, July 2006. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2006.71.

[162] Farn Wang, Li-Wei Yao, and Ya-Lan Yang. Efficient verification of distributed real-time systems with broadcasting behaviors. *Real-Time Systems*, 47(4):285–318, 2011. ISSN 0922-6443. doi: http://dx.doi.org/10.1007/s11241-011-9122-0.

[163] Sergio Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):123–133, 12 1997. doi: http://dx.doi.org/10.1007/s100090050009.

[164] Sergio Yovine. Model checking timed automata. In Grzegorz Rozenberg and Frits W. Vaandrager, editors, *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 114–152. Springer Berlin Heidelberg, November 1998. ISBN 3-540-65193-4. doi: http://dx.doi.org/10.1007/3-540-65193-4_20.

[165] Dezhuang Zhang. *Model Checking for Data-Based Concurrent Systems*. PhD thesis, State University of New York at Stony Brook, December 2005. URL `http://proquest.umi.com/pqdweb?did=1092095351&sid=1&Fmt=6&clientId=41143&RQT=309&VName=PQD`.

[166] Dezhuang Zhang and Rance Cleaveland. Efficient temporal-logic query checking for presburger systems. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE '05)*, pages 24–33, Long Beach, CA, USA, 2005. ACM. ISBN 1-59593-993-4. doi: http://doi.acm.org/10.1145/1101908.1101915.

[167] Dezhuang Zhang and Rance Cleaveland. Fast generic model-checking for data-based systems. In Farn Wang, editor, *Proceedings of the International Conference on the Formal Techniques for Networked and Distributed Systems (FORTE '05)*, volume 3731 of *Lecture Notes in Computer Science*, pages 83–97. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-29189-3. doi: http://dx.doi.org/10.1007/11562436_8.

[168] Dezhuang Zhang and Rance Cleaveland. Fast on-the-fly parametric real-time model checking. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, pages 157–166. IEEE Computer Society, 2005. ISBN 0-7695-2490-7. doi: http://dx.doi.org/10.1109/RTSS.2005.22.