Verification and TCTL Model Checking of Real-Time Systems on Timed Automata and Timed Kripke Structures, and Inductive Conversion Issues in Dense Time

by

Negar Nourollahi

A thesis submitted to the
Department of Computer Science
in conformity with the requirements for
the degree of Master of Science

Bishop's University
Canada
December 2019

Dedicated to my beloved parents

Abstract

The design and verification of concurrent and real-time systems are difficult problems. While model checking proved to be successful as an automatic and effective solution, real-time systems do not benefit directly from classical model checking since they feature infinite clock values. Instead, bisimulation and abstraction can be used to build an abstract finite state machine from the real-time model, which can in turn be model checked.

In the first part of this thesis we consider formal methods for the specification and automatic verification of finite-state real-time systems with dense time. Both automata and temporal logic are extended to allow them to model timing delays and to verify real-time requirements.

Time automata are chosen as the underlying semantic of real-time system because they are the standard modeling method in designing real-time systems, and their state reachability problem is also decidable. We demonstrate how the tools for analysis of untimed finite state systems can also be deployed in order to verify timed systems. While dense time in timed automata generates infinite state spaces, we show that time abstracted bisimulations are decidable for timed automata, and so strong time-abstract bisimulation can be used to reduce the infinite state-space of a given timed automata model to a finite quotient graph and finite transition system. Time-abstract bisimulation also preserves both the linear and branching time properties of the original model sufficiently for verification, while the exact time delays are abstracted away. The strongly non-zeno timed automata are also proposed as an extension of timed automata in order to address the deadlock and timelock issues in verifying timed systems. This solution is based on the finite quotient graph and comes from strong time-abstract bisimulation. We then show how TCTL model checking can be reduced to CTL model-checking using strong time-abstracting bisimulation.

In the second part of the thesis we demonstrate that the satisfaction of TCTL formulas under a natural semantics for both discrete-time and dense-time timed Kripke structures can be reduced to a model-checking problem in the point-wise semantics for timed Kripke structures. Discrete TCTL-preserving abstraction methods of timed Kripke structures, the so-called gcd-transformation and τ-transformation are introduced.

Some effort has been spent in the untimed domain to bride logical approaches (such as model checking) and algebraic approaches to formal methods (such as model-based testing). One of the necessary steps in this direction is a process of establishing an equivalence between the underlying semantic models in the two domains (Kripke structures and labeled transition systems, respectively. In particular, several inductive, algorithmic methods that generate a compact Kripke structure equivalent with given labeled transition systems were developed. We intended to extend this effort to the dense time domain, but we found instead that inductive conversion methods are infeasible in large scale, concurrent real-time systems with dense time domain. While both timed automata and timed Kripke structures can be used for verification in the dense time domain, we cannot feasibility convert timed automata to timed Kripke structures in any feasible way. The reasons include the undecidability of trace properties for timed automata and also an inherent state explosion problem for any inductive conversion algorithm.

Acknowledgements

I would like to appreciate my supervisor Prof. Dr. Stefan D. Bruda for his helpful suggestions, encouragement and advice. I am thankful for his interesting ideas in real-time systems research area and support.

Contents

List of Figures

Notations

U	Until
X	Next time
F	Eventually or in the future
G	Always or globally
R	Release
TS	Transition system
S	Set of states
Act	Set of actions
$\rightarrow$	Transition relation
I	Initial states
AP	Set of atomic propositions
L	Labeling function
Q	Finite set of discrete states
Σ	Alphabet
δ	Transition function
Q_0	Set of initial states
F	Set of accepting or final states
$\lvert A \rvert$	Size of the non-deterministic finite automaton
$\mathcal{L}$	Language accepted by an automaton
$\delta^* \left(q, w\right)$	Set of states that are reachable from q for the input word w (extended transition function)
$\text{Reach}\left(q\right)$	Set of states that are reachable via an arbitrary run starting in state q
DFA	Deterministic finite automaton
q_{trap}	Non-final trap state
σ^ω	Set of all infinite words over $\sum$
$\sim$	Bisimulation equivalence
$Post\left(s\right)$	Every outgoing transition of s
π	Path
$\equiv_{tt}$	Time trace equivalence
$\mathcal{X} = \{x_1, ..., x_n\}$	Finite set of clocks
$\mathbf{R}$	Set of non-negative reals
$\mathbf{v}$	Valuation

$\mathbf{R}^{\mathcal{X}}$	Set of all valuations
TA	Timed Automata
$\textit{disc-pred}(e, S)$	The set of all e-predecessors of states in S
$\textit{time-pred}(S)$	The set of all time-predecessors of states in S
$\approx$	Strong time-abstracting bisimulation (STaB)
$\textit{Props}$	Set of atomic propositions
P	A function associating to each proposition a set of states of the automaton
τ	Time lapse
$\mathcal{C}_0$	Initial partition
ρ	An infinite run
KS	Kripke structures
$\mathcal{TK}$	Timed Kripke Structure
$TCTL$	Timed Computation Tree Logic
A^+	Extended Timed Automaton
$\mathbb{N}$	Set of natural numbers
$\mathbb{Q}_{\geq 0}$	Non-negative rational numbers
$\textit{Intervals}(\mathcal{T})$	Set of all time intervals
$\mathcal{T}$	Time domain satisfying the theory TIME
$\mathcal{C}^p_{\mathcal{TK}}$	Valid configurations of $\mathcal{TK}$ in the point-wise semantic
$\mathcal{C}^c_{\mathcal{TK}}$	Valid configurations of $\mathcal{TK}$ in the continuous semantic
$\text{Paths}^P_{\mathcal{TK}}(\langle s, 0\rangle)$	The set of all the paths in the point-wise interpretation of $\mathcal{TK}$
$\text{Paths}^c_{\mathcal{TK}}(\langle s, \delta\rangle)$	The set of all paths in the continuous interpretation of $\mathcal{TK}$
$\text{tdPaths}^p_{\mathcal{TK}}(\langle s, 0\rangle)$	The set of all time-divergent paths in the point-wise interpretation of $\mathcal{TK}$
$\text{tdPaths}^c_{\mathcal{TK}}(\langle s, \delta\rangle)$	The set of all time-divergent paths in the continuous interpretation of $\mathcal{TK}$
GCD	Greatest common divisor
TKS	Timed Kripke Structures
$\models_p$	The point-wise satisfaction relation
$\models_c$	The continuous satisfaction relation
TCTL_{cb}	TCTL formula with closed interval
$\beta(\varphi)$	A transformation of φ
$\alpha(\varphi)$	The transformation of open intervals φ formula to the closed intervals formula
γ	The greatest common divisor (gcd)
γ_h	The half of the greatest common divisor (gcd)
$\mathcal{TK}^{\gamma}$	gcd-transformation of timed kripke structure
$\mathcal{TK}^{\tau}$	τ -transformation of timed kripke structure
$\text{can}_{\mathcal{TK}^{\tau}}$	The canonical representative
$\text{can}(\sigma)$	The canonical representative of the valid configuration σ
L_a	Extended labeling function
$\varphi!$	Normal form of TCTL formula

$Sat\,(\mathcal{TK}, \varphi!)$	The satisfaction set of the normal form formula
$(SCC\,(\varphi_1))$	The strongly connected component of φ_1-states
Q_{SCC}	States related to some non-trivial strongly connected components of φ_1 states
LTS	Labeled transition system

Chapter 1

Introduction

Computing systems are extensively used nowadays in applications where failure is unacceptable such as electronic commerce, medical instruments, telephone switching networks, air traffic control systems and so on. The need for reliable computing systems is critical, yet we frequently read about incidents where some failure is caused by a computing system error. As the involvement of such systems in our lives increases dramatically, it becomes crucial to ensure their correctness. It is no longer feasible to shut down a malfunctioning system in order to restore safety since we are so very much dependent on such a system. In some cases, devices are even less safe when they are shut down. The consequences of replacing critical code or circuitry can also be economically devastating.

Due to the high growth of the Internet and embedded systems in automobiles, airplanes, and other safety-critical systems, we will be even more dependent on the proper functioning of computing devices in the future. It will therefore become essential to develop methods that raise our confidence in the correctness of these systems [53].

Concurrency and real-time are important techniques, broadly utilized in modern systems, which features unanticipated interactions and race conditions. Ensuring reliability and correctness for concurrent and real-time systems poses numerous difficulties. In the last two decades researchers have extended different formal methods to the analysis of concurrent and real-time systems. We use the term "formal methods" to refer to the variety of formalisms that can be utilized to determine the behavior of a system and to mathematically verify the logical correctness of that system's (functional and nonfunctional) requirements and properties. Formal methods are currently the most promising and automated method utilized at an early stage during the process of software development.

Studies have repeatedly demonstrated that most of the cost of software development stems from design (or requirement) defects. Defects in design can cost a hundred times or more to fix in the testing and maintenance phases than in the design phase. Formal methods enable us to recognize those defects at the early stage of the software life cycle. We can remove these problems earlier and significantly decrease the cost of debugging, maintenance, and re-development. Moreover, formal methods confidently ensure that safety-critical systems satisfy the desired properties in order to avoid disastrous consequences. Many formalisms with different degrees of rigor have been utilized in formal methods for specification, modeling, and verification. Formal methods encompass mathematical logic, graphical notations, state machine models, and process algebras [63].

1.1 Formal Verification

The main validation methods for complex systems are a simulation, testing, deductive verification, and model checking. Simulation and testing [82] both involve making experiments before using the system in the field. While simulation is carried out on an abstraction or a model of the system, testing is done on the actual product. In the case of circuits, simulation is carried out on the design of the circuit, whereas testing is done on the circuit itself. In both situations, these techniques inject signals at specific points in the system and observe the resulting signals at other points. For software, simulation and testing generally involve preparing determined inputs and observing the corresponding outputs. These techniques can be a cost-efficient method to find many errors. However, checking all of the possible interactions and potential pitfalls utilizing simulation and testing techniques is rarely possible.

This problem of design validation (ensuring the correctness of the design at the earliest stage possible) is the main challenge in any responsible system development process, and the tasks intended for its solution occupy a huge portions of the development cycle cost and time budgets. The currently practiced techniques for design validation in most places are still the veteran methods of simulation and testing. Although provably effective in the very early stages of debugging, when the design is still infested with multiple bugs, their effectiveness drops quickly as the design becomes cleaner, and they require an alarmingly increasing amount of time to uncover the more subtle bugs.

A principal disadvantage of these methods is that one can never be sure when a limit has been reached or even an estimate of how many bugs may still lurk in the design. As the complexity of designs increases tremendously, say from 0.5 to 5 million gates per chip, some far-seeing managers predict the complete collapse of these conventional methods and their total inability to scale up [53].

The approach of formal verification is a very attractive and dramatically appealing alternative to simulation and testing. While simulation and testing explore some of the possible behaviors and scenarios of the system, the question of whether the unexplored trajectories contain the fatal bug remains open. Formal verification on the other hand conducts an exhaustive exploration of all possible behaviors. Thus, when a design is pronounced correct by a formal verification method, it means that all the possible behaviors have been explored, so that the questions of proper coverage or a missed behavior become irrelevant.

One of the main approaches to formal verification is model checking. In model checking a desired behavioral property is verified over a given system (the model) through exhaustive (explicit or implicit) enumeration of all the states reachable by the system and the behaviors that lead to them. Compared to other approaches, model checking has three important advantages. First, it is fully automatic, and its application requires no user supervision or expertise in mathematical disciplines such as logic and theorem proving. Anyone who can run simulations of a design is fully eligible and capable of model-checking the same design. In the context of the currently practiced method, model checking can be categorized as the ultimately superior simulation tool. Secondly, when the design fails to satisfy the desired property, the process of model checking always generates a counterexample that shows a behavior which falsifies the property. This faulty trace offers a priceless insight into understanding the real reason for the failure as well as significant clues for solving the problem. Last but not least, model checking has a sound and mathematical underpinning.

These important advantages together with the advent of symbolic model checking, which allows the exhaustive implicit enumeration of an astronomic number of states, completely revolutionized the field of formal verification and transformed it from a purely academic discipline into a viable practical method that can potentially be integrated as an vital technique for design validation in a lot of industrial development processes [53].

1.2 Temporal Logic and Model Checking

Finite-state concurrent systems are used in various areas of computer science, especially in the design of digital circuits and communication protocols. Logical errors found late in the design phase of these systems are an extremely important problem for both circuit designers and programmers. Such errors may delay taking a new product to the market or result in the failure of critical devices that are already in use. As already mentioned, testing or simulation can easily miss significant errors when the number of possible states of the circuit or protocol is very large. Research on theorem provers, term rewriting systems, and proof checkers for verification exist, but these methods are time-consuming and often need a great deal of manual intervention.

The alternative verification method of temporal logic model checking was developed independently by Clarke and Emerson [36] in the United States and by Quielle and Sifakis [91] in France. In this technique specifications are expressed in a propositional temporal logic, and circuit designs and protocols are modeled as state-transition systems. An efficient search is then used to verify if the specification is true of the transition system. In other

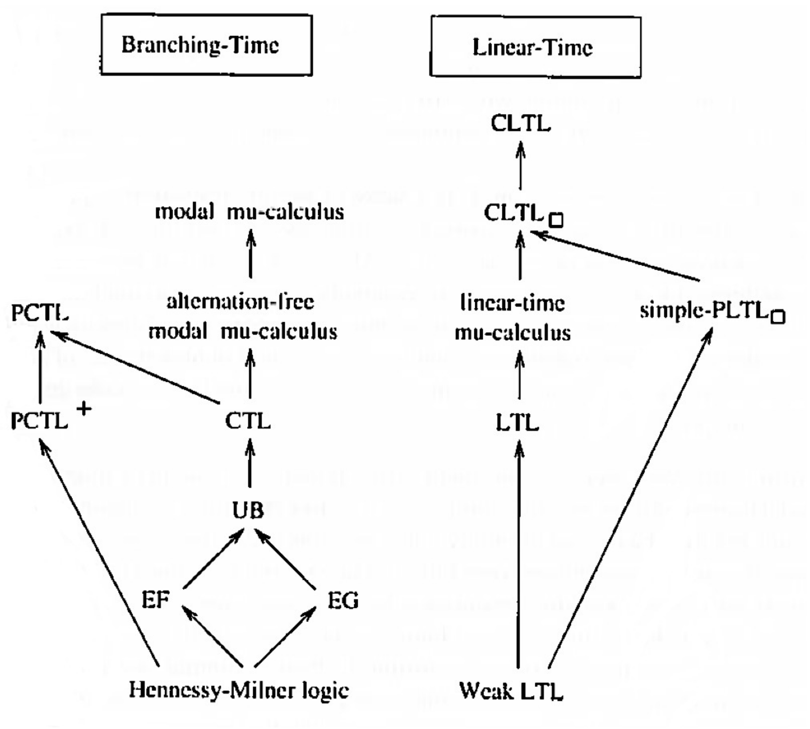


Figure 1.1: Branching and Linear time logics [16].

words, the transition system is checked to see whether it is a model of the specification.

Branching and linear-time logics are both used in model checking and can be categorized based on their expressive power. Figure 1.1 shows the hierarchies of temporal logics. In this figure an arrow $L_1 \rightarrow L_2$ shows that the logic L_2 is strictly more expressive than the logic L_1.

Due to the variation in expressive power, different logics have various degrees of importance for verification. The weakest logics, Hennessy-Milner logic and weak linear-time logic, can only express properties regarding a finite prefix of a system and play consequently only a minor role in themselves. They constitute, however, the basis for the more expressive logics and have some interesting theoretical characteristics. For finite-branching processes, such as for processes where each state accepts only finitely many transitions, it is known that two processes are bisimulation equivalent iff they satisfy the same set of Hennessy-Milner logic formula [56]. The “until” operator has then been introduced to increase the expressiveness of these basic logics (which need infinite formulas to characterize infinite behavior). On the branching-time side this lead to Computation Tree Logic (CTL) [36], while on the linear-time side this yields Linear Temporal Logic (LTL) [88] [16]. The logic utilized for specifications can directly express many of the properties that are required for reasoning about concurrent systems [53].

1.3 Objectives

The objective of this thesis is to model real time systems and demonstrate how the tools for analysis of untimed finite state systems can also be deployed in order to verify timed systems. Also the possibility of inductive conversion equivalences between two timed model systems will be evaluated. In order to reach this goal we considered different timed models such as timed petri nets, timed process algebras, graphs with durations, timed kripke structures and timed automata and we then settled on timed automata, which is a standard modeling technique in designing real-time systems; in additon, the state reachability problem is decidable for timed automata and also timed kripke structurs.

We introduce the time-abstract bisimulation method, which reduces the infinite state-space of a given timed automaton to a finite quotient graph and finite transition system. Strong time-abstract bisimulation refines the dense state space while preserving enough of both the linear and branching time properties of the original model for verification.

We consider two main issues in the verification of timed systems namely, deadlocks and timelocks. A deadlock allows time to pass without the automaton being able to perform any computation actions. Timelocks on the other hand are situations in which time cannot progress beyond a certain point. A solution for deadlock and timelock detection uses finite quotient graphs.

Another step of our approach is in the verification of real-time systems that is, given a model and a property, check whether the model satisfies the property. We focus on Timed

CTL or TCTL for short as property-specification language, which features a branching-time semantics. Model checking of TCTL formulas is also decidable. Timed CTL is a real-time variant of CTL in which clock constraints may act as atomic propositions and the a time interval is added to the until operator. Universal and existential path formulas are described with respect to time-divergent paths. It is showed that TCTL model checking can be reduced to CTL model-checking using strong time-abstracting bisimulation.

Timed kripke structures are kripke structures where each transition has a duration. We shows that by utilizing gcd-transformation and τ-transformation in timed kripke structures, the TCTL model checking of timed kripke structure in point-wise semantics can be used in order to achieve a sound and complete model checking procedure for TCTL formulas in continuous semantics.

The two conversion techniques in the untimed domain are introduced and we investigate the possibility of expanding inductive conversion methods such as these two conversion techniques in the dense time domain. We prove that developing such inductive conversions methods between timed automata and timed kripke structures is infeasible in the dense time domain for concurrent and large-scale real-time systems.

1.4 Organization of the Thesis

The thesis is organized in two parts. The first part deals with timed automata, while the second part describe timed kripke structure system as follows.

In the timed automata part, in Chapter 3 discrete-time and dense-time systems are defined. We also explain equivalence and abstraction, which are used to verify real-time models such as timed automata.

In Chapter 4, we outline the theory of timed automata and we study their closure properties. It is demonstrated that how this theory can be utilized to specify and verify real time systems. Time-abstracting bisimulation is also introduced here. Minimization of timed automata is also explained, and so It is showed that how a finite quotient graph can be generated from a given timed automaton using the partition refinement algorithm. Strongly non-zeno timed automata are defined in order to detect deadlocks and timelocks using the finite quotient graph. Finally, a method for verifying TCTL properties is proposed and it is demonstrated that TCTL model checking can be reduced to CTL model-checking using strong time-abstracting bisimulation.

In the timed Kripke structures part, Chapter 5 is devoted to defining the time domain, timed kripke structure, and zeno-free timed Kripke structure. The point-wise and continuous semantics of TCTL are also described.

Abstraction methods such as GCD-transformation and τ-transformation for TCTL$_{cb}$ formulas are introduced in Chapter 6. These methods reduce the model checking in continuous semantics to the model checking in the point-wise semantics and then the TCTL model checking procedures in point-wise semantics is described.

In Chapter 7, two methods to generate a compact Kripke structure equivalent with a labeled transition system in the untimed domain are introduced. Then we discuss the possibility of extending these methods which are inductive conversion methods to the dense time domain. We prove that inductive conversion methods are infeasible in the dense time domain by identifying several issues that make the inductive conversion methods impossible in the dense time domain.

Chapter 2

Preliminaries

2.1 Process Theory

A process defines the behavior of a system, meaning a machine, a communication protocol, an elementary particle, a network of falling dominoes, a chess player, or any other system. Process theory is concerned with the analysis of processes. Two principal characteristics of process theory are modeling and verification. Modeling is the task of describing processes using mathematical structures in a system description language. Verification is the activity of proving statements regarding processes for example, that the actual behavior of a system is equal to its intended behavior. This is only possible if a criterion has been described, specifying whether or not two processes are equal or behave similarly. Such a criterion establishes the semantics of process theory, often based on an equivalence relation between processes. Which aspects of the behavior of a system are of significance to a specific user rely on the environment in which the system will be running and on the interests of the specific user. Therefore it is not a duty of process theory to discover the 'true' semantics of processes, but rather to specify which process semantics is appropriate for which applications [16].

A process algebra is an algebraic, formal description method for processes, particularly suitable for the description of systems featuring communicating and concurrency. Many process algebras have been expanded, including ACP [15], CCS [77], and CSP [59]. All process algebras have the following main properties: They are all compositional, they are defined in terms of a Plotkin-style [87] structural operational semantics (SOS) that describes the single-step execution capabilities of systems, and behavioural reasoning is accomplished using equivalence relations ("same behaviour") or preorders ("refinement"). An algebraic process description can be "compiled" into a labeled transition system using the SOS.

A process-algebraic approach to system verification usually involves two process descriptions. One of them is the *system model* which capture the design of the actual system, and the other is the *specification* which illustrates the system's desired "high-level" behavior. One may then establish the correctness of the system model in terms of the specification by demonstrating that the system model behaves as same as the specification (if using an equivalence) or by proving that it refines the specification (if using a preorder).

2.2 Model Checking

Model checking implements an exhaustive search of the state space of the system to verify whether some specification is observed by that system. The method will always report a yes/no answer, and is fully algorithmic, with reasonable efficiency. In some circumstances systems that are not finite-state may still be verified using model checking in combination with abstraction and induction principles. Due to its algorithmic nature model checking is preferable to deductive verification, whenever it can be applied. Model checking consists of three steps: *modelling* the system under test into a suitable formalism, *specifying* the desired properties of the system, and *verifying* the system against the specification. A negative response (with the associated error trace) can result from an incorrect model, and incorrrect specification, or an actual negative result (the system not observing the specification).

Compositional Verification The principal disadvantage of model checking is the *state explosion* that can happen if the system being verified has many concurrent components, case in which the number of global system states may increase exponentially with the number of processes.

One of the important approaches pursued to attack the state explosion problem is to produce smaller and equivalent state spaces in terms of specific desired properties for instance deadlock and accessibility. In this technique, irrelevant data can be removed so that verification can be performed more efficiently.

Among methods for building equivalent state spaces, compositional verification is an noteworthy and active research topic [34] [95][98] [102] [109]. Compositional verification exploits the modular approach utilized in modem large-scale system designs. The primary target of modularity is to decrease overall cost by letting modules to be designed independently and in parallel. Moreover, modularity facilitates maintenance and re-usability of computer software. Modules that are designed independently and in parallel can also be analyzed in parallel.

Compositional verification was introduced by [58] [79] and introduces an algebraic treatment of processes with synchronous communication as the primitive tools of interaction among processes [79]. Equivalence and congruence of processes are well founded in process algebras for event-based synchronous processes. A few papers [13] [41] [33] [75] [102] have also addressed the general interest for asynchronous communication as the natural interaction of processes. Several concurrency theories in process algebras can be utilized for the analysis of asynchronous communication as well [63].

Reachability Current theories in compositional verification are extended mostly for event-based systems without the intention of preserving state information (conditions). As a result, current methods can not be utilized to efficiently analyze state-based properties. On the other hand, state reachability has been assumed as an appropriate manner of analyzing critical conditions of systems [81] such as mutual exclusion and buffer overflow. The state reachability property also offers insights into the understanding of a systems. More importantly, state reachability can facilitate the debugging and modification of an improper system design using compositional verification because of the fact that most of the event occurrences are discarded by compositional verification methods in order to prepare a small state space for analysis [63].

2.3 Temporal Logic

A successful method of verification is based on an appropriate formalism for determining central aspects of the intended system behavior. In the case of sequential programs where the input/output behavior together with termination plays a predominant role such formalisms are traditionally relying on state transformer semantics. In fact, sequential programs are usually verified by considering their partial correctness, determined with respect to pre-conditions and post-conditions [50]. Reactive systems, on the other hand, are typically non-terminating, as they maintain an ongoing interaction with the environment. Hence, verification techniques based intrinsically on the existence of a final state are usually not applicable and must be replaced by radically different approaches [65]. Pnueli [88] was the first to propose the use of temporal logic as a language for the specification of concurrent program properties. Temporal logics are tailored to expressing many important correctness properties of reactive systems.

A temporal logics can specify the ordering of events in time without introducing time explicitly. Most of them have an operator like $\mathbf{G}\ f$ that is true in the present if f is always true in the future (i.e., if f is globally true). To assert that two events e_1 and e_2 never happen at the same time, one would write $\mathbf{G}\ (\neg e_1 \vee \neg e_1)$. Temporal logics are often categorized based on whether the time is assumed to have a linear or a branching structure [103]. The "until" operator has been introduced to increase expressiveness and avoid in particular the need of infinite sets of formulas to specify infinite behavior. The operator specifies that property o should hold until a second property w holds. On the branching-time side this lead to Computation Tree Logic (CTL) [36], while on the linear-time side this yields Linear Temporal Logic (LTL) [88] [16].

The temporal-logic model checking algorithms by Clarke and Emerson [36] in the early 1980s permitted this type of reasoning to be automated. Implementation of this method is very efficient since checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models. The algorithm introduced by Clarke and Emerson is suitable for the branching-time logic CTL. This algorithm is polynomial in both the size of the model specified by the program under consideration and in the length of its specification in temporal logic. They also showed that fairness [51] could be controlled without changing the complexity of the algorithm. This was an essential improvement in that the correctness of many concurrent programs based on some type of fairness assumption; for instance, absence of starvation in a mutual exclusion algorithm may be based on the assumption that each process makes progress infinitely often.

At the same time Quielle and Sifakis [91] proposed a model checking algorithm for a subset of CTL, but they did not analyze its complexity. Later, an improved algorithm with linear running time in the product of the length of the formula and the size of the state transition graph was devised by Clarke, Emerson, and Sistla [37]. The algorithm was implemented in the EMC model checker, which was widely utilized to check a number of network protocols and sequential circuits [22] [23] [24] [37] [46] [80].

Alternative methods for verifying concurrent systems have been introduced by a number of other researchers. Many of these techniques utilize automata for specifications as well as for implementations. The implementation is checked to find whether its behavior conforms to that of the specification. For both implementation and specification, the same type of model is utilized, therefore, an implementation at one level can also be utilized as a specification for the next level of refinement [53].

2.3.1 Computation Tree Logic

Computation Tree Logic (CTL) [14] [36] [47] belongs to the family of branching-time logics and describes properties of a computation tree. The formulas are consist of path quantifiers and temporal operators. The path quantifiers specify the branching structure in the computation tree. There are two such quantifiers: **A** (for all computation paths) and **E** (for some computation path). These quantifiers are utilized in a specific state to determine that all of the paths or some of the paths starting at that state have a given property. The temporal operators state properties of a path through the tree. There are five basic such operators:

- **X "next time":** requires that a property holds in the second state of the path.
- **F "eventually" or "in the future":** asserts that a property will hold in some state on the path.
- **G "always" or "globally"** states that a property holds in every state on the path.
- **U "until":** holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.
- **R "release":** is the logical dual of the U operator. It requires that the second property holds along the path up to and including the first state where the first property holds; however, the first property is not required to hold eventually.

Each of the temporal operators **X**, **F**, **G**, **U**, and **R** must be immediately preceded by a path quantifier. This effectively defines ten CTL operators: **AX**, **EX AF**, **EF AG**, **EG AU**, **EU AR**, and **ER**.

Each of the ten operators can be expressed using only the three operators **EX**, **EG**, and **EU** as follows:

- $\mathbf{AX}\, f = \neg \mathbf{EX}(\neg f)$
- $\mathbf{EF}\, f = \mathbf{E}[True\, \mathbf{U} f]$
- $\mathbf{AG}\, f = \neg \mathbf{EF}(\neg f)$
- $\mathbf{AF}\, f = \neg \mathbf{EG}(\neg f)$

- $\mathbf{A}[f\ \mathbf{U}\ g] \equiv \neg\mathbf{E}\,[\neg g\ \mathbf{U}\ \ (\neg f \wedge \neg g)] \wedge \neg\ \mathbf{EG}\neg\ g$
- $\mathbf{A}[f\ \mathbf{R}\ g] \equiv \neg\mathbf{E}\,[\neg f\ \mathbf{U}\ \neg g]$
- $\mathbf{E}[f\ \mathbf{R}\ g] \equiv \neg\mathbf{A}\,[\neg f\ \mathbf{U}\ \neg g]$

2.3.2 Timed Computation Tree Logic

TCTL extends CTL by introducing mechanisms to specify real-time properties.

Definition 1. TIMED COMPUTATIONAL TREE LOGIC (TCTL): Let $\mathcal{I}$ denote the set of all intervals of $\mathbf{R}$ of the form $[c,c']$, $[c,c')$, $(c,c']$, (c,c'), (c,∞) and $[c,\infty)$ where $c,c' \in \mathbf{N}$. A formula in TCTL [4] is described based on the following syntax:

$$\phi ::= \mathit{true}\ |p|\ \neg\phi\,|\phi \vee \phi|\ \exists\ \phi\,\mathcal{U}_I\ \phi\ |\ \forall\ \phi\ \mathcal{U}_I\ \phi$$

where $p \in \mathit{Props}$ is an atomic proposition and $I \in \mathcal{I}$ is an interval.

Let A be a timed automaton with Q the set of discrete states, and let $P : \mathit{Props} \mapsto 2^Q$ be a function associating a set of discrete states of A to each atomic proposition. TCTL formulae are interpreted over states of A. Given a formula ϕ and a state s, the satisfaction relation $s \models_p \phi$ is defined inductively on the syntax of ϕ as in Figure 1 (where the subscript p is omitted for simplicity).

The following abbreviations are also defined:

$$\exists\ \mathbf{F}_I\phi \overset{def}{=} \exists\ \ \mathit{true}\ \mathcal{U}_I\ \ \phi \qquad \forall\ \mathbf{G}_I\phi \overset{def}{=} \neg\exists\ \ \mathbf{F}_I\ \ \neg\phi$$

$$\forall\ \mathbf{F}_I\phi \overset{def}{=} \forall\ \ \mathit{true}\ \mathcal{U}_I\ \ \phi \qquad \exists\ \mathbf{G}_I\phi \overset{def}{=} \neg\forall\ \ \mathbf{F}_I\ \ \neg\phi$$

The notation for intervals is often simplified. For example $\exists\mathbf{F}_{\leq 5}\ \phi$ is often used instead of $\exists\mathbf{F}_{[0,5]}\ \phi$, and $\forall\mathbf{G}\phi$ instead of $\forall\mathbf{G}_{[0,\infty)}\phi$.

It is said that the TA A satisfies a formula ϕ if the initial state of A satisfies ϕ.

CTL [36] can be described as the untimed fragment of TCTL, containing all formula with trivial subscript interval $[0,\infty)$.

$$
\begin{array}{lll}
s \models \mathit{true} & & \\
s \models p & \text{iff} & \mathsf{discrete}(s) \in P(p) \\
s \models \neg\phi_1 & \text{iff} & \text{not } s \models \phi_1 \\
s \models \phi_1 \vee \phi_2 & \text{iff} & s \models \phi_1 \text{ or } s \models \phi_2 \\
s \models \exists\, \phi_1\, \mathcal{U}_I\, \phi_2 & \text{iff} & \exists \rho = s \overset{\delta_1}{\rightarrow}\overset{e_1}{\rightarrow} \cdots \text{ s.t. } \mathsf{time}(\rho) = \infty \text{ and} \\
 & & \quad \exists i \cdot \Sigma_{j \le i}\delta_j \in I \text{ and } \rho(i) + \delta_i \models \phi_2 \text{ and} \\
 & & \quad\quad \forall j \le i \cdot \forall \delta \le \delta_j \cdot \rho(j) + \delta \models \phi_1 \vee \phi_2 \\
s \models \forall\, \phi_1\, \mathcal{U}_I\, \phi_2 & \text{iff} & \forall \rho = s \overset{\delta_1}{\rightarrow}\overset{e_1}{\rightarrow} \cdots \text{ s.t. } \mathsf{time}(\rho) = \infty \cdot \\
 & & \quad \exists i \cdot \Sigma_{j \le i}\delta_j \in I \text{ and } \rho(i) + \delta_i \models \phi_2 \text{ and} \\
 & & \quad\quad \forall j \le i \cdot \forall \delta \le \delta_j \cdot \rho(j) + \delta \models \phi_1 \vee \phi_2
\end{array}
$$

Figure 2.1: TCTL satisfaction.

2.3.3 Transition Systems

Transition systems are often used as models explaining system behavior. They are directed graphs where nodes introduce states and edges model transitions i.e., state changes. We use a general form of transition systems were transitions are labeled with action names and atomic propositions are associated with states. Action names represent communication and we often use letters at the beginning of the Greek alphabet (α, β and so on) to denote actions. We use atomic propositions to specify temporal properties. Intuitively atomic propositions describe simple known facts about the states of the system under consideration. We often use arabic letters from the beginning of the alphabet (a, b, c, etc.) to refer to atomic propositions.

Definition 2. Transition System (TS):: A transition system TS is a tuple $(S, Act, \rightarrow, I, AP, L)$ where S is a set of states, Act is a set of actions, $\rightarrow \subseteq S \times Act \times S$ is a transition relation, $I \subseteq S$ is a set of initial states, AP is a set of atomic propositions, and $L : S \rightarrow 2^{AP}$ is a labeling function.

TS is called finite if S, Act, and AP are finite. The transition system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\rightarrow$. That is, if s is the current state, then a transition $s \xrightarrow{\alpha} s'$ originating from s is selected non-deterministically and performed, meaning that the action α is performed and the transition system evolves from state s into the state s'. This process is repeated in s' and finishes once no outgoing transitions are available.

The labeling function L associates a set $L(s) \in 2^{AP}$ of atomic propositions to any state s. Intuitively $L(s)$ intuitively contains exactly all the atomic propositions $a \in AP$ that are true in state s . Given a propositional logic formula Φ, s satisfies the formula Φ whenever the evaluation induced by $L(s)$ makes the formula Φ true; that is, $s \models \Phi$ iff $L(s) \models \Phi$.

2.4 Finite Automata

A finite automaton is a mathematical model of a device that has access to a constant amount of memory, independent of the size of its input. We will consider finite automata over finite words and finite automata over infinite words (also called ω-automata).

2.4.1 Automata on Finite Words

Definition 3. Nondeterministic Finite Automaton (NFA): A nondeterministic finite automaton (NFA) A is a tuple $A = (Q,\ \Sigma,\ \delta,\ Q_0,\ F)$ where, Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \rightarrow 2^Q$ is a transition function, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting (or final) states.

The size $|A|$ of A is the number of states and transitions in A, i.e.,

$$|A| = |Q| + \sum_{q \in Q} \sum_{A \in \Sigma} |\delta\,(q, A)| \tag{2.1}$$

Σ defines the symbols on which the automaton is defined. The (possibly empty) set Q_0 defines the states in which the automaton may start. The transition function δ can be identified with the relation $\rightarrow \subseteq Q \times \Sigma \times Q$ given by

$$q \xrightarrow{A} q' \quad \text{iff} \quad q' \in \delta\,(q, A) \tag{2.2}$$

Thus, often the notion of transition relation (rather than transition function) is used for δ. Intuitively, $q \xrightarrow{A} q'$ denotes that the automaton can move from state q to state q' when reading the input symbol A.

Definition 4. Runs, Accepted Language of an NFA: Let $A = (Q,\ \Sigma,\ \delta,\ Q_0,\ F)$ be an NFA and $w = A_1...A_n \in \Sigma^*$ a finite word. A run for w in A is a finite sequence of states $q_0 q_1 ... q_n$ such that

$$q_0 \in Q_0 \qquad \text{and} \qquad q_i \xrightarrow{A_{i+1}} q_{i+1} \text{ for all } 0 \leq i < n.$$

Run $q_0 q_1 ... q_n$ is called accepting if $q_n \in F$. A finite word $w \in \Sigma^*$ is called accepted by A if there exists an accepting run for w. The accepted language of A, denoted $\mathcal{L}\left(A\right)$ is the set of finite words in Σ^* accepted by A, i.e.,

$$\mathcal{L}\left(A\right) = \{w \in \Sigma^* | \text{ there exists an accepting run for } w \text{ in } A\}$$

An equivalent alternative characterization of the accepted language of an NFA A is as follows. Let A be an NFA as above. We extend the transition function δ to the function $\delta^* : Q \times \Sigma^* \rightarrow 2^Q$ as follows: $\delta^*\left(q, \varepsilon\right) = \{q\}$ and $\delta^*\left(q, A\right) = \delta\left(q, A\right)$ and

$$\delta^*\left(q, A_1 A_2 ... A_n\right) = \bigcup_{p \in \delta(q, A_1)} \delta^*\left(p, A_2 ... A_n\right)$$

$\delta^*\left(q, w\right)$ is the set of states that are reachable from q for the input word w.

In particular, $\bigcup_{q_0 \in Q_0} \delta^*\left(q_0, w\right)$ is the set of all states where a run for w in A can end. If one of these states is final, then w has an accepting run. Vice versa, if $w \notin \mathcal{L}\left(A\right)$, then none of these states is final. Hence, we have the following alternative characterization of the accepted language of an NFA by means of the extended transition function δ^*:

Definition 5. Alternative Characterization of the Accepted Language: Let A be an NFA. Then:

$$\mathcal{L}\left(A\right) = \{w \in \Sigma^* | \delta^*\left(q_0, w\right) \cap F \neq \varnothing \text{ for some } q_0 \in Q_0\}$$

It can be shown that exactly all the language accepted by nondeterministic finite automata are regular languages [12].

Emptiness A fundamental issue in automata theory is to decide for a given NFA A whether its accepted language is empty, i.e., whether $\mathcal{L}(A) = \varnothing$. This is known as the emptiness problem. From the acceptance condition, it follows directly that $\mathcal{L}(A)$ is nonempty if and only if there is at least one run that ends in some final state. Thus, nonemptiness of $\mathcal{L}(A)$ is equivalent to the existence of an accept state $q \in F$ which is reachable from an initial state $q_0 \in Q_0$. This can easily be determined in time $\mathcal{O}(|A|)$ using a depth-first search traversal that encounters all states that are reachable from the initial states and checks whether one of them is final. For state $q \in Q$, let $\text{Reach}(q) = \bigcup_{w\in\Sigma^*} \delta^*(q, w)$ that is, $\text{Reach}(q)$ is the set of states q' that are reachable via an arbitrary run starting in state q.

Proposition 1. Language Emptiness is Equivalent to Reachability: *Let $A = (Q,\ \Sigma,\ \delta,\ Q_0,\ F)$ be an NFA. Then, $\mathcal{L}(A) \neq \varnothing$ if and only if there exists $q_0 \in Q_0$ and $q \in F$ such that $q \in$ Reach(q_0) [12]*

Definition 6. Deterministic Finite Automaton (DFA): Let $A = (Q,\ \Sigma,\ \delta,\ Q_0,\ F)$ be an NFA. A is called deterministic if $|Q_0| \leq 1$ and $|\delta(q, A)| \leq 1$ for all states $q \in Q$ and all symbols $A \in \Sigma$. We will use the abbreviation DFA for a deterministic finite automaton.

DFA A is called total if $|Q_0| = 1$ and $|\delta(q, A)| = 1$ for all $q \in Q$ and all $A \in \Sigma$.

An NFA is deterministic if it has at most a single initial state and if for each symbol A the successor state of each state q is either uniquely defined if $|\delta(q, A)| = 1$ or undefined if $|\delta(q, A)| = \varnothing$. Total DFAs provide unique successor states, and thus, unique runs for each input word. Any DFA can be turned into an equivalent total DFA by simply adding a non-final trap state, q_{trap} say, that is equipped with a self-loop for any symbol $A \in \Sigma$. From any state $q \neq q_{trap}$, there is a transition to q_{trap} for any symbol A for which q has no A-successor in the given non-total DFA.

Total DFA are often written in the form $A = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$ where q_0 stands for the unique initial state and δ is a (total) transition function $\delta : Q \times \Sigma \to Q$. Also, the extended transition function δ^* of a total DFA can be viewed as a total function $\delta^* : Q \times \Sigma^* \to Q$

which for given state q and finite word w returns the unique state $p = \delta^* (q, w)$ that is reached from state q for the input word w. In particular, the accepted language of a total DFA $A = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$ is given by

$$\mathcal{L}(A) = \{w \in \Sigma^* |\ \delta^* (q_0, w) \in F\}$$

The observation that total DFAs have exactly one run for each input word allows complementing a total DFA A by simply declaring all states to be final that are nonfinal in A and vice versa[12].

2.4.2 Automata on Infinite Words

Finite-state automata accept finite words, i.e., sequences of symbols of finite length, and yield the basis for checking regular safety properties. These ideas are generalized to formalize the requirements for realistic systems. Infinite words over the alphabet Σ are infinite sequences $A_0A_1A_2\ldots$ of symbols $A_i \in \Sigma$. Σ^ω denotes the set of all infinite words over Σ. Any subset of Σ^ω is called a language of infinite words, sometimes also called an ω-language [12].

Because most concurrent systems are designed not to halt during normal execution, we model computations as infinite sequences of states. These automata have the same structure as finite automata over finite words. However, they recognize words from Σ^ω, where the superscript ω indicates an infinite number of repetitions. The simplest automata over infinite words are Büchi automata [28]. A Büchi automaton has the same components as an automaton over finite words. However, F is called the set of accepting states, rather than final states [53].

Part I

Timed Automata

Chapter 3

Verification of Real-Time Systems

Ensuring the correctness of the computer system is a challenging task in life-critical systems especially when the correctness of an action depends not only on the action itself, but also on the actual time when the action is executed (such as in a pacemaker system). Computers are widely utilized in critical applications where predictable response times are vital for correctness. Such systems are called real-time systems; examples include aircraft controllers and industrial machinery. Because of the nature of such applications, errors in real-time systems can be very perilous or even fatal. Guaranteeing the correctness of a complex real-time system is therefore an important (and also nontrivial) task.

Two time semantics are defined for the description of real-time systems. Discrete time [10] specifies that all the time readings are integers and all clocks increment their readings at the same time. The other choice is dense time [7], which means that time readings can be rational numbers or real numbers and all clocks increment their readings at a uniform rate.

3.1 Discrete-Time Systems

When time is discrete, possible clock values are positive integers, and events can only happen at integer time values. This type of model is sufficient for synchronous systems, where all of the components are synchronized to a single global clock. The duration between successive clock ticks is selected as the basic unit for measuring time. This model has been used for many years for reasoning regarding the correctness of synchronous hardware designs.

In discrete-time models, there is a single global clock. Therefore, there is a simple and explicit way [66] to support it: introduce a clock variable "now" whose value shows the current time, and model the passage of time with a "tick" action that increment the variable now. Timing bounds on actions are then determined with one of the following three types of timer variables: a countdown timer which is decreased by the tick action, a count-up timer which is decremented by a tick, and an expiration timer which remains unchanged by any tick. A countdown or count-up timer expires when its value overtakes some specified value; an expiration timer expires when its value minus now overtakes some value. An upper-bound timing constraint on when an action A must happen is represented by an enabling condition on the tick action that violating the condition by a time increment is forbidden. A lower-bound constraint on when A may happen is represented by an enabling condition on A that forbids it from being executed earlier than it should be.

Alternative methods consist of quantitative temporal analyses for discrete-time systems [48] [31]. These approaches expand CTL with a bounded until operator [48] in order to support the specification of timing constraints between two actions. For instance, it is always true that p may be followed by q within 3 time units. CTL model checking verification algorithms are then extended. Symbolic model can be combined with this solution to increase scalability [31].

In discrete time modeling one needs to select some fixed fixed time quantum such that any delay between any two events will be a multiple of this quantum. This in turn restricts the accuracy of the modeling process [53]. The selection of an appropriately small time quantum may also blow up the state space in such a way that verification can no longer be done.

3.2 Dense-Time Systems

Dense time is the natural model for asynchronous systems because the separation of events can be arbitrarily small. This ability helps specifying causally independent events in an asynchronous system. Moreover, no assumptions is required regarding the speed of the events offered by the environment.

Various models of dense-time (also called continuous time) have been introduced [3] [45] [57] [68] [93] [97] [112] [113]. Examples include timed automata [8] [74], Timed Process Algebra [110] [92] [96] , Timed Interval Calculus [49], Timed Statecharts [61], and so on.

The timed automata model of Alur, Courcoubetis, and Dill [27] [45] has become the standard modeling technique in designing real-time models. Timed Automata are finite state machines provided with clock variables (ranging over rational numbers). The passing of time is implemented by updates to clock variables. The execution of the model can be restricted by guard expressions involving clocks. This is actually a general procedure to annotate state transition graphs with timing constraints using finitely many rational-valued clock variables. To achieve a finite representation for the infinite state space thus generated abstraction methods such as clock regions [3], clock zones [45] [111], and bisimulation equivalence can be used. Many efficient verification tools for timed automata have been developed based on these methods. Examples include UPPAAL [69], KRONOS [21], RED [106], Timed COSPAN [99], and Rabbit [17].

3.3 Equivalence and Abstraction

There are two approaches to system verification. The first is equivalence checking, which aims to establish some semantic equivalence between two systems, one of which corresponding to the implementation of the specification given by the other. The second approach is and model checking, which aims to specify whether a given system satisfies some property which is usually given in some modal or temporal logic [16]. In order to prevent the state explosion problem, developing techniques such as abstraction and bisimulation equivalence aim to replace a large structure by a smaller structure which satisfies the same properties. The abstraction is realized by giving a mapping between the actual data values in the system and a small set of abstract data values. The goal of bisimulation equivalence is to identify transition systems with the same branching structure, and which thus can simulate each other in a stepwise manner.

3.3.1 Linear Time and Branching Time Concurrency Semantics

As many as 12 semantics can be defined on uniform concurrency, as shown in Figure 3.1. The coarsest one is trace semantics [58], where the only feature considered is the (partial) trace. The finest semantics is bisimulation [76]. Bisimulation semantics is the standard semantics for the process algebra CCS. Bisimulation equivalence is a refinement of observational equivalence, as shown by Hennessy and Milner [55]. On the domain of finitely branching, concrete, sequential processes, both equivalences coincide. The semantics of De Bakker and Zucker, introduced in [40], also coincides with bisimulation semantics on this domain.

Ten semantics are in between. First, different types of trace semantics can be achieved by utilizing complete traces besides partial ones. Failures semantics is proposed in Brookes, Hoare and Roscoe [59], and utilized in the construction of a model for the process algebra CSP [58]. It is finer than complete trace semantics. Testing equivalence as introduced in De Nicola and Hennessy [43] coincides with failures semantics on the domain of finitely branching, concrete, sequential processes, as do the semantics of Kennaway [64] and Darondeau [39]; this has been established in De Nicola [42]. In Olderog and Hoare [83] readiness semantics is introduced, which is slightly finer than failures semantics. Between readiness and bisimulation semantics one identifies ready trace semantics, as proposed independently in Pnueli [89] (there called barbed semantics), Baeten, Bergstra and Klop [11] and Pomello [90] (under the name exhibited behaviour semantics).

The natural completion of the "square" suggested by failures, readiness and ready trace

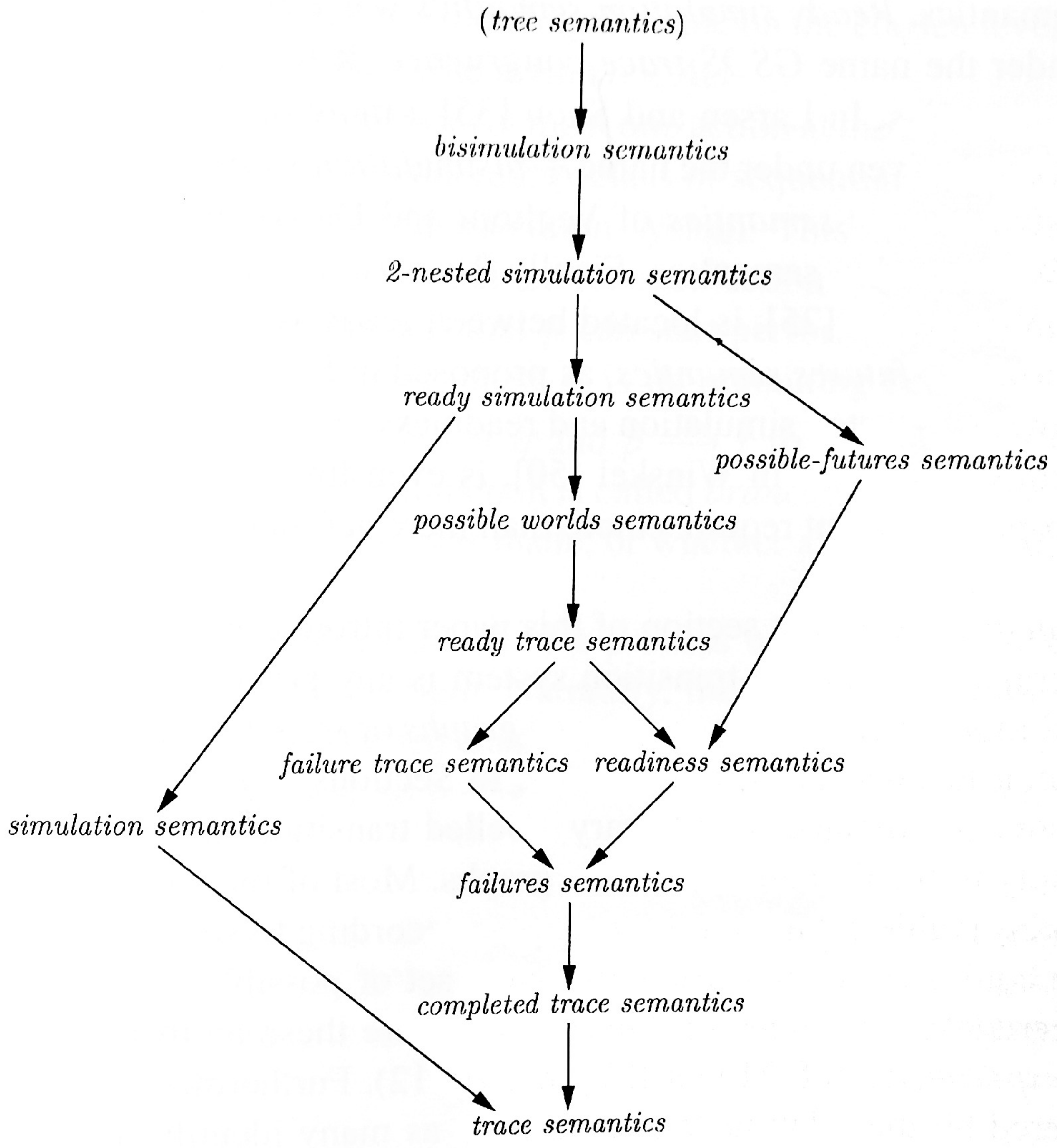


Figure 3.1: The linear time–branching time spectrum [16].

semantics yields the failure trace semantics. For finitely branching processes, this is the same as refusal semantics, proposed in Phillips [86]. Simulation semantics, based on the classical notion of simulation by Park [85], is independent of the last five semantics. Ready simulation semantics was proposed in Bloom, Istrail and Meyer [18] under the name GSOS trace congruence. It is finer than ready trace as well as simulation semantics. In Larsen and Skou [70] a more operational characterization of this equivalence was given under the name $\frac{2}{3}$ -bisimulation equivalence. The notion of possible worlds semantics of Veglioni and De Nicola [104] fits between ready trace and ready simulation semantics. FInally, 2-nested simulation semantics, proposed in Groote and Vaandrager [52], is set between ready simulation and bisimulation semantics, and possible-futures semantics, as introduced in Rounds and Brookes [94], can be located between 2-nested simulation and readiness semantics, Tree semantics, employed in Winskel [108], is even finer than bisimulation semantics. However, this semantics needs more than mere action relations [16].

3.3.2 Abstraction

Verification methods based on abstraction seems to be essential for reasoning about circuits that contain data paths or concurrent programs that include complex data structures. Finite-state verification approaches have been utilized mainly for control-oriented systems. The symbolic methods handle some systems that involve nontrivial data manipulation, but it has high complexity of verification. Data abstraction is based on the observation that the specifications of systems that contain data paths involve fairly simple relationships among the data values in the system. For example, verifying the addition operation in a microprocessor only involves three registers (the value in one register needs to finally be equal to the sum of the values in two other registers). In such situations, abstraction can be utilized to decrease the complexity of model checking. Abstraction is accomplished by giving a mapping between the actual data values in the system and a small set of abstract data values. By developing the mapping to states and transitions, it is possible to generate an abstract version of the system under consideration. The abstract system is much smaller than the actual system, and properties at the abstract level are easier to verify. Clarke, Grumberg, and Long demonstrate that in such a mapping any properties expressible in the logic ACTL that are satisfied by the abstract system will also be true of the actual system [38] [73].

Abstraction is a very important technique for decreasing the state explosion problem. Two main abstraction techniques exist namely, the cone of influence reduction and data abstraction. Both of these methods are executed on a high level description of the system, before the model for the system is built. Thus, we prevent the generation of the unreduced model that might be too big to fit into memory. The cone of influence reduction tries to reduce the size of the state transition graph by focusing on the variables of the system that are referred to in the specification. The abstraction is achieved by eliminating variables that do not affect the variables in the specification. The properties of interest are thus maintained, but the size of the model that needs to be verified is smaller. Data abstraction on the other hand involves discovering a mapping between the actual data values in the system and a small set of abstract data values. By developing this mapping to states and transitions, it is possible to achieve an abstract system that simulates the original system and is much smaller [53].

3.3.3 Bisimulation Equivalence

The only relations that maintain branching-time behavior are bisimulation equivalence and simulation preorder. Bisimulation relates states that mutually mimic all individual transitions, while simulation requires that one state can mimic all stepwise behavior of the other, but not the reverse. Bisimulation equivalence is a technique that replaces a large structure with a smaller structure which satisfies the same properties. More generally, given a logic L and a structure M, the goal is to find a smaller structure M' that satisfies exactly the same set of formulas of the logic L as M [53].

Bisimulation identifies transition systems with the same branching structure, and which thus can simulate each other in a stepwise manner. A transition system TS' can simulate transition system TS if every step of TS can be matched by one (or more) steps in TS'. Bisimulation is first introduced as a binary relation between transition systems over the same set of atomic propositions; later on, bisimulation is also treated as a relation between states of a single transition system. Bisimulation is proposed as the largest relation satisfying certain properties [12].

Definition 7. Direct Successors: Let $TS = (S,\ Act,\ \to,\ I,\ AP,\ L)$, be a transition system. For $s \in S$ and $\alpha \in \text{Act}$, the set of direct α-successors of s is defined as:

$$Post\,(s,\alpha) = \left\{ s' \in S \mid s \overset{\alpha}{\to} s' \right\}, \quad Post\,(s) = \textstyle\bigcup_{\alpha \in \text{Act}} Post\,(s,\alpha)\,.$$

Definition 8. Bisimulation Equivalence: Let $TS_i = (S_i,\ Act_i,\ \to_i,\ I_i,\ AP,\ L_i)\,,\ i = 1,2,$ be transition systems over AP. A bisimulation for (TS_1, TS_2) is a binary relation $R \subseteq S_1 \times S_2$ such that:

1. $\forall s_1 \in I_1\,(\exists\, s_2 \in I_2.\,(s_1,s_2) \in R)$ *and* $\forall s_2 \in I_2\,(\exists\, s_1 \in I_1.\,(s_1,s_2) \in R)\,.$
2. For all $(s_1,s_2) \in R$ it holds that:
   (a) $L_1\,(s_1) = L_2\,(s_2)$
   (b) If $s_1' \in Post(s_1)$ then there exists $s_2' \in Post(s_2)$ with $(s_1', s_2') \in R$.
   (c) If $s_2' \in Post(s_2)$ then there exists $s_1' \in Post(s_1)$ with $(s_1', s_2') \in R$.

TS_1 and TS_2 are bisimulation-equivalent (bisimilar, for short), denoted $TS_1 \sim TS_2$, if there exists a bisimulation R for (TS_1, TS_2). $\sim$ is an equivalence relation (reflexive, transitive, and symmetric).

Condition (1) declares that every initial state of TS_1 corresponds to an initial state of TS_2 and vice versa. According to condition (2a), the states s_1 and s_2 are equally labeled. This ensures the local equivalence of s_1 and s_2. Condition (2b) states that every outgoing transition of s_1 must be matched by an outgoing transition of s_2; the reverse is stated in (2c).

Bisimulation is defined with respect to the direct successors of states. A relation between (finite or infinite) paths can be achieved using an inductive argument.

Definition 9. Bisimulation on Paths: Let TS_1 and TS_2 be transition systems over AP, R a bisimulation for (TS_1, TS_2), and $(s_1, s_2) \in R$. Then for each (finite or infinite) path $\pi_1 = s_{0,1}s_{1,1}s_{2,1}... \in Paths(s_1)$ there exists a path $\pi_2 = s_{0,2}s_{1,2}s_{2,2}... \in Paths(s_2)$ of the same length such that $(s_{j,1}, s_{j,2}) \in R$ for all j.

By symmetry, for each path $\pi_2 \in Paths(s_2)$ there exists a path $\pi_1 \in Paths(s_1)$ of the same length which is statewise related to π_2. As one can construct statewise bisimilar paths, bisimilar transition systems are trace-equivalent. Most of the time it is simpler to demonstrate that two transition systems are bisimilar rather than prove their trace equivalence. The intuitive reason for this discrepancy is that proving bisimulation equivalence just needs local reasoning about state behavior instead of considering entire paths.

Proposition 2. Bisimulation and Trace Equivalence: *$TS_1 \sim TS_2$ implies $Traces(TS_1) = Traces(TS_2)$.*

Bisimulation can also be defined as a relation among states within a single transition system. By considering the quotient transition system under such a relation, smaller models are achieved. This minimization recipe can be used for efficient model checking.

Definition 10. Bisimulation Equivalence as Relation on States: Let $TS = (S,\ Act,\ \rightarrow,\ I,\ AP,\ L)$ be transition systems. A bisimulation for TS is a binary relation R on S such that for all $(s_1, s_2) \in R$:

1. $L(s_1) = L(s_2)$
2. If $s_1' \in Post(s_1)$ then there exists an $s_2' \in Post(s_2)$ with $(s_1', s_2') \in R$.
3. If $s_2' \in Post(s_2)$ then there exists $s_1' \in Post(s_1)$ with $(s_1', s_2') \in R$.

States s_1 and s_2 are bisimulation-equivalent (or bisimilar), denoted $s_1 \sim_{TS} s_2$, if there exists a bisimulation R for TS with $(s_1, s_2) \in R$ [12].

3.3.4 Decidability on Bisimulation-Based and Failure-Based Equivalences

In compositional verification methods the execution time depends on state-space complexity and time complexity of the reduction algorithms. The state-space complexity of algorithms in particular dramatically influences the execution time because the size of a composite state space is specified by the size (complexity) of condensed state spaces utilized in the composition. Bisimulation-based [79] and failure-based [58] equivalences are famous in contemporary process algebras for the compositional verification of concurrent systems.

Definition 11. Timed trace equivalence: Two states q and q' of a timed automaton A are time trace equivalent, written $q \equiv_{tt} q'$, iff , q and q' are trace equivalent with respect to the timed transition system $S_t(A)$. That is, two states are timed trace equivalent iff they generate the same timed words i.e., sequences of input symbols and time increments.

As expected, timed trace equivalence is strictly weaker than timed bisimilarity but incomparable to region equivalence and incomparable to untimed bisimilarity. While timed trace equivalence is a congruence, it is computationally intractable. The undecidability proof for $\equiv_{tt}$ follows the proof that the language inclusion problem for timed automata over infinite words is undecidable [8].

Proposition 3. Decidability on Timed trace equivalence: *The problem of deciding if two initial states of a timed automaton are timed trace equivalent is undecidable [62].*

Bisimulation [78] is a widely used relation to assert equivalence of two systems. The technique has been developed to timed systems as well. Timed bisimulation was first proved to be decidable for timed automata by Cerans utilizing a product construction method on region graphs. It is also known that timed language equivalence is undecidable for timed automata [8].

Bisimulation equivalences are significantly important for timed automata because they have been shown to be decidable [32] [1] [71] [107]. Both timed and time abstracted bisimulations are decidable for timed automata. Timed bisimulation was first proved to be decidable in [32] where a product construction method on region graph was utilized. In [107] the product construction has been used on zones which makes the procedure for deciding timed bisimulation invariant to scaling. Deciding timed bisimulation by applying open maps to then use zone valuation graph is also possible [60]. The decidability of time abstracted bisimulation is inherent in the construction of the zone valuation graph. However a very similar approach for deciding time abstracted bisimulation has earlier been utilized [101] [54]. The methods proposed in [32] [107] use complex product constructions on region graphs and zone graphs respectively.

Chapter 4

Verification of Timed Automata Using Time-Abstracting Bisimulation

We now consider the verification of dense-time systems modeled as timed automata. In this approach, given a timed system A, we construct a finite graph G which captures the behavior of A whereas the exact time delays are abstracted away. Then, an untimed verification methods on G is utilized in order to prove properties on A. The branching-time logic TCTL as property-specification languages is considered. Model checking A against properties recognized as TCTL formulae comes down to using, CTL model-checking algorithms on G. The abstraction of exact delays is formalized under the concept of time-abstracting bisimulations. Three time-abstracting bisimulations can be defined which have different reduction power. The stronger of them keep both linear and branching-time properties, while two weaker ones keep only linear-time properties. The finite graph G is the quotient of A with respect to a time-abstracting bisimulation. Producing G can be performed by adapting a partition-refinement algorithm to the timed case. The adapted algorithm is symbolic that is, equivalence classes are shown as simple polyhedra. When these polyhedra are not convex, operations become expensive; therefore a different partition-refinement technique that preseves convexity is also needed.

4.1 Timed Automata

Timed automata [4] [57] are hereby to model timed systems. These are finite-state automata equipped with clocks used to specify constraints on the amount of time that can elapse between two events.

Definition 12. CLOCKS AND VALUATIONS: A clock is a variable ranging over $\mathbf{R}$, the set of non-negative reals. Given a set of clocks $\mathcal{X} = \{x_1, ..., x_n\}$, an $\mathcal{X}$-valuation is a function $\mathbf{v} : \mathcal{X} \mapsto \mathbf{R}$. The set of all valuations is $\mathbf{R}^{\mathcal{X}}$. $\mathbf{0}$ is the valuation that assigns zero to all clocks. For $X \subseteq \mathcal{X}, \mathbf{v}\left[X := 0\right]$ (reset X to zero) is the valuation $\mathbf{v_1}$ such that $\forall x \in X.\mathbf{v_1}\left(x\right) = 0$ and $\forall x \notin X.\mathbf{v_1}\left(x\right) = \mathbf{v}\left(x\right)$. For $\delta \in \mathbf{R}, \mathbf{v} + \delta$ (time elapse by δ) is the valuation $\mathbf{v_2}$ such that $\forall x \in \mathcal{X}.\mathbf{v_2}\left(x\right) = \mathbf{v}\left(x\right) + \delta$.

Definition 13. HYPERPLANES AND POLYHEDRA : An atomic constraint on $\mathcal{X}$ is an expression of the form $x \sim c$ or $x - y \sim c$, where $x, y \in \mathcal{X}, \sim \in \{<, \leq, \geq, >\}$ and $c \in \mathbf{N}$ is a natural constant. An $\mathcal{X}$- valuation $\mathbf{v}$ satisfies the constraint $x \sim c$ if $\mathbf{v}\left(x\right) \sim c$; $\mathbf{v}$ satisfies $x - y \sim c$ if $\mathbf{v}\left(x\right) - \mathbf{v}\left(y\right) \sim c$. An $\mathcal{X}$- hyperplane is a set of valuations satisfying an atomic clock constraint. The class $\mathcal{H}_{\mathcal{X}}$ of $\mathcal{X}$-polyhedra is described as the smallest subset of $2^{\mathbf{R}^{\mathcal{X}}}$, which contains all $\mathcal{X}$-hyperplanes and is closed under set union, intersection and complementation. The following notation is utilized for polyhedra: we write $x < 5$ for the hyperplane defined by the constraint $x < 5$, $x < 5 \wedge y = 2$ for the polyhedron defined as the intersection of $x < 5$ and $y = 2$, and so on. We also write *true* for $\mathbf{R}^{\mathcal{X}}$, *false* for $\varnothing$, and *zero* for $\{0\}$.

A polyhedron ζ is convex if it is the intersection of a number of hyperplanes. If ζ is non-convex then it can be written as $\zeta_1 \cup ... \cup \zeta_k$, where $\zeta_1, ..., \zeta_k$ are all convex.

Definition 14. TIMED AUTOMATA: A timed automaton (TA) [4] [57] is a tuple $A = (\mathcal{X},\ Q,\ q_0,\ E,\ invar)$, where $\mathcal{X}$ is a finite set of clocks, Q is a finite set of discrete states, $q_0 \in Q$ being the initial discrete state. E is a finite set of edges of the form $e = (q,\ \zeta,\ a,\ X,\ q')$, with $q, q' \in Q$ the *source* and *target* states. Edges are labeled with a discrete actions or a time delay action a in order to model the discrete state transitions and time transitions between source and target states in TA. σ is used for the passage of time transitions and e is utilized for the discrete state transitions. ζ is a conjunction of atomic constraints on $\mathcal{X}$ defining a convex $\mathcal{X}$-polyhedron, called the *guard* of e. $X \subseteq \mathcal{X}$ is a set of clocks to be reset upon crossing the edge. *invar* is a function associating with each discrete state q a convex $\mathcal{X}$-polyhedron called the invariant of q.

Given an edge $e = (q,\ \zeta,\ a,\ X,\ q')$, we write *source*$(e)$, *target*$(e)$, *guard*$(e)$, *label*$(e)$ and *reset*(e) for q, q', ζ, a and X, respectively. Given a discrete state q, we write *in*(q) (resp. *out*(q)) for the set of edges of the form $(-,-,-,-,q)$ (resp. $(q,-,-,-,-)$). We assume that for each $e \in \mathit{out}(q)$, $\mathit{guard}(e) \subseteq \mathit{invar}(q)$.

For a given TA A, we define $c_{max}(A)$ to be the greatest natural constant appearing in an atomic constraint of a guard or an invariant of A.

Definition 15. States: A state of A is a pair $(q, \mathbf{v})$, where $q \in Q$ is a discrete state, and $\mathbf{v} \in \mathit{invar}(q)$ is a valuation satisfying the invariant of q. We write *discrete*(s) to denote q, the discrete part of s. The initial state of A is $s_0 = (q_0, 0)$.

Definition 16. Transitions: Consider a state $(q, \mathbf{v})$. Given an edge $e = (q,\ \zeta,\ a,\ X,\ q')$ such that $\mathbf{v} \in \zeta$ and $\mathbf{v}' = \mathbf{v}\,[\mathit{reset}(e) := 0] \in \mathit{invar}(q')\,, (q, \mathbf{v}) \overset{e}{\rightarrow} (q', \mathbf{v}')$ is a discrete transition of A. $(q', \mathbf{v}')$ is called the e-successor of $(q, \mathbf{v})$.

A time transition from $(q, \mathbf{v})$ has the form $(q, \mathbf{v}) \overset{\delta}{\rightarrow} (q, \mathbf{v} + \delta)$, where $\delta \in \mathbf{R}$ and $\mathbf{v} + \delta \in \mathit{invar}(q)$. For a state $s = (q, \mathbf{v})$, we simply write $s + \delta$ instead of $(q, \mathbf{v} + \delta)$. $s + \delta$ is the δ - successor of s. The concatenation of two time transitions $s \overset{\delta}{\rightarrow} s + \delta$ and $s + \delta \overset{\delta'}{\rightarrow} s + \delta + \delta'$ is a time transition $s \overset{\delta+\delta'}{\rightarrow} s + \delta + \delta'$. Inversely, due to the dense nature of the reals, a time transition $s \overset{\delta}{\rightarrow} s + \delta$ can be split into any number m of consecutive time transitions $s \overset{\delta_1}{\rightarrow} s + \delta_1 \overset{\delta_2}{\rightarrow} \ldots \overset{\delta_m}{\rightarrow} s + \delta$, such that $\delta_1 + \delta_2 + \ldots + \delta_m = \delta$. We write $s \overset{\delta}{\rightarrow}\overset{e}{\rightarrow} s'$ if $s \overset{\delta}{\rightarrow} s + \delta$ is a time transition and $s + \delta \overset{e}{\rightarrow} s'$ is a discrete transition.

Two types of semantics are associated with a TA: a branching-time semantics with respect to a labeled graph and a linear-time semantics in terms of executions.

Definition 17. Semantic graph: The semantic graph of A, indicated by G_A, is defined to be the graph which has the states of A as nodes and two kinds of edges, related to the discrete and time transitions of A. G_A has an uncountable set of nodes and uncountable branching.

Definition 18. Runs: A run of A starting from state s is a finite or infinite sequence $\rho = s_1 \overset{\delta_1}{\to} s_1 + \delta_1 \overset{e_1}{\to} s_2 \overset{\delta_2}{\to} s_2 + \delta_2 \overset{e_2}{\to} ...$, such that $s_1 = s$ and for all $i = 1, 2, ..., s_i + \delta_i$ is the δ - successor of s_i and s_{i+1} is the e_i - successor of $s_i + \delta_i$. That is, a run is a path in the semantic graph of A where discrete transitions are taken infinitely often and consecutive time transitions are concatenated. We denote s_i by $\rho\,(i)\,, \delta_i$ by $\mathit{delay}\,(\rho, i)$, and $\sum_{j<i} \delta_j$ by $\mathit{time}\,(\rho, i)$. The limit of the sequence $\mathit{time}\,(\rho, i)$ as $i \to \infty$ is denoted $\mathit{time}\,(\rho, i)$.

Definition 19. Reachable states: A state s is reachable if there exists a finite run $s_0 \overset{\delta_0}{\to}\overset{e_0}{\to} ... \overset{\delta_k}{\to}\overset{e_k}{\to} s_k \overset{\delta}{\to} s$, where $s_0 = (q_0, 0)$ is the initial state, and $k \in \mathbf{N}$.

A parallel composition operator for timed automata is required in order to explain systems made up of different components. We do not provide a formal definition of this operator in this thesis but more information can be found in [100]. The model of parallelism is based on synchronous passage of time for all components and interleaving of discrete actions. Communication is modeled via transition synchronization. Given automata A_1 ,. . . , A_k, their parallel composition is denoted by $A_1 \,||. . .||\, A_k$.

Definition 20. Symbolic states and operations: A set of states of a timed automaton A is called a symbolic state. A zone is a symbolic state S such that:

1. All states of S are associated with the same discrete state, i.e., for all $s, s' \in S$, $\mathit{discrete}(s) = \mathit{discrete}(s')$; and

2. The set of valuations $\{\mathbf{v} \mid \exists\,(q, \mathbf{v}) \in S\}$ is a convex $\mathcal{X}$- polyhedron ζ.

(q, ζ) is often used for the zone S. Also, *false* is utilized to denote the empty zone.

Let S be a symbolic state and e an edge of A. The following operations on S are defined:

$$\textit{disc-pred}(e, S) \overset{def}{=} \left\{ s \mid \exists\ s' \in S.\ s \xrightarrow{e} s' \right\}$$

$$\textit{time-pred}(S) \overset{def}{=} \left\{ s \mid \exists\ s' \in S,\ \delta \in \mathbf{R}.\ s \xrightarrow{\delta} s' \right\}$$

That is, *disc-pred*(e, S) is the set of all e-predecessors of states in S and *time-pred*(S) is the set of all time-predecessors of states in S. According to these definitions, if S is a zone then *time-pred*(S) and *disc-pred*(e, S) are also zones, meaning that zones are preserved by the above predecessor operations.

4.2 Time-Abstracting Bisimulation

The timed automata utilize a dense time domain which results in an infinite state space. Time-abstracting bisimulations technique reduce the state space of given timed automata into a finite graph (the quotient graph) which preserves sufficient information for verification. The idea behind time-abstracting bisimulations is to refine the dense state space only as much as necessary. Exact time delays are abstracted away from the quantitative aspect of time (we know that some time passes but not how much) while information on the discrete-state changes of the system is retained.

There are three time-abstracting bisimulations methods namely, strong, delay and observational, which are strictly arranged in terms of their reduction power. Various properties can be preserved by these three types of time-abstracting bisimulations. The strong one preserves both linear and branching-time properties, whereas the two weaker ones keep only linear-time properties. In this thesis the strong time-abstracting bisimulation is considered because we use TCTL as property-specification languages, which in turn uses a branching-time semantics.

Definition 21. Strong time-abstracting bisimulation: Consider a TA A with set

$$\begin{tikzcd}
s_1 \arrow[r, dotted, no head] \arrow[d, "\delta_1"'] & s_2 \arrow[d, "\delta_2"] \\
s_3 \arrow[r, dotted, no head] & s_4
\end{tikzcd}$$

$$\begin{tikzcd}
s_1 \arrow[r, dotted, no head] \arrow[d, "e_1"'] & s_2 \arrow[d, "e_2"] \\
s_3 \arrow[r, dotted, no head] & s_4
\end{tikzcd}$$

Figure 4.1: Strong Time-Abstracting Bisimulations [101].

of edges E. A binary relation $\approx$ on the states of A is a strong time-abstracting bisimulation (STaB) if for all states $s_1 \approx s_2$, the following conditions hold:

1. If $s_1 \overset{e_1}{\to} s_3$, for some $e_1 \in E$, then there exists $e_2 \in E$ such that $s_2 \overset{e_2}{\to} s_4$ and, $s_3 \approx s_4$.
2. If $s_1 \overset{\delta_1}{\to} s_3$, then there exists $\delta_2 \in \mathbf{R}$ such that $s_2 \overset{\delta_2}{\to} s_4$ and $s_3 \approx s_4$.
3. The above conditions also hold if the roles of s_1 and s_2 are reversed.

The definition is illustrated in Figure 4.1. The states s_1 and s_2 are called STa-bisimilar. In general, two TA A_1 and A_2 are called STa-bisimilar if there exists a STaB $\approx$ on the states of A_1 and A_2, such that $s_0^1 \approx s_0^2$, where s_0^i is the initial state of A_i.

There is always a STaB definable on any automaton namely, the identity relation. There might be many various STaBs but we usually will be interested in the greatest STaB, with respect to the relation inclusion, that is, the STaB which induces the smallest number of equivalence classes. This STaB is unique.

Sometimes it is beneficial to consider only STaBs that distinguish specific states, such as states satisfying various atomic propositions. For this purpose, let *Props* be a set of atomic propositions and let P be a function associating to each proposition a set of states of the automaton. A STaB $\approx$ respects P if for any pair of states s_1; s_2 such that $s_1 \approx s_2$, for all $P \in$ *Props*, $s_1 \in P(p)$ iff $s_2 \in P(p)$.

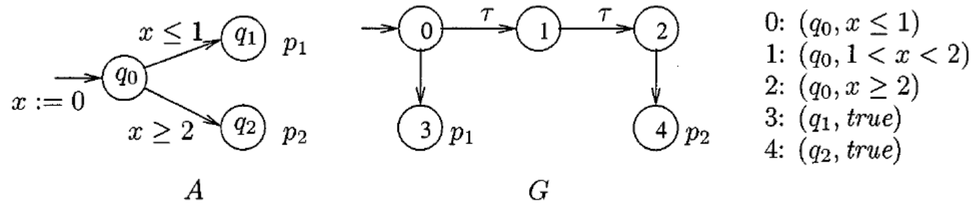


Figure 4.2: Two time-abstracting bisimilar systems. [101].

Definition 22. Time-abstracting quotient graphs: Let $\approx$ be the greatest STaB on a TA A. The $\approx$-quotient graph of A is defined as the graph whose nodes are the classes induced by the STaB and whose edges are of two types:

- $C_1 \xrightarrow{e} C_2$, for some edge e of A, if there exists some state in C_2 which is an e-successor of some state in C_1;
- $C_1 \xrightarrow{\tau} C_2$, where τ is a label denoting time elapse, when C_2 is the immediate time successor of C_1. Formally:

$$\exists s_1 \in C_1, s_2 \in C_2, \delta \in \mathbf{R}.\ s_1 \xrightarrow{\delta} s_2 \wedge \forall \delta' < \delta.\ s_1 + \delta' \in C_1 \cup C_2.$$

In what follows we write $C_1 \rightarrow C_2$ for two classes C_1 and C_2 if either $C_1 \xrightarrow{\tau} C_2$ or $C_1 \xrightarrow{e} C_2$ for some edge e.

Figure 4.2 illustrates the STa-quotient of the TA which is the graph G in the middle of the figure, whose nodes are the symbolic states. They are in fact zones, which are displayed on the right.

Definition 23. Zeno behaviour: An infinite path of a transition system of a timed automaton is zeno if and only it is time convergent and the number of actions executed along path is infinite. A timed automaton is non-zeno if and only if none of its initial states have any zeno path.

Definition 24. Properties of time-abstracting bisimulations: Two main properties of TaBs will be utilized in demonstrating how TA model-checking can be reduced to model-checking in finite graphs (Ta-quotients):

1. **Pre-stability:** Consider a TA A, a TaB $\approx$ on A and two classes C_1 and C_2 in the $\approx$-quotient graph of A. Then, by definition:
   - If $C_1 \xrightarrow{\tau} C_2$ then for each state $s_1 \in C_1$ there exists $s_2 \in C_2$ such that $s_1 \xrightarrow{\delta} s_2$, for some $\delta \in \mathbf{R}$.
   - If $C_1 \xrightarrow{e} C_2$ for some edge e, then if $\approx$ is a TaB then for each state $s_1 \in C_1$ there exists $s_2 \in C_2$ such that $s_1 \xrightarrow{e} s_2$.
2. **Non-zenoness:** TaBs do not preserve non-zenoness since TaBs are insensitive to exact delays.

4.3 Minimization of Timed Automata

The finite graph G is the quotient A with respect to a time-abstracting bisimulation. Producing G entitles minimization and can be performed by adapting a partition-refinement algorithm to the timed case. The algorithm starts with an initial partition of the state space into a finite number of equivalence classes and proceeds by refining the partition until it is stable: for any two classes C_1 and C_2, either all states in C_1 are predecessors of states in C_2, or none is.

The adapted partition-refinement algorithm is symbolic, that is, equivalence classes are symbolic states and set-theoretic operators are utilized to compute the set of predecessor states of a symbolic state. In practice, symbolic states are introduced as simple polyhedra and set-theoretic operators become geometric transformations of polyhedra. When these polyhedra are convex, their machine representation takes $\mathcal{O}\left(n^2\right)$ space, where n is the number of clocks, and the transformations take $\mathcal{O}\left(n^3\right)$ time. However, when the polyhedra

```
Refine ($\mathcal{C}_0$) {
  $\mathcal{C} := \mathcal{C}_0$ ;
  while ($\exists C_1, C_2 \in \mathcal{C} \cdot C_1 \cap \mathsf{preds}(C_2) \notin \{C_1, \emptyset\}$) do
    $\mathcal{C}_{C_1} := \{C_1 \cap \mathsf{preds}(C_2),\ C_1 \setminus \mathsf{preds}(C_2)\}$ ;
    $\mathcal{C} := (\mathcal{C} \setminus \{C_1\}) \cup \mathcal{C}_{C_1}$ ;
  end-while
  return ($\mathcal{C}$) ; }
```

Figure 4.3: A simple partition-refinement algorithm [101].

are not convex, the cost of the transformations becomes exponential both in space and time, due to the lack of a canonical machine representation.

Definition 25. Partition Refinement: Minimization is performed by partition refinement [84]: Given an (untimed) graph $G = (V, \rightarrow)$ (where V is the set of nodes, and $\rightarrow$ define the edges), and an initial partition $\mathcal{C}_0$ of V into disjoint classes, one can successively refines $\mathcal{C}_0$ until a pre-stable partition $\mathcal{C}$ is obtained. That is, for all $C_1, C_2 \in \mathcal{C}$ either, $C_1 \subseteq$ *preds* (C_2) or $C_1\cap$ *preds* $(C_2) = \varnothing$ where:

$$\textit{preds}\ (C) \overset{def}{=} \{v \in V \mid \exists\ v' \in C.\ v \rightarrow v'\}.$$

Refining consists of selecting two classes C_1, C_2 such that C_1 is unstable in terms of C_2, and then replacing C_1 by $C_1\cap$ *preds* (C_2) and $C_1\setminus$ *preds* (C_2). The algorithm is illustrated in Figure 4.3.

The problem with this algorithm is that it refines all classes, whether they are reachable or not (a class is reachable if it contains at least one reachable node). In order to prevent refining unreachable classes, the minimal-model generation algorithm (MMGA) is introduced [19]. The algorithm is shown in Figure 4.4. MMGA utilizes three sets of classes, namely, the current partition $\mathcal{C}$, the set of reachable classes *Access* $\subseteq \mathcal{C}$ and the set of stable classes *stable* $\subseteq \mathcal{C}$. If there exists a reachable class C_1 which may be unstable i.e., $C_1 \in$ *Access* $\setminus$ *Stable*, the algorithm tries to refine it. If it succeeds, C_1 is removed from $\mathcal{C}$ and replaced by its two sub-parts. Otherwise, C_1 is inserted in Stable. The sets Access and Stable are

```
ReachRefine ($\mathcal{C}_0$) {
  $\mathcal{C} := \mathcal{C}_0$ ; $Access := \{C \in \mathcal{C}_0 \mid v_0 \in C\}$ ; $Stable := \{\}$ ;
  while ($\exists C_1 \in Access \setminus Stable$) do
    if ($\exists C_2 \in \mathcal{C} \cdot C_1 \cap \mathsf{preds}(C_2) \not\in \{C_1, \emptyset\}$) then
      $\mathcal{C}_{C_1} := \{C_1 \cap \mathsf{preds}(C_2),\ C_1 \setminus \mathsf{preds}(C_2)\}$ ;
      $Access := (Access \setminus \{C_1\}) \cup \{C \in \mathcal{C}_{C_1} \mid v_0 \in C\}$ ;
      $Stable := Stable \setminus \{C' \in \mathcal{C} \mid C' \cap \mathsf{preds}(C_1) \neq \emptyset\}$ ;
      $\mathcal{C} := (\mathcal{C} \setminus \{C_1\}) \cup \mathcal{C}_{C_1}$ ;
    else
      $Stable := Stable \cup \{C_1\}$ ;
      $Access := Access \cup \{C' \in \mathcal{C} \mid C_1 \cap \mathsf{preds}(C') \neq \emptyset\}$ ;
    end-if
  end-while
  return ($Access$) ; }
```

Figure 4.4: The minimal model generation algorithm [101].

updated accordingly: all predecessor classes of C_1 which were stable are potentially unstable and thus are removed from Stable. Also, a sub-part of C_1 is potentially unreachable, unless it contains the initial node. In the second case, a single-step reachability is performed, to add to the reachable classes all successors of the newly-found stable class.

MMGA can be adapted to infinite state spaces, assuming that they admit effective representations of classes and decision procedures for computing intersection, set-difference, and predecessors of classes, as well as testing whether a class is empty. For termination, it must be ensured that a pre-stable partition always exists. The state space of TA falls in this category, with the difference that there are two kinds of predecessors, related to discrete and time transitions of the TA. Taking this observation into account, the adapted algorithm entitled time-abstracting MMGA (TA-MMGA) is illustrated in Figure 4.5.

In TA-MMGA, the initial partition $\mathcal{C}_0$ is a set of symbolic states. The test for stability of a class S_1 is done in two phases: first stability is checked with respect to time-successors and then with respect to discrete successors. The updates of sets *Stable* and *Access* are modified accordingly. TA-MMGA is parameterized by the discrete-predecessor function $\textit{disc-pred}_{\approx}$,

```
TimeAbstractingReachRefine ($\mathcal{C}_0$, disc-pred$_\approx$()) {
  $\mathcal{C} := \mathcal{C}_0$ ; $Access := \{S \in \mathcal{C}_0 \mid (q_0, \mathbf{0}) \in S\}$ ; $Stable := \{\}$ ;
  while ($\exists S_1 \in Access \setminus Stable$) do
    /* First test stability w.r.t. time transitions */
    if ($\exists S_2 \in \mathcal{C} \cdot S_1 \cap \text{time-pred}(S_2) \notin \{S_1, \emptyset\}$) then
      $\mathcal{C}_{S_1} := \{S_1 \cap \text{time-pred}(S_2),\ S_1 \setminus \text{time-pred}(S_2)\}$ ;
      $Access := (Access \setminus \{S_1\}) \cup \{S \in \mathcal{C}_{S_1} \mid (q_0, \mathbf{0}) \in S\}$ ;
      $Stable := Stable \setminus \{S \in \mathcal{C} \mid S \cap \text{time-pred}(S_1) \neq \emptyset\}$
                        $\setminus \{S \in \mathcal{C} \mid S \cap \text{disc-pred}_\approx(S_1) \neq \emptyset\}$ ;
      $\mathcal{C} := (\mathcal{C} \setminus \{S_1\}) \cup \mathcal{C}_{S_1}$ ;
    /* Then test stability w.r.t. discrete transitions */
    else if ($\exists S_2 \in \mathcal{C} \cdot S_1 \cap \text{disc-pred}_\approx(S_2) \notin \{S_1, \emptyset\}$) then
      $\mathcal{C}_{S_1} := \{S_1 \cap \text{disc-pred}_\approx(S_2),\ S_1 \setminus \text{disc-pred}_\approx(S_2)\}$ ;
      $Access := (Access \setminus \{S_1\}) \cup \{S \in \mathcal{C}_{S_1} \mid (q_0, \mathbf{0}) \in S\}$ ;
      $Stable := Stable \setminus \{S \in \mathcal{C} \mid S \cap \text{time-pred}(S_1) \neq \emptyset\}$
                        $\setminus \{S \in \mathcal{C} \mid S \cap \text{disc-pred}_\approx(S_1) \neq \emptyset\}$ ;
      $\mathcal{C} := (\mathcal{C} \setminus \{S_1\}) \cup \mathcal{C}_{S_1}$ ;
    else
      $Stable := Stable \cup \{S_1\}$ ;
      $Access := Access \cup \{S \in \mathcal{C} \mid S_1 \cap \text{time-pred}(S) \neq \emptyset\}$
                        $\cup \{S \in \mathcal{C} \mid S_1 \cap \text{disc-pred}_\approx(S) \neq \emptyset\}$ ;
    end-if
  end-while
  return ($Access$) ; }
```

Figure 4.5: The time-abstracting minimal model generation algorithm [101].

so that it can be utilized for refinement with respect to any TaB. The function is described as follows:

$$disc\text{-}pred_{\approx}(S) \overset{def}{=} \begin{cases} \bigcup\limits_{e \in E} disc\text{-}pred\ (e, S) & \text{for STaB} \end{cases}$$

4.4 Timelock and Deadlock

Two major issues are part of verifying timed systems. The first issue has to do with the sanity of the model as a whole: it is important to check that the model does not contain deadlocks (same as in the untimed domain) or timelocks (states from which the time cannot advance to infinity). Checking for lack of deadlocks and timelocks is essential when the system consists of several components, the composition of which may lead to unexpected blocking situations. The second issue occurs when checking a specific property: only non-zeno behaviors (behaviors where time progresses without bound) should be considered. The correspondent of this requirement in the untimed domain is the notion of strong fairness.

We start by discussing deadlock and timelock detection using time abstracting quotients. Checking for non-zeno behaviors is more complicated. Untimed strong fairness requirements can be used to remove zeno behaviors from the set of considered behaviors, so we can use a fair CTL semantics and the corresponding techniques. However, these methods dramatically raise the cost of the algorithms. For the sake of simplicity and practicality, the notion of strongly non-zeno systems is defined where a minimal amount of time passes in every cyclic execution. These systems are characterized with simple static conditions, which can be checked compositionally (i.e., if two automata are strongly non-zeno, so it their composition), and it is demonstrated that in strongly non-zeno systems all infinite executions are non-zeno, which exempts us with the burden of checking non-zenoness during model-checking.

Definition 26. DEADLOCKS: Deadlocks are states violating the discrete-progress requirement and the automaton may not be able to perform any further beneficial computation. Discrete progress means it should be possible to take discrete transitions infinitely often. Formally, a state s of a TA A is a deadlock if there is no delay $\delta \in \mathbf{R}$ and edge $e \in E$ such that $s \xrightarrow{\delta}\xrightarrow{e} s'$. A is deadlock-free if none of its reachable states is a deadlock.

Definition 27. NON-ZENO RUNS: Consider an infinite run ρ such that $time\ (\rho) \neq \infty$, that is, there exists $t \in \mathbf{R}$ such that for all i, $time\ (\rho, i) < t$. Such a run, called zeno, corresponds to a pathological situation, since it violates the first of the time-progress requirements. A non-zeno run is a run ρ such that $time(\rho) = \infty$.

Definition 28. TIMELOCKS: Timelocks are states violating the time-progress requirement. Time progress means that it should be possible to let time pass infinitely often and without upper bound. Formally, a state s is a timelock if all infinite runs starting from s are zeno. A is timelock-free if none of its reachable states is a timelock.

Notice that a deadlock is not necessarily a timelock, or the other way around.

Definition 29. Strongly Non-Zeno Timed Automata: Consider a TA A. A structural loop of A is a sequence of distinct edges $e_1...e_m$ such that $\mathit{target}(e_i) = \mathit{source}(e_{i+1})$, for all $i = 1, ..., m$ with the addition that $i + 1$ is modulo m. A is called strongly non-zeno if for every structural loop there exists a clock x and some $0 \leq i, j \leq m$ such that:

1. x is reset in step i , that is, $x \in \mathit{reset}(e_i)$; and

2. x is bounded from below in step j, that is, $(x < 1) \cap \mathit{guard}(e_j) = \mathit{false}$.

This means that at least one unit of time elapses in every loop of A.

If A is strongly non-zeno then every infinite run of A is non-zeno therefore it can be proven that a strongly non-zeno TA is also timelock-free. Strong non-zenoness is interesting because it exempts us from the burden of ensuring time progress. Since strongly non-zeno TA is also timelock-free, checking progress is reduced to checking deadlock-freedom.

If A, A' are strongly non-zeno so is $A \parallel A'$ therefore strong non-zenoness can be checked effectively on large systems, in a component-wise manner.

Deadlock and Timelock Detection Consider a TA A and let G be the quotient of A with respect to a STaB $\approx$. A sink node in G is a node with no successors. It can be proven using the definition of a deadlock and the pre-stability property of $\approx$ that A is deadlock-free iff there is no reachable sink node in G. First, the STa-quotient G of A is produced and then a DFS on G is performed looking for sink nodes. The two steps can be combined so that sink nodes are reported on-the-fly during the construction of G. This can be performed by the time-abstracting partition refinement algorithm.

A strongly non-zeno TA is timelock-free. We must be concerned about timelocks only if A is not strongly non-zeno. In this situation, the following result [57] can be utilized to reduce timelock detection to TCTL model checking: It can be proven that A is timelock-free iff it satisfies the TCTL formula $\forall\, \mathbf{G}\, \exists\, \mathbf{F}_{\geq 1}$ true.

4.5 Verification of TCTL Properties

Our approach is that, given a timed system A, a finite graph G is computed which captures the behavior of A modulo the fact that exact time delays are abstracted away. We then use the branching-time logic TCTL to specify a property. Then, untimed verification techniques on G can be used to prove properties on A. Model checking A against properties specified as a TCTL formula reduces to applying CTL model-checking algorithms on G. The quotient G is generated using strong time abstracting bisimulation (since it is the only one preserving branching-time properties).

If the TCTL formula ϕ to be verified does not involve any timing constraints (i.e., ϕ is a CTL formula), G is generated directly from the original TA A, and a classical CTL model-checking algorithm is utilized to label the nodes of G where ϕ holds. A satisfies ϕ iff ϕ holds in the initial node of G.

Whenever ϕ involves timing constraints, the problem is reduced to CTL verification using a method similar to the one in [4]. G is produced from a TA A^+ which is an extension of A with auxiliary clocks capturing the timing constraints appearing in ϕ. Also, ϕ is transformed into a CTL formula ϕ_{CTL}. Then, CTL model-checking is utilized to label the nodes of G where ϕ_{CTL} holds. A satisfies ϕ iff ϕ_{CTL} holds in the initial node of G.

4.5.1 TCTL Model Checking

Given a TA A to be checked against a TCTL formula ϕ, first A is extended with a set of clocks, to achieve a new automaton A^+; then we transform ϕ to a CTL formula ϕ_{CTL}; finally, the STa-quotient of A^+ is produced and model check it against ϕ_{CTL}. More precisely, let Q and $\mathcal{X}$ be the set of discrete states and set of clocks of A. Also let $I_1, \ldots, I_m$ be the set of non-trivial intervals appearing in ϕ. A^+ has exactly the same structure as A , except that it has an augmented set of clocks $\mathcal{X}^{+} = \mathcal{X} \cup \{y_1, ..., y_m\}$ capturing the timing constraints appearing in ϕ. The set of atomic propositions *Props* is also augmented with two propositions, namely, $p_{yj} = 0$ and $p_{yj} \in I_j$, for each $j = 1, ..., m$. Finally, the formula ϕ is transformed to ϕ_{CTL} recursively as follows:

p	is transformed to	p
$\neg\phi'$	is transformed to	$\neg\phi'_{CTL}$
$\phi' \vee \phi''$	is transformed to	$\phi'_{CTL} \vee \phi''_{CTL}$
$\exists\, \phi'\, \mathcal{U}_{Ij}\, \phi''$	is transformed to	$p_{yj} = 0 \Rightarrow \exists\, \phi'_{CTL}\mathcal{U}\,(\phi''_{CTL} \wedge p_{yj} \in I_j)$
$\forall\, \phi'\, \mathcal{U}_{Ij}\, \phi''$	is transformed to	$p_{yj} = 0 \Rightarrow \forall\, \phi'_{CTL}\mathcal{U}\,(\phi''_{CTL} \wedge p_{yj} \in I_j)$

Regarding the CTL model checking, consider a TA A, a CTL formula ϕ on a set of atomic propositions *Props* and a function P mapping each atomic proposition to a set of discrete states of A. It can be checked whether A satisfies ϕ. It is assumed that A is deadlock-free and strongly non-zeno.

Let $\approx$ be a STaB on A respecting P, that is, if $(q_1, \mathbf{v}_1) \approx (q_2, \mathbf{v}_2)$ then $q_1 \in P(p)$ iff $q_2 \in P(p)$, for any $p \in$ *Props*. Let G be the $\approx$-quotient of A. A formula is said to hold in a node C of G if it is satisfied in some state of C, which implies that the formula is satisfied in any state of C. To check $A \models_p \phi$, the function *ctl-eval*(ϕ) is utilized which is described as follows :

$$\textit{ctl-eval}(p) = \{C \mid \exists\,(q, \mathbf{v}) \in C.\, q \in P(p)\}$$

We determine all the nodes of G where ϕ holds and we also check whether the initial node of G is contained in *ctl-eval*(ϕ). Then $C \in$ *ctl-eval*(ϕ) iff for all $s \in C, s$ satisfies ϕ. According to the above definitions, we conclude that $(q, \mathbf{v}) \models \phi$ iff there exists a state $(q, \mathbf{v}^{+})$ of A^{+} such that $\forall x \in \mathcal{X}.\mathbf{v}(x) = \mathbf{v}^{+}(x)$, and if C is the class of $(q, \mathbf{v}^{+})$ in G then $C \in$ *ctl-eval*(ϕ_{CTL}) [12] [101].

Part II

Timed Kripke Structures

Chapter 5

Timed Kripke Structures and Timed Computation Tree Logic

Timed Kripke structures are Kripke structures where each transition has a duration. There are several TCTL model checking algorithms available in the point-wise semantics for finite timed Kripke structures. The TCTL model checking algorithm we consider refines the one in [67]. We wonder whether such point-wise model checking can be utilized in order to achieve a sound and complete model checking procedure for TCTL formulas in continuous semantics.

Suppose all timed transitions $s \xrightarrow{t} s'$ are split into a sequence of timed transitions $s \xrightarrow{\gamma} s_1 \xrightarrow{\gamma} s_2 \xrightarrow{\gamma} ...s_k \xrightarrow{\gamma} s'$ with the same total duration, and where each s_i is some state that satisfies the same atomic propositions that s does. The duration γ is the half of the greatest common divisor of all the following time values: all non-zero durations happening in the timed Kripke structure and all non-zero finite time values appearing as bounds in the TCTL formula under consideration. Point-wise model checking can then be used on the resulting, still finite, timed Kripke structure. Essentially a sound and complete model checking method in the continuous semantics is obtained under specified conditions that are satisfied by many interesting classes of systems. Point-wise model checking can be used on the resulting timed Kripke structure, which is still finite.

The idea outlined above applies to those discrete-event systems where the time when the next event happens is given deterministically, and where the progress of time does not modify the valuation of the atomic propositions. The latter condition usually holds in practice because time spends only influences timers and clocks, whose values rarely affect the validity of an atomic proposition.

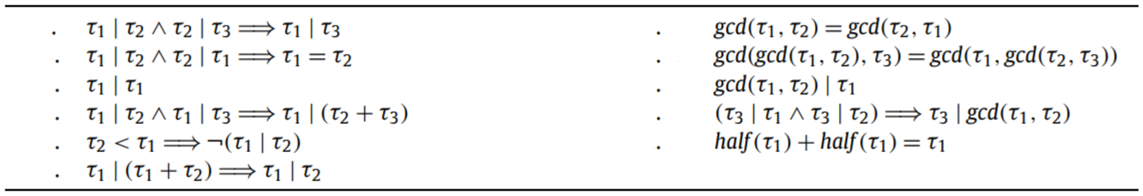
In all, the model checking of a TCTL formula under the more natural continuous semantics is reduced to a TCTL model checking problem in the point-wise semantics. There are straightforward model checking algorithms are available for timed Kripke structures which are parametric in the (discrete or dense) time domain. In particular, the methods which are explained latter are independent of the formalism utilized to explain the timed Kripke structure.

5.1 Time Domains

A time domain is either discrete, such as the set of natural numbers $\mathbb{N}$, or dense, such as the non-negative rational numbers $\mathbb{Q}_{\geq 0}$. Instead of considering a specific time domain, a general algebraic abstract specification of time is utilized. Time is thus a linearly ordered commutative monoid $\mathcal{T} = (\mathit{Time}, +, 0, <)$ with additional operators, such as $\dot{-}$, where $\tau \dot{-} \tau'$ signify $\tau - \tau'$ if $\tau < \tau'$ and 0 otherwise, and *max*, where $\mathit{max}(\tau, \tau') = \tau'$ if $\tau < \tau'$ and τ otherwise. $\mathcal{T}$ is then a time domain satisfying the theory TIME. To simplify the notation, the $\mathcal{T}$ is used for both the algebra and the carrier of the algebra, and write $0, +, ...$ for the interpretations $0_{\mathcal{T}}, +_{\mathcal{T}}$ of $0, +, ...$ in $\mathcal{T}$. We use $\tau, \tau', \tau_1, ...$ to denote time values, and write $k\tau$ or $k.\tau$ for $\tau + \tau + \cdots + \tau$ k times. Time $\setminus \{0\}$ is denoted by $\mathrm{Time}_{>0}$. The theory TIME_{∞} stands for TIME with the new infinity element $\infty \notin$ Time, such that $\mathrm{Time}_{\infty} = \mathrm{Time} \cup \{\infty\}$.

A time interval is a non-empty interval of the form $[a, b]$, $(a, b]$, $[a, b_{\infty})$ or (a, b_{∞}), where $a, b \in \mathcal{T}$ and $b_{\infty} \in \mathcal{T}_{\infty}$. Let *Intervals* ($\mathcal{T}$) be the set of all time intervals in $\mathcal{T}$. For some $I \in \mathit{Intervals}(\mathcal{T})$ we have:

$$\inf(I) = \max\{\tau \in \mathcal{T}_{\infty} | \forall \tau' \in I.\tau \leq \tau'\} \text{ to be the infimum of I;}$$



. $\tau_1 \mid \tau_2 \wedge \tau_2 \mid \tau_3 \Longrightarrow \tau_1 \mid \tau_3$	. $gcd(\tau_1, \tau_2) = gcd(\tau_2, \tau_1)$
. $\tau_1 \mid \tau_2 \wedge \tau_2 \mid \tau_1 \Longrightarrow \tau_1 = \tau_2$	. $gcd(gcd(\tau_1, \tau_2), \tau_3) = gcd(\tau_1, gcd(\tau_2, \tau_3))$
. $\tau_1 \mid \tau_1$	. $gcd(\tau_1, \tau_2) \mid \tau_1$
. $\tau_1 \mid \tau_2 \wedge \tau_1 \mid \tau_3 \Longrightarrow \tau_1 \mid (\tau_2 + \tau_3)$	. $(\tau_3 \mid \tau_1 \wedge \tau_3 \mid \tau_2) \Longrightarrow \tau_3 \mid gcd(\tau_1, \tau_2)$
. $\tau_2 < \tau_1 \Longrightarrow \neg(\tau_1 \mid \tau_2)$	. $half(\tau_1) + half(\tau_1) = \tau_1$
. $\tau_1 \mid (\tau_1 + \tau_2) \Longrightarrow \tau_1 \mid \tau_2$	

Figure 5.1: Axioms for the theory $TIME^{gcd}$.

$$\sup(I) = \min\{\tau \in \mathcal{T}_\infty | \forall \tau' \in I.\tau' \leq \tau\} \text{ to be the supremum of I;}$$

An interval $I \in \mathit{Intervals}(\mathcal{T})$ satisfies $(\forall \tau \in \mathcal{T}.\ \inf(I) < \tau < \sup(I) \Rightarrow \tau \in I)$ The theory TIME^{gcd} adds the following to TIME:

1. $\mid : \text{Time}_{>0} \times \text{Time}_{>0} \to \text{Bool}$
2. $\text{gcd} : \text{Time}_{>0} \times \text{Time}_{>0} \to \text{Time}_{>0}$
3. $\text{half} : \text{Time}_{>0} \to \text{Time}_{>0}$

This theory satisfies the axioms in Figure 5.1 for each $\tau_1, \tau_2, \tau_3 \in \text{Time}_{>0}$. Intuitively, $\tau_1 | \tau_2$ (τ_1 divides τ_2 or τ_1 is a divisor of τ_2) is true if adding up τ_1 for a finite number of times gives τ_2; $\text{gcd}(\tau_1, \tau_2)$ denotes the greatest common divisor of τ_1 and τ_2 which always exist; and half (τ_1) denotes the half of τ_1, that is 2 . half $(\tau_1) = \tau_1$. For example, $\mathbb{Q}_{\geq 0}$ satisfies this theory with the standard interpretation of divisors and the greatest common divisor operator. The theory $TIME^{gcd}_\infty$ adds to $TIME^{gcd}$ the infinity element ∞.

5.2 Timed Kripke Structures

Different timed extensions of Kripke structures (KS) exist [67] [48] [29] [30] cite-laroussinie2003expressivity [31]. The timed Kripke structures used here annotates transitions with durations.

Definition 30. Timed Kripke Structures: Given a set AP of atomic propositions, a timed Kripke structure over AP is a tuple $\mathcal{TK} = (S, \mathcal{T}, \to, L)$, where S is a set of states, $\mathcal{T}$

is a time domain satisfying the theory TIME, $\rightarrow \subseteq S \times \mathcal{T} \times S$ is a total transition relation with duration (i.e., for each $s \in S$ there exist τ and s' such that $(s, \tau, s') \in \rightarrow$), and L is a labeling function $L : S \rightarrow \mathcal{P}(AP)$. $s \xrightarrow{\tau} s'$ is used for $(s, \tau, s') \in \rightarrow$. A transition $s \xrightarrow{0} s'$ is called instantaneous and a transition $s \xrightarrow{\tau} s'$ with $\tau > 0$ is a tick transition. $\mathcal{TK}$ is finite iff $\rightarrow$ is finite.

A transition $s \xrightarrow{\tau} s'$, with $\tau > 0$, can be interpreted in two ways: In the point-wise interpretation an "atomic tick step" of duration τ means that we jump directly from s_0 to s_1 without "visiting" any intermediate time in between. Alternatively, the continuous interpretation assumes that time advance continuously from s_0 to s_1. These interpretations can be formalized by defining the notion of configuration of a timed Kripke structure.

Definition 31. Configurations: A configurations $\langle s, \delta \rangle \in S \times \mathcal{T}$ of $\mathcal{TK}$ specifies that the system has been in state s for time δ.

Definition 32. Interpretations of a Timed Kripke Structure: The continuous interpretation of $\mathcal{TK}$ is given by the one-step transition relation $\hookrightarrow \subseteq (S \times \mathcal{T}) \times \mathcal{T} \times (S \times \mathcal{T})$ defined by the following rules:

$$\frac{s \xrightarrow{\tau} s' \quad \delta + \tau' < \tau}{\langle s, \delta \rangle \overset{\tau'}{\hookrightarrow} \langle s, \delta + \tau' \rangle} \text{Rule}_{tick1} \qquad \frac{s \xrightarrow{\tau} s' \quad \delta + \tau' = \tau}{\langle s, \delta \rangle \overset{\tau'}{\hookrightarrow} \langle s', 0 \rangle} \text{Rule}_{tick2} \qquad \frac{s \xrightarrow{0} s'}{\langle s, 0 \rangle \overset{0}{\hookrightarrow} \langle s', 0 \rangle} \text{Rule}_{inst}$$

where $\tau, \tau' > 0$. The point-wise interpretation of $\mathcal{TK}$ is given by the operational semantics consisting of the rules Rule_{tick2} and Rule_{inst}.

The rules Rule_{tick1} and Rule_{tick2} signify tick steps which permit time progress, whereas rule Rule_{inst} defines instantaneous steps. The tick steps explained by Rule_{tick2} are maximal, since they permit time to elapse by the maximal possible duration of the given transition. In contrast, Rule_{tick1} permits time to elapse in a state $s \in S$ by a value less than the duration of the longest tick step from s. The point-wise interpretation of a timed Kripke structure allows only instantaneous and maximal tick steps, and therefore all reachable configurations $\langle s, \delta \rangle$ have $\delta = 0$.

Definition 33. Valid Configurations in Point-Wise and Continuous Semantics: Given a state $s \in S$, we denote by $\mathcal{T}_{\mathcal{TK}}(s) \subset \mathcal{T}$ the set $\left\{\tau \in \mathcal{T} | \, \exists s' \in S \, . \, s \xrightarrow{\tau} s' \wedge 0 < \tau\right\}$ of positive tick step durations from s, and the sets $\mathcal{C}^p_{\mathcal{TK}} \subseteq S \times \mathcal{T}$ are defined respectively. $\mathcal{C}^c_{\mathcal{TK}} \subseteq S \times \mathcal{T}$ of valid configurations of $\mathcal{TK}$ in the point-wise and continuous interpretation, respectively are defined as follows:

$$\mathcal{C}^p_{\mathcal{TK}} = \{\langle s, 0\rangle \, | s \in S\}$$

$$\mathcal{C}^c_{\mathcal{TK}} = \mathcal{C}^p_{\mathcal{TK}} \cup \{\langle s, \delta\rangle \mid \exists \tau \in \mathcal{T}_{\mathcal{TK}}(s) \, .\delta < \tau\}$$

Definition 34. Path in the Point-Wise Interpretation: A path π of $\mathcal{TK}$ in the point-wise interpretation is an infinite sequence of steps $\langle s_0, 0\rangle \overset{\tau_0}{\hookrightarrow} \langle s_1, 0\rangle \overset{\tau_1}{\hookrightarrow} \langle s_2, 0\rangle \overset{\tau_2}{\hookrightarrow}$ ..., where $\langle s_0, 0\rangle \in \mathcal{C}^p_{\mathcal{TK}}$ and each $\langle s_i, 0\rangle \overset{\tau_i}{\hookrightarrow} \langle s_{i+1}, 0\rangle$ is a step allowed in the point-wise interpretation of $\mathcal{TK}$. The $s^\pi_i = s_i$, and $c^\pi_i = \sum_{j=0}^{i-1} \tau_j$ are defined.

Definition 35. Path in the Continuous Interpretation: A path π of $\mathcal{TK}$ in the continuous interpretation is an infinite sequence of steps $\langle s_0, \delta_0\rangle \overset{\tau_0}{\hookrightarrow} \langle s_1, \delta_1\rangle \overset{\tau_1}{\hookrightarrow} \langle s_2, \delta_2\rangle \overset{\tau_2}{\hookrightarrow}$ ..., where $\langle s_0, \delta_0\rangle \in \mathcal{C}^c_{\mathcal{TK}}$ and each $\langle s_i, \delta_i\rangle \overset{\tau_i}{\hookrightarrow} \langle s_{i+1}, \delta_{i+1}\rangle$ a step allowed in the continuous interpretation of $\mathcal{TK}$. The $s^\pi_i = s_i$, and $c^\pi_i = \sum_{j=0}^{i-1} \tau_j$ and $\delta^\pi_i = \delta_i$ are defined.

$\text{Paths}^P_{\mathcal{TK}}(\langle s, 0\rangle)$ denotes the set of all the paths in the point-wise interpretation of $\mathcal{TK}$ starting in a valid configuration $\langle s, 0\rangle \in \mathcal{C}^p_{\mathcal{TK}}$. Similarly, $\text{Paths}^c_{\mathcal{TK}}(\langle s, \delta\rangle)$ is the set of all paths in the continuous interpretation of $\mathcal{TK}$ starting in $\langle s, \delta\rangle \in \mathcal{C}^c_{\mathcal{TK}}$ [72].

5.3 Zeno-Free Timed Kripke Structures

The continuous interpretation of a timed Kripke structure permits Zeno paths.

Definition 36. Time-Divergent Path: A path π of $\mathcal{TK}$ is time-divergent if for each $\tau \in \mathcal{T}$ there is an $i \in \mathbb{N}$ such that $c^\pi_i > \tau$, and is time-convergent or Zeno otherwise.

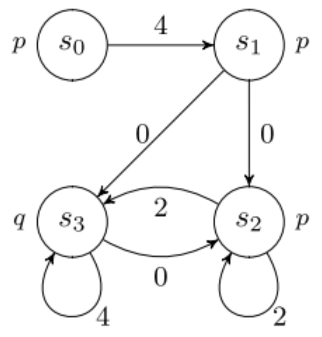


Figure 5.2: A timed Kripke structure.

Figure 6.1 shows a timed Kripke structure with the time domain $\mathbb{Q}_{\geq 0}$. When interpreted continuously, the path $\langle s_0, 0\rangle \overset{1}{\hookrightarrow} \langle s_0, 1\rangle \overset{1/2}{\hookrightarrow} \langle s_0, 1 + 1/2\rangle \overset{1/4}{\hookrightarrow} \langle s_0, 1 + 1/2 + 1/4\rangle \overset{1/8}{\hookrightarrow} \ldots$ is possible. This path is Zeno i.e., it has an infinite length but time 2 is never reached.

The unrealizable behaviour modeled by Zeno paths, paths (like the one in the above example) cannot be excluded by suitable modelling. The Zeno paths together with those paths that execute infinitely many instantaneous steps in finite time must be taken into the account as a modelling flaw. In the rest of the thesis, Zeno-free timed Kripke structures will be considered. A finite timed Kripke structure is Zeno-free whenever it does not contain any loops consisting of instantaneous transitions only (zero-loops).

The notion of reachability is limited to configurations that are reachable by time-divergent paths. The notation $\text{tdPaths}^p_{\mathcal{TK}}(\langle s, 0\rangle)$ is utilized for the set of all time-divergent paths in the point-wise interpretation of $\mathcal{TK}$ starting in a valid configuration $\langle s, 0\rangle \in \mathcal{C}^p_{\mathcal{TK}}$. The notation $\text{tdPaths}^c_{\mathcal{TK}}(\langle s, \delta\rangle)$ is used for the set of all time-divergent paths in the continuous interpretation of $\mathcal{TK}$ starting in $\langle s, \delta\rangle \in \mathcal{C}^c_{\mathcal{TK}}$.

A configuration $\langle s', 0\rangle \in \mathcal{C}^p_{\mathcal{TK}}$ is reachable from $\langle s, 0\rangle \in \mathcal{C}^p_{\mathcal{TK}}$ in time τ in the point-wise interpretation iff there exists a path $\pi \in \text{tdPaths}^p_{\mathcal{TK}}(\langle s, 0\rangle)$ and some $i \in \mathbb{N}$ such that $s^\pi_i = s'$ and $c^\pi_i = \tau$. The configuration $\langle s', \delta'\rangle \in \mathcal{C}^c_{\mathcal{TK}}$ is reachable from $\langle s, \delta\rangle \in \mathcal{C}^c_{\mathcal{TK}}$ in time τ in the continuous interpretation iff there exists a path $\pi \in \text{tdPaths}^c_{\mathcal{TK}}(\langle s, \delta\rangle)$ and some $i \in \mathbb{N}$ such that $s^\pi_i = s'$, $\delta^\pi_i = \delta'$ and $c^\pi_i = \tau$. A state $s' \in S$ is reachable from

$s \in S$ iff there exist two time values $\delta', \tau \in \mathcal{T}$ such that the configuration $\langle s', \delta' \rangle$ is valid and reachable from $\langle s, 0 \rangle$ in time τ. We note that $\mathrm{Paths}^{p}_{\mathcal{TK}}(\langle s, 0 \rangle) \subseteq \mathrm{Paths}^{c}_{\mathcal{TK}}(\langle s, 0 \rangle)$ and $\mathrm{tdPaths}^{p}_{\mathcal{TK}}(\langle s, 0 \rangle) \subseteq \mathrm{tdPaths}^{c}_{\mathcal{TK}}(\langle s, 0 \rangle)$.

Definition 37. Timed Configuration: The set $\mathcal{TC}_{\mathcal{TK}} \subseteq S \times \mathcal{T} \times \mathcal{T}$ of timed configurations of $\mathcal{TK}$ is defined as $\mathcal{TC}_{\mathcal{TK}} = \{(s, \delta, c) \,|\, \langle s, \delta \rangle \in \mathcal{C}_{\mathcal{TK}} \text{ and } c \in \mathcal{T}\}$. A timed configuration (s, δ, c) is denoted by $\langle s, \delta \rangle\, @c$. Let $\pi \in \mathrm{Paths}_{\mathcal{TK}}(\langle s_0, \delta_0 \rangle)$, for some $\langle s_0, \delta_0 \rangle \in \mathcal{C}_{\mathcal{TK}}$ be the path $\langle s_0, \delta_0 \rangle \overset{\tau_0}{\hookrightarrow} \langle s_1, \delta_1 \rangle \overset{\tau_1}{\hookrightarrow} \ldots$; then the corresponding timed path$@\pi$ is denoted by $\langle s_0, \delta_0 \rangle\, @c_0 \overset{\tau_0}{\hookrightarrow} \langle s_1, \delta_1 \rangle\, @c_1 \overset{\tau_1}{\hookrightarrow} \ldots$ with $c_i = c_i^{\pi}$ for all $i \in \mathbb{N}$. The notions defined for untimed configurations are utilized in the same way as long as they make sense for timed configurations; for example we say that a timed path$@\pi$ is time-divergent (time-convergent) if its corresponding path π is time-divergent (time-convergent).

According to the above definition, timed paths begin at the global time 0, i.e., $c_0 = 0$. In the point-wise interpretation of a timed Kripke structure, all reachable timed configurations $\langle s, \delta \rangle\, @c$ have $\delta = 0$.

Definition 38. Point-Wise and Continuous Positions in a Path: Assume a path $\pi = \langle s_0, \delta_0 \rangle \overset{\tau_0}{\hookrightarrow} \langle s_1, \delta_1 \rangle \overset{\tau_1}{\hookrightarrow} \ldots \in \mathrm{Paths}_{\mathcal{TK}}(\langle s_0, \delta_0 \rangle)$ starting in some configuration $\langle s_0, \delta_0 \rangle \in \mathcal{C}_{\mathcal{TK}}$. A point-wise position in π is any timed configuration $\langle s_i, \delta_i \rangle\, @c_i^{\pi}$ with $i \in \mathbb{N}$. For each pointwise position $\langle s_i, \delta_i \rangle\, @c_i^{\pi}$ in π the set is defined as follows:

$$\mathrm{pre}^{p}_{\pi}\left(\langle s_i, \delta_i \rangle\, @c_i^{\pi}\right) = \left\{ \langle s_j, \delta_j \rangle\, @c_j^{\pi} \mid j \in \mathbb{N} \text{ and } 0 \le j < i \right\}$$

of its point-wise predecessor positions.

A continuous position in π is any timed configuration $\langle s_i, \delta_i + \tau \rangle\, @c_i^{\pi} + \tau$ with $i \in \mathbb{N}$ and $0 \le \tau < \tau_i$. For each continuous position $\langle s_i, \delta_i + \tau \rangle\, @c_i^{\pi} + \tau$ in π the set is defined as follows:

$$\mathrm{pre}^{c}_{\pi}\left(\langle s_i, \delta_i + \tau \rangle\, @c_i^{\pi} + \tau\right) =$$
$$\left\{ \langle s_j, \delta_j + \tau' \rangle\, @c_j^{\pi} + \tau' \mid (i = j \text{ and } 0 \le \tau' < \tau) \text{ or } (0 \le j < i \text{ and } 0 \le \tau' < \tau_j) \right\}$$

of its continuous predecessor positions. The point-wise (continuous) positions in $@\pi$ are the point-wise (continuous) positions in π.

5.4 Timed Computation Tree Logic

Various timed extensions of temporal logics have been designed [9] [105]. Timed CTL (TCTL) [4] is in particular an extension of CTL [36] with interval time constraints on temporal operators.

Definition 39. Timed computation tree logic: Given a set AP of atomic propositions and a time domain $\mathcal{T}$, TCTL formulas φ are built using the following grammar:

$$\varphi ::= \mathit{true} \ \mid \ p \ \mid \neg\varphi \mid \ \varphi \wedge \varphi \mid \mathrm{E}\,\psi \mid \ \mathrm{A}\,\psi$$

$$\psi ::= \ \ \varphi \ U_I \ \varphi$$

where $p \in AP$ and $I \in \text{Intervals}\,(\mathcal{T})$.

The path formula $\psi = \ \ \varphi_1 \ U_I \ \varphi_2$ holds for a given path π if φ_2 holds at a position $\langle s, \delta \rangle\, @c$ in π happening at time $c \in I$ and φ_1 holds for all predecessor positions. The existentially quantified state formula $\mathrm{E}\,\psi$ holds in a given configuration if there exists some path that begins from that configuration and satisfies ψ, while the universally quantified state formula $\mathrm{A}\,\psi$ holds in a given configuration if ψ holds in each path that starts from that configuration. The time interval subscript is often described relationally, using $= a$, $\leq b$, $< b$, $\geq a$, and $> a$ for $[a, a]$, $[0, b]$, $[0, b)$, $[a, \infty)$, and (a, ∞), respectively. TCTL_{cb} is the fragment of TCTL without open finite bounds i.e., where all time intervals are of the form $[a, b]$ with $a \leq b$, or $[a, \infty)$. The universal and the existential fragments TACTL and TECTL of TCTL [53] allow negation only in front of atomic propositions and limit quantification to the universal quantifier and to the existential quantifier, respectively.

Definition 40. TCTL Point-Wise Semantics: For a timed Kripke structure $\mathcal{TK} = (S, \mathcal{T}, \rightarrow, L)$, a valid configuration of $\mathcal{TK}$ in the pointwise interpretation $\langle s, 0 \rangle \in \mathcal{C}^{p}_{\mathcal{TK}}$, and a

TCTL formula φ, the point-wise satisfaction relation $\mathcal{TK}, \langle s, 0\rangle \models_p \varphi$ is defined inductively below. Then $\mathcal{TK}, s \models_p \varphi$ iff $\mathcal{TK}, \langle s, 0\rangle \models_p \varphi$.

$\mathcal{TK}, \langle s, 0\rangle \models_p true$		always holds.
$\mathcal{TK}, \langle s, 0\rangle \models_p p$	iff	$p \in L(s)$.
$\mathcal{TK}, \langle s, 0\rangle \models_p \neg\varphi_1$	iff	$\mathcal{TK}, \langle s, 0\rangle \not\models_p \varphi_1$.
$\mathcal{TK}, \langle s, 0\rangle \models_p \varphi_1 \wedge \varphi_2$	iff	$\mathcal{TK}, \langle s, 0\rangle \models_p \varphi_1$ and $\mathcal{TK}, \langle s, 0\rangle \models_p \varphi_2$.
$\mathcal{TK}, \langle s, 0\rangle \models_p E\,\psi$	iff	$\mathcal{TK}, \pi \models_p \psi$ for some $\pi \in \mathrm{tdPaths}^p_{\mathcal{TK}}(\langle s, 0\rangle)$.
$\mathcal{TK}, \langle s, 0\rangle \models_p A\,\psi$	iff	$\mathcal{TK}, \pi \models_p \psi$ for all $\pi \in \mathrm{tdPaths}^p_{\mathcal{TK}}(\langle s, 0\rangle)$.
$\mathcal{TK}, \pi \models_p \varphi_1\, U_I\, \varphi_2$	iff	there is a pointwise position $\langle s'', 0\rangle\,@c''$ in π $s.t$ $c'' \in I$, $\mathcal{TK}, \langle s'', 0\rangle \models_p \varphi_2$ and $\mathcal{TK}, \langle s', 0\rangle \models_p \varphi_1$ for all pointwise positions $\langle s', 0\rangle\,@c'$ in $\mathrm{pre}^p_\pi(\langle s'', 0\rangle\,@c'')$.

Definition 41. TCTL Continuous Semantics: Given $\mathcal{TK}$ as above, $\langle s, \delta\rangle \in \mathcal{C}^c_{\mathcal{TK}}$ a valid configuration in the continuous interpretation of $\mathcal{TK}$, and a TCTL formula φ. The continuous satisfaction relation $\models_c$ is defined similarly to $\models_p$, but using the notion of continuous position instead of point-wise position, and $\mathrm{tdPaths}^c_{\mathcal{TK}}(\langle s, d\rangle)$ instead of $\mathrm{tdPaths}^p_{\mathcal{TK}}(\langle s, d\rangle)$. The resulting definition is shown below. Then $\mathcal{TK}, s \models_c \varphi$ iff $\mathcal{TK}, \langle s, 0\rangle \models_c \varphi$.

$\mathcal{TK}, \langle s, \delta\rangle \models_c true$		always holds.
$\mathcal{TK}, \langle s, \delta\rangle \models_c p$	iff	$p \in L(s)$.
$\mathcal{TK}, \langle s, \delta\rangle \models_c \neg\varphi_1$	iff	$\mathcal{TK}, \langle s, \delta\rangle \not\models_c \varphi_1$.
$\mathcal{TK}, \langle s, \delta\rangle \models_c \varphi_1 \wedge \varphi_2$	iff	$\mathcal{TK}, \langle s, \delta\rangle \models_c \varphi_1$ and $\mathcal{TK}, \langle s, \delta\rangle \models_c \varphi_2$.
$\mathcal{TK}, \langle s, \delta\rangle \models_c E\,\psi$	iff	$\mathcal{TK}, \pi \models_c \psi$ for some $\pi \in \mathrm{tdPaths}^c_{\mathcal{TK}}(\langle s, \delta\rangle)$.
$\mathcal{TK}, \langle s, \delta\rangle \models_c A\,\psi$	iff	$\mathcal{TK}, \pi \models_c \psi$ for all $\pi \in \mathrm{tdPaths}^c_{\mathcal{TK}}(\langle s, \delta\rangle)$.
$\mathcal{TK}, \pi \models_c \varphi_1\, U_I\, \varphi_2$	iff	there is a continuous position $\langle s'', \delta''\rangle\,@c''$ in π $s.t$ $c'' \in I$, $\mathcal{TK}, \langle s'', \delta''\rangle \models_c \varphi_2$ and $\mathcal{TK}, \langle s', \delta'\rangle \models_c \varphi_1$ for all continuous positions $\langle s', \delta'\rangle\,@c'$ in $\mathrm{pre}^c_\pi(\langle s'', \delta''\rangle\,@c'')$.

In the point-wise semantics checking a formula happens only immediately after a discrete event. By contrast, in the continuous semantics checking happens continuously through a tick step at all possible times. Specifically, in the continuous semantics a tick transition $s \xrightarrow{\tau} s'$ needs to take into account any possible splitting $\langle s, 0\rangle \overset{\tau_0}{\hookrightarrow} \langle s, \tau_0\rangle \overset{\tau_1}{\hookrightarrow} \ldots \overset{\tau_k}{\hookrightarrow} \langle s, \tau_0 + \ldots + \tau_k\rangle \overset{\tau_{k+1}}{\hookrightarrow} \langle s', 0\rangle$, with $\tau_0 + \tau_1 + \ldots + \tau_k + \tau_{k+1} = \tau$. Furthermore, any possible continuous position in any path needs to be considered.

Chapter 6

Reducing Continuous to Point-Wise Semantics in Model Checking

We now show how TCTL model checking in the continuous semantics can be reduced to TCTL model checking in the point wise semantics, which is amenable to explicit-state model checking. The transformation of a timed Kripke structure $\mathcal{TK}$ will result in the model checking of the resulting timed Kripke structure in the point-wise semantics being equivalent to model checking $\mathcal{TK}$ in the continuous semantics. The transformation involves two main steps as follows:

Firstly, a gcd-transformation mapping the timed Kripke structure $\mathcal{TK}$ to another timed Kripke structure $\mathcal{TK}_a^{\mathrm{half}(GCD(\mathcal{TK},\varphi))}$ is performed so that

$$\mathcal{TK}, s \models_c \varphi \;\;\Leftrightarrow\;\; \mathcal{TK}_a^{\mathrm{half}(GCD(\mathcal{TK},\varphi))}, (s,0) \models_p \beta\left(\varphi\right),$$

for a TCTL$_{cb}$ formula φ. All temporal operators in φ are annotated with closed intervals $[a,b]$ or $[a,\infty)$; $(s,0)$ is the state in $\mathcal{TK}_a^{\mathrm{half}(GCD(\mathcal{TK},\varphi))}$ a corresponding to s, and $\beta\left(\varphi\right)$ is a transformation of φ.

Then we convert a given TCTL formula (with possibly open intervals) φ to a TCTL$_{cb}$ formula $\alpha\left(\varphi\right)$, so that

$$\mathcal{TK}_a^{\mathrm{half}(GCD(\mathcal{TK},\varphi))}, (s,0) \models_c \varphi \;\;\Leftrightarrow\;\; \mathcal{TK}_a^{\mathrm{half}(GCD(\mathcal{TK},\varphi))}, (s,0) \models_c \alpha\left(\varphi\right),$$

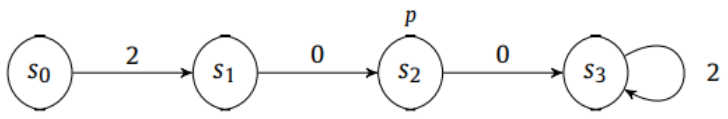


Figure 6.1: A timed Kripke structure.

The above results will be merged together according to the fact that the gcd-transformation does not influence the continuous semantics of timed Kripke structures, i.e. $\mathcal{TK}, s \models_c \varphi \;\Leftrightarrow\; \mathcal{TK}_a^{\text{half}(GCD(\mathcal{TK},\varphi))}, (s,0) \models_c \varphi$. The desired equivalence is achieved which decrease the model checking problem in the continuous semantics to a model checking problem in the point-wise semantics:

$$\mathcal{TK}, s \models_c \varphi \;\Leftrightarrow\; \mathcal{TK}_a^{\text{half}(GCD(\mathcal{TK},\varphi))}, (s,0) \models_p \beta\left(\alpha\left(\varphi\right)\right),$$

for any TCTL formula φ [72].

6.1 Abstraction of TKS by GCD-Transformations and τ-Transformations

To be sure that time progress stops at any time point where a time bound in the formula can be attained, any tick transition is split into a sequence of tick transitions of a smaller duration τ that divides the duration of each tick transition and each finite non-zero time bound in the formula.

The example from Figure 6.1 demonstrates that it is not adequate to always advance time by the greatest common divisor (gcd) γ of these values to achieve a sound and complete model checking in the continuous semantics. In the point-wise interpretation, $\mathcal{TK}$ has only one behaviour from s_0, which is illustrated with respect to the validity of the atomic proposition p along the path $\pi = \neg p \overset{2}{\to} \neg p \overset{0}{\to} p \overset{0}{\to} \neg p \overset{2}{\to} \neg p \overset{2}{\to} \ldots$ (i.e., $\neg p$ forever). The only p-state is reachable in exactly time 2. All tick transitions have duration 2. Consider then $\varphi = E\;\varphi_1 U_{=2}$ true, where φ_1 is the formula $E\;F_{=2}\;p$. The formula φ shows that φ_1 must hold in all positions until time 2 is reached. The greatest common divisor of all tick transition durations and all time values in φ is 2, therefore converting $\mathcal{TK}$ by splitting each tick transition into a sequence of transitions, each of duration 2 leaves it unchanged.

In the point-wise semantics this conversion (i.e., the above behaviour) satisfies φ in terms of the initial state s_0. However,$\mathcal{TK}$, $s_0 \models_c \varphi$ does not hold, since φ_1 does not hold in the continuous positions along the first tick step; for example, it does not hold at time 1. The basis of the method is to capture these "intermediate" states by further splitting the tick transitions of duration equal to the gcd into two smaller ones. The time is advanced not by the greatest common divisor γ, but by a time γ_h such that $2 \,.\, \gamma_h = \gamma$, that is, by "half" the gcd, in each tick step. A generalization of the gcd-transformation is τ -transformations which splits a tick transition $s \overset{\tau'}{\rightarrow} s'$ into k transitions of duration τ followed by a transition of duration $\tau' \dot{-} k\tau$, and needs k new states (s, δ) at times $\delta = \tau, 2\tau, ..., k\tau$.

Definition 42. τ- TRANSFORMATION: Let $\tau \in \mathcal{T}$ be a non-zero time value. The τ -transformation of $\mathcal{TK}$ is the timed Kripke structure $\mathcal{TK}^{\tau} = (S^{\tau}, \mathcal{T}, \rightarrow^{\tau}, L^{\tau})$ with

- $S^{\tau} = \{(s, \delta) \in S \times \mathcal{T} | \delta = 0 \vee (\exists n \in \mathbb{N}, \tau' \in \mathcal{T}_{\mathcal{TK}}(s) . \delta = n\tau < \tau')\}$.
- $((s_1, \delta_1), \tau_1, (s_2, \delta_2)) \in \rightarrow^{\tau}$ if and only if
  1. $s_1 \overset{0}{\rightarrow} s_2$ and $\tau_1 = \delta_1 = \delta_2 = 0$; or
  2. $s_1 \overset{\tau_1'}{\rightarrow} s_2$ and $s_2 = s_1$, $\tau_1 = \tau$ $\delta_2 = \delta_1 + \tau < \tau_1'$; or
  3. $s_1 \overset{\tau_1'}{\rightarrow} s_2$ and $\tau_1 \leq \tau$, $\delta_1 + \tau_1 = \tau_1'$, $\delta_2 = 0$
- $L^{\tau}(s, \delta) = L(s)$.

A state (s, δ) in $\mathcal{TK}^{\tau}$ shows that the system is in state s and that time has advanced by δ since the last $\mathcal{TK}$-transition. Instantaneous $\mathcal{TK}$-transitions are introduced as transitions $(s_1, 0) \overset{0}{\rightarrow}{}^{\tau}$ $(s_2, 0)$ in $\mathcal{TK}^{\tau}$, and a tick $\mathcal{TK}$-transition $s_1 \overset{k\tau+\tau_1}{\longrightarrow} s_2'$ is demonstrated by the sequence $(s_1, 0) \overset{\tau}{\rightarrow} (s_1, \tau) \overset{\tau}{\rightarrow} ... \overset{\tau}{\rightarrow} (s_1, k\tau) \overset{\tau_1}{\rightarrow} (s_2, 0)$.

Proposition 4. *Let $\langle s, n\tau + \delta\rangle \in \mathcal{C}^c_{\mathcal{TK}}$ for some $s \in S, n \in \mathbb{N}, \tau \in \mathcal{T}$ with $\tau > 0$ and $\delta \in \mathcal{T}$ with $\delta < \tau$. Then*

$$\mathcal{TK}\ \ , \langle s, n\tau + \delta\rangle \models_c \varphi \textit{ iff } \mathcal{TK}^{\tau}\ , \langle (s, n\tau), \delta\rangle \models_c \varphi.$$

In otger words, a τ-transformation does not affect the continuous semantics of timed Kripke structures.

Definition 43. $GCD\,(\mathcal{TK}, \varphi)$: Given a timed Kripke structure $\mathcal{TK} = (S, \mathcal{T}, \rightarrow, L)$ over a set of atomic propositions AP, where the time domain $\mathcal{T}$ satisfies TIME^{gcd}, and a TCTL formula φ over AP, it is defined:

$$\mathcal{T}_{\mathcal{TK}} = \left\{\tau \in \mathcal{T} \mid \exists\, s, s' \in S\ .\ s \xrightarrow{\tau} s' \wedge 0 < \tau\right\}$$

$$\mathcal{T}_{\varphi} = \{\tau \in \{inf\,(I)\,, sup\,(I)\}\ \setminus \{0, \infty\} \mid \text{there exists a subformula } \ \varphi_1\ U_I\ \varphi_2 \ \text{ of } \ \varphi\}$$

$$GCD\,(\mathcal{TK}, \varphi) = gcd\,(\mathcal{T}_{\mathcal{TK}} \cup \mathcal{T}_{\varphi})$$

where, for $\mathcal{T}' \subseteq \mathcal{T}$ a finite non-empty set of time values, $gcd\,(\mathcal{T}')$ is the greatest common divisor of these time values, and is recursively defined as $gcd\,(\{\tau\}) = \tau$ and $gcd\,(\{\tau_1, \tau_2\} \uplus \mathcal{T}') = gcd\,(\{gcd\,(\tau_1, \tau_2)\} \cup \mathcal{T}')$.

If $\mathcal{TK}$ is finite and has at least one tick transition then the set of time values $\mathcal{T}_{\mathcal{TK}} \cup \mathcal{T}_{\varphi}$ is non-empty and finite. In the rest of this thesis it is assumed that the time domain of $\mathcal{TK}$ satisfies TIME^{gcd} and that $\mathcal{TK}$ is finite and Zeno-free and has at least one tick transition. The γ is utilized to denote $GCD\,(\mathcal{TK}, \varphi)$, and γ_h to denote half(γ).

6.2 Reducing Continuous to Point-Wise Semantics in TCTL$_{\mathbf{cb}}$ Model Checking

Let φ be a TCTL$_{cb}$ formula. A non-Zeno path in $\mathcal{TK}^{\gamma_h}$ beginning in state $(s_0, 0)$ with only maximal tick steps consists of an alternating sequence of zero or more instantaneous steps, followed by a sequence of one or more tick steps having total duration multiple of γ:

$$\langle (s_0,0),0\rangle \underbrace{\overset{0}{\hookrightarrow} \dots \overset{0}{\hookrightarrow}}_{k_0 \text{ times}} \langle (s_{k_0},0),0\rangle \quad \underbrace{\overset{\gamma_h}{\hookrightarrow} \dots \overset{\gamma_h}{\hookrightarrow}}_{\text{total duration } \gamma n_0} \quad \langle (s'_{k_0},0),0\rangle \underbrace{\overset{0}{\hookrightarrow} \dots \overset{0}{\hookrightarrow}}_{k_1 \text{ times}} \dots$$

where for each $i \in \mathbb{N}$, $k_i, n_i \in \mathbb{N}$, and $n_i > 0$, there exists a tick transition of duration γn_i from s_{ki} to s'_{ki} in $\mathcal{TK}$. This maximal path in $\mathcal{TK}^{\gamma_h}$ corresponds to the following maximal path in $\mathcal{TK}$:

$$\langle s_0,0\rangle \underbrace{\overset{0}{\hookrightarrow} \dots \overset{0}{\hookrightarrow}}_{k_0 \text{ times}} \langle s_{k_0},0\rangle \overset{\gamma n_0}{\hookrightarrow} \langle s'_{k_0},0\rangle \underbrace{\overset{0}{\hookrightarrow} \dots \overset{0}{\hookrightarrow}}_{k_1 \text{ times}} \dots$$

Definition 44. Canonical Representative of a Configuration σ: Given $0 < \tau \in \mathcal{T}$ such that $GCD(\mathcal{TK},\varphi)$ is a multiple of 2τ and $\mathcal{TK}^\tau = (S,\mathcal{T},\to,L)$ the τ -transformation of $\mathcal{TK}$. The function $\text{can}_{\mathcal{TK}^\tau} : \mathcal{C}^c_{\mathcal{TK}^\tau} \to \mathcal{C}^c_{\mathcal{TK}^\tau}$ on valid configurations $\langle (s,n\tau),\delta\rangle$ of $\mathcal{TK}^\tau$ is defined as

$$\text{can}_{\mathcal{TK}^\tau}(\langle (s,n\tau),\delta\rangle) = \begin{cases} \langle (s,n\tau),0\rangle & \text{if } \delta = 0 \text{ or } n \text{ is odd} \\ \langle (s,(n+1)\tau,0\rangle) & \text{otherwise} \end{cases}$$

and call $\text{can}_{\mathcal{TK}^\tau}(\langle (s,n\tau),\delta\rangle)$ the canonical representative of $\langle (s,n\tau),\delta\rangle$. We omit the subscript $\mathcal{TK}^\tau$ when it is clear from the context [72].

The canonical representative of configurations $\langle (s,0),0\rangle$ in $\mathcal{TK}^{\gamma_h}$, which corresponds to the state s in $\mathcal{TK}$, is the identity. The canonical representative of configurations $\langle (s,\delta),\delta'\rangle$, with $0 < \delta' < \gamma_h$, is a configuration $\langle (s,\delta''),0\rangle$, such that δ'' is an odd multiple of γ_h, i.e.,$\delta'' = (2n+1)\gamma_h$ for some $n \in \mathbb{N}$.

Figure 6.2 display the mapping of configurations to their canonical representatives on the time segment corresponding to the path of $\mathcal{TK}$ in Figure 6.1 Dotted arrows map a

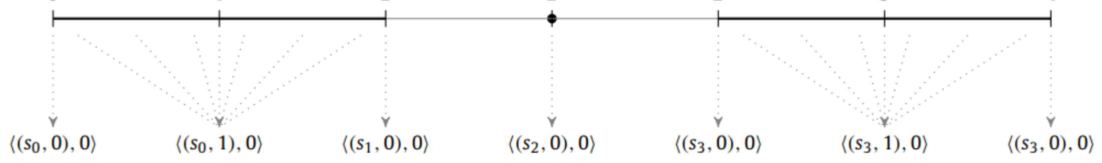


Figure 6.2: Mapping configurations to their canonical representative.

configuration on the top to its canonical representative on the bottom. The continuous positions along tick steps are defined by a thick black line, and the continuous position after a discrete step at time 2 is shown by a black circle.

γ divides all finite non-zero time bounds in any time interval from φ. Therefore, any finite bound in such time intervals is a multiple of $2\gamma_h$ (e.g., if $\gamma_h = 1/2$ and $I = (2, \infty)$ is $(2\gamma_h \,.\, 2, \infty)$). Any time interval I in a TCTL$_{cb}$ formula has the form $[2m_l \,.\, \gamma_h, 2m_u \,.\, \gamma_h]$ for some $m_l, m_u \in \mathbb{N}$ with $m_l \leq m_u$, or $I = [2m_l \,.\, \gamma_h, \infty)$ for some $m_l \in \mathbb{N}$.

To show the correctness of the reduction the following definition shows that if the time duration between two configurations along a path is in a given time interval I (appearing in φ) then the time duration between the corresponding canonical representatives is also in I:

Proposition 5. *Let $\tau \in \mathcal{T}, \tau > 0$ and I a time interval such that either $I = [2m_l \,.\, \tau, 2m_u \,.\, \tau]$ for some $m_l, m_u \in \mathbb{N}$ with $m_l \leq m_u$, or $I = [2m_l \,.\, \tau, \infty]$ for some $m_l \in \mathbb{N}$. Let furthermore $\delta_i \in \mathcal{T}$ with $\delta_i = 2n_i\tau + \tau_i$ for some $n_i \in \mathbb{N}$ and $\tau_i \in \mathcal{T}$, $0 \leq \tau_i < 2\tau$ for $i = 0, 1$. It is defined*

$$\delta_i^* = \begin{cases} \delta_i & \text{if } \tau_i = 0 \\ 2n_i\tau + \tau & \text{if } \tau_i > 0 \end{cases}$$

for $i = 0, 1$.Then

$$\delta_1 - \delta_0 \in I \;\; \Rightarrow \;\; \delta_1^* - \delta_0^* \in I$$

Proposition 6. *Let φ be a TCTL$_{cb}$ formula over AP, $0 < \tau \in \mathcal{T}$ such that $GCD\,(\mathcal{TK}, \varphi)$ is a multiple of 2τ and $\mathcal{TK}^\tau\,(S, \mathcal{T}, \rightarrow, L)$ the τ-transformation of $\mathcal{TK}$. Then for each valid configuration σ of $\mathcal{TK}^\tau$ it holds that*

$$\mathcal{TK}^\tau, \sigma \models_c \varphi \;\; \Leftrightarrow \;\; \mathcal{TK}^\tau, can\,(\sigma) \models_c \varphi$$

In other words, a configuration σ and its canonical representative can$\,(\sigma)$ satisfy the same sub-formulas of φ in the continuous semantics. This shows that the mapping to the canonical representatives maintains the equivalence for any TCTL$_{cb}$ formula with time bounds that are multiples of the computed greatest common divisor γ. Additionally, for a tick step of duration γ_h in $\mathcal{TK}^{\gamma_h}$, all continuous configurations during this tick step (i.e., all configurations $\langle (s,\delta), \delta' \rangle$ with $0 < \delta < \gamma_h$) satisfy the same sub-formulas of φ. Therefore, it is possible to abstract these configurations using the respective canonical representative (i.e., $\mathrm{can}\left(\langle (s,\delta), \delta' \rangle\right)$), which, has the form $\langle (s, (2n+1)\gamma_h), 0 \rangle$ for some $n \in \mathbb{N}$. Abstract configurations are the canonical representatives for sets of configurations, while concrete configurations (of the form $\langle (s, (2n)\gamma_h), 0 \rangle$) only represent themselves. In Figure 6.2 the abstract configurations are $\langle (s_0, 1), 0 \rangle$ and $\langle (s_3, 1), 0 \rangle$ while the concrete configurations are $\langle (s_0, 0), 0 \rangle$, $\langle (s_1, 0), 0 \rangle$, $\langle (s_2, 0), 0 \rangle$ and $\langle (s_3, 0), 0 \rangle$.

If a path begins in an abstract configuration, then all the positions at an even multiple of γ_h are also abstract, and all positions at an odd multiple of γ_h are concrete. If a path begins in a concrete configuration, then all positions at an even multiple of γ_h are also concrete, and all positions at an odd multiple of γ_h are abstract. This holds for any $\mathcal{TK}^{\tau}$ such that γ_h is a multiple of τ .

Given an abstract configuration (respectively, a concrete configuration) σ of $\mathcal{TK}^{\tau}$, such that γ_h is a multiple of τ and a path $\pi \in \mathrm{Paths}_{\mathcal{TK}^{\tau}}(\sigma)$, it can be proven that both the following facts are true:

- For all positions $\sigma'@c$ in π such that c is an even multiple of τ, it holds that σ' is an abstract configuration (respectively, a concrete configuration).

- For all positions $\sigma'@c$ in π such that c is an odd multiple of τ, it holds that σ' is a concrete configuration (respectively, an abstract configuration).

A modification $\beta(\varphi)$ of formula φ is introduce which captures the following conditions, and therefore has the same satisfaction (in the continuous semantics) as φ. A path π satisfies the formula $\varphi_1\, U_I\, \varphi_2$ in the continuous semantics iff one of the following formulas holds (in both the continuous and point-wise semantics):

1. $\varphi_1\, U_I$ (φ_2 and "concrete"), that is φ_2 holds at a position in I corresponding to some concrete configuration (and φ_1 holds in all positions before), or

2. $\varphi_1 \, U_I \, (\varphi_1 \wedge \varphi_2$ and "abstract"), that is $\varphi_1 \wedge \varphi_2$ holds at a position in I corresponding to some abstract configuration (and φ_1 holds in all positions before), or

3. the interval I includes time 0 and φ_2 holds immediately (in particular, when the configuration at the first position is an abstract one).

In order to obtain an elegant definition of the modified formula $\beta\left(\varphi\right)$, all states in $\mathcal{TK}^{\gamma_h}$ are labeled as $\left(s, \left(2n+1\right)\gamma_h\right)$ and a new proposition p_a corresponding to the abstract configurations is utilized. Configurations corresponding to some abstract state (i.e., p_a-states) can be separated from other concrete configurations (i.e., configurations corresponding to some $\neg p_a$-state) directly in a TCTL formula. The gcd-transformation $\mathcal{TK}_a^{\gamma_h}$ of a timed Kripke structure is defined to be $\mathcal{TK}^{\gamma_h}$, where the labeling function uses the new atomic proposition p_a.

Definition 45. Abstract τ -transformation: Let φ be a TCTL formula over AP, $0 < \tau \in \mathcal{T}$ such that $GCD\left(\mathcal{TK}, \varphi\right)$ is a multiple of 2τ and $\mathcal{TK}^{\tau}\left(S, \mathcal{T}, \rightarrow, L\right)$ be the τ -transformation of $\mathcal{TK}$. It is defined $AP_a = AP \uplus \{p_a\}$ and an extended labeling function $L_a : S \rightarrow \mathcal{P}\left(AP_a\right)$, such that:

$$\forall \left(s, \delta\right). \; L_a\left(s, \delta\right) = \begin{cases} L\left(s, \delta\right) \cup \{p_a\} & \text{if } \exists\, n \in \mathbb{N} : \delta = \left(2n+1\right)\tau, \\ L\left(s, \delta\right) & \text{otherwise.} \end{cases}$$

The timed Kripke structure $\left(S, \mathcal{T}, \rightarrow, L_a\right)$ is denoted by $\mathcal{TK}_a^{\tau}$ and we refer to it as the abstract τ -transformation of $\mathcal{TK}$. When $2\tau = GCD\left(\mathcal{TK}, \varphi\right)$, we denote the abstract τ -transformation of $\mathcal{TK}$ by $\mathcal{TK}_a^{\gamma_h}$ and we refer to it as the gcd-transformation of $\mathcal{TK}$.

The abstract τ -transformation does not influence the continuous semantics of the τ -transformation i.e., $\mathcal{TK}^{\tau}, \left(s, 0\right) \models_c \varphi' \;\Leftrightarrow\; \mathcal{TK}_a^{\tau}, \left(s, 0\right) \models_c \varphi'$, where φ is a TCTL formula over the atomic propositions AP.

Definition 46. TCTL$_{cb}$ formula $\beta\left(\varphi\right)$: Let φ be a TCTL$_{cb}$ formula over AP, $0 < \tau \in \mathcal{T}$ such that $GCD\left(\mathcal{TK}, \varphi\right)$ is a multiple of 2τ and $\mathcal{TK}_a^{\tau} = \left(S, \mathcal{T}, \rightarrow, L\right)$ is the abstract τ -transformation of $\mathcal{TK}$, so that $L : S \rightarrow \mathcal{P}\left(AP_a\right)$ and $AP_a = AP \uplus \{p_a\}$. The TCTL$_{cb}$

formula $\beta(\varphi)$ is inductively defined as follows:

$$\begin{array}{lcl}
\beta(true) & = & \text{true};\\
\beta(p) & = & p;\\
\beta(\neg\,\varphi_1) & = & \neg\beta(\varphi_1);\\
\beta(\varphi_1 \wedge \varphi_2) & = & \beta(\varphi_1) \wedge \beta(\varphi_2);\\
\beta(E\,\varphi_1\,U_I\,\varphi_2) & = & \begin{cases} \beta(\varphi_2) \vee E\,\psi & \text{if } 0 \in I\\ E\,\psi & \text{if } 0 \notin I; \end{cases}\\
\beta(A\,\varphi_1\,U_I\,\varphi_2) & = & \begin{cases} \beta(\varphi_2) \vee A\,\psi & \text{if } 0 \in I\\ A\,\psi & \text{if } 0 \notin I, \end{cases}
\end{array}$$

where $\psi = \beta(\varphi_1) \;\; U_I \;\; (\beta(\varphi_2) \wedge (\neg p_a \vee \beta(\varphi_1)))$.

Proposition 7. *Let φ be a TCTL$_{cb}$ formula over AP, $0 < \tau \in \mathcal{T}$ such that $GCD(\mathcal{TK}, \varphi)$ is a multiple of 2τ and $\mathcal{TK}_a^\tau = (S, \mathcal{T}, \rightarrow, L)$ the abstract τ -transformation of $\mathcal{TK}$, so that $L : S \rightarrow \mathcal{P}(AP_a)$ and $AP_a = AP \uplus \{p_a\}$ are as given in the definition of $\mathcal{TK}_a^\tau$. Then for each state (s, δ) of $\mathcal{TK}_a^\tau$ and each sub-formula φ' of φ, it holds that*

$$\mathcal{TK}_a^\tau, (s, \delta) \models_c \varphi' \;\;\Leftrightarrow\;\; \mathcal{TK}_a^\tau, (s, \delta) \models_p \beta(\varphi')$$

That is, model checking $\beta(\varphi)$ in $\mathcal{TK}_a^{\gamma_h}$ in the point-wise semantics is equivalent to model checking the TCTL$_{cb}$ formula φ in $\mathcal{TK}$ under the continuous semantics.

Proposition 8. *[72] Let AP be a set of atomic propositions and $\mathcal{TK}$ a timed Kripke structure over AP whose time domain satisfies the theory TIME^{gcd}. Let φ be a TCTL$_{cb}$ formula over AP, $0 < \tau \in \mathcal{T}$ such that $GCD(\mathcal{TK}, \varphi)$ is a multiple of 2τ and $\mathcal{TK}_a^\tau = (S, \mathcal{T}, \rightarrow, L)$ the abstract τ -transformation of $\mathcal{TK}$. Then for each state s of $\mathcal{TK}$ and each subformula φ' of φ it holds that*

$$\mathcal{TK}, s \models_c \varphi' \;\;\Leftrightarrow\;\; \mathcal{TK}_a^\tau, (s, 0) \models_p \beta(\varphi')$$

6.3 Reducing Continuous to Point-Wise Semantics in TCTL Model Checking

The result from the previous section are developed to the entire TCTL logic by transforming a TCTL formula φ with possibly open interval bounds to a TCTL$_{cb}$ formula $\alpha(\varphi)$, so that

$\alpha(\varphi)$ holds in $\mathcal{TK}_a^{\gamma_h}$ in the continuous semantics if and only if φ holds in $\mathcal{TK}_a^{\gamma_h}$ in the continuous semantics:

$$\mathcal{TK}_a^{\gamma_h}, (s, \delta) \models_c \varphi \;\;\Leftrightarrow\;\; \mathcal{TK}_a^{\gamma_h}, (s, \delta) \models_c \alpha(\varphi)$$

This in turn implies that model checking $\beta(\alpha(\varphi))$ in $\mathcal{TK}_a^{\gamma_h}$ in the point-wise semantics is equivalent to model checking φ in $\mathcal{TK}$ in the continuous one:

$$\mathcal{TK}, s \models_c \varphi \;\;\Leftrightarrow\;\; \mathcal{TK}_a^{\gamma_h}, (s, 0) \models_p \beta(\alpha(\varphi))$$

The key observation is that a path starting from a concrete configuration will reach the finite bounds $inf(I)$ and $sup(I)$ at some concrete configurations. A path begining from an abstract configuration will obtain the above interval bounds at some abstract configurations. A path π satisfies the TCTL path formula $\varphi_1 \; U_{(a,b)} \; \varphi_2$ with the open bound (a, b), if and only if one of the following two conditions hold:

1. The initial configuration is concrete and the formula $\varphi_1 \; U_{[a+\gamma_h, b-\gamma_h]} \; \varphi_2$, with the contracted interval $[a + \gamma_h, b - \gamma_h]$ holds in π.

2. The initial configuration is abstract and the formula $\varphi_1 \; U_{[a,b]} \; \varphi_2$, with the extended interval $[a, b]$ holds in π.

The above conditions are captured by the modification of a TCTL formula φ into the TCTL$_{cb}$ formula $\alpha(\varphi)$, defined (for a general $\mathcal{TK}_a^{\tau}$ such that τ is a divisor of γ_h) as follows.

Definition 47. TCTL$_{cb}$ FORMULA $\alpha(\varphi)$: Let φ be a TCTL formula over AP, $0 < \tau \in \mathcal{T}$ such that $GCD(\mathcal{TK}, \varphi)$ is a multiple of 2τ and $\mathcal{TK}_a^{\tau} = (S, \mathcal{T}, \rightarrow, L)$ is the abstract τ - transformation of $\mathcal{TK}$, so that $L : S \rightarrow \mathcal{P}(AP_a)$ and $AP_a = AP \uplus \{p_a\}$. The TCTL$_{cb}$ formula $\alpha(\varphi)$ is inductively defined as follows:

$$\begin{array}{rcl}
\alpha(true) & = & \text{true;} \\
\alpha(p) & = & \text{p;} \\
\alpha(\neg\varphi_1) & = & \neg\alpha(\varphi_1); \\
\alpha(\varphi_1 \wedge \varphi_2) & = & \alpha(\varphi_1) \wedge \alpha(\varphi_2); \\
\alpha(E\,\varphi_1\, U_I\, \varphi_2) & = & (\neg p_a \wedge E\,\alpha(\varphi_1)\; U_{I_1}\; \alpha(\varphi_2)) \vee (p_a \wedge E\,\alpha(\varphi_1)\; U_{I_2}\; \alpha(\varphi_2)); \\
\alpha(A\,\varphi_1\, U_I\, \varphi_2) & = & (\neg p_a \wedge A\,\alpha(\varphi_1)\; U_{I_1}\; \alpha(\varphi_2)) \vee (p_a \wedge A\,\alpha(\varphi_1)\; U_{I_2}\; \alpha(\varphi_2));
\end{array}$$

where I_1 is the time interval $[a,b]$, with a and b two time values such that

$$a = \begin{cases} inf\,(I) & \text{if } inf\,(I) \in I, \\ inf\,(I) + \tau & \text{otherwise.} \end{cases}$$

$$b = \begin{cases} sup\,(I) & \text{if } sup\,(I) \in I, \\ sup\,(I) \dot{-} \tau & \text{otherwise.} \end{cases}$$

and $I_2 = [inf\,(I)\,, sup\,(I)]$.

Proposition 9. *[72] Let φ be a TCTL formula over AP, $0 < \tau \in \mathcal{T}$ a time value such that $GCD\,(\mathcal{TK}, \varphi)$ is a multiple of 2τ and $\mathcal{TK}_a^\tau = (S, \mathcal{T}, \rightarrow, L)$ the abstract τ-transformation of $\mathcal{TK}$, so that $L : S \rightarrow \mathcal{P}\,(AP_a)$ and $AP_a = AP \uplus \{p_a\}$. Then the following holds for each state (s, δ) in $\mathcal{TK}_a^\tau$ and each subformula φ' of φ:*

$$\mathcal{TK}_a^\tau, (s,\delta) \models_c \varphi' \;\; \Leftrightarrow \;\; \mathcal{TK}_a^\tau, (s,\delta) \models_p \alpha\,(\varphi')\,.$$

Proposition 10. *Let AP be a set of atomic propositions and $\mathcal{TK}$ a timed Kripke structure over AP whose time domain satisfies the theory $TIME^{gcd}$. Let φ be a TCTL formula over AP, $0 < \tau \in \mathcal{T}$ such that $GCD\,(\mathcal{TK}, \varphi)$ is a multiple of 2τ and $\mathcal{TK}_a^\tau = (S, \mathcal{T}, \rightarrow, L)$ the abstract τ -transformation of $\mathcal{TK}$. Then for each state s of $\mathcal{TK}$ and each sub-formula φ' of φ:*

$$\mathcal{TK}, s \models_c \varphi' \;\; \Leftrightarrow \;\; \mathcal{TK}_a^\tau, (s,0) \models_p \beta\,(\alpha\,(\varphi'))\,.$$

6.4 Model Checking for Discrete Time

In discrete time TCTL model checking for a formula φ and a timed Kripke structure $\mathcal{TK}$ can be obtained by exhaustively visiting all time instants, which can be performed by splitting each tick transition in $\mathcal{TK}$ with smaller ones of duration one (i.e., by model checking the 1-transformation of $\mathcal{TK}$). When the time domain is $\mathbb{N}$, all transitions in $\mathcal{TK}^1$ have duration 0 or 1.

Proposition 11. *Let AP be a set of atomic propositions and $\mathcal{TK}$ a timed Kripke structure over AP whose time domain is $\mathbb{N}$. Let φ be a TCTL formula over AP, and $\mathcal{TK}^1 =$*

$(S, \mathbb{N}, \rightarrow, L)$ *the 1-transformation of* $\mathcal{TK}$*. Then for each state* s *of* $\mathcal{TK}$ *and each subformula* φ' *of* φ*:*

$$\mathcal{TK}, s \models_c \varphi' \;\;\Leftrightarrow\;\; \mathcal{TK}^1, (s, 0) \models_p \varphi'.$$

The continuous interpretation of $\mathcal{TK}^1$ is equivalentand to the point-wise interpretation of $\mathcal{TK}^1$, since each tick step in $\mathcal{TK}^1$ is a maximal one, and therefore its one-step transition relation uses only rules $Rule_{tick1}$ and $Rule_{tick2}$ in both interpretations. Model checking $\mathcal{TK}^1$ in the continuous semantics is equivalent to model checking it in the point-wise semantics i.e., for each state s in $\mathcal{TK}^1$ and each sub-formula φ' of a TCTL formula φ over AP:

$$\mathcal{TK}^1, (s, 0) \models_c \varphi' \;\;\Leftrightarrow\;\; \mathcal{TK}^1, (s, 0) \models_p \varphi'.$$

The heavy price of obtaining soundness and completeness for discrete time in this method has to do with the fact that visiting all discrete times typically result in a state space explosion that renders model checking infeasible. For instance, this method is not sufficient if discrete events do not occur for relatively long time intervals, e.g., if each tick transition duration in the system and each finite time bound appearing in the formula is a multiple of 10000 time units.

A more efficient model checking method can be obtained by adapting the gcd-transformation to the time domain $\mathbb{N}$. The gcd-transformation is directly applicable to timed Kripke structures over the time domain $\mathbb{N}$, if the greatest common divisor γ is an even number, i.e., if $\gamma_h = \text{half}\,(\gamma)$ is defined.

If the greatest common divisor is odd but larger than 1, we can multiply each finite constant appearing in the timed Kripke structure and in the TCTL formula by 2. This transformation does not influence the TCTL semantics of the formula, but the greatest common divisor becomes even, such that the procedure can be utilized. The following fact formalizes that this scaling is satisfiability-preserving: For each state s in $\mathcal{TK}$ with time domain $\mathbb{N}$ and each sub-formula φ' of a TCTL formula φ over AP with $GCD\,(\mathcal{TK}, \varphi) > 1$:

$$\mathcal{TK}, s \models_c \varphi' \;\;\Leftrightarrow\;\; (\,2.\,\mathcal{TK})_a^{\gamma_h}, (s, 0) \models_p 2\,.\,\beta\,(\alpha\,(\varphi'))\,.$$

where $2 . \mathcal{TK}$ is achieved from $\mathcal{TK}$ by multiplying each finite transition duration by 2, and $2 . \beta(\alpha(\varphi'))$ is achieved from $\beta(\alpha(\varphi'))$ by multiplying each finite time value appearing in some interval bound by 2.

If the greatest common divisor is 1 then the formula might evaluate in various way for discrete and for dense time domains, since satisfaction might rely on whether there is any configuration reachable between two successive discrete time points. Scaling up durations in a discrete time domain with greatest common divisor 1 would correspond to inserting such reachable configurations and therefore it could affect the evaluation of TCTL formulas in the continuous semantics [72].

6.5 TCTL Model Checking in Point-Wise Semantics

TCTL model checking in point-wise semantics relies on the explicit-state CTL model checking approach [12] that recursively calculates the set of reachable states satisfying the sub-formula for each sub-formula of the desired TCTL formula:

1. show that any TCTL formula is equivalent to one in normal form, and
2. describe a model checking procedure for TCTL formulas in normal form.

The method is also based on the assumption that in the point-wise interpretation of timed Kripke structures, all paths in the timed Kripke structure are time-divergent. A finite timed Kripke structure $\mathcal{TK}$ is zeno-free iff it has no zero-time loops. Hence checking zeno-freeness of $\mathcal{TK}$ can be performed in linear time as follows: Let $\mathcal{TK}_0$ be $\mathcal{TK}$ where all tick transitions have been deleted such as only instantaneous transitions are left. Then $\mathcal{TK}$ is Zeno-free iff $\mathcal{TK}_0$ has no loops.

The model described here adapts the one in [67], which is enhanced to handle formulas with interval time bounds. Some core methods are also simplified, which is possible since the assumption of Zeno-freeness excludes zero-time loops. For the reminder of the thesis s stands for the configuration $\langle s, 0\rangle$ and $s \xrightarrow{\tau} s'$ is used for a step $\langle s, 0\rangle \overset{\tau}{\hookrightarrow} \langle s', 0\rangle$.

6.5.1 TCTL Normal Form

Definition 48. TCTL Normal Form: Given a set AP of atomic propositions and a time domain $\mathcal{T}$, TCTL formulas in normal form are built using the following grammar:

$$\varphi ::= \mathit{true} \ \mid p \mid \neg\varphi \mid \ \varphi \wedge \varphi \mid \mathrm{E}\, G\ \varphi \ \mid \mathrm{E}\ \varphi\ U_I\ \varphi \mid \mathrm{A}\ \varphi\ U_{I>0}\ \varphi$$

where $p \in AP,\ I,\ I_{>0}\ \in \mathit{Intervals}\,(\mathcal{T})$, and $\mathit{inf}\,(I_{>0}) = 0 \notin\ I_{>0}$.

Figure 6.3 shows the list of the equivalences that can be applied to the sub-formulas of a TCTL formula φ to reduce it to its normal form $\varphi!$. Equivalences might have a validity condition (i.e., a condition under which they are applicable). The equivalences to formulas are on sub-formulas that are not yet in normal form, as described by the application conditions. They are used with a priority that relies on the order in which they are listed in the figure: when more than one equivalence can be utilized for reducing the same formula, the one listed first is selected. The right-hand side of an equivalence is not necessarily in normal form and may require to be further reduced by utilizing other equivalences.

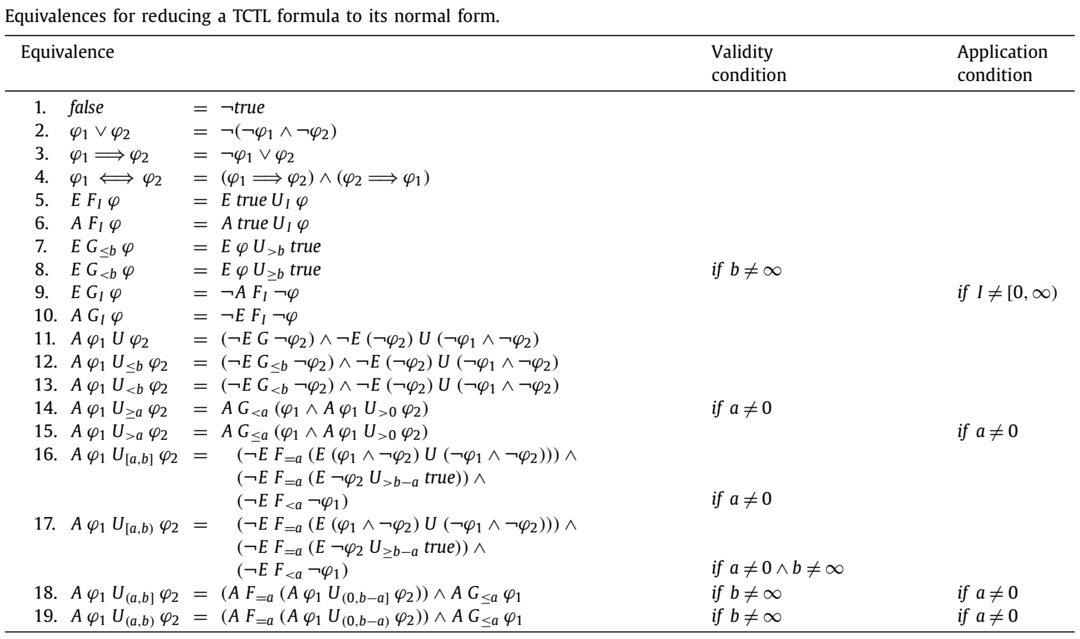
Equivalences (1)-(10) are given by the syntactic sugar in TCTL, with the exception of Equivalences (7) and (8), which hold due to time divergence. Equivalences (11)-(19) describe the reduction of universal quantifiers until formulas, refining time intervals until only the permitted form $\mathrm{A}\ \varphi_1\ U_{I>0}\ \varphi_2$ is used. The unbounded case (11) is a well-known CTL equivalence. Equivalences (12) and (13) follow directly from the following Equivalence [67]:

(E_1) $\quad \mathrm{A}\ \varphi_1\ U_{\leq c}\ \varphi_2 = (\mathrm{A}\ \mathrm{F}_{\leq c}\ \varphi_2) \wedge \neg\mathrm{E}\ (\neg\varphi_2)\ U\ (\neg\varphi_1 \wedge \neg\varphi_2)$

Equivalences (14) and (15), which hold due to the assumption of time-divergence, follow directly from the following Equivalence [67]:

(E_2) $\quad \mathrm{A}\ \varphi_1\ U_{\geq c}\ \varphi_2 = \mathrm{A}\ \mathrm{G}_{<c}\,(\varphi_1\ \wedge \mathrm{A}\ \varphi_1\ \mathrm{U}_{>0}\ \varphi_2) \quad$ if $c > 0$

Equivalences (16) and (17) extend (12) and (13) to a "general interval" time bound. The formula on the right-hand side of Equivalence (16) is a conjunction, where $\neg\mathrm{E}\ \mathrm{F}_{<a}\ \neg\varphi_1$ specifies that φ_1 must hold all the time before bound a is obtained in each behaviour,


Equivalences for reducing a TCTL formula to its normal form.

Equivalence			Validity condition	Application condition
1. *false*	$=$	$\neg true$		
2. $\varphi_1 \vee \varphi_2$	$=$	$\neg(\neg\varphi_1 \wedge \neg\varphi_2)$		
3. $\varphi_1 \Longrightarrow \varphi_2$	$=$	$\neg\varphi_1 \vee \varphi_2$		
4. $\varphi_1 \Longleftrightarrow \varphi_2$	$=$	$(\varphi_1 \Longrightarrow \varphi_2) \wedge (\varphi_2 \Longrightarrow \varphi_1)$		
5. $E\ F_I\ \varphi$	$=$	$E\ true\ U_I\ \varphi$		
6. $A\ F_I\ \varphi$	$=$	$A\ true\ U_I\ \varphi$		
7. $E\ G_{\leq b}\ \varphi$	$=$	$E\ \varphi\ U_{>b}\ true$		
8. $E\ G_{<b}\ \varphi$	$=$	$E\ \varphi\ U_{\geq b}\ true$	*if* $b \neq \infty$	
9. $E\ G_I\ \varphi$	$=$	$\neg A\ F_I\ \neg\varphi$		*if* $I \neq [0,\infty)$
10. $A\ G_I\ \varphi$	$=$	$\neg E\ F_I\ \neg\varphi$		
11. $A\ \varphi_1\ U\ \varphi_2$	$=$	$(\neg E\ G\ \neg\varphi_2) \wedge \neg E\ (\neg\varphi_2)\ U\ (\neg\varphi_1 \wedge \neg\varphi_2)$		
12. $A\ \varphi_1\ U_{\leq b}\ \varphi_2$	$=$	$(\neg E\ G_{\leq b}\ \neg\varphi_2) \wedge \neg E\ (\neg\varphi_2)\ U\ (\neg\varphi_1 \wedge \neg\varphi_2)$		
13. $A\ \varphi_1\ U_{<b}\ \varphi_2$	$=$	$(\neg E\ G_{<b}\ \neg\varphi_2) \wedge \neg E\ (\neg\varphi_2)\ U\ (\neg\varphi_1 \wedge \neg\varphi_2)$		
14. $A\ \varphi_1\ U_{\geq a}\ \varphi_2$	$=$	$A\ G_{<a}\ (\varphi_1 \wedge A\ \varphi_1\ U_{>0}\ \varphi_2)$	*if* $a \neq 0$	
15. $A\ \varphi_1\ U_{>a}\ \varphi_2$	$=$	$A\ G_{\leq a}\ (\varphi_1 \wedge A\ \varphi_1\ U_{>0}\ \varphi_2)$		*if* $a \neq 0$
16. $A\ \varphi_1\ U_{[a,b]}\ \varphi_2$	$=$	$(\neg E\ F_{=a}\ (E\ (\varphi_1 \wedge \neg\varphi_2)\ U\ (\neg\varphi_1 \wedge \neg\varphi_2))) \wedge$ $(\neg E\ F_{=a}\ (E\ \neg\varphi_2\ U_{>b-a}\ true)) \wedge$ $(\neg E\ F_{<a}\ \neg\varphi_1)$	*if* $a \neq 0$	
17. $A\ \varphi_1\ U_{[a,b)}\ \varphi_2$	$=$	$(\neg E\ F_{=a}\ (E\ (\varphi_1 \wedge \neg\varphi_2)\ U\ (\neg\varphi_1 \wedge \neg\varphi_2))) \wedge$ $(\neg E\ F_{=a}\ (E\ \neg\varphi_2\ U_{\geq b-a}\ true)) \wedge$ $(\neg E\ F_{<a}\ \neg\varphi_1)$	*if* $a \neq 0 \wedge b \neq \infty$	
18. $A\ \varphi_1\ U_{(a,b]}\ \varphi_2$	$=$	$(A\ F_{=a}\ (A\ \varphi_1\ U_{(0,b-a]}\ \varphi_2)) \wedge A\ G_{\leq a}\ \varphi_1$	*if* $b \neq \infty$	*if* $a \neq 0$
19. $A\ \varphi_1\ U_{(a,b)}\ \varphi_2$	$=$	$(A\ F_{=a}\ (A\ \varphi_1\ U_{(0,b-a)}\ \varphi_2)) \wedge A\ G_{\leq a}\ \varphi_1$	*if* $b \neq \infty$	*if* $a \neq 0$

Figure 6.3: Equivalences for reducing a TCTL formula to its normal form.

$\neg$E F$_{=a}$ (E $\neg\varphi_2\ U_{>b-a}$ true) specifies that in each behaviour φ_2 must hold somewhere in the time interval $[a,b]$, and $\neg$E F$_{=a}$ (E$\,(\varphi_1 \wedge \neg\varphi_2)\,U\,(\neg\varphi_1 \wedge \neg\varphi_2)$) specifies that in each behaviour, once it is in the interval, φ_1 must hold until a φ_2-state is eventually reached. Equivalences (18) and (19) also extend Equivalences (14) and (15) to a "general interval" time bound. The right-hand side of (18) is the conjunction of two formulas, where A G$_{\leq=a}\ \varphi_1$ specifies that φ_1 must hold all the time before the interval is obtained, while A F$_{=a}$ $\big($A $\varphi_1\ U_{(0,b-a]}\ \varphi_2\big)$ specifies that in any path at time a a state is obtained where (again in any path) φ_1 holds until a φ_2-state is achieved within the time-interval $(0, b-a]$ ("shifting" the interval $(0, b-a]$ of a time units leadss to the interval $(0+a, b-a+a] = (a,b]$). Since each equivalence in normal form maintains validity in the point-wise semantics, it can be proven that:

$$\mathcal{TK}, s \models_p \varphi \;\;\Leftrightarrow\;\; \mathcal{TK}, s \models_p \varphi!.$$

6.5.2 Core Model Checking Procedures

Procedures 1 to 3 are shown in Figure 6.4.

Procedure 1 shows a high-level overview of our TCTL model checking procedure. Consider a timed Kripke structure $\mathcal{TK} = (S, \mathcal{T}, \rightarrow, L)$, a state $s \in S$ and a TCTL formula φ. The model checking procedure MC $(\mathcal{TK}, s, \varphi)$ establishes whether $\mathcal{TK}, s \models_p \varphi$ by recursively calculating the satisfaction set of the normal form $\varphi!$ of φ:

The recursive calculation of the satisfaction set $Sat\,(\mathcal{TK}, \varphi!)$ is performed as Procedure 2. It based on six core procedures, each taking care of a certain normal form modality

Procedure 3 outlines the algorithm $Sat - \mathrm{EU}\,(\mathcal{TK}, \varphi)$ for calculating the satisfaction set of a formula φ of the type $\mathrm{E}\,\varphi_1\,U\,\varphi_2$. The classic explicit model checking algorithm for CTL is utilized [12]. All states satisfying φ_2 also satisfy φ. The set of φ_2-states (i.e., $\mathrm{Q} = \{s \in Sat\,(\varphi_2)\}$) is recursively closed with the set Q_{pre} of all φ_1-states, which are not already in Q and which can reach some state in Q in one step, until a fixed-point is obtained.

Procesudes 4 and 5 are shown in Figure 6.5.

Procedure 4 describes the algorithm $Sat - \mathrm{EG}\,(\mathcal{TK}, \varphi)$ for computing the satisfaction set for a formula φ of the form E G φ_1. The classic explicit model checking algorithm for CTL is applied. $Tarjan\,(\mathrm{Q}, \rightarrow)$ calculates all the states which are related to some non-trivial strongly connected component of φ_1-states $(SCC\,(\varphi_1))$ over the given transition relation, utilizing the well-known Tarjan algorithm for finding strongly connected components. From each state in some $(SCC\,(\varphi_1))$ there exists an infinite behaviour including φ_1-states, since each SCC consists of looping behaviours, therefore φ holds in these states. Then, this initial set of states $\mathrm{Q} = \{s \in Sat\,(\varphi_1)\}$ is recursively closed with the set Q_{pre} of all φ_1-states, which are not already in Q and which can reach some state in Q in one step, until a fixed-point is obtained.

Procedure 5 describes the algorithm $Sat - \mathrm{EU}_b\,(\mathcal{TK}, \varphi)$ for calculating the satisfaction set for a formula φ of the form $\mathrm{E}\,\varphi_1\,U_{\sim b}\,\varphi_2$ with $\sim \in \{\leq, <\}$. The satisfaction set of such a formula is calculated using the same procedure as the one in [67]. The procedure

Procedure 1 MC($\mathcal{TK}, s, \varphi$)

```
Input  : $\mathcal{TK} = (S, \mathcal{T}, \longrightarrow, L)$,
         TCTL formula $\varphi$.
Output : $\mathcal{TK}, s \models_p \varphi$.

Q := Sat($\mathcal{TK}, \varphi!$);          // Recursively compute satisfaction set of normalized formula $\varphi!$
return $s \in$ Q;
```

(i). Sat-EU for formulas $E\ \varphi_1\ U\ \varphi_2$;
(ii). Sat-EG for formulas $E\ G\ \varphi$;
(iii). Sat-EU$_b$ for formulas $E\ \varphi_1\ U_{\sim b}\ \varphi_2$, with $\sim \in \{\leq, <\}$, except the modality covered in (i);
(iv). Sat-EU$_a$ for formulas $E\ \varphi_1\ U_{\sim a}\ \varphi_2$, with $\sim \in \{\geq, >\}$, except the modality covered in (i);
(v). Sat-EU$_{a,b}$ for formulas $E\ \varphi_1\ U_I\ \varphi_2$, except the modalities covered in (i), (iii), and (iv); and
(vi). Sat-AU$_0$ for formulas $A\ \varphi_1\ U_{I_{>0}}\ \varphi_2$.

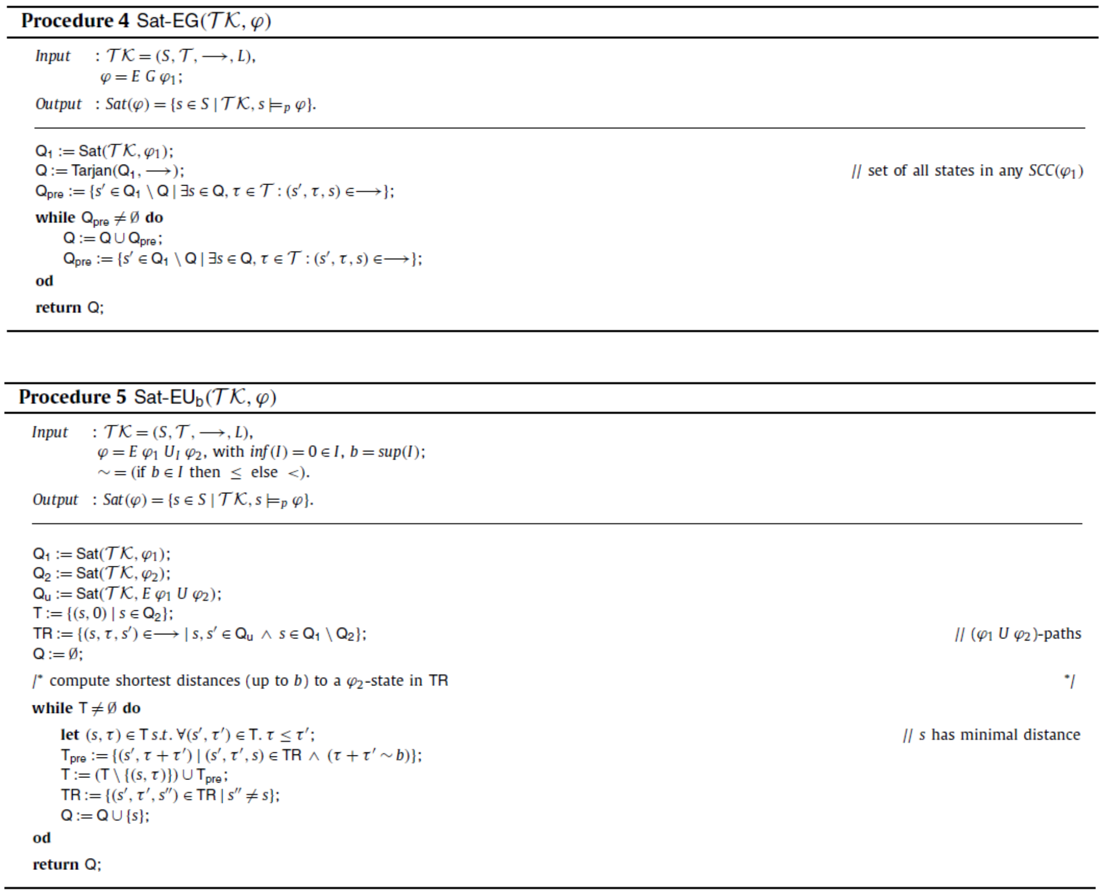
Procedure 2 Sat($\mathcal{TK}, \varphi$)

```
Input  : $\mathcal{TK} = (S, \mathcal{T}, \longrightarrow, L)$,
         TCTL formula $\varphi$ in normal form.
Output : $Sat(\varphi) = \{s \in S \mid \mathcal{TK}, s \models_p \varphi\}$.

                                                  // Recursively evaluate all subformulas of $\varphi$
switch $\varphi$ :
    $true$                          : return $S$;
    $p$                             : return $\{s \mid p \in L(s)\}$;
    $\neg\varphi_1$                 : return $S \setminus$ Sat($\mathcal{TK}, \varphi_1$);
    $\varphi_1 \wedge \varphi_2$    : return Sat($\mathcal{TK}, \varphi_1$) $\cap$ Sat($\mathcal{TK}, \varphi_2$);
    $\varphi_1 \vee \varphi_2$      : return Sat($\mathcal{TK}, \varphi_1$) $\cup$ Sat($\mathcal{TK}, \varphi_2$);
    $E\ \varphi_1\ U\ \varphi_2$    : return Sat-EU($\mathcal{TK}, \varphi$);              // do Procedure 3
    $E\ G\ \varphi_1$               : return Sat-EG($\mathcal{TK}, \varphi$);              // do Procedure 4
    $E\ \varphi_1\ U_{\leq b}\ \varphi_2$ :
    $E\ \varphi_1\ U_{<b}\ \varphi_2$     : return Sat-EU$_b$($\mathcal{TK}, \varphi$);    // do Procedure 5
    $E\ \varphi_1\ U_{\geq a}\ \varphi_2$ :
    $E\ \varphi_1\ U_{>a}\ \varphi_2$     : return Sat-EU$_a$($\mathcal{TK}, \varphi$);    // do Procedure 6
    $E\ \varphi_1\ U_{[a,b]}\ \varphi_2$  :
    $E\ \varphi_1\ U_{(a,b]}\ \varphi_2$  :
    $E\ \varphi_1\ U_{[a,b)}\ \varphi_2$  :
    $E\ \varphi_1\ U_{(a,b)}\ \varphi_2$  : return Sat-EU$_{a,b}$($\mathcal{TK}, \varphi$); // do Procedure 7
    $A\ \varphi_1\ U_{(0,\infty)}\ \varphi_2$ :
    $A\ \varphi_1\ U_{(0,b]}\ \varphi_2$  :
    $A\ \varphi_1\ U_{(0,b)}\ \varphi_2$  : return Sat-AU$_0$($\mathcal{TK}, \varphi$);   // do Procedure 8
endsw
```

Procedure 3 Sat-EU($\mathcal{TK}, \varphi$)

```
Input  : $\mathcal{TK} = (S, \mathcal{T}, \longrightarrow, L)$,
         $\varphi = E\ \varphi_1\ U\ \varphi_2$;
Output : $Sat(\varphi) = \{s \in S \mid \mathcal{TK}, s \models_p \varphi\}$.

Q$_1$ := Sat($\mathcal{TK}, \varphi_1$);
Q := Sat($\mathcal{TK}, \varphi_2$);
Q$_{pre}$ := $\{s' \in$ Q$_1 \setminus$ Q $\mid \exists s \in$ Q$, \tau \in \mathcal{T} : (s', \tau, s) \in \longrightarrow\}$;
while Q$_{pre} \neq \emptyset$ do
    Q := Q $\cup$ Q$_{pre}$;
    Q$_{pre}$ := $\{s' \in$ Q$_1 \setminus$ Q $\mid \exists s \in$ Q$, \tau \in \mathcal{T} : (s', \tau, s) \in \longrightarrow\}$;
od
return Q;
```

Figure 6.4: Core model checking procedures 1 to 3.



Procedure 4 Sat-EG($\mathcal{TK}, \varphi$)

```
Input  : $\mathcal{TK} = (S, \mathcal{T}, \longrightarrow, L)$,
         $\varphi = E\ G\ \varphi_1$;
Output : $Sat(\varphi) = \{s \in S \mid \mathcal{TK}, s \models_p \varphi\}$.

$Q_1 := Sat(\mathcal{TK}, \varphi_1)$;
$Q := Tarjan(Q_1, \longrightarrow)$;                                    // set of all states in any $SCC(\varphi_1)$
$Q_{pre} := \{s' \in Q_1 \setminus Q \mid \exists s \in Q, \tau \in \mathcal{T} : (s', \tau, s) \in \longrightarrow\}$;
while $Q_{pre} \neq \emptyset$ do
    $Q := Q \cup Q_{pre}$;
    $Q_{pre} := \{s' \in Q_1 \setminus Q \mid \exists s \in Q, \tau \in \mathcal{T} : (s', \tau, s) \in \longrightarrow\}$;
od
return Q;
```

Procedure 5 Sat-EU$_b$($\mathcal{TK}, \varphi$)

```
Input  : $\mathcal{TK} = (S, \mathcal{T}, \longrightarrow, L)$,
         $\varphi = E\ \varphi_1\ U_I\ \varphi_2$, with $inf(I) = 0 \in I$, $b = sup(I)$;
         $\sim =$ (if $b \in I$ then $\leq$ else $<$).
Output : $Sat(\varphi) = \{s \in S \mid \mathcal{TK}, s \models_p \varphi\}$.

$Q_1 := Sat(\mathcal{TK}, \varphi_1)$;
$Q_2 := Sat(\mathcal{TK}, \varphi_2)$;
$Q_u := Sat(\mathcal{TK}, E\ \varphi_1\ U\ \varphi_2)$;
$T := \{(s, 0) \mid s \in Q_2\}$;
$TR := \{(s, \tau, s') \in \longrightarrow \mid s, s' \in Q_u \land s \in Q_1 \setminus Q_2\}$;      // $(\varphi_1\ U\ \varphi_2)$-paths
$Q := \emptyset$;
/* compute shortest distances (up to $b$) to a $\varphi_2$-state in TR                  */
while $T \neq \emptyset$ do
    let $(s, \tau) \in T$ s.t. $\forall (s', \tau') \in T.\ \tau \leq \tau'$;              // $s$ has minimal distance
    $T_{pre} := \{(s', \tau + \tau') \mid (s', \tau', s) \in TR \land (\tau + \tau' \sim b)\}$;
    $T := (T \setminus \{(s, \tau)\}) \cup T_{pre}$;
    $TR := \{(s', \tau', s'') \in TR \mid s'' \neq s\}$;
    $Q := Q \cup \{s\}$;
od
return Q;
```

Figure 6.5: Core model checking procedures 4 and 5.

calculates the shortest distances from each φ_1-state to a φ_2-state along a $(\varphi_1\, U\, \varphi_2)$-path. The satisfaction set Q_u of the untimed formula $\mathrm{Q}_u = \mathrm{E}\, \varphi_1\, U\, \varphi_2$ is computed first, and then the shortest path calculation is limited to the graph $\mathrm{G} = (\mathrm{Q}_u, \mathrm{TR})$, where TR is the set of transitions between φ_u-states, excluding outgoing transitions from φ_2-states (because these transitions would not contribute to discovering such shortest paths). The shortest distance from each φ_1-state to a φ_2-state along the paths in TR is calculated using a variant of Dijkstra's shortest path algorithm. This is obtained by initially including in the set T of (state, distance) pairs the φ_2-states in Q_u with distance zero, and then iteratively considering the pair $(s, \tau) \in \mathrm{T}$ with the minimum distance and adding it to T the set of pairs T_{pre} of predecessors of s that can obtain a φ_2-state through a transition in TR within the bound b. The procedure returns the states having shortest distance within the bound, which are stored in Q.

Procedure 6, shown in Figure 6.6 demonstrates the algorithm $Sat - \mathrm{EU}_a(\mathcal{TK}, \varphi)$ for calculating the satisfaction set for a formula φ of the form $\mathrm{E}\, \varphi_1\, U_{\sim a}\, \varphi_2$ with $\sim \in \{\geq, >\}$ and either $\sim$ is equal to $>$ or $a \neq 0$. This procedure is an adaptation of the one in [67]. In general, a state satisfying φ appears on a path of the following type:

$$\begin{tikzcd}
 & \text{(a)} & & & \text{(b)} & \\
\varphi_1 \arrow[r] & \varphi_1 \arrow[r, bend right] & \varphi_1 \arrow[l, bend right] \arrow[r] & \varphi_1 \arrow[r] & \varphi_1 \arrow[r] & \varphi_2
\end{tikzcd}$$

The procedure will calculate the following two sets of φ-states (intuitively identified above by dashed boxes):

1. The set of states that satisfy φ because of a looping sequence of φ_1-states leading to some φ_2-state; having a finite set of states, looping sequences of φ_1-states can be discovered by detecting $SCC(\varphi_1)$. Due to the assumption of time-divergence, any non-trivial strongly connected component in $\mathcal{TK}$ has at least one tick transition with

```
Procedure 6 Sat-EU_a($\mathcal{TK}, \varphi$)

Input   : $\mathcal{TK} = (S, \mathcal{T}, \longrightarrow, L)$,
          $\varphi = E\ \varphi_1\ U_I\ \varphi_2$, with $a = inf(I)$ and $sup(I) = \infty$
          $\sim =$ (if $a \in I$ then $\geq$ else $>$).
Output  : $Sat(\varphi) = \{s \in S \mid \mathcal{TK}, s \models_p \varphi\}$.

$Q_1 := Sat(\mathcal{TK}, \varphi_1)$;
$Q_u := Sat(\mathcal{TK}, E\ \varphi_1\ U\ \varphi_2)$;
$Q_{SCC} := Tarjan(Q_1, \longrightarrow)$;
$Q := Q_u \cap Q_{SCC}$;                                   // $s \in (Q_u \cap Q_{SCC}) \Longrightarrow s \in Sat(\varphi)$
$Q_{pre} := \{s' \in Q_1 \setminus Q \mid \exists s \in Q, \tau \in \mathcal{T} : (s', \tau, s) \in\longrightarrow\}$;
/* recursively compute $Q = Sat(E\ \varphi_1\ U\ (\text{“}SCC(\varphi_1)\text{-state”} \wedge E\ \varphi_1\ U\ \varphi_2))$   */
while $Q_{pre} \neq \emptyset$ do
    $Q := Q \cup Q_{pre}$;
    $Q_{pre} := \{s' \in Q_1 \setminus Q \mid \exists s \in Q, \tau \in \mathcal{T} : (s', \tau, s) \in\longrightarrow\}$;
od

$Q_{DAG} := Q_u \setminus Q$;                              // the states in $Q_u \setminus Q$ induce a DAG in $\mathcal{TK}$
$TR_{DAG} := \{(s, \tau, s') \in\longrightarrow \mid s, s' \in Q_{DAG} \wedge s \in Q_1\}$;   // $(\varphi_1\ U\ \varphi_2)$-paths in $Q_{DAG}$

/* Backward traverse $DAG = (Q_{DAG}, TR_{DAG})$ to find longest paths from $\varphi_1$-states to $\varphi_2$-states;
$T^*$ : {($Q_{DAG}$-state without outgoing edges in $TR_{DAG}$, distance)}, yet to visit;
$T$ : {($Q_{DAG}$-state, current longest distance)}.                                        */
$T^* := \{(s, 0) \mid s \in Q_{DAG} \wedge (\forall (s', \tau, s'') \in TR_{DAG}.\ s' \neq s)\}$;
$T := \emptyset$;
while $T^* \neq \emptyset$ do
    let $(s, \tau) \in T^*$;
    $T^* := T^* \setminus \{(s, \tau)\}$;
    if $\tau \sim a$ then $Q := Q \cup \{s\}$ fi;
    foreach $(s', \tau', s) \in TR_{DAG}$ do
        $TR_{DAG} := TR_{DAG} \setminus \{(s', \tau', s)\}$;
        if $(\exists \tau'' \in \mathcal{T} : (s', \tau'') \in T)$ then
            if $\tau + \tau' > \tau''$ then
                $\tau_{max} := \tau + \tau'$;    $T := (T \setminus \{(s', \tau'')\}) \cup \{(s', \tau_{max})\}$;
            else
                $\tau_{max} := \tau''$;
            fi
        else
            $\tau_{max} := \tau + \tau'$;    $T := T \cup \{(s', \tau_{max})\}$;
        fi
        /* if $s'$ has no more outgoing transitions in $TR_{DAG}$                   */
        if $\{s'' \xrightarrow{\tau''} s''' \in TR_{DAG} \mid s'' = s'\} = \emptyset$ then
            $T^* := T^* \cup \{(s', \tau_{max})\}$;
        fi
    endeach
od
return $Q$;
```

Figure 6.6: Core model checking procedure 6.

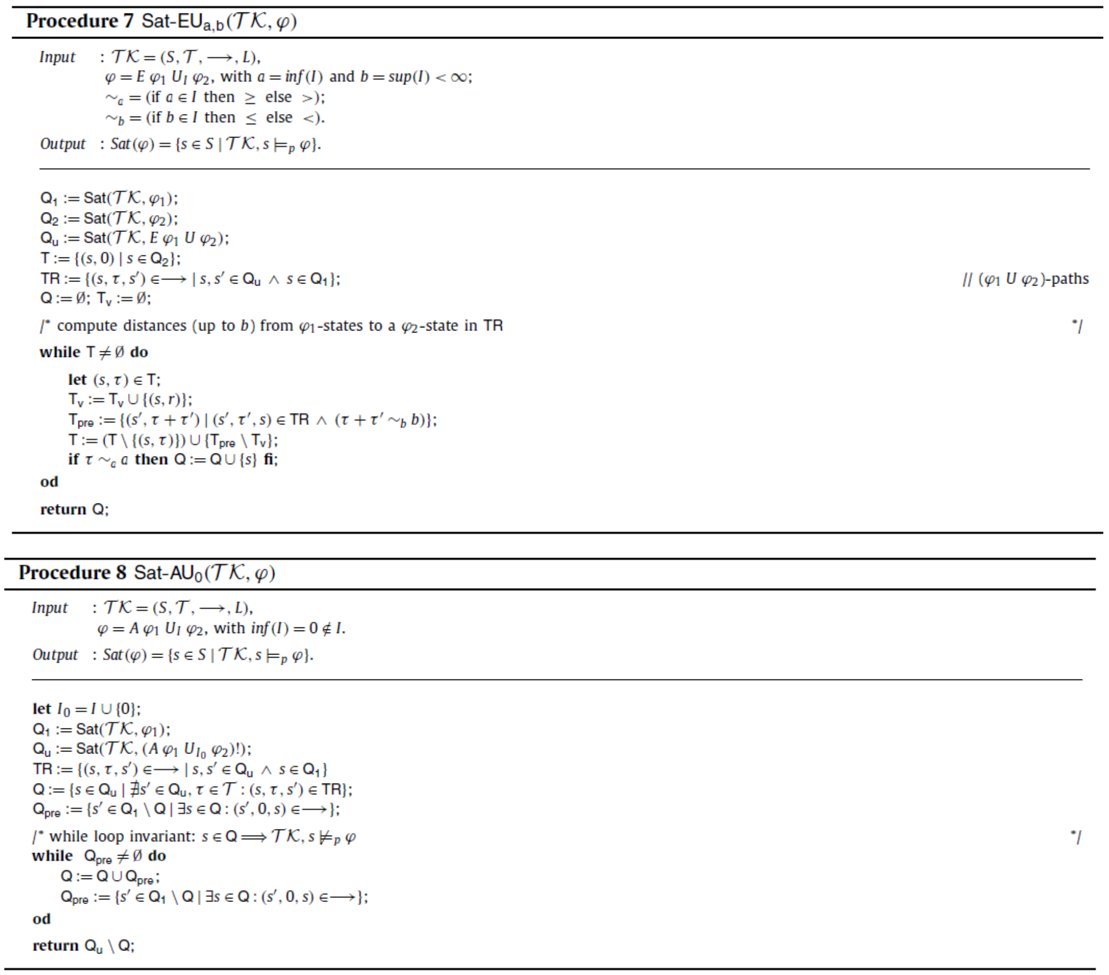
a duration greater than 0.

2. The set of states that satisfy φ because of a simple (non-looping) $(\varphi_1\, U\, \varphi_2)$-path, whose duration is higher than the required time bound.

For calculating the first set of states the sets Q_1 and Q_u are first calculated as in Procedure 5. Then the set Q_{SCC} of states related to some non-trivial strongly connected components of φ_1 states in $\mathcal{TK}$ is determined by calling Tarjan $(\mathrm{Q}_1, \rightarrow)$. Q initially contains states which are both in Q_{SCC} and satisfying $\mathrm{E}\, \varphi_1\, U\, \varphi_2$, and hence satisfy φ. The set Q is recursively closed with the set Q_u of all φ_1-states, which are not already in Q and which can reach some state in Q in one step, until a fixed-point is obtained. For the second set of states we limit ourselves to the graph DAG $(\mathrm{Q}_{DAG}, \mathrm{TR}_{DAG})$, where Q_{DAG} is the set of $(\mathrm{E}\, \varphi_1\, U\, \varphi_2)$-states which do not relate to the first set of states, and TR_{DAG} is the set of outgoing transitions from φ_1-states in Q_{DAG}.

Outgoing transitions from $(\varphi_2\, \wedge\, \neg\varphi_1)$-states are not included in TR_{DAG}, because they are irrelevant to our aim of calculating the maximal durations from each φ_1-state from Q_{DAG} to any φ_2-state along $(\varphi_1\, U\, \varphi_2)$-paths. Having removed all cycles, DAG is a directed acyclic graph, and the longest paths from a φ_1-state to some φ_2 state in it can be calculated by traversing DAG in a backward-reachability fashion, such that the finding of the longest distances follows an inverted topological ordering. During the calculation, T^* includes those states whose longest distance is established i.e., those states which have no outgoing transitions left to visit; T includes the maximal currently known distances for those states which have been detected so far, but still have some outgoing transition left to visit.

The calculation starts by setting in T^* the distance of states without outgoing edges (that are φ_2-states by construction) to zero, while T will be initially empty. The algorithm continues by taking and removing a longest-distance pair (s, τ) in T^*, adding s to Q if τ is greater than the time bound and iteratively backward visiting its incoming edges. For each detected predecessor s' of s, if s' is found for the first time then it is inserted together with



Procedure 7 Sat-EU$_{a,b}(\mathcal{TK}, \varphi)$

```
Input  : $\mathcal{TK} = (S, \mathcal{T}, \longrightarrow, L)$,
         $\varphi = E\ \varphi_1\ U_I\ \varphi_2$, with $a = inf(I)$ and $b = sup(I) < \infty$;
         $\sim_a$ = (if $a \in I$ then $\geq$ else $>$);
         $\sim_b$ = (if $b \in I$ then $\leq$ else $<$).
Output : $Sat(\varphi) = \{s \in S \mid \mathcal{TK}, s \models_p \varphi\}$.

$Q_1 := Sat(\mathcal{TK}, \varphi_1)$;
$Q_2 := Sat(\mathcal{TK}, \varphi_2)$;
$Q_u := Sat(\mathcal{TK}, E\ \varphi_1\ U\ \varphi_2)$;
$T := \{(s, 0) \mid s \in Q_2\}$;
$TR := \{(s, \tau, s') \in \longrightarrow \mid s, s' \in Q_u \wedge s \in Q_1\}$;        // $(\varphi_1\ U\ \varphi_2)$-paths
$Q := \emptyset$; $T_v := \emptyset$;
/* compute distances (up to $b$) from $\varphi_1$-states to a $\varphi_2$-state in TR        */
while $T \neq \emptyset$ do
    let $(s, \tau) \in T$;
    $T_v := T_v \cup \{(s, r)\}$;
    $T_{pre} := \{(s', \tau + \tau') \mid (s', \tau', s) \in TR \wedge (\tau + \tau' \sim_b b)\}$;
    $T := (T \setminus \{(s, \tau)\}) \cup \{T_{pre} \setminus T_v\}$;
    if $\tau \sim_a a$ then $Q := Q \cup \{s\}$ fi;
od
return Q;
```

Procedure 8 Sat-AU$_0(\mathcal{TK}, \varphi)$

```
Input  : $\mathcal{TK} = (S, \mathcal{T}, \longrightarrow, L)$,
         $\varphi = A\ \varphi_1\ U_I\ \varphi_2$, with $inf(I) = 0 \notin I$.
Output : $Sat(\varphi) = \{s \in S \mid \mathcal{TK}, s \models_p \varphi\}$.

let $I_0 = I \cup \{0\}$;
$Q_1 := Sat(\mathcal{TK}, \varphi_1)$;
$Q_u := Sat(\mathcal{TK}, (A\ \varphi_1\ U_{I_0}\ \varphi_2)!)$;
$TR := \{(s, \tau, s') \in \longrightarrow \mid s, s' \in Q_u \wedge s \in Q_1\}$
$Q := \{s \in Q_u \mid \nexists s' \in Q_u, \tau \in \mathcal{T} : (s, \tau, s') \in TR\}$;
$Q_{pre} := \{s' \in Q_1 \setminus Q \mid \exists s \in Q : (s', 0, s) \in \longrightarrow\}$;
/* while loop invariant: $s \in Q \Longrightarrow \mathcal{TK}, s \not\models_p \varphi$        */
while $Q_{pre} \neq \emptyset$ do
    $Q := Q \cup Q_{pre}$;
    $Q_{pre} := \{s' \in Q_1 \setminus Q \mid \exists s \in Q : (s', 0, s) \in \longrightarrow\}$;
od
return $Q_u \setminus Q$;
```

Figure 6.7: Core model checking procedures 7 and 8.

its current distance in T; if s' was already in T then, if needed, its distance is updated to the current one; in any case, if all outgoing edges from s' have been visited, then its longest distance is known and this information is saved in T*. The procedure terminates when T* becomes empty. In the end, Q includes all the states satisfying φ.

Procedures 7 and 8 are shown in Figure 6.7.

Procedure 7 describes the algorithm $Sat - \mathrm{EU}_{a,b}\,(\mathcal{TK}, \varphi)$ for calculating the satisfaction set for a formula φ of the form E $\varphi_1\ U_I\ \varphi_2$ with finite interval bounds. This procedure is the same as Procedure 5, except that instead of recursively calculating the shortest paths from φ_1-states to φ_2-states along $(\varphi_1 \; U \; \varphi_2)$-paths in $\mathcal{TK}$, it calculates all the durations of such paths within the upper time bound. It first calculates the satisfaction set Q_u of the untimed formula $\mathrm{Q}_u = \mathrm{E}\, \varphi_1 \; U \; \varphi_2$, and limits the path duration calculation to the graph $\mathrm{G} = (\mathrm{Q}_u, \mathrm{TR})$, where TR is the set of transitions between Q_u-states, excluding outgoing transitions from $\neg\varphi_1$-states, since these transitions do not relate to any $(\varphi_1 \; U \; \varphi_2)$ - path. Then, it recursively computes distances from each φ_1-state to some φ_2-state along the paths in TR. This is obtained by initially putting in the set T of (state, distance) pairs the φ_2-states in Q_u with distance zero, and then iteratively taking any pair $(s, \tau) \in \mathrm{T}$ and adding to T the set of pairs T_{pre}, of (not already visited) predecessors of s that can reach a φ_2-state through a transition in TR within the upper time bound b. The procedure returns the states having the shortest distance within the interval I, which are stored in Q. Due to the assumption of time-divergence, which guarantees the absence of zero-time loops, the above backward computation of the distances will eventually stop, because the distance will eventually reach the time bound b.

Procedure 8 outlines the algorithm $Sat-\mathrm{AU}_0\,(\mathcal{TK}, \varphi)$ for calculating the satisfaction set for a formula φ of the form A $\varphi_1 \; U_I \; \varphi_2$ with $inf\,(I) = 0 \notin I$. The satisfaction set Q_u of the formula $\varphi_u = \mathrm{A}\; \varphi_1 \; U_{I_0} \; \varphi_2$ where $I_0 = I \cup \{0\}$ is calculated; this formula can then be reduced to its normal form utilizing Equivalences (11), (12), and (13). The remaining computation is limited to the graph $\mathrm{G} = (\mathrm{Q}_u, \mathrm{TR})$, where TR is the set of transitions between Q_u-states, excluding outgoing transitions from $\neg\varphi_1$-states, since these transitions do not relate to any $(\varphi_1 \; U \; \varphi_2)$ - path. Then the set of states which do not satisfy φ is calculated by recursively closing the initial set of states

$$\mathrm{Q} = \{s \in Sat\,(\mathrm{Q}_u) \;\mid \text{s has no outgoing transitions to } Q_u\text{-states}\}$$

With those states that can obtain a state in Q in zero-time. Due to the time-divergent assumption, which excludes the presence of zero-time loops in G, this can be performed by a backward closure over the zero-time transitions that reach a state in Q. The satisfaction set of φ is hence given by excluding from the set of states satisfying φ_u the states in Q.

6.5.3 Correctness of the Model Checking Procedures

Let τ_{min} (τ_{max}) be the smallest (largest) non-zero transition duration notation in $\mathcal{TK}$, $|\varphi|$ the number of sub-formulas in the TCTL formula φ, τ_φ the largest finite time constant appearing in the interval bounds of φ, and $\bar{\tau} = max\,(\tau_{max}, \tau_\varphi)$. The procedures required to add two time values together or check that a time value is less than another time value, etc. Because the time domain is not fixed, the complexity of the model checking procedure relies on the complexity of these operations on the time values. In the following complexity results it is assumed that the time values are natural numbers with the standard operations.

Proposition 12. *Given a finite Zeno-free timed Kripke structure $\mathcal{TK} = (S, \mathcal{T}, \rightarrow, L)$ and a TCTL formula φ in normal form, $Sat\,(\mathcal{TK}, \varphi)$ terminates and has time complexity $\mathcal{O}\left(|\varphi|\ .\ log\,(\bar{\tau})\ .\ \left(|S|^2 + |S|\ .\ |{\rightarrow}|\ .\ \bar{K}\right)\right)$,where $\bar{K} = \lfloor \tau_\varphi / \tau_{min} \rfloor + 1$.*

The normalization of the formula φ may increase its size. In the worst case, it can be shown that $|\varphi!|$ is $\mathcal{O}\left(5^{d(\varphi)}\ .\ |\varphi|\right)$, where $d\,(\varphi)$ is the maximal depth of nested until modalities in φ. This follows from the fact that MC $(\mathcal{TK}, s, \varphi)$ includes a call to the procedure $Sat\,(\mathcal{TK}, \varphi!)$.

Proposition 13. *Given a finite Zeno-free timed Kripke structure $\mathcal{TK} = (S, \mathcal{T}, \rightarrow, L)$ and a TCTL formula φ, $MC\,(\mathcal{TK}, s, \varphi)$ terminates and has time complexity*
$\mathcal{O}\left(5^{d(\varphi)}\ .\ |\varphi|\ \ .\ log\,(\bar{\tau})\ .\ \left(|S|^2 + |S|\ .\ |{\rightarrow}|\ .\ \bar{K}\right)\right)$.

$Sat\,(\mathcal{TK}, \varphi)$ calculates the satisfaction set of the given (normalized) TCTL formula in the point-wise semantics. It follows that the procedure MC $(\mathcal{TK}, s, \varphi)$ returns $\mathcal{TK}, s \models_p \varphi$.

Proposition 14. *Given a finite Zeno-free timed Kripke structure $\mathcal{TK} = (S, \mathcal{T}, \rightarrow, L)$ and a TCTL formula φ in normal form,*

$$Sat\,(\varphi) = \{s \in S \mid \mathcal{TK}, s \models_p \varphi\} = Sat\,(\mathcal{TK}, \varphi)$$

Proposition 15. *Given a finite Zeno-free timed Kripke structure $\mathcal{TK} = (S, \mathcal{T}, \rightarrow, L)$, a state $s \in S$, and a TCTL formula φ, on termination of $MC(\mathcal{TK}, s, \varphi)$:*

$$\mathcal{TK}, s \models_p \varphi \;\;\Leftrightarrow\;\; MC(\mathcal{TK}, s, \varphi)$$

Point-wise model checking can be utilized to model check the (normalized) formula φ in $\mathcal{TK}$ in the continuous semantics by model checking the modified formula $\beta\,(\alpha\,(\varphi))$ in the gcd-transformation $\mathcal{TK}_a^{\gamma_h}$ of $\mathcal{TK}$ in the point-wise semantics.

In practice, the modification $\beta\,(\alpha\,(\varphi))$ of φ does not rise the size of the formula. In particular, $\alpha\,(\mathrm{E}\,\varphi_1\,U_I\,\varphi_2) = (\neg p_a \wedge \mathrm{E}\,\alpha\,(\varphi_1)\,U_{I1}\;\;\alpha\,(\varphi_2)\,) \vee\;(p_a \wedge \mathrm{E}\,\alpha\,(\varphi_1)\;\,U_{I2}\,\alpha\,(\varphi_2))$ and, for each state, depending on whether the state is a p_a -state or not, only one of the two until formulas $\mathrm{E}\,\alpha\,(\varphi_1)\,U_{I1}\;\;\alpha\,(\varphi_2)$ and $\mathrm{E}\,\alpha\,(\varphi_1)\,U_{I2}\;\;\alpha\,(\varphi_2)$ needs to be calculated. A similar reasoning can be performed in the transformation $\beta\,(\alpha)$ for formulas of the type A $\varphi_1\,U_I\,\varphi_2$. In particular, $\beta\,(\varphi_2)\wedge(\neg p_a \vee \beta\,(\varphi_1)) = (\neg p_a \wedge \beta\,(\varphi_2))\vee(p_a \wedge \beta\,(\varphi_1) \wedge \beta\,(\varphi_2))$ and depending on whether the state is a p_a-state or not, only one of the two formulas $\beta\,(\varphi_1)$ and $\beta\,(\varphi_1) \wedge \beta\,(\varphi_2)$ needs to be calculated. Computing the conjunctions or disjunctions introduced in the transformed formula increases the computation time only by a constant value. Checking whether a state is a p_a-state or not also increases the time complexity by a constant value. The slightly changed time bounds in the modification $\beta\,(\alpha\,(\varphi))$ are irrelevant with respect to the time complexity. In particular, the modified formula $\beta\,(\alpha)$ has the same time bounds as φ, and for the modified formula $\alpha\,(\varphi)$ we may have $\tau_{\alpha(\varphi)} = (K_\varphi - 1)\;\;.\;\gamma_h$, for an integer K_φ such that $\tau_\varphi = K_\varphi\,\gamma_h$. Therefore, utilizing the point-wise model checker for model checking a TCTL formula φ in the continuous semantics (which corresponds to the procedure call $\mathrm{MC}\,(\mathcal{TK}_a^\tau, s, \beta\,(\alpha\,(\varphi)))$ has the same time complexity as the procedure call $\mathrm{MC}\,(\mathcal{TK}_a^\tau, s, \varphi)$ [72].

Proposition 16. *Let AP be a set of atomic propositions and $\mathcal{TK}$ a finite Zeno-free timed Kripke structure over AP whose time domain satisfies the theory $TIME^{gcd}$.*

Let φ be a TCTL formula over AP, $0 < \tau \in \mathcal{T}$ a time value such that $GCD\,(\mathcal{TK}, \varphi)$ is a multiple of 2τ and $\mathcal{TK}_a^\tau$ the abstract τ -transformation of $\mathcal{TK}$. Then for each state s of $\mathcal{TK}$ it holds that $MC(\mathcal{TK}_a^\tau, s, \varphi)$ has time complexity $\mathcal{O}\left(5^{d(\varphi)} \,.\, |\varphi| \,.\, log\,(\bar{\tau}) \,.\, \left((|S| \,.\, k_{max})^2 + |S| \,.\, |{\rightarrow}| \,.\, k_{max} \,.\, (k_\varphi + 1)\right)\right)$ where k_φ is an integer value such that the greatest finite time bound in φ is equal to $k_\varphi \,.\, \tau$, and k_{max} is an integer value such that the largest non-zero transition duration in $\rightarrow$ is equal to $k_{max} \,.\, \tau$.

Chapter 7

On the Equivalence of Timed Automata and Timed Kripke Structures in Dense Time

We have shown throughout this thesis that verification of real-time systems has proven feasible for both Timed Automata and Timed Kripke Structures in the dense time domain. They accomplish the same target which is verification and in the respect of this purpose they are equivalent. Now we wonder how can we establish and actual algorithmic equivalence between these two models for example, as a conversion between timed automata (TA) and timed Kripke structures (TK) in the dense time domain. This has been answered positively in the untimed domain, where different constructive equivalence relations under CTL between labeled transition system (LTS) and Kripke structures in the untimed domain have been developed. The equivalences are inductive and algorithmic.

In this chapter we investigate the possibility of expanding the inductive conversion methods such as the two equivalence methods introduced in [26] to the dense time domain. We identify the issues which make inductive conversions methods difficult and impractical in dense time domain for concurrent and large scale real time systems.

7.1 Constructing a Kripke Structure Equivalent with a Given LTS

An LTS [25] is a tuple $M = (S, A, \rightarrow, s_0)$ where S is a countable, non empty set of states, $s_0 \in S$ is the initial state, and A is a countable set of actions. The actions in A are called visible (or observable), by contrast with the special, unobservable action $\tau \notin A$ (also called internal action). The relation $\rightarrow \subseteq S \times (A \cup \{\tau\}) \times S$ is the transition relation; we use $p \overset{a}{\rightarrow} q$ instead of $(p, a, q) \in \rightarrow$. A transition $p \overset{a}{\rightarrow} q$ means that state p becomes state q after performing the (visible or internal) action a.

A path (or run) π starting from state p' is a sequence $p' = p_0 \overset{a_1}{\rightarrow} p_1 \overset{a_2}{\rightarrow} ...p_{k-1} \overset{a_k}{\rightarrow} p_k$ with $k \in \mathbb{N} \cup \{\omega\}$ such that $p_{i-1} \overset{a_i}{\rightarrow} p_i$ for all $0 < i \leq k$. We use $|\pi|$ to refer to k, the length of π. If $|\pi| \in \mathbb{N}$, then we say that π is finite. The trace of π is the sequence $\text{trace}\,(\pi) = (a_i)_{0<i\leq|\pi|, a_i\neq\tau} \in A^*$ of all the visible actions that occur in the run listed in their order of occurrence and including duplicates. Note in particular that internal actions do not appear in traces. The set of finite traces of a process p is defined as $\text{Fin}\,(p) = \{tr \in \text{traces}\,(p) : |tr| \in \mathbb{N}\}$. If we are not interested in the intermediate states of a run then we use the notation $p \overset{w}{\Rightarrow} q$ to state that there exists a run π starting from state p and ending at state q such that $\text{trace}\,(\pi) = w$. We also use $p \overset{w}{\Rightarrow}$ instead of $\exists p' : p \overset{w}{\Rightarrow} p'$.

A process p that has no outgoing internal action cannot make any progress unless it performs a visible action. We say that such a process is stable [96]. We write $p \downarrow$ whenever we want to say that process p is stable. Formally, $p \downarrow = \neg \left(\exists p' \neq p : p \overset{\epsilon}{\Rightarrow} p'\right)$. A stable process p responds predictably to any set of actions $X \subseteq A$, in the sense that its response depends exclusively on its outgoing transitions. Whenever there is no action $a \in X$ such that $p \overset{a}{\rightarrow}$ we say that p refuses the set X. Only stable processes are able to refuse actions; unstable processes refuse actions "by proxy": they refuse a set X whenever they can internally become a stable process that refuses X. Formally, p refuses X (written p ref X) if and only if $\forall a \in X : \neg \left(\exists p' : \left(p \overset{\epsilon}{\Rightarrow} p'\right) \wedge p' \downarrow \wedge p' \overset{a}{\rightarrow}\right)$.

Method 1: Function $\mathbb{K}$ Converts an LTS into an Equivakent Kripke Structure

Definition 49. Equivalence between Kripke Structurs and LTS: Given a Kripke structure K and a set of states Q of K, the pair K, Q is equivalent to a process p, written $K, Q \simeq p$ (or $p \simeq K, Q$), if and only if for any CTL^* formula $K, Q \models f$ if and only if $p \models f$.

Theorem 1. *There exists an algorithmic function $\mathbb{K}$ which converts a labeled transition system p into a Kripke structure K and a set of states Q such that $p \simeq (K, Q)$. Specifically, for any labeled transition system $p = (S, A, \rightarrow, S_0)$, its equivalent Kripke structure $k = \mathbb{K}(p)$ is defined as $k = (S', Q, R,' L')$ where*

1. $S' = \{\langle s, x\rangle : s \in S,\ x \subseteq init\,(s)\}$
2. $Q = \{\langle s_0, x\rangle \in S'\}$
3. *R' contains exactly all the transitions $(\langle s, N\rangle, \langle t, O\rangle)$ such that $\langle s, N\rangle, \langle t, O\rangle \in S'$, and*
   (a) *for any $n \in N, s \overset{n}{\Rightarrow} t$*
   (b) *for some $q \in S$ and for any $o \in O, t \overset{o}{\Rightarrow} q$, , and item if $N = \varnothing$ then $O = \varnothing$ and $t = s$ (these loops ensure that the relation R' is complete).*
4. $L' : S' \rightarrow 2^{AP}$ *such that* $L'(s, x) = x$ *where* $AP = A$

By utilizing function $\mathbb{K}$ conversion method, the semantics of CTL^* formulae with respect to a process rather than Kripke structure can be definedd. The resulting Kripke structure is very compact, but a new satisfaction operator for sets of Kripke states is needed [26] (since one state of a process can generate multiple initial Kripke states).

Method 2: Function $\mathbb{X}$ Converts an LTS into an Equivalent Kripke Structure
The need of a supplementary satisfaction operator can be eliminated using a different conversion function [44], at the expense of a considerably larger Kripke structure.

Theorem 2. *There exists an algorithmic function $\mathbb{X}$ which converts a labeled transition system into an equivalent Kripke structure. The function $\mathbb{X}$ is defined as follows: with $\triangle$ a fresh symbol not in A, given an LTS $p = (S, A, \rightarrow, s_0)$ the Kripke structure $\mathbb{X}(p) = (S', Q, R', L)$ is given by:*

1. *$AP = A \uplus \triangle$;*
2. *$S' \cup \left\{(r, a, s) : a \in A \text{ and } r \overset{a}{\rightarrow} s\right\}$;*
3. *$Q = \{s_0\}$;*
4. *$R' = \left\{(r, s) : r \overset{\tau}{\rightarrow} s\right\} \cup \left\{(r, (r, a, s)) : r \overset{a}{\rightarrow} s\right\} \cup \left\{((r, a, s), s) : r \overset{a}{\rightarrow} s\right\}$;*
5. *For $r, s \in S$ and $a \in A : L(s) = \{\triangle\}$ and $L((r, a, s)) = \{a\}$.*

Then $p \simeq \mathbb{X}(p)$.

In the resulting Kripkle structure, instead of combining each state with its corresponding actions in the LTS (and thus possibly splitting the LTS state into multiple Kripke structure states), the new symbol $\triangle$ is used to stand for the original LTS states. Every $\triangle$ state of the Kripke structure is the LTS state, and all the other states in the Kripke structure are the actions in the LTS. This ensures that all states in the Kripke structure corresponding to actions that are outgoing from a single LTS state have all the same parent. This in turn eliminates the need for the weaker satisfaction operator over sets of states. However, a relatively straightforward modification to the CTL satisfaction operator is needed (to "jump over" $\triangle$ states).

7.2 Issues in Constructing Equivalence Relations between Timed Automata and Timed Kripke Structures in Dense Time

We now investigate the possibility of expanding inductive conversion methods such as the two conversion techniques in the untimed domain which are introduced in Section 7.1 to the dense time domain. We find out that developing such inductive conversions methods between timed automata and timed Kripke structures is difficult and impractical in the dense time domain for concurrent and large-scale real-time systems.

One of the main reasons which make the conversion of two systems impractical is the usage of the converted systems.

In the untimed domain the conversion is utilized in the following framework: The LTS semantics is used in model-based testing, where a test runs in parallel with the system under test and synchronizes with it over visible actions. A run of a test t and a process p represents a possible sequence of states and actions of t and p running synchronously. The outcome of such a run is either success ($\top$) or failure ($\bot$), and the outcome of applying tests on processes establishes the failure trace testing preorder over processes. On the other hand process equivalence can be determined with respect to their refusals, as follows: To describe the behaviour of a process we need to record each refusal together with the trace that causes that refusal. An observation of a refusal plus the trace that causes it is called a stable failure. The stable failure preorder is then defined based on the relation between traces and stable failures of the two processes. The stable failure preorder (which is based on the behaviour of processes) can be readily converted into a testing-based preorder (based on the outcomes of tests applied to processes).

While there are several other testing scenarios for the untimed domain, we focus on failure trace testing and stable failures because the conversion functions mentioned earlier are developed based on stable failures. Indeed, the conversion functions preserve CTL properties which as a consequence were found equivalent to failure trace testing [26].

In the dense time domain however, it turns out that traces and by extension stable failures are not useful. Two states are timed trace equivalent iff they generate the same timed words i.e., sequences of input symbols and time increments. Timed trace equivalence is strictly weaker than timed bisimilarity but incomparable to region equivalence and incomparable to untimed bisimilarity. While timed trace equivalence is a congruence, it is computationally intractable. The undecidability proof for trace equivalence follows form the proof that the language inclusion problem for timed automata over infinite words is undecidable.

Proposition 17. *The problem of deciding if two initial states of a timed automaton are timed trace equivalent is undecidable [62].*

Proposition 18. *The problem of deciding if two initial states of a timed automaton are timed trace congruent is undecidable [62].*

Proposition 19. *Language (or trace) equivalence and inclusion checking are undecidable in timed automata [7].*

Proposition 20. *Universality of timed automata is undecidable [8].*

According to these results, timed trace equivalence, which would be the basis of stable failure preorder, is undecidable on timed automata. Thus other equivalence relation (usually some form of bisimilarity) are used in the dense time domain.

The problem of automated verification for timed concurrent programs is understood as the problem of checking that a finite-state graph (model) related to the program satisfies a given property formula. The complexity of checking a formula in the model is linear in the size of the model, therefore, the principal barrier in the model generation is related to its size, which can be prohibitive. Knowing in advance the properties, we can affect the process of producing a model in such a way that the model we achieve is a minimal one preserving these properties. Rather than producing minimal models for specific formulas, minimal models preserving whole sub-classes of formulas (sub-languages) are produced. This is performed by recognizing the equivalence preserved by a selected sub-language of the logic and then producing a minimal model preserving this equivalence. We described the methods used to produce minimal models preserving bisimulation (so CTL* and TCTL properties). Time abstract bisimulation in particular preserves linear and branching time properties [19] [5] [6].

Another reason that makes the inductive conversion methods impractical in dense time, especially for large-scale concurrent real-time systems is the infinite state space that comes with the dense time domain. Timed automata are finite-state automata equipped with clocks utilized to specify constraints on the amount of time that can elapse between two events. The particularity of this model is that it utilizes a dense time domain which leads to an infinite state space. We therefore need to apply minimization techniques such as time-abstract bisimulation in order to obtain a finite state space. The time-abstracting bisimulation equivalences converts the state space of a given timed automaton into a finite graph (the quotient graph) which preserves sufficient information for verification. This finite object is close to a Timed Kripke Structures which also has a finite representation. The nodes of the finite quotient graph are classes (in fact, zones) each consisting of a set of S of symbolic states and the edges represent either discrete state transitions or the passage of an arbitrary amount of time. Symbolic states are represented as simple polyhedra and ζ is a conjunction of atomic constraints on $\mathcal{X}$ (a finite set of clocks) defining a convex $\mathcal{X}$ - polyhedron called the guard.

In other words, we cannot consider the original time automata for conversion (which would produce an infinite Kripke structure), but we need to consider the quotient graph instead. The nodes of the quotient graph are zones, each consisting of a set of S of symbolic states which are represented as simple polyhedra, and a conjunction of atomic constraint guards which are represented as convex $\mathcal{X}$ - polyhedra. It follows that they cannot be converted to one state and one delay label, which would be needed in inductive conversion methods. Indeed, should we convert all zones (set of states) and all $\mathcal{X}$ - polyhedron guards to individual labelled states as the ones in timed Kripke structures we will face a huge state explosion problem. We therefore conclude that inductive conversions methods do not work for dense time domain in concurrent real-time systems.

State explosion is a famous problem that impedes analysis and testing relies on state-space exploration. This problem is particularly serious in real-time systems since unbounded time values cause the state space to be infinite. The major weakness of any approach that ignores this issue is that the size of the state space grows exponentially with the number of processes. Kripke structures represent the state space of the system under investigation, and thus their size is exponential in the size of the system description. Therefore, even for systems of relatively modest size, it is often impossible to compute their Kripke structures [35].

The conversion methods developed for the untimed domain are inductive and so they suffer themselves to a certain degree from a state explosion problem since one state in the original LTS can result in multiple equivalent states in the resulting Kripke structure. If similar inductive conversion methods are applied in dense time domain, then all zones and guards from the quotient graph will generate individual states as nodes and delays, with an unmanageable state explosion situation.

This being said, it should be noted that many techniques have been proposed to overcome state explosion. The main concept shared by all these techniques is the concept of state equivalence, which allows the analysis of a system model utilizing another, equivalent and more compact system model. The notion of equivalence is very useful for reducing the state explosion when we focus on special behavior or properties of systems. In this context, bisimulation-based equivalence relations are more effective for state-space reduction in synchronous processes in comparison with failure-based equivalence relations, because all the minimization algorithms developed based on failure-based equivalences increase the number of states. That is, failure-based equivalences enlarge the number of states compared to bisimulation-based equivalences, which in turn causes higher execution time for the verification algorithms [63].

All failure-based equivalences method such as Stable failure equivalence, CFFD equivalence, CSP-failure equivalence, and IOT-failure equivalence, are defined in terms of trace semantics. In fact we argue that any stable failure model must be based on trace semantics. Indeed, any stable failure must feature a trace component. Thus any minimization algorithm in a failure-based model must be at least PSPACE-complete. Add to this the undecidability of many trace problems mentioned earlier, and we can conclude that a failure-based approach is not useful for the dense time domain.

Chapter 8

Conclusions

In this thesis, we summarized that the Strong Time-Abstract Bisimulation method and its properties which enable the infinite state-space of a given timed automata model to be reduced to a finite quotient graph and finite transition system. We also show that time-abstracted equivalence is decidable. Strong time-abstract bisimulation refines the dense state-space while preserving both linear and branching time properties of the original model sufficiently for verification. After the conversion one can utilize untimed verification tools in order to verify real-time systems.

A procedure to check whether a system is free from timelocks and deadlocks was then described. By extending the timed automata model systems to strongly non-zeno timed automata the deadlocks and timelocks can be detected using the finite quotient graph. The quotient graph can also be utilized to reduce model checking of TA to the untimed case so that classical finite-state system verification methods such as CTL model checking can be applied. In particular, TCTL model checking can be reduced to CTL model-checking using strong time-abstracting bisimulation.

On the other hand, we summarized how the satisfaction of TCTL formulas under the natural continuous semantics for both discrete-time and dense-time timed Kripke structures can be reduced to a model-checking problem in the point-wise semantics for timed Kripke structures. Discrete TCTL-preserving abstraction methods on timed Kripke structures, the so-called gcd-transformation and τ-transformation are introduced.

We show that the gcd-transformation and τ-transformation form a sound and complete abstraction for timed Kripke structures with respect to the model checking of TCTL formulas whose time intervals are closed time intervals. Model checking the abstraction in the point-wise semantics is equivalent to model checking the original timed Kripke structure in the continuous semantics. Definitions and abstract methods are developed that achieves soundness and completeness for the whole TCTL logic (including open time intervals in the temporal operators).

We then intended to put everything together by developing methods for converting timed automata to equivalent time Kripke structures, using an approach similar to the one that exists in the untimed domain. We found however that such inductive conversion methods are infeasible in large-scale concurrent and real-time systems in the dense time domain. One reason is that trace equivalence is undecidable for timed automata. The other reason is that the nodes of the finite quotient graph of an infinite timed automaton system are labelled with sets of states as well as constraints over clocks defining a convex $\mathcal{X}$ - polyhedron. Converting these labels into atomic propositions and delays (like in timed Kripke structures) will result in an intractable state explosion problem.

In all, we show that the establishment of an equivalence relation between timed automata and timed Kripke structures based on inductive conversion methods is infeasible in the dense time domain. We should also note in passing that the timed kripke structure system has limited expressive power in comparison with timed automata in a dense time domain.

Bibliography

[1] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba, *Reactive systems: modelling, specification and verification*, cambridge university press, 2007.

[2] R. Alur, *Techniques for automatic verification of real-time systems*, PhD thesis, stanford university, 1991.

[3] R. Alur, C. Courcoubetis, and D. Dill, *Model-checking for real-time systems*, in [1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science, IEEE, 1990, pp. 414–425.

[4] ——, *Model-checking in dense real-time*, Information and computation, 104 (1993), pp. 2–34.

[5] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi, *An implementation of three algorithms for timing verification based on automata emptiness*, in [1992] Proceedings Real-Time Systems Symposium, IEEE, 1992, pp. 157–166.

[6] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi, *Minimization of timed transition systems*, in International Conference on Concurrency Theory, Springer, 1992, pp. 340–354.

[7] R. Alur and D. Dill, *Automata for modeling real-time systems*, in International Colloquium on Automata, Languages, and Programming, Springer, 1990, pp. 322–335.

[8] R. Alur and D. L. Dill, *A theory of timed automata*, Theoretical computer science, 126 (1994), pp. 183–235.

[9] R. Alur and T. A. Henzinger, *Logics and models of real time: A survey*, in Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems), Springer, 1991, pp. 74–106.

[10] ——, *A really temporal logic*, Journal of the ACM (JACM), 41 (1994), pp. 181–203.

[11] J. C. Baeten, J. A. Bergstra, and J. W. Klop, *Ready-trace semantics for concrete process algebra with the priority operator*, The Computer Journal, 30 (1987), pp. 498–506.

[12] C. Baier and J.-P. Katoen, *Principles of model checking*, MIT press, 2008.

[13] T. Basten and M. Voorhoeve, *An algebraic semantics for hierarchical p/t nets*, in International Conference on Application and Theory of Petri Nets, Springer, 1995, pp. 45–65.

[14] M. Ben-Ari, A. Pnueli, and Z. Manna, *The temporal logic of branching time*, Acta informatica, 20 (1983), pp. 207–226.

[15] J. A. Bergstra and J. W. Klop, *Process algebra for synchronous communication*, Information and control, 60 (1984), pp. 109–137.

[16] J. A. Bergstra, A. Ponse, and S. A. Smolka, *Handbook of process algebra*, Elsevier, 2001.

[17] D. Beyer, C. Lewerentz, and A. Noack, *Rabbit: A tool for bdd-based verification of real-time systems*, in International Conference on Computer Aided Verification, Springer, 2003, pp. 122–125.

[18] B. Bloom, S. Istrail, and A. R. Meyer, *Bisimulation can't be traced*, Journal of the ACM (JACM), 42 (1995), pp. 232–268.

[19] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel, *Minimal state graph generation*, Science of Computer Programming, 18 (1992), pp. 247–269.

[20] P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, J. Ouaknine, and J. Worrell, *Model checking real-time systems*, in Handbook of Model Checking, Springer, 2018, pp. 1001–1046.

[21] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, *Kronos: A model-checking tool for real-time systems*, in International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Springer, 1998, pp. 298–302.

[22] M. Browne and E. Clarke, *Sml: A high level language for the design and verification of finite state machines*, in IFIP WG, vol. 10, pp. 1035–1044.

[23] M. Browne, E. Clarke, and D. Dill, *Checking the correctness of sequential circuits*, in 1985 IEEE Proceedings of the International Conference on Computer Design, 1985, pp. 545–548.

[24] M. C. Browne, E. M. Clarke, and D. L. Dill, *Automatic circuit verification using temporal logic: Two new examples*, Formal Aspects of VLSI Design, (1986), pp. 113–124.

[25] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, *Model-based testing of reactive systems*, in Volume 3472 of Springer LNCS, Springer, 2005.

[26] S. D. Bruda, S. Singh, A. Uddin, Z. Zhang, and R. Zuo, *A constructive equivalence between computation tree logic and failure trace testing*, arXiv preprint arXiv:1901.10925, (2019).

[27] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, *Formal hardware verification by symbolic ternary trajectory evaluation*, (1975).

[28] J. R. Büchi, *On a decision method in restricted second order arithmetic*, in The Collected Works of J. Richard Büchi, Springer, 1990, pp. 425–435.

[29] S. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi, *Computing quantitative characteristics of finite-state real-time systems*, in RTSS, 1994, pp. 266–270.

[30] S. Campos, M. Teixeira, M. Minea, A. Kuehlmann, and E. Clarke, *Model checking semi-continuous time models using bdds*, Electronic Notes in Theoretical Computer Science, 23 (2001), pp. 75–87.

[31] S. V. Campos and E. M. Clarke, *Real-time symbolic model checking for discrete time models*, tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1994.

[32] K. Čerāns, *Decidability of bisimulation equivalences for parallel timer processes*, in International Conference on Computer Aided Verification, Springer, 1992, pp. 302–315.

[33] Y. Chen, W.-T. Tsai, and D. Chao, *Dependency analysis-a petri-net-based technique for synthesizing large concurrent systems*, IEEE Transactions on parallel and distributed systems, 4 (1993), pp. 414–426.

[34] S. C. Cheung and J. Kramer, *Context constraints for compositional reachability analysis*, ACM Transactions on Software Engineering and Methodology (TOSEM), 5 (1996), pp. 334–377.

[35] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, *Progress on the state explosion problem in model checking*, in Informatics, Springer, 2001, pp. 176–194.

[36] E. M. Clarke and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, in Workshop on Logic of Programs, Springer, 1981, pp. 52–71.

[37] E. M. Clarke, E. A. Emerson, and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems (TOPLAS), 8 (1986), pp. 244–263.

[38] E. M. Clarke, O. Grumberg, and D. E. Long, *Model checking and abstraction*, ACM transactions on Programming Languages and Systems (TOPLAS), 16 (1994), pp. 1512–1542.

[39] P. Darondeau, *An enlarged definition and complete axiomatization of observational congruence of finite processes*, in International Symposium on Programming, Springer, 1982, pp. 47–62.

[40] J. W. de Bakker and J. I. Zucker, *Processes and the denotational semantics of concurrency*, Information and Control, 54 (1982), pp. 70–120.

[41] F. S. de Boer, J. N. Kok, C. Palamidessi, and J. J. Rutten, *On blocks: locality and asynchronous communication*, in Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems), Springer, 1992, pp. 73–90.

[42] R. De Nicola, *Extensional equivalences for transition systems*, Acta Informatica, 24 (1987), pp. 211–237.

[43] R. De Nicola and M. C. Hennessy, *Testing equivalences for processes*, Theoretical computer science, 34 (1984), pp. 83–133.

[44] R. De Nicola and F. Vaandrager, *Action versus state based logics for transition systems*, in LITP Spring School on Theoretical Computer Science, Springer, 1990, pp. 407–419.

[45] D. L. Dill, *Timing assumptions and verification of finite-state concurrent systems*, in International Conference on Computer Aided Verification, Springer, 1989, pp. 197–212.

[46] D. L. Dill and E. M. Clarke, *Automatic verification of asynchronous circuits using temporal logic*, IEE Proceedings E (Computers and Digital Techniques), 133 (1986), pp. 276–282.

[47] E. A. Emerson and E. M. Clarke, *Characterizing correctness properties of parallel programs using fixpoints*, in International Colloquium on Automata, Languages, and Programming, Springer, 1980, pp. 169–181.

[48] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan, *Quantitative temporal reasoning*, in International Conference on Computer Aided Verification, Springer, 1990, pp. 136–145.

[49] C. J. Fidge, I. J. Hayes, A. P. Martin, and A. Wabenhorst, *A set-theoretic model for real-time specification and reasoning*, in International Conference on Mathematics of Program Construction, Springer, 1998, pp. 188–206.

[50] R. W. Floyd, *Assigning meanings to programs*, in Program Verification, Springer, 1993, pp. 65–81.

[51] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, *On the temporal analysis of fairness*, in Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1980, pp. 163–173.

[52] J. F. Groote and F. Vaandrager, *Structured operational semantics and bisimulation as a congruence*, Information and computation, 100 (1992), pp. 202–260.

[53] O. Grumberg, E. Clarke, and D. Peled, *Model checking*, The MIT Press Cambridge, 1999.

[54] S. Guha, C. Narayan, and S. Arun-Kumar, *Deciding timed bisimulation for timed automata using zone valuation graph*, 2012.

[55] M. Hennessy and R. Milner, *On observing nondeterminism and concurrency*, in International Colloquium on Automata, Languages, and Programming, Springer, 1980, pp. 299–309.

[56] ——, *Algebraic laws for nondeterminism and concurrency*, Journal of the ACM (JACM), 32 (1985), pp. 137–161.

[57] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, *Symbolic model checking for real-time systems*, Information and computation, 111 (1994), pp. 193–244.

[58] C. Hoare, *Communicating sequential processes*, Prentice-Hall, 1985.

[59] C. A. R. Hoare, S. D. Brookes, and A. W. Roscoe, *A theory of communicating sequential processes*, Oxford University Computing Laboratory, Programming Research Group, 1981.

[60] T. Hune and M. Nielsen, *Timed bisimulation and open maps*, in International Symposium on Mathematical Foundations of Computer Science, Springer, 1998, pp. 378–387.

[61] F. Jahanian and A. K. Mok, *Modechart: A specification language for real-time systems*, IEEE Transactions on Software engineering, 20 (1994), pp. 933–947.

[62] B. Jonsson and J. Parrow, *CONCUR'94: Concurrency Theory: 5th International Conference, Uppsala, Sweden, August 22-25, 1994. Proceedings*, vol. 836, Springer, 2006.

[63] E. Y. Juan and J. J. Tsai, *Compositional Verification of Concurrent and Real-Time Systems*, vol. 676, Springer Science & Business Media, 2012.

[64] J. R. Kennaway, *Formal semantics of nondeterminism and parallelism*, PhD thesis, University of Oxford, 1981.

[65] L. Lamport, *Verification and specification of concurrent programs*, in Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems), Springer, 1993, pp. 347–374.

[66] ——, *Real-time model checking is really simple*, in Advanced Research Working Conference on Correct Hardware Design and Verification Methods, Springer, 2005, pp. 162–175.

[67] F. Laroussinie, N. Markey, and P. Schnoebelen, *Efficient timed model checking for discrete-time systems*, Theoretical Computer Science, 353 (2006), pp. 249–271.

[68] K. G. Larsen, P. Pettersson, and W. Yi, *Compositional and symbolic model-checking of real-time systems*, in Proceedings 16th IEEE Real-Time Systems Symposium, IEEE, 1995, pp. 76–87.

[69] ——, *Uppaal in a nutshell*, International Journal on Software Tools for Technology Transfer (STTT), 1 (1997), pp. 134–152.

[70] K. G. Larsen and A. Skou, *Bisimulation through probabilistic testing*, Information and computation, 94 (1991), pp. 1–28.

[71] K. G. Larsen and W. Yi, *Time abstracted bisimulation: Implicit specifications and decidability*, in International Conference on Mathematical Foundations of Programming Semantics, Springer, 1993, pp. 160–176.

[72] D. Lepri, E. Ábrahám, and P. C. Ölveczky, *Sound and complete timed ctl model checking of timed kripke structures and real-time rewrite theories*, Science of Computer Programming, 99 (2015), pp. 128–192.

[73] D. Long, *Model checking, abstraction, and compositional reasoning*, Ph. D. Thesis, Carnegie Mellon University, (1993).

[74] N. Lynch and F. Vaandrager, *Action transducers and timed automata*, Formal Aspects of Computing, 8 (1996), pp. 499–538.

[75] N. A. Lynch and M. S. Tuttle, *Hierarchical correctness proofs for distributed algorithms.*, tech. rep., MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, 1987.

[76] R. Milner, *Calculi for synchrony and asynchrony*, Theoretical computer science, 25 (1983), pp. 267–310.

[77] ——, *Lectures on a calculus for communicating systems*, in International Conference on Concurrency, Springer, 1984, pp. 197–220.

[78] ——, *Communication and concurrency*, vol. 84, Prentice hall New York etc., 1989.

[79] ——, *Operational and algebraic semantics of concurrent processes*, in Formal Models and Semantics, Elsevier, 1990, pp. 1201–1242.

[80] B. Mishra and E. Clarke, *Hierarchical verification of asynchronous circuits using temporal logic*, Theoretical Computer Science, 38 (1985), pp. 269–291.

[81] T. Murata, *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE, 77 (1989), pp. 541–580.

[82] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*, vol. 2, Wiley Online Library, 2004.

[83] E.-R. Olderog and C. A. R. Hoare, *Specification-oriented semantics for communicating processes*, Acta Informatica, 23 (1986), pp. 9–66.

[84] R. Paige and R. E. Tarjan, *Three partition refinement algorithms*, SIAM Journal on Computing, 16 (1987), pp. 973–989.

[85] D. Park, *Concurrency and automata on infinite sequences*, in Theoretical computer science, Springer, 1981, pp. 167–183.

[86] I. Phillips, *Refusal testing*, Theoretical Computer Science, 50 (1987), pp. 241–284.

[87] G. D. Plotkin, *A structural approach to operational semantics*, (1981).

[88] A. Pnueli, *The temporal logic of programs*, in 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), IEEE, 1977, pp. 46–57.

[89] ——, *Linear and branching structures in the semantics and logics of reactive systems*, in International Colloquium on Automata, Languages, and Programming, Springer, 1985, pp. 15–32.

[90] L. Pomello, *Some equivalence notions for concurrent systems. an overview*, in European Workshop on Applications and Theory in Petri Nets, Springer, 1985, pp. 381–400.

[91] J.-P. Queille and J. Sifakis, *Specification and verification of concurrent systems in cesar*, in International Symposium on programming, Springer, 1982, pp. 337–351.

[92] G. M. Reed and A. W. Roscoe, *A timed model for communicating sequential processes*, in International Colloquium on Automata, Languages, and Programming, Springer, 1986, pp. 314–323.

[93] T. G. Rokicki and C. J. Myers, *Automatic verification of timed circuits*, in International Conference on Computer Aided Verification, Springer, 1994, pp. 468–480.

[94] W. C. Rounds and S. D. Brookes, *Possible futures, acceptances, refusals, and communicating processes*, in 22nd Annual Symposium on Foundations of Computer Science (sfcs 1981), IEEE, 1981, pp. 140–149.

[95] K. K. Sabnani, A. M. Lapone, and M. U. Uyar, *An algorithmic procedure for checking safety properties of protocols*, IEEE Transactions on Communications, 37 (1989), pp. 940–948.

[96] S. Schneider, *Concurrent and Real-time systems*, John Wiley and Sons, 2000.

[97] R. H. Sloan and U. Buy, *Stubborn sets for real-time petri nets*, Formal Methods in System Design, 11 (1997), pp. 23–40.

[98] K.-C. Tai and P. V. Koppol, *An incremental approach to reachability analysis of distributed programs*, in Proceedings of the 7th international workshop on Software specification and design, IEEE Computer Society Press, 1993, pp. 141–150.

[99] S. TaŞiran, R. Alur, R. P. Kurshan, and R. K. Brayton, *Verifying abstractions of timed systems*, in International Conference on Concurrency Theory, Springer, 1996, pp. 546–562.

[100] S. Tripakis, *The formal analysis of timed systems in practice*, PhD thesis, PhD thesis, Université Joseph Fourier, 1998.

[101] S. Tripakis and S. Yovine, *Analysis of timed systems using time-abstracting bisimulations*, Formal Methods in System Design, 18 (2001), pp. 25–68.

[102] A. Valmari, *Compositional analysis with place-bordered subnets*, in International Conference on Application and Theory of Petri Nets, Springer, 1994, pp. 531–547.

[103] M. Y. Vardi, *Alternating automata and program verification*, in Computer Science Today, Springer, 1995, pp. 471–485.

[104] S. Veglioni and R. De Nicola, *Possible worlds process algebras*, in International Conference on Concurrency Theory, Springer, 1998, pp. 179–193.

[105] F. Wang, *Formal verification of timed systems: A survey and perspective*, Proceedings of the IEEE, 92 (2004), pp. 1283–1305.

[106] F. Wang, R.-S. Wu, and G.-D. Huang, *Verifying timed and linear hybrid rule-systems with red.*, in SEKE, 2005, pp. 448–454.

[107] C. Weise and D. Lenzkes, *Efficient scaling-invariant checking of timed bisimulation*, in Annual Symposium on Theoretical Aspects of Computer Science, Springer, 1997, pp. 177–188.

[108] G. Winskel, *Synchronization trees*, Theoretical Computer Science, 34 (1984), pp. 33–82.

[109] W. J. Yeh and M. Young, *Compositional reachability analysis using process algebra*, in Symposium on Testing, Analysis, and Verification, 1991, pp. 49–59.

[110] W. Yi, *Ccs+ time= an interleaving model for real time systems*, in International Colloquium on Automata, Languages, and Programming, Springer, 1991, pp. 217–228.

[111] W. Yi, P. Pettersson, and M. Daniels, *Automatic verification of real-time communicating systems by constraint-solving*, in Formal Description Techniques VII, Springer, 1995, pp. 243–258.

[112] T. Yoneda and B.-H. Schlingloff, *Efficient verification of parallel real–time systems*, Formal Methods in System Design, 11 (1997), pp. 187–215.

[113] T. Yoneda, A. Shibayama, B.-H. Schlingloff, and E. M. Clarke, *Efficient verification of parallel real-time systems*, in International Conference on Computer Aided Verification, Springer, 1993, pp. 321–332.

Appendix A

Timed Automata

Proposition 21. *The reachability problem in timed automata is PSPACE- complete [8].*

Proposition 22. *Any two automata being timed bisimilar are also time-abstracted bisimilar [20].*

Proposition 23. *Time-abstracted similarity and bisimilarity are decidable for timed automata [20].*

Proposition 24. *Timed similarity and bisimilarity are decidable for timed automata [32].*

Proposition 25. *TCTL model checking is PSPACE-complete [4].*

Proposition 26. *TCTL satisfiability is undecidable [4].*

Proposition 27. *The language-inclusion problem for timed automata is undecidable, both over finite and infinite words [8].*

Proposition 28. *The problem of deciding if two initial states of a timed automaton are timed trace equivalent is undecidable [62].*

Proof. The undecidability proof for $\equiv_{tt}$ follows the proof that the language inclusion problem for timed automata over infinite words is undecidable [8]. □

Theorem 3. *If A is strongly non-zeno then every infinite run of A is non-zeno.*

Proof. Let $\rho = s_1 \overset{\delta_1\, e_1}{\rightarrow\rightarrow} s_2 \overset{\delta_2\, e_2}{\rightarrow\rightarrow} ...$ be an infinite run of A. Since A has only a finite number of edges, there exist some $i_1, i_2, ..., i_m$ such that $e_{i1}, e_{i2}, ..., e_{im}$ form a structural loop and ρ takes infinitely often every discrete transition e_{i_j}. There exist also a clock x and $j_1, j_2 \in \{i_1, i_2, ..., i_m\}$ such that $x \in \text{reset}\,(e_{j1})$ and $(x < 1) \cap \text{guard}\,(e_{j2}) = \varnothing$. Now, each time ρ takes an e_{j1}-transition, clock x is reset to 0. The next time ρ takes an e_{j2}-transition, at least 1 time unit has passed, since x must be greater or equal to 1 for e_{j2} to be taken. Since e_{j1}- and e_{j2} -transitions are taken infinitely often, an infinite number of 1-time-unit delays are accumulated, so ρ is non-zeno. □

Theorem 4. *If A, A' are strongly non-zeno so is $A \parallel A'$.*

Proof. Note that any structural loop of $A \parallel A'$ contain a structural loop of either A, or A', or both. Therefore, any structural loop of $A \parallel A'$ satisfies the conditions of Theorem 3 which implies that $A \parallel A'$ is strongly non-zeno. □

Theorem 5. *A strongly non-zeno TA is also timelock-free.*

Proof. A state s is a timelock if all infinite runs starting from s are zeno. A is timelock-free if none of its reachable states is a timelock, and by Theorem 3 if A is strongly non-zeno then every infinite run of A is non-zeno. □

Theorem 6. *If S is a zone then time-pred(S) and disc-pred(e, S) are also zones.*

Proof. Nore first that the reset, backward projection and intersection operations preserve convexity of polyhedra.

A polyhedron ζ is called convex if it can be specified as the intersection of a number of hyperplanes. If ζ is non-convex then it can be written as $\zeta_1 \cup ... \cup \zeta_k$, where $\zeta_1, ..., \zeta_k$ are all convex.

The operations $\zeta\,[Y := 0]$ (post-reset) and $[Y := 0]\,\zeta$ (pre-reset) are defined as:

$$\zeta\,[Y := 0] \overset{def}{=} \{\mathbf{v}\,[Y := 0] \mid \mathbf{v} \in \zeta\}$$

$$[Y := 0]\,\zeta \overset{def}{=} \{\mathbf{v} \mid \mathbf{v}\,[Y := 0] \in \zeta\}$$

The backward projection of an $\mathcal{X}$-polyhedron ζ is defined as :

$$\swarrow \zeta \overset{def}{=} \{\mathbf{v} \mid \exists\, \delta \in \mathbf{R}\,.\,\mathbf{v} + \delta \in \zeta\}$$

It can be seen that if ζ is convex then $\swarrow \zeta$ is also convex. □

The region graph is too large to be of any practical interest: its size is exponential in the number of clocks of the system as well as in the size of the constants utilized in the timing constraints. In order to manage the sate explosion, time-abstracting bisimulations were proposed. They induce a much coarser untimed state space. In fact, the region equivalence is a strong time-abstracting bisimulation.

Theorem 7. *The region equivalence is a strong time-abstracting bisimulation.*

Proof. It can be proven that the region equivalence (RegEq) [4] is a strong TaB. This implies in particular that the quotient of a TA A in terms of the greatest STaB defined on A is finite.

Informally, two states $(q, \mathbf{v})$ and $(q, \mathbf{v}')$ are region equivalent if $\mathbf{v}$ and $\mathbf{v}'$ agree on the integral parts of all clock values and have the same ordering of the fractional parts of all pairs of clock values. More formally, let $\lfloor \delta \rfloor$ be the greatest integer smaller than δ, for $\delta \in \mathbf{R}$. Let $\langle \delta \rangle$ be $\delta - \lfloor \delta \rfloor$. Consider a TA A with set of clocks $\mathcal{X}$ and let $c \geq c_{max}\,(\mathrm{A})$. Two $\mathcal{X}$-valuations $\mathbf{v}$ and $\mathbf{v}'$ are region equivalent, denoted $\mathbf{v} \simeq_c \mathbf{v}'$ if: (a) for all $x \in \mathcal{X}$, either $\lfloor \mathbf{v}\,(x) \rfloor = \lfloor \mathbf{v}'\,(x) \rfloor$ or $(\mathbf{v}\,(x) > c \wedge \mathbf{v}'\,(x) > c)$;(b) for all $x, y \in \mathcal{X}$, either $\lfloor \mathbf{v}\,(x) - \mathbf{v}\,(y) \rfloor = \lfloor \mathbf{v}'\,(x) - \mathbf{v}'\,(y) \rfloor$, or $(\mathbf{v}\,(x) - \mathbf{v}\,(y) > c \wedge \mathbf{v}'\,(x) - \mathbf{v}'\,(y) > c)$.

For any $c \in \mathbf{N}$, $\simeq_c$ is an equivalence relation, whose equivalence classes are called regions. The region equivalence can be extended to states of A so that $(q, \mathbf{v})$ is equivalent to $(q', \mathbf{v}')$ if $q = q'$ and $\mathbf{v} \simeq_c \mathbf{v}'$.

Let $(q, \mathbf{v}) \simeq_c (q, \mathbf{v}')$. Observe that:

1. For any c-closed $\mathcal{X}$-polyhedron ζ (in particular, any guard or invariant of A), $\mathbf{v} \in \zeta$ iff $\mathbf{v}' \in \zeta$;

2. For any set of clocks $X \subseteq \mathcal{X}, \mathbf{v}[X := 0] \simeq \mathbf{v}'[X := 0]$;

3. For any $\delta \geq 0$ there exists $\delta' \geq 0$ such that $\mathbf{v} + \delta \simeq \mathbf{v}' + \delta'$.

Now, if $(q, \mathbf{v}) \xrightarrow{e} (q_1, \mathbf{v}_1)$ is a discrete transition, then $(q, \mathbf{v}') \xrightarrow{e} (q_1, \mathbf{v}'_1)$ is also a discrete transition, since both $\mathbf{v}$ and $\mathbf{v}'$ satisfy guard(e). Also, $(q_1, \mathbf{v}_1) \simeq (q_1, \mathbf{v}'_1)$, since $\mathbf{v}[\text{reset}(e) := 0] \simeq \mathbf{v}'[\text{reset}(e) := 0]$. Let $(q, \mathbf{v}) \xrightarrow{\delta} (q, \mathbf{v} + \delta)$ be a time transition. There exists $\delta' \geq 0$ such that $\mathbf{v} + \delta \simeq \mathbf{v}' + \delta'$. $\mathbf{v}'$ and $\mathbf{v}' + \delta'$ satisfy invar(q), since $\mathbf{v}$ and $\mathbf{v} + \delta$ do. $\mathbf{v}' + \delta''$ satisfies invar(q) for any $\delta'' < \delta'$, by convexity of invar(q). Thus, $(q, \mathbf{v}') \xrightarrow{\delta'} (q, \mathbf{v}' + \delta')$ is also a time transition. □

The following definition is useful to prove an important property of STaBs which is based on the passage of time namely, that STaBs preserve branching-time properties.

Definition 50. Time transitions traversing classes: Consider a TA A and a STaB $\approx$ on A. Given a time transition of A, $s \xrightarrow{\delta} s + \delta$, and m different classes $C_1, ..., C_m$, the transition is said to traverses $C_1, ..., C_m$ if:

1. $s \in C_1$ and $s + \delta \in C_m$.

2. For all $0 < \delta' < \delta$, there exists $1 \leq i \leq m$ such that $s + \delta' \in C_i$.

Theorem 8. *Any time transition traverses a unique (finite) set of classes.*

Proof. Consider a time transition $s \xrightarrow{\delta} s + \delta$ and let m be the number of different points $0 < \delta_1 < ... < \delta_m < \delta$ such that $s + \delta_i$ and $s + \delta_{i+1}$ belong to different classes (there is a finite number of such points since the quotient is finite). The proof is by induction on m. If $m = 0$, then $s, s + \delta \in C_1$, for some class C_1. We show that $s + \delta_1 \in C_1$ for all $0 < \delta_1 < \delta$. Assume the opposite, $s + \delta_1 \in C_2$ for some $C_2 \neq C_1$. Then, since $C_1 \xrightarrow{\tau} C_2$ (from the fact that $s \xrightarrow{\delta_1} s + \delta'$) and $C_2 \xrightarrow{\tau} C_1$ (from the fact that $s + \delta_1 \xrightarrow{\delta - \delta_1} s + \delta$), we can build an infinite sequence $C_1 \xrightarrow{\tau} C_2 \xrightarrow{\tau} C_1 \xrightarrow{\tau} C_2 ....$ But this is not possible, since we assumed $\approx$ to be weaker than the region equivalence $\simeq_c$ and after the upper bound c all states are equivalent. The induction step is straightforward. □

Theorem 9. *If $s \approx s'$ then for any time transition $s \xrightarrow{\delta} s+\delta$, there exists a time transition $s' \xrightarrow{\delta'} s'+\delta'$ such that $s+\delta \approx s'+\delta'$ and the two transitions traverse the same classes.*

Proof. Let $C_1, ..., C_m$ be the classes traversed by $s \xrightarrow{\delta} s+\delta$. The proof is by induction on m. If $m = 1$, then $s \approx s+\delta$ and it suffices to take $\delta' = 0$. For simplicity, instead of proving the general induction step, we assume that $m = 2$, that is, $s \xrightarrow{\delta} s+\delta$ traverses classes C_1, C_2. The extension to any $m > 1$ is easy using the induction hypothesis. We have: $s, s' \in C_1, s+\delta, s+\delta' \in C_2$, for some δ'. We want to show that for all ${\delta_1}' < \delta', s'+{\delta_1}' \in C_1 \cup C_2$. Assume this is not the case, that is, $s'+{\delta_1}' \in C$ and C is different from C_1, C_2. Since $s' \approx s$, there exists $s \xrightarrow{\delta_1} s+\delta_1$ such that $s+\delta_1 \in C$. From the fact that $s \xrightarrow{\delta} s+\delta$ traverses C_1, C_2 and condition 2 of the definition of traversal, it must be that $\delta_1 > \delta$. Thus, $C_2 \xrightarrow{\tau} C$ (from the fact that $s+\delta \xrightarrow{\delta_1-\delta} s+\delta_1$). On the other hand, $C \xrightarrow{\tau} C_2$ (from $s'+{\delta_1}' \xrightarrow{\delta'-{\delta_1}'} s+\delta'$). We can build an infinite sequence $C_2 \xrightarrow{\tau} C \xrightarrow{\tau} C_2 \xrightarrow{\tau} C$... contradicting the hypotheses. □

Theorem 10. *Let A be a strongly non-zeno TA and $\approx$ be a strong time-abstracting bisimulation on A. For any CTL formula ϕ and any pair of states $s \approx s', s \models \phi$ iff $s' \models \phi$*

Proof. This theorem can be proved by induction on the syntax of ϕ. The basis comes directly from the hypothesis that $\approx$ respects P. The interesting induction steps are for $\phi = \forall \phi_1 \mathcal{U} \phi_2$ or $\phi = \exists \phi_1 \mathcal{U} \phi_2$. The latter case is considered for proving, the former being similar. Assume that $s \models \exists \phi_1 \mathcal{U} \phi_2$. Then, there exists a non-zeno run $\rho = s \xrightarrow{\delta_1} \xrightarrow{e_1} ...$ and some point i along ρ such that $\rho(i) + \delta_i \models \phi_2$ and for all $j \leq i,\ \delta \leq \delta_j,\ \rho(j) + \delta \models \phi_1 \vee \phi_2$. From the fact that $s \approx s'$, a run $\rho' = s' \xrightarrow{{\delta_1}'} \xrightarrow{{e_1}'} ...$ is built, such that $s_j \approx s'_j$ and $s_j + \delta_j \approx s'_j + \delta'_j$, for all j. From the strongly non-zeno hypothesis, ρ' is non-zeno. From the induction hypothesis, $\rho'(i) + \delta'_i \models \phi_2$ and for all $j \leq i,\ \rho'(j) + \delta'_j \models \phi_1 \vee \phi_2$. It must be shown that $\rho'(j) + \delta' \models \phi_1 \vee \phi_2$, for all $\delta' \leq \delta'_j$. Then for any $\delta' \leq \delta'_j$, there exists $\delta \leq \delta_j$ such that $s'_j + \delta' \approx s_j + \delta$. The result follows from the induction hypothesis. □

Theorem 11. *Every run ρ is inscribed in a unique path π in $G \approx$. Inversely, if $\pi = C_1 \rightarrow$*

$C_2 \rightarrow ...$ is a path in $G \approx$ then for all $s_1 \in C_1$ there exists a run ρ starting from s_1 and inscribed in π.

Proof. A straightforward modification of the proof of Lemma 3.35 of [2]. $\square$

Theorem 12. *$C \in$ ctl-eval(ϕ) iff for all $s \in C$, s satisfies ϕ.*

Proof. The theorem can be proved by induction on the syntax of ϕ. The basis (ϕ is an atomic proposition) comes from the fact that $\approx$ respects P. The case for $\phi_1 \vee \phi_2$ is trivial. In the case where ϕ is of the form $\exists\ \phi_1 \mathcal{U} \phi_2$, assume that $C' \in \text{ctl-eval}(\phi), C \in \text{ctl-eval}(\phi_1)$ and $C \xrightarrow{\tau} C'$. Let $s \in C$. There exists δ such that $s \xrightarrow{\delta} s + \delta$ and $s + \delta \in C'$. By induction, $s + \delta$ satisfies ϕ and s satisfies ϕ_1. Now, for any $\delta' < \delta, s + \delta' \in C \cup C'$ (here the fact that C' are the immediate time successors of C is utilized). By Theorem 9 and the fact that $s + \delta'$ is STa-bisimilar either to s or $s + \delta$, we have $s + \delta' \models \phi_1$ or $s + \delta' \models \phi$, thus, $s \models \phi$. The case $C \xrightarrow{e} C'$ is similar.
Consider the case where ϕ is of the form $\forall \phi_1 \mathcal{U} \phi_2$. Let $C \notin \text{ctl-eval}(\phi)$. Since A is deadlock-free, there is an infinite path in G, $\pi = C \rightarrow C_1 \rightarrow ...$, and some i, such that $C_i \notin \text{ctl-eval}(\phi_1)$ and for all $j < i, C_j \notin \text{ctl-eval}(\phi_2)$. Also note that π contains only a finite number of τ-transitions, since there are no τ -self-loops in G. Finally, π is non-zeno, since A is strongly non-zeno. Thus, by Theorem 10, we can extract from π a non-zeno run which falsifies $\forall \phi_1 \mathcal{U} \phi_2$. $\square$

Development of Partition Refinement Algorithm to the Timed Case The partition refinement algorithm form the timed case will generate the finite graph G. The state space of TA consist of two types of predecessors, corresponding to discrete and time transitions of the TA. The MMGA algorithm is adapted to infinite state spaces; they accept an effective representations of classes and decision ways in order to calculate intersection, set-difference and predecessors of classes, and testing whether a class is empty. It must be ensured that a pre-stable partition always exists for termination. The adapted algorithm is called time-abstracting MMGA (TA-MMGA).

Correctness of the algorithm is based on the definitions of the predecessor operators. Termination is ensured by Theorem 6. which means, the algorithm will produce the partition induced by the region equivalence in the worst case.