TECHNIQUES FOR AUTOMATIC VERIFICATION OF REAL-TIME SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Rajeev Alur
August 1991

© Copyright 1991 by Rajeev Alur
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

David Dill
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Zohar Manna
(Coadvisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Moshe Vardi

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

This thesis proposes formal methods for specification and automatic verification of *finite-state real-time* systems. The traditional formalisms for reasoning about programs abstract away from quantitative time and, consequently, are inadequate for reasoning about real-time systems. We extend the methods based on automata and temporal logics to allow them to model timing delays and to verify real-time requirements.

We introduce *timed automata* to model the behavior of real-time systems over time. Our definition provides a simple, and yet powerful, way to annotate state-transition graphs with timing constraints using finitely many real-valued *clocks*. A timed automaton accepts *timed words* — strings in which a real-valued time of occurrence is associated with each symbol. We study timed automata from the perspective of formal language theory: we consider closure properties, decision problems, and subclasses.

We present two conservative extensions of the existing temporal logics to allow them to specify timing properties. The *metric interval temporal logic* (MITL) uses linear-time semantics, and its syntax allows temporal operators to be subscripted with intervals restricting their scope in time. The *timed computation tree logic* (TCTL) uses branching-time semantics, and its syntax provides access to time through a novel kind of time quantifier.

In the proposed verification method, a finite-state system is modeled as a composition of timed automata, and the correctness is specified either as a *deterministic* timed automaton, or as a formula of MITL or TCTL. In each case we develop an algorithm for *model checking*. The distinguishing feature of our work is the use of the set of reals to model time; we argue that the *denseness* of the time domain is crucial for modeling event-driven asynchronous systems. The thesis also clarifies the relationship between different models and logics for real-time, and answers some basic questions regarding complexity, decidability, and expressiveness.

Acknowledgments

First of all I thank my advisors, David Dill and Zohar Manna, for offering me technical, financial, and moral support during the last four years. My reading committee comprised of David, Zohar, and Moshe Vardi, and I consider myself fortunate that I had access to valuable guidance from all three of them. I am also thankful to them for their critical reading of the draft. Zohar introduced me to temporal logics, and directed me to the relatively unexplored area of real-time logics. He also made possible an extremely productive visit to the Weizmann Institute in Israel. Much of the research reported in this thesis is inspired by my discussions with David about his ideas on coupling automata with timing constraints. Moshe is one of the leading proponents of the automata-theoretic approach to verification, and his expertise on the subject has been very useful to me.

It has been a great pleasure for me to work closely with Tom Henzinger. We learned to do research together; we solved many problems together; it would be futile to pinpoint his innumerable contributions towards my work. Special thanks also go to my other colleagues: Costas Courcoubetis and Tomas Feder. Costas's unbounded enthusiasm to attack new problems has been a source of inspiration to me. The decision procedure for MITL builds upon some insightful observations made by Tomas.

I have had the opportunity to discuss my research with many scientists at Stanford, at IBM Almaden Research Center, and at various conferences and seminars. I thank all of them for being helpful and encouraging. I am particularly grateful to Joe Halpern, Dinesh Katiyar, John Mitchell, Amir Pnueli, Howard Wong-Toi, and Mihalis Yannakakis.

My allegiance to computer science is mainly due to the excellent education I received at the Indian Institute of Technology at Kanpur, and I am thankful to the faculty on its Computer Science Department. Also this thesis would not exist without the love and support of my family, especially, my parents. I would also like to use this opportunity to thank all my friends on Stanford campus; because of them my stay here has been very enjoyable and memorable.

Contents

Abstract **iv**

Acknowledgments **v**

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Background: formalisms for qualitative reasoning . . . 2

1.2.1 Temporal logics . . . 3

1.2.2 Automata-theoretic approach . . . 4

1.2.3 Other approaches . . . 5

1.3 Overview . . . 6

1.3.1 Verification methodology . . . 6

1.3.2 Contributions . . . 8

1.4 Related research . . . 10

1.4.1 Modeling real-time systems . . . 10

1.4.2 Specification languages . . . 12

1.4.3 Verification . . . 14

1.5 Organization of the thesis . . . 16

2 Adding Time to Semantics **18**

2.1 Trace semantics . . . 18

2.2 Timed traces . . . 20

2.2.1 Adding timing to traces . . . 20

2.2.2 Discrete-time model . . . 23

2.2.3 Dense-time model . . . 24

2.2.4 Fictitious-clock model . . . 24
2.3 A case for the dense-time Model . . . 27
2.3.1 Correctness . . . 27
2.3.2 Expressiveness . . . 32
2.3.3 Compositionality . . . 33
2.3.4 Complexity . . . 34

3 Automata-Theoretic Approach 36
3.1 ω-automata . . . 36
3.2 Timed automata . . . 39
3.2.1 Timed languages . . . 39
3.2.2 Transition tables with timing constraints . . . 40
3.2.3 Clock constraints and clock interpretations . . . 42
3.2.4 Timed transition tables . . . 43
3.2.5 Timed regular languages . . . 45
3.2.6 Properties of timed regular languages . . . 47
3.2.7 Timed Muller automata . . . 49
3.3 Checking emptiness . . . 51
3.3.1 Restriction to integer constants . . . 51
3.3.2 Clock regions . . . 52
3.3.3 The region automaton . . . 54
3.3.4 The untiming construction . . . 60
3.3.5 Complexity of checking emptiness . . . 61
3.4 Intractable problems . . . 63
3.4.1 A Σ_1^1-complete problem . . . 64
3.4.2 Undecidability of the universality problem . . . 65
3.4.3 Inclusion and equivalence . . . 67
3.4.4 Nonclosure under complement . . . 69
3.5 Deterministic timed automata . . . 70
3.5.1 Definition . . . 70
3.5.2 Closure properties . . . 72
3.5.3 Decision problems . . . 74
3.5.4 Expressiveness . . . 75

3.6 Variants of timed automata 76
3.6.1 Clock constraints 76
3.6.2 Timed automata with ϵ-transitions 79
3.7 Verification 82
3.7.1 ω-automata and verification 82
3.7.2 Verification using timed automata 84
3.7.3 Verification example 86
3.7.4 Implementation 90

4 Linear Temporal Logic **92**
4.1 Propositional temporal logic: PTL 92
4.2 Metric interval temporal logic 94
4.2.1 Intervals 95
4.2.2 Timed state sequences 95
4.2.3 Syntax and semantics of MITL 97
4.2.4 Defined operators 98
4.2.5 Refining the models 99
4.2.6 Real versus rational time 101
4.2.7 Allowing rational constants 103
4.2.8 Avoiding undecidability 103
4.3 Interval automata 105
4.4 Deciding MITL 112
4.4.1 Restricting the problem 112
4.4.2 Intuition for the algorithm 115
4.4.3 Witnessing intervals 116
4.4.4 Type-1 and type-2 formulas 118
4.4.5 Type-3 and type-4 formulas 119
4.4.6 Constructing the interval automaton 121
4.4.7 Complexity of MITL 128
4.5 Verification using MITL 131
4.6 Expressiveness 132
4.6.1 Comparison with fictitious-clock logics 132
4.6.2 Comparison with interval automata 138

5 Branching-Time Logic **140**

5.1 Computation tree logic 140

5.2 The logic TCTL 143

5.2.1 Syntax 143

5.2.2 Semantics 144

5.2.3 On the choice of syntax 146

5.2.4 Undecidability 148

5.2.5 Interval automata as TCTL-structures 149

5.3 Model checking 152

5.3.1 Introducing formula clocks 152

5.3.2 Clock regions 153

5.3.3 The region graph 155

5.3.4 Labeling algorithm 161

5.3.5 Complexity of the algorithm 162

5.3.6 Complexity of model-checking 164

5.3.7 TCTL with fairness 165

6 Concluding Remarks **166**

Bibliography **169**

Chapter 1

Introduction

1.1 Motivation

With the increasing use of computers in safety-critical applications there is a pressing need for designing more reliable systems. As a result, developing formal methods for the design and analysis of concurrent systems has been an active area of computer science research. The conventional approach to testing the correctness of a system involves simulation on some test cases. This method is quite inadequate for developing bug-free complex concurrent systems. One approach to assure *correctness* is to employ *automatic verification* methods. A verification formalism comprises of

1. A formal semantics which assigns mathematical meanings to system components and correctness criteria.
2. A language for describing the essential aspects of the system components, and constructs for combining them.
3. A specification language for expressing the correctness requirements.
4. A verification algorithm to check if the correctness criteria are fulfilled in every possible execution of the system.

In this thesis we provide formalisms for automatic verification of *finite-state real-time systems*.

The class of systems to which our methods are applicable includes asynchronous circuits, communication protocols, and controllers (such as a flight controller, or a controller for a

manufacturing plant). The essential characteristics of such systems are:

- *Finite-state:* The system can be in one of the finitely many discrete states. If we focus only on the control aspect of the system, ignoring the computational aspect, then this is an useful abstraction in many cases. State-transitions are triggered by events which are instantaneous.

- *Reactive:* The system constantly interacts with the environment reacting to stimuli. So we are interested in the ongoing behavior over time. This is quite unlike the traditional "transformational" view of the programs where the functional relationship between the input state and the output state defines the meaning of a program. The system comprises of a collection of components operating concurrently and communicating with each other.

- *Real-time:* The correctness of the system depends on the actual magnitudes of the timing delays of the components. This is obviously the case when the system needs to meet *hard* real-time deadlines: the system needs to respond to a stimulus within a certain fixed time bound. Also there are cases when the logical correctness of the system depends on the lengths of various delays.

Real-time systems are used in safety-critical applications such as controllers for nuclear plants. Failures in such systems can be very expensive and even life-threatening. Because of the intricacies of the timing relationships, real-time systems are quite hard to model, specify, and design. Consequently, there is a great demand for formal methods applicable to real-time systems.

1.2 Background: formalisms for qualitative reasoning

Several different formalisms have been proposed to reason about *reactive* systems. These include Petri nets, process algebras, temporal logics, automata-theoretic techniques, and partial-order models.

These methodologies abstract from time, retaining the information about the causality and/or the temporal order of occurrence of observable events. Even though there is no general agreement about what is the right semantics of concurrency, some of these techniques have provided the foundations for building verifiers for hardware and communication protocols, and some have suggested structured disciplines for writing concurrent programs. We

will briefly review these approaches; the methods for automatic verification of finite-state systems are of main interest to us.

1.2.1 Temporal logics

The use of temporal logic as a formalism for specifying the behavior of a reactive system over time was first proposed by Pnueli in 1977 [Pnu77]. The subject has been extensively studied since then [BMP81, MP81, EC82, OL82, Lam83, MW84, BKP86, CES86, Pnu86, MP89, Lam91]. Temporal logic is a modal logic with modalities such as $\Diamond$ meaning "eventually", and $\Box$ meaning "always" (see [Eme90] for an overview). Temporal logics provide a succinct and natural way of expressing the desired temporal requirements. Two types of temporal logics have been proposed: *linear-time* and *branching-time*.

In the linear-time framework, a system is viewed as a set of computations, where each computation is a sequence of system-states recording all the transitions over the course of time. A linear temporal logic formula is interpreted over such state sequences [Pnu77, OL82]. The branching-time logics, on the other hand, are interpreted over tree models [BMP81, EC82, EL85]. The system is viewed as a finitely-branching tree; the paths in the tree correspond to the possible executions of the system.

In the traditional approach to verification of concurrent programs, the correctness of the program is expressed by a formula in first-order temporal logic. The verification problem reduces to proving a theorem in a deductive system. For example, Manna and Pnueli [MP89] have developed the model of fair transition systems to describe the implementation, and give a proof system to verify temporal logic specifications. Though the technique is quite general, constructing a proof needs to be done manually and requires a great deal of understanding of the program. The only extent of automation one can hope for is to have the proof checked by a machine and possibly to have some limited heuristics in finding the proof.

Model checking provides a different approach to checking properties of finite-state systems [CES86, LP85, EL85, BCD$^+$90, GW91]. In this approach, the global state-transition graph is viewed as a finite Kripke structure (with fairness requirements, if necessary). The specification of the system is given as a formula of a propositional temporal logic. The model-checking algorithm then decides whether the system meets the specification in all possible scenarios. For the linear-time case, the complexity of model-checking is linear in the size of the state-transition graph and exponential in the size of the specification, and in

the branching-time case, it is linear both in the size of the state-transition graph and the length of the temporal logic specification. Various aspects of the model-checking problem for the logic CTL [EC82] have been studied. This approach has been successfully applied to verify circuits and protocols, and to find bugs in previously-published, non-trivial protocols and circuits [CES86, BCDM86].

The model-checking approach to program verification is probably the most exciting advance in the theory of program correctness in recent years. It has been extended to probabilistic systems [Var85, PZ86, CY88], to real-time systems [EMSS89, AH89, ACD90, Lew90, HLP90], and to probabilistic real-time systems [HJ89, ACD91a, ACD91b].

The main difficulty in using model-checking approach is the *state-explosion* problem: the size of the global state-transition graph grows exponentially with the number of components in the system. This problem has received great attention recently, and different ways to cope with the problem have been proposed [BCD+90, God90, GW91].

1.2.2 Automata-theoretic approach

A related approach to verification of finite-state systems uses ω-automata [WVS83, Var87]. The computation of a reactive program is viewed as an infinite word over the alphabet of events (or states). This gives rise to an intimate connection between reasoning about reactive systems and the formal language theory. A system is modeled as an automaton generating infinite sequences which correspond to the possible computations of the system. The automata over infinite words were first studied by Büchi in relation to the theory S1S, the second order monadic theory of natural numbers with successor [Büc62]. Büchi automata and their variants have been studied in great detail since then [Cho74, Mul63, McN66], leading to a beautiful theory of ω-regular languages (see [Tho90] for an overview).

In the automata-theoretic framework, a system is modeled as a composition of several automata. The implementation automaton I is the product of these automata, and acts as a *generator*. The specification is given as another automaton S which acts as an *acceptor*. Alternatively, from a linear temporal logic specification, one can construct an automaton which accepts all the computations that satisfy the given formula. The implementation is correct iff every behavior generated by I is accepted by S. Thus the verification problem reduces to the language inclusion problem. Consequently the known effective constructions for intersection, complementation, and test for emptiness can be used

as a basis for automatic verification. Checking for language inclusion involves complementing the specification automaton which can be expensive, particularly for nondeterministic automata [SVW87, Saf88]. Alternative ways which use simulation relations have been proposed [DHW91].

Another advantage of automata specifications is the possibility of hierarchical verification. Since both implementation and specification are automata, there is no real distinction between them; they can be viewed as descriptions of the system at different levels of detail. Consequently verification can be carried out by starting with a high-level model and applying successive refinements. The system COSPAN developed at Bell Laboratories is based on the automata based selection/resolution model [AKS83], and has been used successfully to verify some of the commercially used protocols [HK90].

Apart from the automatic verification approach, other automata based techniques also have been proposed. Lynch et. al. have defined input-output automata as a model of computation in asynchronous distributed networks, and have developed methods to construct modular correctness proofs of distributed algorithms [LT87]. Alpern and Schneider show how to derive proof obligations from Büchi automata specifications, and give proof rules for checking these obligations against a concurrent program [AS89].

1.2.3 Other approaches

Petri nets provide a succinct and elegant way to model concurrency and causal dependencies in reactive systems [Pet81]. An extensive literature exists on the topic, and the formalism has been widely used in the specification, modeling and performance evaluation of systems. Reachability analysis on Petri nets can be used to detect if something "bad" will ever happen.

Milner introduced CCS (Calculus of Communicating Systems) as a model for concurrent systems [Mil80]. CCS views the system computation as a finitely-branching tree. The calculus provides operators such as nondeterministic choice, parallel composition, and hiding to build complex terms from simpler ones, with an associated array of algebraic laws. The verification methodology based on CCS defines an equivalence relation on CCS terms, and the verification problem is to decide whether or not the specification term is equivalent to the implementation term. Various notions of observational equivalences and preorders have been proposed and studied.

Another popular formalism for concurrency is Hoare's theory of Communicating Sequential Processes [Hoa78, Hoa85]. CSP provides a small, and yet powerful, set of constructs for writing concurrent programs, and laws for reasoning with them. In one of the possible models for CSP, each process is modeled as a collection of sequences called *traces*, where a trace records the order of events that may be observed when the process runs.

1.3 Overview

The thesis extends the finite-state verification techniques based on automata and temporal logics to real-time systems. The techniques discussed in the previous section abstract away from quantitative time and, hence, are unsuitable for modeling and specifying real-time systems. We develop real-time extensions of automata which can model timing delays between system transitions, and real-time extensions of temporal logics which can specify hard real-time requirements.

1.3.1 Verification methodology

Formal semantics

The standard notion of a computation models only the sequencing of events or state-transitions. In the event-based model of automata, we introduce real-time by assigning a real-valued time to every event occurrence. Similarly, we incorporate real-time in the model of state sequences by associating an interval of the real number line with every state. The feature that distinguishes our work from most of the earlier work on formalisms for real-time reasoning is the use of a *dense domain* for choosing the time values.

We consider both linear-time and branching-time logic specifications. In the linear-time world, the system is modeled as a set of (dense) linear executions. In the branching-time world, the system is viewed as a tree over such executions; however, because of the infinitely many choices for the time of the next transition, the tree is no longer finitely branching.

Modeling the system

To augment the state-transition graph of a system with its timing constraints, we propose the formalism of *timed automata*. Our definition is inspired by the model introduced by Dill [Dil89]. A timed automaton is a finite state-transition graph with a finite set of real-valued

clocks. The clocks can be reset to 0 (independently of each other) with the state-transitions of the system, and keep track of the time elapsed since the last reset. To express the timing delays of the system, we associate with the transitions, or alternatively with the states, constraints that compare clock values with constants. With this mechanism we can model timing properties such as "the channel delivers every message within 3 to 5 time units of its receipt".

Timed automata can model several interesting aspects of real-time systems: qualitative features such as liveness, fairness, and nondeterminism; and quantitative features such as periodicity, bounded response, and timing delays. The model of timed automata is *compositional*, and we provide an algorithm to construct the global automaton from the automata describing the behaviors of different components.

Specification languages

As in the qualitative case, a timed automaton defines a formal language, and thus, when viewed as an acceptor, provides a specification formalism. We propose that *deterministic timed automata*, coupled with *Muller acceptance* conditions, be used as a specification language.

We also consider real-time extensions of temporal logics. In the linear-time case, we define the logic *metric interval temporal logic* (MITL) by extending the linear temporal logic PTL. In MITL, the temporal modalities are subscripted with time intervals restricting their scope in time. For instance, $\Box_{(2,4)}$ means "throughout the interval $(2,4)$." With this extension one can express a bounded response requirement that "every p-state is followed by some q-state within 5 time units," by the formula

$$\Box\,[\,p \rightarrow \Diamond_{[0,3]}\,q\,].$$

In the branching-time case we define the logic *timed computation tree logic* (TCTL) by extending the (qualitative) branching-time logic CTL. The syntax of TCTL expresses timing requirements by using variables ranging over the time domain of reals along with a novel form of quantification. In this logic, the above bounded response property is written

$$\forall\Box x.\,[\,p \rightarrow \forall\Diamond y.\,(q \wedge y \leq x+3)\,].$$

The time quantifier "$x.$" binds the associated variable x to the "current" time.

We address complexity and expressiveness issues relating to these three specification languages.

Verification

In a typical verification problem the system behavior is described as a timed automaton I presented as a composition of several smaller automata. The desired requirement S is specified in one of the following forms:

- Deterministic timed Muller automaton
- Formula of the linear-time logic MITL
- Formula of the branching-time logic TCTL

For the above three specification languages, we present model-checking algorithms which check if the implementation I is correct with respect to the specification S. The complexity of the algorithm is exponential in the number of components in the system (as is the case for qualitative verification). The timing considerations introduce an additional blow-up by the actual magnitudes of the constants appearing as bounds for the timing delays.

1.3.2 Contributions

Automatic verification in dense-time: decidability and complexity

The main contribution of the thesis is the proposed technique for automatic verification of timing properties. All the formalisms proposed previously are either undecidable or use a discrete time domain.

The undecidability results proved for different cases identify the boundary between decidability and undecidability for different formalisms for real-time reasoning. For instance, we show that for the real-time logic MITL, introduction of modalities such as $\Diamond_{=3}$ makes the satisfiability and model-checking problems undecidable; thus the restriction that the intervals subscripting the modalities be *nonsingular* is crucial. Such a restriction is *not* required if we choose to interpret the logic over one of the discrete models. More surprisingly, in the branching-time case, the logic TCTL uses the dense-time semantics, allows equality constraints, and yet has a decidable model-checking problem.

We characterize the complexity classes of all the verification problems defined here. For example, model-checking for the branching-time logic TCTL is PSPACE-complete. Thus for this problem, the complexity class stays unchanged as we move from the qualitative case to the discrete-time case to the dense-time case. Introducing real-time considerations involves an additional blow-up by the actual magnitudes of the constants bounding the timing delays. This blow-up is observed for all decidable problems considered in this thesis.

Timed automata and theory of timed languages

Timed automata provide a simple, and yet powerful, way of annotating state-transition graphs with timing constraints. Judging by the response we have received so far, we feel hopeful that it will provide the "canonical" model for finite-state real-time systems.

We study timed automata from the perspective of formal language theory. A *timed language* comprises of infinite words over a finite alphabet, in which each symbol has a real-valued time of occurrence. Timed automata define *timed regular languages*. We consider closure properties and decision problems for timed automata. The class of nondeterministic automata is closed under union, intersection, but not under complement. For these automata testing emptiness is PSPACE-complete, but checking for universality is undecidable — Π_1^1-hard. On the other hand, the deterministic automata are closed under complement also, and problems such as universality and language inclusion are solvable in PSPACE. We have focused on the application of this theory for verification problems, but the theory may hold interest of its own.

Models for real-time

Several real-time extensions of existing formalisms have been proposed previously, but the issue of choosing the "right" model for introducing time in semantics has not been addressed before. We precisely characterize three different models: *dense-time*, *discrete-time* and *fictitious-clock*.

The simplest of all these models is the discrete-time model: time is assumed to be isomorphic with the set of natural numbers. Thus events are assumed to happen synchronously with the ticks of a global clock. This assumption is relaxed in the *fictitious-clock* approach. Here time is viewed as a global state variable that ranges over the domain of natural numbers, and is incremented by one with every *tick* transition of a global, asynchronous, fictitious, discrete clock. The timing delay between two events is measured by counting the number of *tick*s between them. Finally, there is the possibility of choosing the real line itself to model time. The occurrence times for events are real numbers in this model, and consequently it is the most realistic model for asynchronous systems.

We compare these models, and study how the choice of a model affects the complexity of different decision problems. The model of our choice is the dense-time model: in Chapter 2 we make a case for our preference.

1.4 Related research

The problem of formal methods to deal with the real-time aspect explicitly has received relatively little attention in the past. However, over the last couple of years there has been an explosion on the number of papers on this subject, and several real-time logics and real-time algebras have been proposed. In this section we provide an overview of some this work; the proceedings of a recent REX workshop on the topic "Real Time: Theory in Practice" will provide an excellent starting point for anybody who wishes to explore the wide range of research that is under way [dR91].

1.4.1 Modeling real-time systems

The idea of introducing real-time into qualitative models of behavior by associating a time of occurrence with every event or state-transition is fairly standard, and is used in most of the approaches considered here.

Timed automata

Our definition of timed automata is a modified version of the formalism of timed automata introduced by Dill [Dil89]. In the original definition, the automaton has a finite set of *timers*. There are two special events associated with every timer: *set* and *expire*. A timer is activated by the *set* event to an arbitrary value between specified bounds. All the timers count down at the same rate, and the *expire* event corresponding to a timer gets fired when its value becomes 0. To express a constraint on the delay between two events e_1 and e_2, a timer with appropriate bounds is set along with e_1, and the event e_2 is required to be synchronized with the expiration of the timer. In our formalism, the automaton has a finite set of *clocks* which count *up* showing the elapsed time since the last reset. The clocks can be reset to 0 along with the state-transitions, and timing delays are expressed by annotating the transitions with constraints comparing clock values with constant bounds. Apart from some technical conveniences in developing the emptiness algorithm and proving its correctness, the reformulation allows a simple syntactic characterization of determinism for timed automata.

A model similar to Dill's was independently proposed and studied by Lewis [Lew89]. He defines *state-diagrams*, and gives a way of translating a circuit description to a state-diagram. A state-diagram is a finite-state machine where every edge is annotated with a

matrix of intervals. The machine essentially remembers information about delays between a finite number of transition pairs that have occurred in the past. With every transition the associated delay matrix is used to check the consistency of previous delays, and the time of the current transition is used to update the delay information. In essence, like timed automata, state diagrams also express constant bounds on delays along paths. The definition of timed automata, however, is much simpler.

Transition systems

Perhaps the most standard way of introducing timing information in a process model is by associating lower and upper bounds with transitions. For example, Ostroff [Ost90b, Ost90a] and Henzinger [HMP91, Hen91] define *real-time transition systems* by extending the framework of fair transition systems [MP89] by associating lower and upper bounds with transitions. The timing constraints for a legal computation require that a transition with lower bound l and upper bound u is continuously enabled for at least l time units before it is taken, and is never enabled for u time units at a stretch without being taken.

Timed I/O automata [LA90] are a similar extension of I/O automata. As in I/O automata there is a useful distinction between input events and output events. Their semantics of timed traces is similar to ours, except that the events appearing at the same time are clustered in a set in our definition, and their definition considers all possible linearizations. Aggarwal and Kurshan [AK83] show how to incorporate timing information in their selection/resolution model (based on automata) in a similar way. Jahanian and Stuart describe a modular and graphical language *Modechart* for expressing the control and timing information of a real-time system [JS88]. Timing extensions of the Petri net model also have been considered [Ram74, CR83]. In *timed Petri nets* each transition has an associated real-valued time (or an interval) giving its duration; a transition is fired only after it is continuously enabled for a time period equal to this duration.

The models of transition systems or timed I/O automata have a strong operational flavor compared to timed automata. In a timed automaton there is no explicit notion of enabled transitions, lower bounds, or upper bounds. The timing properties of the system are expressed more abstractly. For finite-state systems, transition system style descriptions can be compiled into timed automata in a straightforward way. In principle, timed automata can express more complex timing constraints than transition systems.

Process algebras

The standard way to introduce real-time in algebraic models is to add some form of a "delay" construct to the calculus [Mil83, RR88, Zwa88, MT90, NRSV90, Wan90, BB91]. The following discussion should provide a flavor of these constructs. Reed and Roscoe [RR88] extend CSP to *timed CSP* by introducing an additional WAIT construct. The construct "WAIT t" models the process that terminates successfully after t time units. The semantics is defined in terms of the *timed stability* model which uses a dense domain. In this model, with every event its time of occurrence is recorded, along with the earliest time at which the process can engage in the next action. A similar approach is Zwarico's model of *timed acceptances* [Zwa88]. The semantics uses discrete-time, and is an extension of the acceptance model for CSP: after every event all the possible choices the process may execute are also recorded.

Milner's SCCS [Mil83] is a calculus in the style of CCS. It makes the assumption of *strong synchrony*: all processes execute in lock-step, performing one event with each passing unit of time. Sifakis et. al. have defined ATP — an *algebra for timed processes* [NRSV90] based on the fictitious-clock model. The vocabulary of actions contains a distinguished element corresponding to the *tick* of a global clock. The syntax of the calculus does not allow explicit references to this *tick* event, but has a binary unit-delay operator. For two terms t_1 and t_2, this operator gives a process that behaves as t_1 if started before the next *tick*, and behaves as t_2 otherwise. Another timed process algebra is TCCS [Wan90]. The syntax of CCS is extended with a delay construct; "$\epsilon(t).P$" represents a process that waits for t time units and then behaves like P. The set of actions contains a special *timeout* action, and an action $\epsilon(t)$ for every real value t.

In the untimed case, a few constructs of process algebras are rich enough to model complex aspects of concurrent systems. However, this does not seem to be the case for the timing properties. None of the above calculi seem to be able to model all the features such as lower and upper bounds, timeouts, and periodicity.

1.4.2 Specification languages

As far as we know, there has been no other attempt to use automata to specify correctness of real-time systems. However, a large number of real-time extensions of temporal logics have been proposed.

Dense-time

The earliest proposal for specifying real-time requirements is found in a paper by Bernstein and Harter [BH81]. They define an extension of PTL by introducing operators such as $\phi \xrightarrow{<n} \psi$, meaning that "every ϕ-state is followed by a ψ-state within n time units". The paper gives some examples of specifications and proofs.

Koymans [KVdR83, Koy90] defines *metric temporal logic* as a specification language for time-critical systems. The logic allows temporal modalities for $\Diamond_{<3}$ to write down real-time requirements. The logic has a very rich and powerful syntax, and the definition of the semantics makes very weak assumptions about the time domain. His thesis gives specifications for many interesting problems.

In [Lew90], Lewis considers a real-time branching-time logic. The syntax is an extension of CTL with interval subscripts on temporal operators. The logic is interpreted over state diagrams — his abstraction for finite-state systems.

In [AH89] we showed that the dense-time semantics leads to undecidability of the satisfiability problem in the presence of operators such $\Diamond_{=3}$. Later in [AFH91] we found a decidable subset — the logic MITL which disallows singular interval subscripts. Thus, of all the temporal logics interpreted over dense models, only MITL is decidable.

Fictitious-clock

One approach to specifying real-time requirements using temporal logic is to employ first-order temporal logic where one of the state variables denotes the value of the global clock. The logic RTTL of Ostroff [Ost90b] uses this approach. It is based on the fictitious-clock semantics. The syntax uses a dynamic state variable T to denote the current time, and static variables over the time domain to express timing constraints. For example, the property that "every p-state is followed by some q-state within time 3," is expressed by the formula

$$\Box[\,(p \wedge T = x) \rightarrow \Diamond(q \wedge T \leq x + 3)].$$

The logic is undecidable even if we restrict to finite-state systems. Consequently, several different decidable fragments of RTTL have been identified.

In [AH89] we defined the logic TPTL. For decidability, TPTL requires that the timing constraints be of a very simple form, involving only comparisons and addition by constant values. Secondly, it achieves elementary complexity through a novel form of time quantifier "$x.$" which captures the current time. The notation also provides abstraction from explicit

references to the time variable. The syntax for TCTL used in this thesis is based upon this TPTL notation.

Another decidable fragment of RTTL is the logic MTL [AH90]. MTL uses Koyman's notation of subscripted temporal operators, and is interpreted over fictitious-clock models. Unlike TPTL, MTL also allows *past* operators such as $\diamondminus_{<3}$ meaning "sometime within the past 3 time units".

The logic XCTL introduced by Pnueli and Harel [PH88, Har88] is also a restricted fragment of RTTL, and is decidable. Unlike all the formalisms we consider, XCTL allows the addition primitive; however it allows only one form of quantification, and consequently, the logic is not closed under negation.

For a detailed comparison of the above logics see [AH90] or [Hen91].

Discrete-time

As an example of a real-time logic with discrete-time semantics consider the branching-time logic RTCTL of [EMSS89]. In this logic one can write formulas such as $\exists\Diamond^{\leq 4} p$, meaning some p-state is reachable within time 4. Since each transition takes unit time, the real-time operators are merely abbreviations for sequences of next operators of CTL. The paper gives complexity results on satisfiability and model-checking for RTCTL.

Mok and his group have done extensive work on specifying real-time systems. The logic RTL is an event-based logic which allows stating relationships between occurrence times of different events [JM86, JM87]. The specification style of RTL is quite different from the conventional temporal logics. For instance, the RTL formula

$$\forall i.\, [\, @(b,i) \,\leq\, (@a, i+5)\,]$$

states the property that "the i-th occurrence of the b event is within 5 time units of the i-th occurrence of the a event, for all choices of i." Theoretically, it is an extension of Presburger arithmetic with an uninterpreted unary function symbol corresponding to every event symbol. RTL is undecidable; there seems to be no natural decidable fragment.

1.4.3 Verification

Dill [Dil89] gives an algorithm to check *qualitative* temporal properties of finite-state systems. The essential construction involves checking consistency of timing constraints of a

timed automaton with timers. The untiming construction for timed automata presented in this thesis has a better worst-case running time.

Lewis also gives an algorithm to check consistency of timing information for a system modeled by his state diagrams. In [Lew90], he gives an algorithm to check properties written in a branching-time logic that is a fragment of TCTL. The algorithm is quite different from ours, and makes the assumption of *progressiveness*: in a bounded interval of time only a bounded number of events happen (note that this is not the same as the discrete-time assumption). Also the worst-case complexity of our algorithm is better.

Lynch and Attiya [LA90] provide a formal basis for comparing two descriptions, not necessarily finite-state, presented as timed I/O automata and show how to use it for reasoning about timing properties of protocols.

Ostroff [Ost90b] extends the proof system for temporal logic to handle RTTL formulas. He also gives algorithms for checking a restricted class of RTTL specifications against systems modeled using real-time transition systems [Ost90a].

Henzinger's thesis [Hen91] studies several aspects of the verification problem based on fictitious-clock temporal logics. The model-checking algorithms for the finite-state case are reported in [AH89, AH90, HLP90]. He also gives axiomatization for MTL [Hen90], and proof rules for checking certain types of real-time specifications such as bounded response and bounded invariance [HMP91]. With its integration in the existing proof systems for temporal logic, it provides a general system for reasoning about real-time programs.

The algebraic approaches [Zwa88, RR88, Wan90, NRSV90] provide an array of operators and laws for reasoning with them. The verification is defined using the notions of process containment and process equivalence. In the context of timed Petri nets [Ram74, CR83] analysis techniques for solving specific performance-related problems have been considered for subclasses. Verification methods based on time Petri nets have been considered in [BD91, YKT91]. Aggarwal and Kurshan [AK83] using their timing extension of the selection/resolution model, show how to compute elapsed time between different events, and argue that the timing information helps to reduce the number of reachable states.

In the context of RTL, Jahanian and Mok [JM86] show how to do safety analysis of timing properties. In [JS88] the semantics for Modecharts is defined using RTL-formulas, thus reducing the verification problem to proving a theorem in RTL. For RTL specifications of a specific form algorithms for model-checking have been developed.

1.5 Organization of the thesis

In Chapter 2, we consider three alternative models for real-time, namely, discrete-time, dense-time, and fictitious-clock, in the context of trace semantics. We give justification for our choice of model on the basis of issues such as correctness, expressiveness, complexity and compositionality.

In Chapter 3, we develop a theory of *timed automata*. We define the formalism, and study its closure properties. We consider automata with both Büchi and Muller acceptance conditions. The main results include a product construction for timed automata, a decision procedure for testing emptiness, the undecidability of language inclusion, and the subclass of deterministic timed Muller automata for which the language inclusion is solvable. We also consider extensions, variants, and expressiveness issues. We show how this theory can be used to model, specify, and verify real-time systems.

In Chapter 4, we consider a real-time extension of the linear-time temporal logic PTL. We define the logic *metric interval temporal logic* (MITL) by extending the syntax of PTL with nonsingular interval subscripts for the temporal operators. We also define *interval automata*, a state-based variant of timed automata, as a model for finite-state systems. We reduce the satisfiability problem for MITL to the emptiness problem for interval automata, and show the logic to be EXPSPACE-complete. We also present a model-checking algorithm for MITL specifications. We consider variants of this logic, and show how the choice of syntax and semantics affects decidability. Finally we compare the expressiveness of MITL to that of the fictitious-clock temporal logic MTL.

Chapter 5 is devoted to the study of the logic *timed computation tree logic* (TCTL), a real-time extension of the branching-time logic CTL. We define the semantics of *continuous computation trees* for TCTL. We show that, although the denseness makes the satisfiability problem for TCTL undecidable, the model-checking problem (that is, the problem of checking TCTL specifications against an interval automaton) is decidable in PSPACE.

The concluding chapter indicates some directions for ongoing and future research.

Joint work

Timed automata of Chapter 3 were first introduced in a joint paper with David Dill presented at the *17th International Colloquium on Automata, Languages, and Programming*

in June 1990 [AD90]. The results of Chapter 4 appear in a joint paper with Thomas Henzinger and Tomas Feder [AFH91] presented at the *Tenth ACM Symposium on Principles of Distributed Computing* in August 1991 [AFH91]. Chapter 5 generalizes the definition and the results appearing in a joint paper with Costas Courcoubetis and David Dill at the *Fifth IEEE Symposium on Logic in Computer Science* in June 1990 [ACD90].

Chapter 2

Adding Time to Semantics

In this chapter we define three different ways of introducing time in linear trace semantics for concurrent processes. We compare these models, and justify our preference for the dense-time model.

2.1 Trace semantics

In trace semantics, we associate a set of observable *events* with each process, and model the process by the set of all its *traces*. A trace is a (linear) sequence of events that may be observed when the process runs. For example, an event may denote an assignment of a value to a variable, or pressing a button on the control panel, or arrival of a message. All events are assumed to occur instantaneously. Actions with duration are modeled using events marking the beginning and the end of the action. Hoare originally proposed such a model for CSP [Hoa78]. In our model, we allow several events to happen simultaneously. Also we consider only infinite sequences, which model nonterminating interaction of reactive systems with their environments. This is no serious limitation; a finite sequence representing a terminating behavior can be extended to an infinite sequence using a suffix comprising of infinite repetition of a dummy event.

Formally, given a set A of events, a *trace* $\sigma = \sigma_1\sigma_2\ldots$ is an infinite word over $\mathcal{P}^+(A)$ — the set of nonempty subsets of A. An *untimed process* is a pair (A, X) comprising of the set A of its observable events and the set X of its possible traces.

Example 2.1 Consider a channel P connecting two components. Let a represent the arrival of a message at one end of P, and let b stand for the delivery of the message at the

other end of the channel. The channel cannot receive a new message until the previous one has reached the other end. Consequently the two events a and b alternate. Assuming that the messages keep arriving, the only possible trace is

$$\sigma_P \ : \ \{a\} \rightarrow \{b\} \rightarrow \{a\} \rightarrow \{b\} \rightarrow \cdots .$$

Often we will denote the singleton set $\{a\}$ by the symbol a. The process P is represented by $(\{a,b\},(ab)^\omega)$ [1]. ■

Various operations can be defined on processes; these are useful for describing complex systems using the simpler ones. We will consider only the most important of these operations, namely, parallel composition. The parallel composition of a set of processes describes the joint behavior of all the processes running concurrently. In our framework, processes synchronize via common events, and concurrency is modeled by all possible interleavings of the causally independent events.

The parallel composition operator can be conveniently defined using the projection operation. The *projection* of $\sigma \in \mathcal{P}^+(A)^\omega$ onto $B \subseteq A$ (written $\sigma\lceil B$) is formed by intersecting each event set in σ with B and deleting all the empty sets from the sequence. For instance, in Example 2.1 $\sigma_P\lceil\{a\}$ is the trace a^ω. Notice that the projection operation may result in a finite sequence; but we will consider the projection of a trace σ onto B only when $\sigma_i \cap B$ is nonempty for infinitely many i.

For a set of processes $\{P_i = (A_i, X_i) \mid i = 1, 2, \ldots n\}$, their *parallel composition* $\|_i \ P_i$ is a process with the event set $\cup_i A_i$ and the trace set

$$\{\sigma \in \mathcal{P}^+(\cup_i A_i)^\omega \mid \ \wedge_i \ \sigma\lceil A_i \in X_i\}.$$

Thus σ is a trace of $\|_i \ P_i$ iff $\sigma\lceil A_i$ is a trace of P_i for each $i = 1, \ldots n$. When there are no common events the above definition corresponds to the unconstrained interleavings of all the traces. On the other hand, if all event sets are identical then the trace set of the composition process is simply the set-theoretic intersection of all the component trace sets.

Example 2.2 Consider another channel Q connected to the channel P of Example 2.1. The event of message arrival for Q is same as the event b. Let c denote the delivery of the message at the other end of Q. The process Q is given by $(\{b,c\},(bc)^\omega)$.

[1] We will use the notation of ω-regular expressions freely. The expression e^* stands for a finite repetition of e, and the expression e^ω stands for an infinite repetition of e.

When P and Q are composed we require them to synchronize on the common event b, and between every pair of b's we allow the possibility of the event a happening before the event c, the event c happening before a, and both occurring simultaneously. Thus $[\, P \parallel Q \,]$ has the event set $\{a, b, c\}$, and has an infinite number of traces. An example trace is

$$\{a\} \rightarrow \{b\} \rightarrow \{c\} \rightarrow \{a\} \rightarrow \{b\} \rightarrow \{a, c\} \rightarrow \{b\} \rightarrow \{a\} \rightarrow \{c\} \rightarrow \{b\} \rightarrow \cdots$$

■

In this framework, the verification question is presented as an inclusion problem. Both the implementation and the specification are given as untimed processes. The implementation process is typically a composition of several smaller component processes. We say that an implementation (A, X_I) is *correct* with respect to a specification (A, X_S) iff $X_I \subseteq X_S$.

Example 2.3 Consider the channels of Example 2.2. The implementation process is $[\, P \parallel Q \,]$. The specification is given as the process $S = (\{a, b, c\}, (abc)^\omega)$. Thus the specification requires the message to reach the other end of Q before the next message arrives at P. In this case, $[P \parallel Q]$ does not meet the specification S, for it has too many other traces, specifically, the trace $ab(acb)^\omega$. ■

2.2 Timed traces

In this section we explore different possible definitions for introducing time in trace semantics.

2.2.1 Adding timing to traces

An untimed process models the sequencing of events but not the actual times at which the events occur. Thus the description of the channel in Example 2.1 gives only the sequencing of the events a and b, and not the delays between them. Timing can be added to a trace by coupling it with a sequence of time values. We assume that these values are chosen from a domain *TIME* with linear order $\leq$. Different choices for *TIME* will lead to different ways of modeling the behavior. The examples in this subsection will use the set of natural numbers as *TIME*.

A *time sequence* $\tau = \tau_1 \tau_2 \ldots$ is an infinite sequence of time values $\tau_i \in TIME$ with $\tau_i > 0$, satisfying the following constraints:

- *Monotonicity:* τ increases strictly monotonically; that is, $\tau_i < \tau_{i+1}$ for all $i \geq 1$.
- *Progress:* For all $t \in TIME$, there is some $i \geq 1$ such that $t < \tau_i$.

A *timed trace* over a set of events A is a pair (σ, τ) where σ is a trace over A, and τ is a time sequence.

In a timed trace (σ, τ), each τ_i gives the time at which the events in σ_i occur. In particular, τ_1 gives the time of the first observable event; we always assume $\tau_1 > 0$, and define $\tau_0 = 0$. Sometimes we will represent the timed trace (σ, τ) by the infinite sequence

$$(\sigma_1, \tau_1) \rightarrow (\sigma_2, \tau_2) \rightarrow (\sigma_3, \tau_3) \rightarrow \cdots$$

Observe that the progress condition implies that only a finite number of events can happen in a bounded interval of time. In particular, it rules out convergent time sequences such as $1/2, 3/4, 7/8, \ldots$ representing the possibility that the system participates in infinitely many events before time 1.

A *timed process* is a pair (A, L) where A is a finite set of events, and L is a set of timed traces over A.

Example 2.4 Consider the channel P of Example 2.1 again. Assume that the first message arrives at time 1, and the subsequent messages arrive at fixed intervals of length 3 time units. Furthermore, it takes 1 time unit for every message to traverse the channel. The process has a single timed trace

$$\rho_P = (a, 1) \rightarrow (b, 2) \rightarrow (a, 4) \rightarrow (b, 5) \rightarrow \cdots$$

and it is represented as a timed process $P^T = (\{a, b\}, \{\rho_P\})$. ■

The operations on untimed processes are extended in the obvious way to timed processes. To get the projection of (σ, τ) onto $B \subseteq A$, we first intersect each event set in σ with B and then delete all the empty sets along with the associated time values. The definition of parallel composition remains unchanged, except that it uses the projection for timed traces. Thus in parallel composition of two processes, we require that both the processes should participate in the common events at the same time. This rules out the possibility of interleaving: parallel composition of two timed traces is either a single timed trace or is empty.

Example 2.5 As in Example 2.2 consider another channel Q connected to P. For Q, as before, the only possible trace is $\sigma_Q = (bc)^\omega$. In addition, the timing specification of Q says that the time taken by a message for traversing the channel, that is, the delay between b and the following c, is always 1 unit. The timed process Q^T has infinitely many timed traces, and it is given by

$$[\,\{b, c\},\ \{((bc)^\omega, \tau) \mid \forall i.\,(\tau_{2i} - \tau_{2i-1} = 1)\}\,].$$

The description of $[\,P^T \parallel Q^T\,]$ is obtained by composing ρ_P with each timed trace of Q^T. However only one timed trace of Q^T is consistent with the timing of b events in ρ_P. The resulting process has the event set $\{a, b, c\}$, and a unique timed trace

$$(a,1) \rightarrow (b,2) \rightarrow (c,3) \rightarrow (a,4) \rightarrow (b,5) \rightarrow (c,6) \rightarrow \cdots$$

■

The time values associated with the events can be discarded by the *Untime* operation. For a timed process $P = (A, L)$, $\mathit{Untime}[(A, L)]$ is the untimed process with the event set A and the trace set consisting of traces σ such that $(\sigma, \tau) \in L$ for some time sequence τ.

Note that

$$\mathit{Untime}(P_1 \parallel P_2) \subseteq \mathit{Untime}(P_1) \parallel \mathit{Untime}(P_2).$$

However, as Example 2.6 shows, the two sides are not necessarily equal. In other words, the timing information retained in the timed traces constrains the set of possible traces when two processes are composed.

Example 2.6 Consider the channels of Example 2.5. Observe that $\mathit{Untime}(P^T) = P$ and $\mathit{Untime}(Q^T) = Q$. As seen before, $[\,P^T \parallel Q^T\,]$ has a unique trace $(abc)^\omega$. On the other hand, $[\,P \parallel Q\,]$ has infinitely many traces; between every pair of b events all possible orderings of an event a and an event c are admissible. ■

The verification problem is again posed as an inclusion problem. Now the implementation is given as a composition of several timed processes, and the specification is also given as a timed process.

Example 2.7 Consider the verification problem of Example 2.3 again. If we model the implementation as the timed process $[\,P^T \parallel Q^T\,]$ then it meets the specification S. The

specification S is now a timed process $(\{a,b,c\}, \{((abc)^\omega, \tau)\})$. Observe that, though the specification S constrains only the sequencing of events, the correctness of $[\, P^T \parallel Q^T \,]$ with respect to S crucially depends on the timing constraints of the two channels. Consider another specification S' which requires, in addition to S, the timing delay between every event a and the next following c to be at most 2. The implementation meets S' also. ■

2.2.2 Discrete-time model

In Section 2.2.1, while defining timed traces, we did not commit to choosing a specific time domain. Different choices for *TIME* give us different models of real-time.

Choosing *TIME* to be the set of natural numbers, N, gives us the *discrete-time* model. In this model events can happen only at the integer time values. This describes the behavior of *synchronous* systems, where all components are driven by a common global clock. The duration between the successive clock ticks is chosen as the time unit. The discrete-time model is the traditional model for synchronous hardware. The processes considered in Example 2.5 use this model.

The advantage of this model is its simplicity. In fact, timed traces are not even necessary to model the behavior. A timed trace (σ, τ) over a set of events A may be viewed as an infinite sequence σ' over 2^A: for all $i \geq 1$, let σ'_i be σ_j if $\tau_j = i$, and let σ'_i be $\emptyset$ if for all $j \geq 1$ $\tau_j \neq i$. Thus the i-th element of σ' gives the events happening at time i. For instance, the timed trace

$$(\{a\},1) \to (\{b\},2) \to (\{a,b\},4) \to (b,5) \to \cdots$$

is represented by the sequence

$$\{a\} \to \{b\} \to \emptyset \to \{a,b\} \to \cdots$$

Thus a timed process can be modeled as a set of traces over 2^A; the empty set of events corresponds to the passage of time. The parallel composition operator can be defined directly within this framework by modifying the definition of the projection operation on traces: the projection of a trace σ over 2^A onto $B \subseteq A$ is formed by simply intersecting each event set in σ with B (that is, empty sets are not deleted).

No substantially new techniques are needed to analyze the timing behavior in this model; the techniques used in the verification of untimed processes can be modified in a straightforward way.

2.2.3 Dense-time model

Choosing *TIME* to be the set of real numbers, R, gives the *dense-time* model. In this model, we assume that events happen at arbitrary points in time over the real-line, and with each event we associate its real-valued time of occurrence. As it turns out, with regards to complexity and expressiveness issues, the crucial aspect of the underlying domain is its denseness — the property that between every two time values there is a third one, and not its continuity; we may replace R by some other dense linear order, say, the set of rational numbers, Q.

The dense-time model is a natural model for asynchronous systems. It allows events to happen arbitrarily close to each other; that is, there is no lower bound on the separation between events. This is a desirable feature for representing two causally independent events in an asynchronous system. While defining the semantics of a system, no assumptions regarding the speed of the environment need to be made.

Example 2.8 Let us consider the channels P and Q of Example 2.5 again. We keep the timing specification of the channel P the same. For channel Q assume that the delay between the event b and the following c is some real value between 1 and 2. The timed process Q^T is now given by

$$[\,\{b, c\},\ \{(\sigma_Q, \tau) \mid \forall i.\,(\tau_{2i-1} + 1 < \tau_{2i} < \tau_{2i-1} + 2)\}\,].$$

The composition process $[\, P^T \parallel Q^T \,]$ has uncountably many timed traces. An example trace is

$$(a, 1) \rightarrow (b, 2) \rightarrow (c, 3.8) \rightarrow (a, 4) \rightarrow (b, 5) \rightarrow (c, 6.02) \rightarrow \cdots$$

■

2.2.4 Fictitious-clock model

In this section we consider an alternative way to introduce time in traces. We assume that there is an external, discrete clock ticking at a fixed rate, asynchronously with the other components in the system. Time is viewed as a discrete counter, and is incremented by one with every tick of this fictitious clock. With each event we associate, instead of its exact (real) time of occurrence, the value of this counter. Thus events happening between consecutive ticks have the time-stamp, and only the ordering of their times of occurrences is known.

We formalize this model using *observation traces*. An observation trace over a set of events A consists of a trace σ over A and an infinite sequence $\tau = \tau_1\tau_2\tau_3\ldots$ over $\mathbb{N}$ satisfying the progress constraint and the weak monotonicity constraint that $\tau_i \leq \tau_{i+1}$ for all $i \geq 1$. Thus an observation trace is similar to a timed trace for the discrete-time model, but instead of requiring the time sequence to be strictly increasing we simply require it to be nondecreasing. If τ_i equals τ_{i+1}, it means that the events in the $(i+1)$-th set happen after the events in the i-th set, but before the next clock tick. Also τ_1 may be 0 indicating that the first event happens before the first clock tick. An alternative way to define the fictitious-clock semantics would be to use the model of untimed processes with a special *tick* event common to all processes. Thus a timed process with an event set A is modeled as an untimed process with the event set $A \cup \{tick\}$.

Example 2.9 Consider the channel P with alternating events a and b. In addition we know that an observer with a clock ticking at fixed intervals observes at most 1 tick between every event a and the successive b, and precisely 2 ticks between every pair of successive a's. The timed process P^T is given by

$$[\{a,b\}, \{((ab)^\omega, \tau) \mid \forall i.\, (\tau_{2i} \leq \tau_{2i-1} + 1) \wedge (\tau_{2i+1} = \tau_{2i-1} + 2)\}].$$

An example observation trace is

$$(a,0) \rightarrow (b,0) \rightarrow (a,2) \rightarrow (b,3) \rightarrow (a,4) \rightarrow \cdots$$

The process can be represented using *tick* events also. For instance, the above trace can be represented by the trace $a, b, tick, tick, a, tick, b, tick, a \ldots$ ■

The projection operation is defined for observation traces as in the case of timed traces. Also the definition of the parallel composition operator is unchanged. Thus when composing two observation traces of two processes, we require synchronization on every successive clock tick in addition to the common events. As in the qualitative model, we get all possible interleavings of the events in the two traces between every pair of successive clock ticks. Consequently, the result of composing two behaviors is not unique as in the discrete-time or the dense-time model, but is more constrained than the qualitative model because of the additional synchronization of ticks.

Example 2.10 Consider another channel Q as in our previous examples. The timing specification of Q requires precisely 1 tick between every event b and the next c. An example observation trace of Q^T is

$$(b,0) \rightarrow (c,1) \rightarrow (b,3) \rightarrow (c,4) \rightarrow \cdots$$

Composing this trace with the observation trace of P^T shown in Example 2.9 allows arbitrary ordering of the events a and c at time 4. For instance, the composition contains the trace

$$(a,0) \rightarrow (b,0) \rightarrow (c,1) \rightarrow (a,2) \rightarrow (b,3) \rightarrow (a,4) \rightarrow (c,4) \rightarrow \cdots$$

In this case $Untime[\, P^T \parallel Q^T \,]$ equals $[\, P \parallel Q \,]$. ■

The fictitious-clock model can be considered as a generalization of the discrete-time model where successive event sets can have the same time. This model allows arbitrarily many transitions between the successive tick events, and hence, unlike the discrete-time model, makes no assumptions regarding the speed of the environment. On the other hand, it can be viewed as an approximation to the dense-time model where the time value associated with each event set is truncated. The observation traces only record the observations of the actual behaviors with respect to a discrete clock. Since time is considered discrete, the verification algorithms based on this model are simpler compared to those based on the dense-time model. The techniques used in the verification of finite-state (untimed) processes can be adopted to develop verification methods for this model.

However since only incomplete timing information is retained it cannot model the timing delays of a system accurately. The timing delay between two events is measured by counting the number of *tick*s between them. When we require that there be k *tick*s between two transitions, we can only infer that the delay between them is larger than $k-1$ time units and smaller than $k+1$ time units. Consequently, it is impossible to state precisely certain simple requirements on the delays such as "the delay between two transitions equals 2 seconds." This leads to some unintuitive properties in the model. For instance, consider the property:

> if a precedes b, and c happens 1 time unit later than a, and d happens 1 unit later than b, then c precedes d.

This is a valid property of timed traces in the discrete-time or the dense-time models, but it is not a valid property of observation traces in the fictitious-clock model. The observation

trace

$$(a,0) \rightarrow (b,0) \rightarrow (d,1) \rightarrow (c,1) \rightarrow \cdots$$

satisfies all the conditions in the antecedent, but violates the precedence requirement of the consequent.

2.3 A case for the dense-time Model

Several researchers have used either the discrete-time or the fictitious-clock model saying that it is good enough for all practical purposes provided the unit for time (the rate at which the clock ticks) is small enough. However, this argument has not been supported by any precise mathematical claims. Below we make an attempt to give different justifications to our preference for the dense-time model over the other two.

2.3.1 Correctness

For event-driven asynchronous systems the dense-time model gives different results compared to the other models. We discuss this issue in context of a reachability problem for asynchronous circuits with bounded inertial delays. The network model we use and some of the examples are borrowed from [BS91].

A network N consists of single-output gates connected by wires $1, 2, \ldots m$. Each gate can be identified by its unique output wire. Wires are assumed to have no delays, however each gate is assumed to have a bounded delay; the delay of a gate j can be an arbitrary real value in the interval $[l(j), u(j)]$. Thus $l(j)$ gives the lower bound for the delay, and $u(j)$ gives the upper bound. All the bounds are nonnegative integers. Wires are assumed to have only binary states, and the changes are assumed to be instantaneous. A state s of the circuit is an m-tuple over $\{0, 1\}$ giving the values of all the wires; the j-th component $s(j)$ represents the value of the wire j. The value assigned to the output wire of a gate by a state may not be consistent with the values assigned to its input wires. For a state s and a gate j, let $out(s, j)$ denote the required value of the wire j according to the values of the input wires to gate j in state s and the functional laws for the gate j. For instance, if a state s assigns the value 0 to the input wire of an inverter gate j, then $out(s, j)$ is 1. If $s(j)$ differs from $out(s, j)$ then the gate j is unstable in state s.

A state s_n is reachable from a state s_0 iff there exists a sequence of the form

$$s_0 \xrightarrow[\tau_1]{A_1} s_1 \xrightarrow[\tau_2]{A_2} \cdots\cdots s_{n-1} \xrightarrow[\tau_n]{A_n} s_n$$

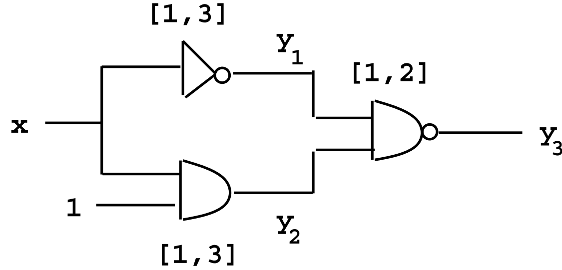


Figure 2.1: Circuit C_1

of states s_i, increasing time values $\tau_i \in \mathsf{R}$, and nonempty sets of gates A_i. The state changes from s_{i-1} to s_i due to the toggling of the output wires of gates in A_i. Hence we require that $s_{i-1}(j)$ and $s_i(j)$ differ iff $j \in A_i$. Note that the states s_1 through s_n can be inferred from the initial state s_0 and the sets $A_1, \ldots A_n$. The timing requirements are given by the following two laws:

1. *Lower Bound:* Each gate must be unstable for at least $l(j)$ time units before it changes its value: if the gate j changes its value at time τ_i (that is, $j \in A_i$) then for some k with $\tau_k \leq \tau_i - l(j)$, the gate j is unstable in all the intermediate states during the time period $[\tau_k, \tau_i)$ (that is, $out(s_{k'}, j) \neq s_{k'}(j)$ for all $k \leq k' < i$), and its value stays unchanged during this period (that is, $j \notin A_{k'}$ for $k < k' < i$).

2. *Upper Bound:* A gate cannot be unstable for $u(j)$ time units without changing its value: for every gate j, and every pair of transition points k_1 and k_2 such that $\tau_{k_1} + u(j) \geq \tau_{k_2}$, if the gate j is unstable in all the intermediate states between k_1 and k_2 (that is, $out(s_k, j) \neq s_k(j)$ for all $k_1 \leq k < k_2$), then the gate must have changed its value sometime during this period (that is, $j \in A_k$ for all $k_1 < k < k_2$).

Given a network N with an initial state s_0, let $R(N, s_0)$ denote the set of states reachable from s_0. If s_0 is stable with respect to every gate then clearly no other state is reachable. If s_0 is unstable then $R(N, s_0)$ gives all the states the network can possibly visit before stabilizing.

Example 2.11 Consider the circuit C_1 shown in Figure 2.1. The gates are labeled with the lower and upper bounds for the delays. The wire x is the input wire, and y_1, y_2 and y_3

give the wires corresponding to the three gates. The wire y_1 is the output of the inverter gate, y_2 is the output of the AND-gate, and y_3 is the output of the NAND-gate. Consider the stable state with $x = 0$ and $y = [101]$. Now suppose the input changes to $x = 1$. The stable state is $y = [011]$. However, before this state is reached the gates can change their values in different orders. A possible behavior of the circuit is shown below (only the values for y_1, y_2, and y_3 are listed):

$$[101] \xrightarrow[1.2]{\{y_2\}} [111] \xrightarrow[2.5]{\{y_3\}} [110] \xrightarrow[2.8]{\{y_1\}} [010] \xrightarrow[4.5]{\{y_3\}} [011]$$

In the above behavior, the outputs of the AND-gate and the NAND-gate change before the output of the inverter changes. The inverter responds with a delay 2.8, the AND-gate responds with a delay 1.2. The NAND-gate responds with a delay of 1.3 when the wire y_2 changes, and with a delay of 1.7 when the wire y_1 changes.

Another possible behavior is when the inverter changes its output before the AND-gate toggles, giving the state [001], which then changes to the stable state [011].

The set of reachable states is given by

$$R(C_1, [x = 1, y = 101]) = \{[101], [111], [110], [010], [011], [001]\}.$$

■

In the dense-time model, the transition times can be arbitrary real numbers. Clearly, an analysis based on this model can compute the set $R(N, s_0)$ correctly. Note that if we require all the transition times to be rationals, the set of reachable states will remain unchanged. The question is whether this set can be computed by other models, if we choose a sufficiently fast clock.

Consider the discrete-time model with a clock ticking at fixed intervals of length $(1/k)$ time units. With the assumption that gates can change values only with the ticks of the clock, this model requires the delays for all gates and the transition times to be multiples of $(1/k)$. Let $R^d(N, s_0, k)$ denote the set of reachable states based on this discrete-time model. It is clear that $R^d(N, s_0, k) \subseteq R^d(N, s_0, k+1) \subseteq R(N, s_0)$ for all $k \geq 1$. From the examples given in [BS91] it follows that for all $k \geq 1$, there exists a network N with an initial state s_0 such that $R^d(N, s_0, k) \neq R(N, s_0)$. Example 2.12 shows that $R^d(N, s_0, 1) \neq R(N, s_0)$.

Example 2.12 [from BS91]

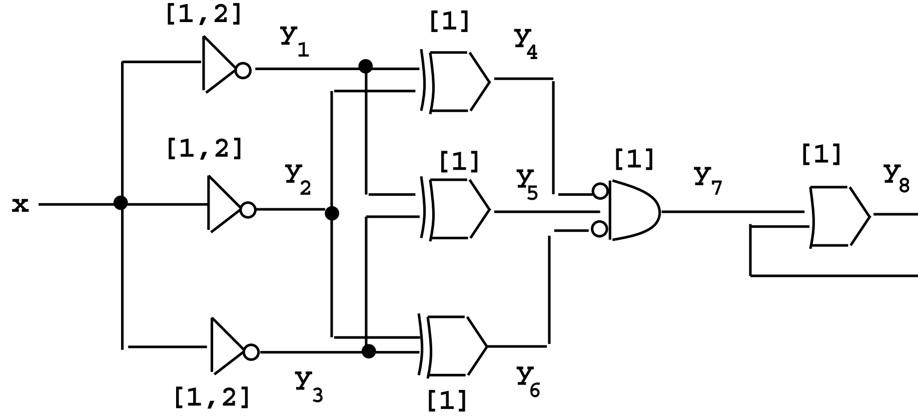


Figure 2.2: Circuit C_2

Consider the circuit C_2 of Figure 2.2. The gates 4, 5, and 6 are *exclusive-or* (XOR) gates. The gates labeled with [1] have delay precisely equal to 1 time unit.

Initially the cicuit is in the stable state $x = 0$, and $y = 11100000$. Now consider the behavior of the circuit when the input x changes to 1. First observe that a state in which the output y_8 of the OR-gate toggles is reachable if we choose the delays for the inverters 1, 2, and 3 to be 1, 1.5, and 2 time units, respectively. In this case the XOR-gates 4 and 6 are unstable only for an interval of length 0.5, and hence do not change their outputs. On the other hand, the XOR-gate 5 is unstable for 1 time unit, and hence, y_5 changes its value. Consequently, the gate 7 is unstable for 1 time unit, and the wire y_7 toggles. This propagates to the OR-gate, and y_8 toggles.

On the other hand, if we require the delays for the inverters to be discrete values, either 1 or 2, it follows that the XOR-gate 5 is unstable for 1 time unit iff at least one of the XOR-gates 4 or 6 is unstable for 1 time unit. This means that the gate 7, and consequently the OR-gate 8, are never unstable. Thus in every reachable state in $R^d(C_2, [x = 1, y = 11100000], 1)$ the value of y_8 is 0. ■

Thus the reachability analysis based on the discrete-time model will not be able to detect some reachable states if k is fixed *a priori*. Notice that for the circuit C_2 of Example 2.2, a discrete-time analysis using a time granularity of 0.5 time units detects all the transient states. In general, for each network N there is some k for which $R^d(N, s_0, k) = R(N, s_0)$.

Because of this, one may argue that discrete-time model is good enough, provided that the time unit is not fixed in advance. But this approach has its own problems. Firstly, this appropriate value of the granularity k is not obvious, and as far as we know nobody has proposed an algorithm to find this value. Indeed, the algorithm is likely to be as complex as the algorithms for the dense-time analysis presented in this thesis. Secondly, increasing the granularity adversely affects the complexity of the verification algorithms. Thirdly, for properties more complex than reachability, in particular those involving infinite traces, such a result does not hold. In Section 3.2 we show that such a result does not hold for timed automata. We give an example of a timed automaton whose trace set is empty if the transition times are required to be multiples of $(1/k)$, irrespective of the choice of k, but is nonempty with the dense-time semantics.

Now consider the fictitious-clock model with a clock ticking at the rate of $(1/k)$ time units. Now the transition times are multiples of $(1/k)$ as in the discrete-time model, but the sequence of time values is required to be only nondecreasing. The lower bound requirement says that once a gate becomes unstable there need to be at least $l(j)$ ticks before it can change its state, and the upper bound requirement says that there cannot be more than $u(j)$ ticks with some gate staying unstable. The actual lower bound on the delay is $(l(j) - 1/k)$, and the upper bound is $(u(j)+1/k)$. Let $R^f(N, s_0, k)$ denote the set of reachable states using the analysis based on this model. It can be shown that $R^f(N, s_0, k+1) \supseteq R^f(N, s_0, k) \supseteq R(N, s_0)$ for all $k \geq 1$. To see the second containment, observe that truncating all the transition times from a real behavior gives a behavior permitted by the fictitious-clock model. However, since the delays are modeled only imprecisely, the fictitious-clock model admits some extra behaviors also. The following example shows that there are networks for which $R^f(N, s_0, k)$ differs from $R(N, s_0)$ for all $k \geq 1$.

Example 2.13 Consider the circuit C_3 of Figure 2.3 with two inverter gates each of delay 1. Suppose initially $x = 0$ and $y = [11]$. Now if the input x changes to 1 then in the dense-time model both the outputs y_1 and y_2 will change to 0 at time 1, and hence,

$$R(C_3, [x = 1, y = 11]) = \{[11], [00]\}.$$

On the other hand, in the fictitious-clock model with $k = 1$, the following is an admissible timed state sequence:

$$[11] \xrightarrow[1]{\{y_1\}} [01] \xrightarrow[1]{\{y_2\}} [00].$$

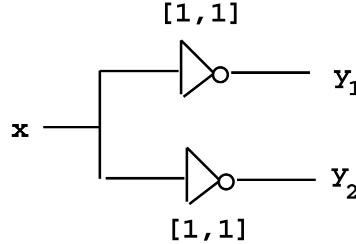


Figure 2.3: Circuit C_3

Increasing the value of the parameter k is of no use — independent of its choice, there is no way to model the fact that the delays of the two inverters are the same. In fact, for all $k \geq 1$,

$$R^f(C_3, [x = 1, y = 11], k) = \{[11], [01], [10], [00]\}.$$

■

Thus for the reachability analysis, the set of reachable states computed using a fictitious-clock approximation is a subset of the set computed using the dense-time analysis. For more complex properties, the relationship between the results given by the two models is not well understood. Henzinger [Hen91] shows that an analysis based on the fictitious-clock model can be used to solve the dense-time verification problem for systems and specifications of a particular type.

2.3.2 Expressiveness

The dense-time model is the most general of all the three.

The discrete-time model is a special case of the dense-time model. While modeling a discrete-time system within a formalism based on dense-time semantics, we add an extra clock process that ticks after a fixed interval, and require events of all other processes to be synchronized with the ticks. Similarly, in a specification language with dense-time semantics, if we add the requirement "$\tau_{i+1} = \tau_i + 1$ for all i," we get the discrete-time models. In all specification formalisms studied in this thesis this requirement is expressible.

The fictitious-clock model also can be simulated in the dense-time framework. While modeling a system, we add an extra clock process that ticks at a fixed rate. The timing

constraints of the system are modified so that they count ticks instead of the elapsed time. The same trick can be applied to the specification languages also. In Section 4.6 we show that the real-time logic MITL, which uses a dense-time semantics and disallows the use of equality constraints, is more expressive than the real-time logic MTL which uses the fictitious-clock model, irrespective of the choice of the separation between the successive clock ticks.

2.3.3 Compositionality

The main problem in defining semantics with the discrete models is that the semantics cannot be defined without fixing the time unit, but a reasonable choice for the time unit depends upon the timing constraints of all the components and the property to be verified. Consequently, a component cannot be given its own semantics independently of the other components. Furthermore, the semantic definition depends on the specification also. This is because, as Example 2.14 illustrates, the semantics of timed traces does not have enough information to refine the time unit. The example uses the discrete-time model, but the fictitious-clock model has the same problem.

Example 2.14 Consider a process P with two events a and b which alternate. The timing specification says that the delay between a and the following b is 1 second. Consider the discrete-time semantics for P with time unit equal to 1 second. P is represented by the set $T(P)$.

Consider another process Q with the same alternating events a and b. Its timing specification says that the delay between every occurrence of a and the following b is 1 second, and furthermore, the delay between b and the following a is at least 1 second. The process Q is modeled as a set of timed traces $T(Q)$. It is clear that $T(P) = T(Q)$. Thus within trace semantics both P and Q have same properties. This is acceptable as long as the time unit is 1 second.

Now suppose we introduce a third process R with time unit 0.5 second. In order to compose P and R, one needs to change the semantics of P to reflect the change of time unit. Let $T'(P)$ denote the semantics for P with this new time unit. If (σ, τ) is a timed trace in $T(P)$ then $(\sigma, 2\tau)$ is a timed trace in $T'(P)$. All such timed traces require the event a to happen at even time values, but $T'(P)$ has additional timed traces like

$$(a, 1) \rightarrow (b, 3) \rightarrow (a, 4) \rightarrow (b, 6) \rightarrow \cdots$$

Thus $T'(P)$ cannot be obtained from $T(P)$, one needs to refer to the original description of P to define $T'(P)$.

Also now $T'(Q)$ is different from $T'(P)$; above timed trace belongs only to $T'(P)$. Thus P and Q are no longer equivalent. This shows that properties need to be reproved when the time unit is refined. ■

The above example shows that in the discrete models one can reason about a system only after the descriptions of all the components and all the properties to be verified are known.

For compositional reasoning, we should be able to define the semantics of a component and prove its properties without knowing the details of the other parts of the system. Shifting to dense-time semantics offers a natural and mathematically clean way to this effect. In this model, the composition operator can be defined on timed processes describing different components in a straightforward way. Also the properties proved for one component hold independently of the speeds of the others.

2.3.4 Complexity

Since the dense-time model admits the possibility of an unbounded number of events in an interval of finite time length, some problems related to the verification of finite-state systems turn out to be, unlike the other models, undecidable. For example, the language inclusion problem for timed automata is undecidable; if we had chosen one of the discrete models, the problem would have been solvable. However, this problem does not turn out to be fatal. The thesis shows that checking properties of finite-state real-time systems is possible for several reasonably powerful specification languages based on the dense-time model. With the availability of these positive results, we hope that more researchers will explore the dense-time model further.

For the problems that are decidable, the complexity is almost the same for all the three models. As an example, let us consider the following model-checking problem. The system I is described as a composition of several components, each with its delay properties. The correctness specification S is given as a formula of a real-time extension of the branching-time logic CTL. The problem of testing whether I satisfies S is PSPACE-complete irrespective of the choice of the model. In fact, even for the qualitative case which ignores all the timing constraints, the complexity is PSPACE in the descriptions of the component processes of

I (though polynomial in S). The factors contributing to the exponential complexity of the model-checking algorithm are different in each case. In all cases, the running time is proportional to the product of the number of states in individual components, and the length of the specification formula. When we add real-time, that is, when we account for the delay properties of the components and allow time bounds in the formula, the complexity blows up by a factor of the product of all the constants bounding the delays. This factor is independent of the time model we use. In the case of dense-time, we need to introduce one clock per each component to model independent simultaneously active delays, and as a result, the complexity blows up by an additional factor proportional to the factorial of the number of clocks. In the discrete models, all delays are measured with respect to only one clock, and hence this factor does not appear. On the other hand, in these models, one may be forced to choose a smaller time unit to get correct results. This increases the magnitudes of all the constants blowing up the complexity of the verification algorithm.

Chapter 3

Automata-Theoretic Approach

In this chapter we develop a theory of timed automata, and show how it can be used for formal verification.

3.1 ω-automata

In this section we will briefly review the relevant aspects of the theory of *ω-regular languages*.

The more familiar definition of a formal language is as a set of finite words over some given alphabet (see, for example, [HU79]). As opposed to this, an ω-language consists of infinite words. Thus an ω-language over an alphabet Σ is a subset of Σ^ω — the set of all infinite words over Σ. ω-automata provide a finite representation for certain types of ω-languages. An ω-automaton is essentially the same as a nondeterministic finite-state automaton, but with the acceptance condition modified suitably so as to handle infinite input words. Various types of ω-automata have been studied in the literature [Büc62, McN66, Cho74, Tho90]. We will mainly consider two types of ω-automata: Büchi automata and Muller automata.

A *transition table* A is a tuple $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{E} \rangle$, where Σ is an input alphabet, S is a finite set of automaton states, $\mathrm{S}_0 \subseteq \mathrm{S}$ is a set of start states, and $\mathrm{E} \subseteq \mathrm{S} \times \mathrm{S} \times \Sigma$ is a set of edges. The automaton starts in an initial state, and if $\langle s, s', a \rangle \in \mathrm{E}$ then the automaton can change its state from s to s' reading the input symbol a.

For $\sigma \in \Sigma^\omega$, we say that

$$r : \; s_0 \xrightarrow{\sigma_1} s_1 \xrightarrow{\sigma_2} s_2 \xrightarrow{\sigma_3} \cdots$$

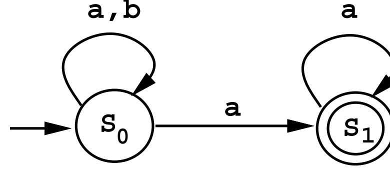


Figure 3.1: Büchi automaton accepting $(a+b)^*a^\omega$

is a *run* of A over σ, provided $s_0 \in \mathrm{S}_0$, and $\langle s_{i-1}, s_i, \sigma_i \rangle \in \mathrm{E}$ for all $i \geq 1$. For such a run, the set $inf(r)$ consists of the states $s \in \mathrm{S}$ such that $s = s_i$ for infinitely many $i \geq 0$.

Different types of ω-automata are defined by adding an acceptance condition to the definition of the transition tables. A *Büchi automaton* A is a transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{E} \rangle$ with an additional set $\mathrm{F} \subseteq \mathrm{S}$ of accepting states. A run r of A over a word $\sigma \in \Sigma^\omega$ is an *accepting run* iff $inf(r) \cap \mathrm{F} \neq \emptyset$. In other words, a run r is accepting iff some state from the set F repeats infinitely often along r. The language $L(\mathcal{A})$ accepted by A consists of the words $\sigma \in \Sigma^\omega$ such that A has an accepting run over σ.

Example 3.1 Consider the 2-state automaton of Figure 3.1 over the alphabet $\{a, b\}$. The state s_0 is the start state and s_1 is the accepting state. Every accepting run of the automaton has the form

$$r \ : \ s_0 \ \xrightarrow{\sigma_1} \ s_0 \ \xrightarrow{\sigma_2} \ \cdots \ \xrightarrow{\sigma_n} \ s_0 \ \xrightarrow{a} \ s_1 \ \xrightarrow{a} \ s_1 \ \xrightarrow{a} \ \cdots$$

The automaton accepts all words with only a finite number of b's; that is, the language $L_0 = (a+b)^*a^\omega$. ■

An ω-language is called *ω-regular* iff it is accepted by some Büchi automaton. Thus the language L_0 of Example 3.1 is an ω-regular language.

The class of ω-regular languages is closed under all the Boolean operations. Language intersection is implemented by a product construction for Büchi automata [Cho74, WVS83]. There are known constructions for complementing Büchi automata [SVW87, Saf88].

When Büchi automata are used for modeling finite-state concurrent processes, the verification problem reduces to that of language inclusion. The inclusion problem for ω-regular languages is decidable. To test whether the language of one automaton is contained in the other, we check for emptiness of the intersection of the first automaton with the complement

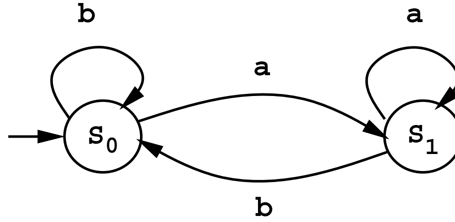


Figure 3.2: Deterministic Muller automaton accepting $(a+b)^*a^\omega$

of the second. Testing for emptiness is easy; we only need to search for a cycle that is reachable from a start state and includes at least one accepting state. In general, complementing a Büchi automaton involves an exponential blow-up in the number of states, and the language inclusion problem is known to be PSPACE-complete [SVW87]. However, checking whether the language of one automaton is contained in the language of a *deterministic* automaton can be done in polynomial time [Kur87].

A transition table $\mathcal{A} = \langle \pm, \mathrm{S}, \mathrm{S}_{\prime}, \mathrm{E} \rangle$ is *deterministic* iff (i) there is a single start state, that is, $|\mathrm{S}_0| = 1$, and (ii) the number of a-labeled edges starting at s is at most one for all states $s \in \mathrm{S}$ and for all symbols $a \in \Sigma$. Thus, for a deterministic transition table, the current state and the next input symbol determine the next state uniquely. Consequently, a deterministic automaton has at most one run over a given word. Unlike the automata on finite words, the class of languages accepted by deterministic Büchi automata is strictly smaller than the class of ω-regular languages. For instance, there is no deterministic Büchi automaton which accepts the language L_0 of Example 3.1. Muller automata (defined below) avoid this problem at the cost of a more powerful acceptance condition.

A *Muller automaton* A is a transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{E} \rangle$ with an *acceptance family* $\mathcal{F} \subseteq 2^{\mathrm{S}}$. A run r of A over a word $\sigma \in \Sigma^\omega$ is an *accepting run* iff $\mathit{inf}(r) \in \mathcal{F}$. That is, a run r is accepting iff the set of states repeating infinitely often along r equals some set in $\mathcal{F}$. The language accepted by A is defined as in case of Büchi automata.

The class of languages accepted by Muller automata is the same as that accepted by Büchi automata, and, more importantly, also equals that accepted by deterministic Muller automata.

Example 3.2 The deterministic Muller automaton of Figure 3.2 accepts the language L_0 consisting of all words over $\{a, b\}$ with only a finite number of b's. The Muller acceptance

family is $\{\{s_1\}\}$. Thus every accepting run can visit the state s_0 only finitely often. ∎

Thus deterministic Muller automata form a strong candidate for representing ω-regular languages: they are as expressive as their nondeterministic counterpart, and they can be complemented in polynomial time. Algorithms for constructing the intersection of two Muller automata and for checking the language inclusion are known [CDK89].

3.2 Timed automata

In this section we define timed words by coupling a real-valued time with each symbol in a word. Then we augment the definition of ω-automata so that they accept timed words, and use them to develop a theory of timed regular languages analogous to the theory of ω-regular languages.

3.2.1 Timed languages

Recall the definition of timed traces in Section 2.2.1. We define timed words so that a timed trace over the event set A is a timed word over the alphabet $\mathcal{P}^+(A)$. As in the case of the dense-time model, the set of nonnegative real numbers, R, is chosen as the time domain. A word σ is coupled with a time sequence τ as defined below:

Definition 3.3 A *time sequence* $\tau = \tau_1\tau_2\cdots$ is an infinite sequence of time values $\tau_i \in$ R with $\tau_i > 0$, satisfying the following constraints:

1. *Monotonicity:* τ increases strictly monotonically; that is, $\tau_i < \tau_{i+1}$ for all $i \geq 1$.
2. *Progress:* For every $t \in$ R, there is some $i \geq 1$ such that $\tau_i > t$.

A *timed word* over an alphabet Σ is a pair (σ, τ) where $\sigma = \sigma_1\sigma_2\ldots$ is an infinite word over Σ and τ is a time sequence. A *timed language* over Σ is a set of timed words over Σ. ∎

If a timed word (σ, τ) is viewed as an input to an automaton, it presents the symbol σ_i at time τ_i. If each symbol σ_i is interpreted to denote an event occurrence then the corresponding component τ_i is interpreted as the time of occurrence of σ_i. Let us consider some examples of timed languages.

Example 3.4 Let the alphabet be $\{a, b\}$. Define a timed language L_1 to consist of all timed words (σ, τ) such that there is no b after time 5.6. Thus the language L_1 is given by

$$L_1 \;=\; \{(\sigma, \tau) \mid \forall i.\, ((\tau_i > 5.6) \rightarrow (\sigma_i = a))\}.$$

Another example is the language L_2 consisting of timed words in which a and b alternate, and the time difference between the successive pairs of a and b keeps increasing. L_2 is given as

$$L_2 \;=\; \{((ab)^\omega, \tau) \mid \forall i.\, ((\tau_{2i} - \tau_{2i-1}) < (\tau_{2i+2} - \tau_{2i+1}))\}.$$

■

The language-theoretic operations such as intersection, union, complementation are defined for timed languages as usual. In addition we define the *Untime* operation which discards the time values associated with the symbols, that is, it considers the projection of a timed trace (σ, τ) on the first component.

Definition 3.5 For a timed language L over Σ, $Untime(L)$ is the ω-language consisting of $\sigma \in \Sigma^\omega$ such that $(\sigma, \tau) \in L$ for some time sequence τ. ■

For instance, referring to Example 3.4, $Untime(L_1)$ is the ω-language $(a + b)^* a^\omega$, and $Untime(L_2)$ consists of a single word $(ab)^\omega$.

3.2.2 Transition tables with timing constraints

Now we extend transition tables to *timed transition tables* so that they can read timed words. When an automaton makes a state-transition, the choice of the next state depends upon the input symbol read. In case of a timed transition table, we want this choice to depend also upon the time of the input symbol relative to the times of the previously read symbols. For this purpose, we associate a finite set of (real-valued) *clocks* with each transition table. A clock can be set to zero simultaneously with any transition. At any instant, the reading of a clock equals the time elapsed since the last time it was reset. With each transition we associate a clock constraint, and require that the transition may be taken only if the current values of the clocks satisfy this constraint. Before we define the timed transition tables formally, let us consider some examples.

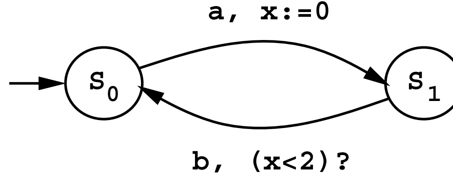


Figure 3.3: Example of a timed transition table

Example 3.6 Consider the timed transition table of Figure 3.3. The start state is s_0. There is a single clock x. An annotation of the form $x := 0$ on an edge corresponds to the action of resetting the clock x when the edge is traversed. Similarly an annotation of the form $(x < 2)?$ on an edge gives the clock constraint associated with the edge.

The automaton starts in state s_0, and moves to state s_1 reading the input symbol a. The clock x gets set to 0 along with this transition. While in state s_1, the value of the clock x shows the time elapsed since the occurrence of the last a symbol. The transition from state s_1 to s_0 is enabled only if this value is less than 2. The whole cycle repeats when the automaton moves back to state s_0. Thus the timing constraint expressed by this transition table is that the delay between a and the following b is always less than 2; more formally, the language is

$$\{((ab)^\omega, \tau) \mid \forall i.\, (\tau_{2i} < \tau_{2i-1} + 2)\}.$$

■

Thus to constrain the delay between two transitions e_1 and e_2, we require a particular clock to be reset on e_1, and associate an appropriate clock constraint with e_2. Note that clocks can be set asynchronously of each other. This means that different clocks can be restarted at different times, and there is no lower bound on the difference between their readings. Having multiple clocks allows multiple concurrent delays, as in the next example.

Example 3.7 The timed transition table of Figure 3.4 uses two clocks x and y, and accepts the language

$$L_3 \;=\; \{((abcd)^\omega, \tau) \mid \forall j.\, ((\tau_{4j+3} < \tau_{4j+1} + 1) \,\wedge\, (\tau_{4j+4} > \tau_{4j+2} + 2))\}.$$

The automaton loops between the states s_0, s_1, s_2 and s_3. The clock x gets set to 0 each time it moves from s_0 to s_1 reading a. The check $(x < 1)?$ associated with the c-transition

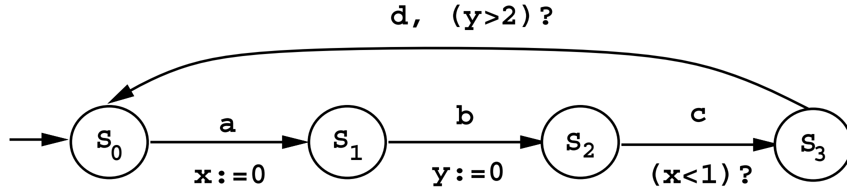


Figure 3.4: Timed transition table with 2 clocks

from s_2 to s_3 ensures that c happens within time 1 of the preceding a. A similar mechanism of resetting another independent clock y while reading b and checking its value while reading d, ensures that the delay between b and the following d is always greater than 2. ■

Notice that in the above example, to constrain the delay between a and c and between b and d the automaton does not put any bounds on the time difference between a and the following b, or c and the following d. This is an important advantage of having access to multiple clocks which can be set independently of each other. The above language L_3 is the intersection of the two languages L_3^1 and L_3^2 defined as

$$\begin{aligned} L_3^1 &= \{((abcd)^\omega, \tau) \mid \forall j.\, (\tau_{4j+3} < \tau_{4j+1} + 1)\}, \\ L_3^2 &= \{((abcd)^\omega, \tau) \mid \forall j.\, (\tau_{4j+4} > \tau_{4j+2} + 2)\}. \end{aligned}$$

Each of the languages L_3^1 and L_3^2 can be expressed by an automaton which uses just one clock; however to express their intersection we need two clocks.

We remark that the clocks of the automaton do not correspond to the local clocks of different components in a distributed system. All the clocks increase at the uniform rate counting time with respect to a fixed global time frame. They are fictitious clocks invented to express the timing properties of the system. Alternatively, we can view the automaton to be equipped with a finite number of stop-watches which can be started and checked independently of one another, but all stop-watches refer to the same clock.

3.2.3 Clock constraints and clock interpretations

To define timed automata formally, we need to say what type of clock constraints are allowed on the edges. The simplest form of a constraint compares a clock value with a time

constant. We allow only the Boolean combinations of such simple constraints. Any value from Q, the set of nonnegative rationals, can be used as a time constant. Later we will show that allowing more complex constraints, such as those involving addition of clock values, leads to undecidability.

Definition 3.8 For a set X of clocks, the set $\Phi(X)$ of *clock constraints* δ is defined inductively by

$$\delta := x \leq c \mid c \leq x \mid \neg\delta \mid \delta_1 \wedge \delta_2,$$

where x is a clock in X and c is a constant in Q. ■

Observe that constraints such as **true**, $(x = c)$, $x \in [2, 5)$ can be defined as abbreviations.

A *clock interpretation* ν for a set X of clocks assigns a real value to each clock; that is, it is a mapping from X to R. We say that a clock interpretation ν for X satisfies a clock constraint δ over X iff δ evaluates to true using the values given by ν.

For $t \in$ R, $\nu + t$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + t$, and the clock interpretation $t \cdot \nu$ assigns to each clock x the value $t \cdot \nu(x)$. For $Y \subseteq X$, $[Y \mapsto t]\nu$ denotes the clock interpretation for X which assigns t to each $x \in Y$, and agrees with ν over the rest of the clocks.

3.2.4 Timed transition tables

Now we give the precise definition of timed transition tables.

Definition 3.9 A *timed transition table* is a tuple $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$, where

- Σ is a finite alphabet,
- S is a finite set of states,
- $\mathrm{S}_0 \subseteq \mathrm{S}$ is a set of start states,
- C is a finite set of clocks, and
- $\mathrm{E} \subseteq \mathrm{S} \times \mathrm{S} \times \Sigma \times 2^{\mathrm{C}} \times \Phi(\mathrm{C})$ gives the set of transitions. An edge $\langle s, s', a, \lambda, \delta \rangle$ represents a transition from state s to state s' on input symbol a. The set $\lambda \subseteq \mathrm{C}$ gives the clocks to be reset with this transition, and δ is a clock constraint over C.

■

Given a timed word (σ, τ), the timed transition table A starts in one of its start states at time 0 with all its clocks initialized to 0. As time advances the values of all clocks change, reflecting the elapsed time. At time τ_i, A changes state from s to s' using some transition of the form $\langle s, s', \sigma_i, \lambda, \delta \rangle$ reading the input σ_i, if the current values of clocks satisfy δ. With this transition the clocks in λ are reset to 0, and thus start counting time with respect to it. This behavior is captured by defining *runs* of timed transition tables. A run records the state and the values of all the clocks at the transition points. For a time sequence $\tau = \tau_1 \tau_2 \ldots$ we define $\tau_0 = 0$.

Definition 3.10 A run r, denoted by $(\bar{s}, \bar{\nu})$, of a timed transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$ over a timed word (σ, τ) is an infinite sequence of the form

$$r : \langle s_0, \nu_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, \nu_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \langle s_2, \nu_2 \rangle \xrightarrow[\tau_3]{\sigma_3} \cdots$$

with $s_i \in \mathrm{S}$ and $\nu_i \in [\mathrm{C} \to \mathsf{R}]$, for all $i \geq 0$, satisfying the following requirements:

- *Initiation:* $s_0 \in \mathrm{S}_0$, and $\nu_0(x) = 0$ for all $x \in \mathrm{C}$.
- *Consecution:* for all $i \geq 1$, there is an edge in E of the form $\langle s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i \rangle$ such that $(\nu_{i-1} + \tau_i - \tau_{i-1})$ satisfies δ_i and ν_i equals $[\lambda_i \mapsto 0](\nu_{i-1} + \tau_i - \tau_{i-1})$.

The set $\mathit{inf}(r)$ consists of $s \in \mathrm{S}$ such that $s = s_i$ for infinitely many $i \geq 0$. ■

Example 3.11 Consider the timed transition table of Example 3.7. Consider a timed word

$$(a, 2) \to (b, 2.7) \to (c, 2.8) \to (d, 5) \to \cdots$$

Below we give the initial segment of the run. A clock interpretation is represented by listing the values $[x, y]$.

$$\langle s_0, [0,0] \rangle \xrightarrow[2]{a} \langle s_1, [0,2] \rangle \xrightarrow[2.7]{b} \langle s_2, [0.7, 0] \rangle \xrightarrow[2.8]{c} \langle s_3, [0.8, 0.1] \rangle \xrightarrow[5]{d} \langle s_0, [3, 2.3] \rangle \cdots$$

■

Along a run $r = (\bar{s}, \bar{\nu})$ over (σ, τ), the values of the clocks at time t between τ_i and τ_{i+1} are given by the interpretation $(\nu_i + t - \tau_i)$. When the transition from state s_i to s_{i+1} occurs, we use the value $(\nu_i + \tau_{i+1} - \tau_i)$ to check the clock constraint; however, at time τ_{i+1}, the value of a clock that gets reset is defined to be 0.

Note that a transition table $\mathcal{A} = \langle \pm, \mathrm{S}, \mathrm{S}_{\prime}, \mathrm{E} \rangle$ can be considered to be a timed transition table $\mathcal{A}'$. We choose the set of clocks to be the empty set, and replace every edge $\langle s, s', a \rangle$ by $\langle s, s', a, \emptyset, \mathbf{true} \rangle$. The runs of $\mathcal{A}'$ are in an obvious correspondence with the runs of $\mathcal{A}$.

3.2.5 Timed regular languages

We can couple acceptance criteria with timed transition tables, and use them to define timed languages.

Definition 3.12 A *timed Büchi automaton* (in short TBA) is a tuple $\langle \Sigma, S, S_0, C, E, F \rangle$, where $\langle \Sigma, S, S_0, C, E \rangle$ is a timed transition table, and $F \subseteq S$ is a set of *accepting* states.

A run $r = (\bar{s}, \bar{\nu})$ of a TBA over a timed word (σ, τ) is called an *accepting run* iff $inf(r) \cap F \neq \emptyset$.

For a TBA A, the language $L(\mathcal{A})$ of timed words it accepts is defined to be the set $\{(\sigma, \tau) \mid \text{ A has an accepting run over } (\sigma, \tau)\}$. ■

In analogy with the class of languages accepted by Büchi automata, we call the class of timed languages accepted by TBAs timed regular languages.

Definition 3.13 A timed language L is a *timed regular language* iff $L = L(\mathcal{A})$ for some TBA A. ■

Example 3.14 The language L_3 of Example 3.7 is a timed regular language. The timed transition table of Figure 3.4 is coupled with the acceptance set consisting of all the states.

For every ω-regular language L over Σ, the timed language $\{(\sigma, \tau) \mid \sigma \in L\}$ is regular.

A typical example of a nonregular timed language is the language L_2 of Example 3.4. It requires that the time difference between the successive pairs of a and b form an increasing sequence.

Another nonregular language is $\{(a^\omega, \tau) \mid \forall i. (\tau_i = 2^i)\}$. ■

The automaton of Example 3.15 combines the Büchi acceptance condition with the timing constraints to specify an interesting convergent response property:

Example 3.15 The automaton of Figure 3.5 accepts the timed language L_{crt} over the alphabet $\{a, b\}$.

$$L_{crt} = \{((ab)^\omega, \tau) \mid \exists i. \forall j \geq i. (\tau_{2j} < \tau_{2j-1} + 2)\}.$$

The start state is s_0, the accepting state is s_2, and there is a single clock x. The automaton starts in state s_0, and loops between the states s_0 and s_1 for a while. Then, nondeterministically, it moves to state s_2 setting its clock x to 0. While in the loop between

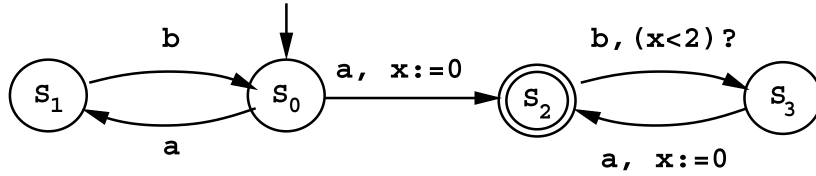


Figure 3.5: Timed Büchi automaton accepting L_{crt}

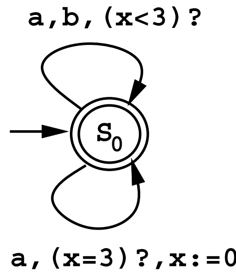


Figure 3.6: Timed automaton specifying periodic behavior

the states s_2 and s_3, the automaton resets its clock while reading a, and ensures that the next b is within 2 time units. Interpreting the symbol b as a response to a request denoted by the symbol a, the automaton models a system with a *convergent response time*; the response time is "eventually" always less than 2 time units. ■

The next example shows that timed automata can specify periodic behavior also.

Example 3.16 The automaton of Figure 3.6 accepts the following language over the alphabet $\{a, b\}$.

$$\{(\sigma, \tau) \mid \forall i.\, \exists j.\, (\tau_j = 3i \wedge \sigma_j = a)\}$$

The automaton has a single state s_0, and a single clock x. The clock gets reset at regular intervals of period 3 time units. The automaton requires that whenever the clock equals 3 there is an a symbol. Thus it expresses the property that a happens at all time values that are multiples of 3. ■

3.2.6 Properties of timed regular languages

The next theorem considers some closure properties of timed regular languages.

Theorem 3.17 The class of timed regular languages is closed under (finite) union and intersection.

PROOF. Consider TBAs $\mathcal{A}_\rangle = \langle \pm, \mathrm{S}_\rangle, \mathrm{S}_{\rangle,}, \mathrm{C}_\rangle, \mathrm{E}_\rangle, \mathrm{F}_\rangle \rangle$, $i = 1, 2, \ldots n$. Assume without loss of generality that the clock sets C_i are disjoint. We construct TBAs accepting the union and intersection of $L(\mathcal{A}_\rangle)$.

Since TBAs are nondeterministic the case of union is easy. The required TBA is simply the disjoint union of all the automata.

Intersection can be implemented by a trivial modification of the standard product construction for Büchi automata [Cho74]. The set of clocks for the product automaton A is $\cup_i \mathrm{C}_i$. The states of A are of the form $\langle s_1, \ldots s_n, k \rangle$, where each $s_i \in \mathrm{S}_i$, and $1 \leq k \leq n$. The i-th component of the tuple keeps track of the state of $\mathcal{A}_\rangle$, and the last component is used as a counter for cycling through the accepting conditions of all the individual automata. Initially the counter value is 1, and it is incremented from k to $(k+1)$ (modulo n) iff the current state of the k-th automaton is an accepting state. Note that we choose the value of n *mod* n to be n.

The initial states of A are of the form $\langle s_1, \ldots s_n, 1 \rangle$ where each s_i is a start state of $\mathcal{A}_\rangle$. A transition of A is obtained by coupling the transitions of the individual automata having the same label. Let $\{\langle s_i, s_i', a, \lambda_i, \delta_i \rangle \in \mathrm{E}_i \mid i = 1, \ldots n\}$ be a set of transitions with the same label a. Corresponding to this set, there is a joint transition of A out of each state of the form $\langle s_1, \ldots s_n, k \rangle$ labeled with a. The new state is $\langle s_1', \ldots s_n', j \rangle$ with $j = (k+1)$ *mod* n if $s_k \in \mathrm{F}_k$, and $j = k$ otherwise. The set of clocks to be reset with this transition is $\cup_i \lambda_i$, and the associated clock constraint is $\wedge_i \delta_i$.

The counter value cycles through the whole range $1, \ldots n$ infinitely often iff the accepting conditions of all the automata are met. Consequently, we define the accepting set for A to consist of states of the form $\langle s_1, \ldots s_n, n \rangle$, where $s_n \in \mathrm{F}_n$. ■

In the above product construction, the number of states of the resulting automaton is $n \cdot \Pi_i |\mathrm{S}_i|$. The number of clocks is $\Sigma_i |\mathrm{C}_i|$, and the size of the edge set is $n \cdot \Pi_i |\mathrm{E}_i|$. Note that $|\mathrm{E}|$ includes the length of the clock constraints assuming binary encoding for the constants.

Observe that even for the timed regular languages arbitrarily many symbols can occur in a finite interval of time. Furthermore, the symbols can be arbitrarily close to each other.

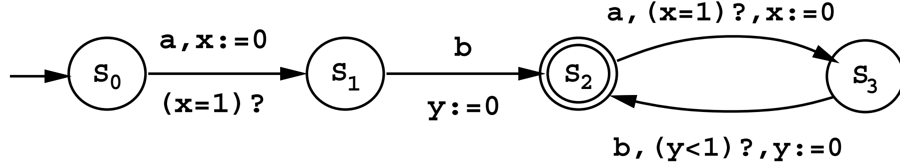


Figure 3.7: Timed automaton accepting $L_{converge}$

The following example shows that there is a timed regular language L such that for every $(\sigma, \tau) \in L$, there exists some $\epsilon \geq 0$ such that the sequence $\{(\tau_{i+1} - \tau_i) \mid i \geq 1\}$ converges to the limit ϵ.

Example 3.18 The language accepted by the automaton in Figure 3.7 is

$$L_{converge} \;=\; \{((ab)^\omega, \tau) \mid \forall i.\, (\tau_{2i-1} = i \wedge (\tau_{2i} - \tau_{2i-1} > \tau_{2i+2} - \tau_{2i+1}))\}.$$

Every word accepted by this automaton has the property that the sequence of time differences between a and the following b converges. A sample word accepted by the automaton is

$$(a,1) \to (b,1.5) \to (a,2) \to (b,2.25) \to (a,3) \to (b,3.125) \to \cdots$$

■

This example illustrates that the model of reals is indeed different from the discrete-time model. If we require all the time values τ_i to be multiples of some fixed constant ϵ, however small, the language accepted by the automaton of Figure 3.7 will be empty.

On the other hand, timed automata do not distinguish between the set of reals R and the set of rationals Q. Only the denseness of the underlying domain plays a crucial role. In particular, Theorem 3.19 shows that if we require all the time values in time sequences to be rational numbers, the untimed language $Untime[L(\mathcal{A})]$ of a timed automaton A stays unchanged.

Theorem 3.19 *Let L be a timed regular language. For every word σ, $\sigma \in Untime(L)$ iff there exists a time sequence τ such that $\tau_i \in$ Q for all $i \geq 1$, and $(\sigma, \tau) \in L$.*

Proof. Consider a timed automaton A, and a word σ. If there exists a time sequence τ with all rational time values such that $(\sigma, \tau) \in L(\mathcal{A})$, then clearly, $\sigma \in Untime[L(\mathcal{A})]$.

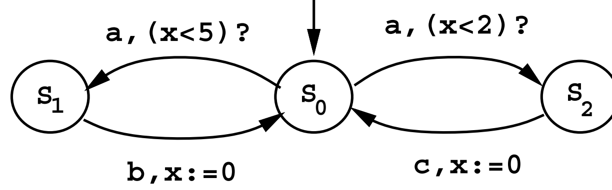


Figure 3.8: Timed Muller automaton

Now suppose for an arbitrary time sequence τ, $(\sigma, \tau) \in L(\mathcal{A})$. Let $\epsilon \in \mathbb{Q}$ be such that every constant appearing in the clock constraints of A is an integral multiple of ϵ. Let $\tau'_0 = 0$, and $\tau_0 = 0$. If $\tau_i = \tau_j + n\epsilon$ for some $0 \leq j < i$ and $n \in \mathbb{N}$, then choose $\tau'_i = \tau'_j + n\epsilon$. Otherwise choose $\tau'_i \in \mathbb{Q}$ such that for all $0 \leq j < i$, for all $n \in \mathbb{N}$, $(\tau_i - \tau_j) < n\epsilon$ iff $(\tau'_i - \tau'_j) < n\epsilon$. Note that because of the denseness of $\mathbb{Q}$ such a choice of τ'_i is always possible.

Consider an accepting run $r = (\overline{s}, \overline{\nu})$ of A over (σ, τ). Because of the construction of τ', if a clock x is reset at the i-th transition point, then its possible values at the j-th transition point along the two time sequences, namely, $(\tau_j - \tau_i)$ and $(\tau'_j - \tau'_i)$, satisfy the same set of clock constraints. Consequently it is possible to construct an accepting run $r' = (\overline{s}, \overline{\nu}')$ over (σ, τ') which follows the same sequence of edges as r. In particular, choose $\nu'_0 = \nu_0$, and if the i-th transition along r is according to the edge $\langle s_{i-1}, s_i, \sigma_i, \lambda_i, \delta_i \rangle$, then set $\nu'_i = [\lambda_i \mapsto 0](\nu'_{i-1} + \tau'_i - \tau'_{i-1})$. Consequently, $\mathcal{A}$ accepts (σ, τ'). ■

3.2.7 Timed Muller automata

We can define timed automata with Muller acceptance conditions also.

Definition 3.20 A *timed Muller automaton* (TMA) is a tuple $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}, \mathcal{F} \rangle$, where $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$ is a timed transition table, and $\mathcal{F} \subseteq 2^{\mathrm{S}}$ specifies an acceptance family.

A run $r = (\overline{s}, \overline{\nu})$ of the automaton over a timed word (σ, τ) is an accepting run iff $\mathit{inf}(r) \in \mathcal{F}$.

For a TMA A, the language $L(\mathcal{A})$ of timed words it accepts is defined to be the set $\{(\sigma, \tau) \mid \text{A has an accepting run over } (\sigma, \tau)\}$. ■

Example 3.21 Consider the automaton of Figure 3.8 over the alphabet $\{a, b, c\}$. The start state is s_0, and the Muller acceptance family consists of a single set $\{s_0, s_2\}$. So any accepting run should loop between states s_0 and s_1 only finitely many times, and between states s_0 and s_2 infinitely many times. Every word (σ, τ) accepted by the automaton satisfies: (1) $\sigma \in (a(b+c))^*(ac)^\omega$, and (2) for all $i \geq 1$, the difference $(\tau_{2i-1} - \tau_{2i-2})$ is less than 2 if the $(2i)$-th symbol is c, and less than 5 otherwise. ■

Recall that Büchi automata and Muller automata have the same expressive power. The following theorem states that the same holds true for TBAs and TMAs. Thus the class of timed languages accepted by TMAs is the same as the class of timed regular languages. The proof of the following theorem closely follows the standard argument that an ω-regular language is accepted by a Büchi automaton iff it is accepted by some Muller automaton.

Theorem 3.22 A timed language is accepted by some timed Büchi automaton iff it is accepted by some timed Muller automaton.

PROOF. Let $\mathcal{A} = \langle \pm, \mathrm{S}, \mathrm{S}_{\prime}, \mathrm{C}, \mathrm{E}, \mathrm{F} \rangle$ be a TBA. Consider the TMA $\mathcal{A}'$ with the same timed transition table as that of A, and with the acceptance family $\mathcal{F} = \{\mathrm{S}' \subseteq \mathrm{S} : \mathrm{S}' \cap \mathrm{F} \neq \emptyset\}$. It is easy to check that $L(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. This proves the "only if" part of the claim.

In the other direction, given a TMA, we can construct a TBA accepting the same language using the simulation of Muller acceptance condition by Büchi automata. Let A be a TMA given as $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E}, \mathcal{F} \rangle$. First note that $L(\mathcal{A}) = \cup_{\mathrm{F} \in \mathcal{F}} \mathcal{L}(\mathcal{A}_\mathrm{F})$ where $\mathcal{A}_\mathrm{F} = \langle \pm, \mathrm{S}, \mathrm{S}_{\prime}, \mathrm{C}, \mathrm{E}, \{\mathrm{F}\} \rangle$, so it suffices to construct, for each acceptance set F, a TBA $\mathcal{A}'_\mathrm{F}$ which accepts the language $L(\mathcal{A}_\mathrm{F})$. Assume $\mathrm{F} = \{s_1, \ldots s_k\}$. The automaton $\mathcal{A}'_\mathrm{F}$ uses nondeterminism to guess when the set F is entered forever, and then uses a counter to make sure that every state in F is visited infinitely often. States of $\mathcal{A}'_\mathrm{F}$ are of the form $\langle s, i \rangle$, where $s \in \mathrm{S}$ and $i \in \{0, 1, \ldots k\}$. The set of initial states is $\mathrm{S}_0 \times \{0\}$. The automaton simulates the transitions of A, and at some point nondeterministically sets the second component to 1. For every transition $\langle s, s', a, \lambda, \delta \rangle$ of A, the automaton $\mathcal{A}'_\mathrm{F}$ has a transition $\langle \langle s, 0 \rangle, \langle s', 0 \rangle, a, \lambda, \delta \rangle$, and, in addition, if $s' \in \mathrm{F}$ it also has a transition $\langle \langle s, 0 \rangle, \langle s', 1 \rangle, a, \lambda, \delta \rangle$.

While the second component is nonzero, the automaton is required to stay within the set F. For every A-transition $\langle s, s', a, \lambda, \delta \rangle$ with both s and s' in F, for each $1 \leq i \leq k$, there is an $\mathcal{A}'_\mathrm{F}$-transition $\langle \langle s, i \rangle, \langle s', j \rangle, a, \lambda, \delta \rangle$ where $j = (i+1)$ *mod* k, if s equals s_i, else $j = i$. The only accepting state is $\langle s_k, k \rangle$. ■

3.3 Checking emptiness

In this section we develop an algorithm for checking the emptiness of the language of a timed automaton. The existence of an infinite accepting path in the underlying transition table is clearly a necessary condition for the language of an automaton to be nonempty. However, the timing constraints of the automaton rule out certain additional behaviors. We will show that a Büchi automaton can be constructed that accepts exactly the set of untimed words that are consistent with the timed words accepted by a timed automaton.

3.3.1 Restriction to integer constants

Recall that our definition of timed automata allows clock constraints which involve comparisons with rational constants. The following lemma shows that, for checking emptiness, we can restrict ourselves to timed automata whose clock constraints involve only integer constants. For a timed sequence τ and $t \in \mathsf{R}$, let $t \cdot \tau$ denote the timed sequence obtained by multiplying all τ_i by t.

Lemma 3.23 Consider a timed transition table A, a timed word (σ, τ), and $t \in \mathsf{R}$. $(\overline{s}, \overline{\nu})$ is a run of A over (σ, τ) iff $(\overline{s}, t{\cdot}\overline{\nu})$ is a run of $\mathcal{A}_{\sqcup}$ over $(\sigma, t{\cdot}\tau)$, where $\mathcal{A}_{\sqcup}$ is the timed transition table obtained by replacing each constant d in each clock constraint labeling the edges of A by $t \cdot d$.

Proof. The lemma can be proved easily from the definitions using induction. ■

Thus there is an isomorphism between the runs of A and the runs of $\mathcal{A}_{\sqcup}$. If we choose t to be the least common multiple of all the constants appearing in the clock constraints of A, then the clock constraints for $\mathcal{A}_{\sqcup}$ use only integer constants. In this translation, the values of the individual constants grow with the product of the denominators of all the original constants. We assume binary encoding for the constants. Let us denote the length of the clock constraints of A by $|\delta(\mathcal{A})|$. It is easy to prove that $|\delta(\mathcal{A}_{\sqcup})|$ is bounded by $|\delta(\mathcal{A})|^2$. Observe that this result depends crucially on the fact that we encode constants in binary notation; if we use unary encoding then $|\delta(\mathcal{A}_{\sqcup})|$ can be exponential in $|\delta(\mathcal{A})|$.

Observe that $L(\mathcal{A})$ is empty iff $L[\mathcal{A}_{\sqcup}]$ is empty. Hence, to decide the emptiness of $L(\mathcal{A})$ we consider $\mathcal{A}_{\sqcup}$. Also *Untime*$[L(\mathcal{A})]$ equals *Untime*$[L(\mathcal{A}_{\sqcup})]$. In the remainder of the section we assume that the clock constraints use only integer constants.

3.3.2 Clock regions

At every point in time the future behavior of a timed transition table is determined by its state and the values of all its clocks. This motivates the following definition:

Definition 3.24 For a timed transition table $\langle \Sigma, S, S_0, C, E\rangle$, an *extended state* is a pair $\langle s, \nu\rangle$ where $s \in S$ and ν is a clock interpretation for C. ■

Since the number of such extended states is infinite (in fact, uncountable), we cannot possibly build an automaton whose states are the extended states of A. But if two extended states with the same A-state agree on the integral parts of all clock values, and also on the ordering of the fractional parts of all clock values, then the runs starting from the two extended states are very similar. The integral parts of the clock values are needed to determine whether or not a particular clock constraint is met, whereas the ordering of the fractional parts is needed to decide which clock will change its integral part first. For example, if two clocks x and y are between 0 and 1 in an extended state, then a transition with clock constraint $(x = 1)$ can be followed by a transition with clock constraint $(y = 1)$, depending on whether or not the current clock values satisfy $(x < y)$.

The integral parts of clock values can get arbitrarily large. But if a clock x is never compared with a constant greater than c, then its actual value, once it exceeds c, is of no consequence in deciding the allowed paths.

Now we formalize this notion. For any $t \in \mathsf{R}$, $\mathit{fract}(t)$ denotes the fractional part of t, and $\lfloor t \rfloor$ denotes the integral part of t; that is, $t = \lfloor t \rfloor + \mathit{fract}(t)$. We assume that every clock in C appears in some clock constraint.

Definition 3.25 Let $\mathcal{A} = \langle \pm, S, S_{\prime}, C, E\rangle$ be a timed transition table. For each $x \in C$, let c_x be the largest integer c such that $(x \leq c)$ or $(c \leq x)$ is a subformula of some clock constraint appearing in E.

The equivalence relation $\sim$ is defined over the set of all clock interpretations for C; $\nu \sim \nu'$ iff all the following conditions hold:

1. For all $x \in C$, either $\lfloor \nu(x) \rfloor$ and $\lfloor \nu'(x) \rfloor$ are the same, or both $\nu(x)$ and $\nu'(x)$ are greater than c_x.
2. For all $x, y \in C$ with $\nu(x) \leq c_x$ and $\nu(y) \leq c_y$, $\mathit{fract}(\nu(x)) \leq \mathit{fract}(\nu(y))$ iff $\mathit{fract}(\nu'(x)) \leq \mathit{fract}(\nu'(y))$.

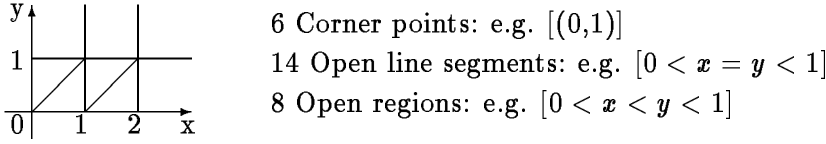


Figure 3.9: Clock regions

3. For all $x \in C$ with $\nu(x) \leq c_x$, $fract(\nu(x)) = 0$ iff $fract(\nu'(x)) = 0$.

A *clock region* for A is an equivalence class of clock interpretations induced by $\sim$. ∎

We will use $[\nu]$ to denote the clock region to which ν belongs. Each region can be uniquely characterized by a (finite) set of clock constraints it satisfies. For example, consider a clock interpretation ν over two clocks with $\nu(x) = 0.3$ and $\nu(y) = 0.7$. Every clock interpretation in $[\nu]$ satisfies the constraint $(0 < x < y < 1)$, and we will represent this region by $[0 < x < y < 1]$. The nature of the equivalence classes can be best understood through an example.

Example 3.26 Consider a timed transition table with two clocks x and y with $c_x = 2$ and $c_y = 1$. The clock regions are shown in Figure 3.9. ∎

Note that there are only a finite number of regions. Also note that for a clock constraint δ, if $\nu \sim \nu'$ then ν satisfies δ iff ν' satisfies δ. We say that a clock region α satisfies a clock constraint δ iff every $\nu \in \alpha$ satisfies δ. Each region can be represented by specifying

(1) for every clock x, one clock constraint from the set

$$\{x = c \mid c = 0, 1, \ldots c_x\} \cup \{c - 1 < x < c \mid c = 1, \ldots c_x\} \cup \{x > c_x\},$$

(2) for every pair of clocks x and y such that $c - 1 < x < c$ and $d - 1 < y < d$ appear in (1) for some c, d, whether $fract(x)$ is less than, equal to, or greater than $fract(y)$.

By counting the number of possible combinations of equations of the above form, we get the upper bound in the following lemma.

Lemma 3.27 The number of clock regions is bounded by $[|\mathrm{C}|! \cdot 2^{|\mathrm{C}|} \cdot \Pi_{x \in \mathrm{C}}(2c_x + 2)]$. ■

Henceforth, we assume that the number of regions is $O[2^{|\delta(\mathcal{A})|}]$; remember that $\delta(\mathcal{A})$ stands for the length of the clock constraints of $\mathcal{A}$ assuming binary encoding. Note that if we increase $\delta(\mathcal{A})$ without increasing the number of clocks or the size of the largest constants the clocks are compared with, then the number of regions does not grow with $|\delta(\mathcal{A})|$. Also observe that a region can be represented in space linear in $|\delta(\mathcal{A})|$.

3.3.3 The region automaton

The first step in the decision procedure for checking emptiness is to construct a transition table whose paths mimic the runs of A in a certain way. We will denote the desired transition table by $R(\mathcal{A})$, the *region automaton* of A. A state of $R(\mathcal{A})$ records the state of the timed transition table $\mathcal{A}$, and the equivalence class of the current values of the clocks. It is of the form $\langle s, \alpha \rangle$ with $s \in$ S and α being a clock region. The intended interpretation is that whenever the extended state of A is $\langle s, \nu \rangle$, the state of $R(\mathcal{A})$ is $\langle s, [\nu] \rangle$. The region automaton starts in some state $\langle s_0, [\nu_0] \rangle$ where s_0 is a start state of A, and the clock interpretation ν_0 assigns 0 to every clock. The transition relation of $R(\mathcal{A})$ is defined so that the intended simulation is obeyed. It has an edge from $\langle s, \alpha \rangle$ to $\langle s', \alpha' \rangle$ labeled with a iff A in state s with the clock values $\nu \in \alpha$ can make a transition reading a to the extended state $\langle s', \nu' \rangle$ for some $\nu' \in \alpha'$.

The edge relation can be conveniently defined using a *time-successor* relation over the clock regions. The time-successors of a clock region α are all the clock regions that will be visited by a clock interpretation $\nu \in \alpha$ as time progresses.

Definition 3.28 A clock region α' is a time-successor of a clock region α iff for each $\nu \in \alpha$, there exists a positive $t \in \mathsf{R}$ such that $\nu + t \in \alpha'$. ■

Example 3.29 Consider the clock regions shown in Figure 3.9 again. The time-successors of a region α are the regions that can be reached by moving along a line drawn from some point in α in the diagonally upwards direction (parallel to the line $x = y$). For example, the region $[(1 < x < 2), (0 < y < x - 1)]$ has, other than itself, the following regions as time-successors: $[(x = 2), (0 < y < 1)]$, $[(x > 2), (0 < y < 1)]$, $[(x > 2), (y = 1)]$ and $[(x > 2), (y > 1)]$. ■

Now let us see how to construct all the time-successors of a clock region. Recall that a clock region α is specified by giving (1) for every clock x, a constraint of the form $(x = c)$ or $(c-1 < x < c)$ or $(x > c_x)$, and (2) for every pair x and y such that $(c-1 < x < c)$ and $(d-1 < y < d)$ appear in (1), the ordering relationship between $fract(x)$ and $fract(y)$. To compute all the time-successors of α we proceed as follows. First observe that the time-successor relation is a transitive relation. We consider different cases.

First suppose that α satisfies the constraint $(x > c_x)$ for every clock x. The only time-successor of α is itself. This is the case for the region $[(x > 2), (y > 1)]$ in Figure 3.9.

Now suppose that the set C_0 consisting of clocks x such that α satisfies the constraint $(x = c)$ for some $c \leq c_x$, is nonempty. In this case, as time progresses the fractional parts of the clocks in C_0 become nonzero, and the clock region changes immediately. The time-successors of α are same as the time-successors of the clock region β specified as below:

(1) For $x \in C_0$, if α satisfies $(x = c_x)$ then β satisfies $(x > c_x)$, otherwise if α satisfies $(x = c)$ then β satisfies $(c < x < c+1)$. For $x \notin C_0$, the constraint in β is the same as that in α.

(2) For clocks x and y such that $(c-1 < x < c)$ and $(d-1 < y < d)$ appear in (1), the ordering relationship in β between their fractional parts is same as in α.

For instance, in Figure 3.9, the time-successors of $[(x = 0), (0 < y < 1)]$ are same as the time-successors of $[0 < x < y < 1]$.

If both the above cases do not apply, then let C_0 be the set of clocks x with maximal fractional part; that is, for all $y \in C$, $fract(y) \leq fract(x)$ is a constraint of α. In this case, as time progresses, the clocks in C_0 assume integer values. Let β be the clock region specified by

(1) For $x \in C_0$, if α satisfies $(c-1 < x < c)$ then β satisfies $(x = c)$. For $x \notin C_0$, the constraint in β is same as that in α.

(2) For clocks x and y such that $(c-1 < x < c)$ and $(d-1 < y < d)$ appear in (1), the ordering relationship in β between their fractional parts is same as in α.

In this case, the time-successors of α include α, β, and all the time-successors of β. For instance, in Figure 3.9, time-successors of $[0 < x < y < 1]$ include itself, $[(0 < x < 1), (y = 1)]$, and all the time-successors of $[(0 < x < 1), (y = 1)]$.

Now we are ready to define the region automaton.

Definition 3.30 For a timed transition table $\mathcal{A} = \langle \pm, \mathrm{S}, \mathrm{S}_{\prime}, \mathrm{C}, \mathrm{E}\rangle$, the corresponding region automaton $R(\mathcal{A})$ is a transition table over the alphabet Σ.

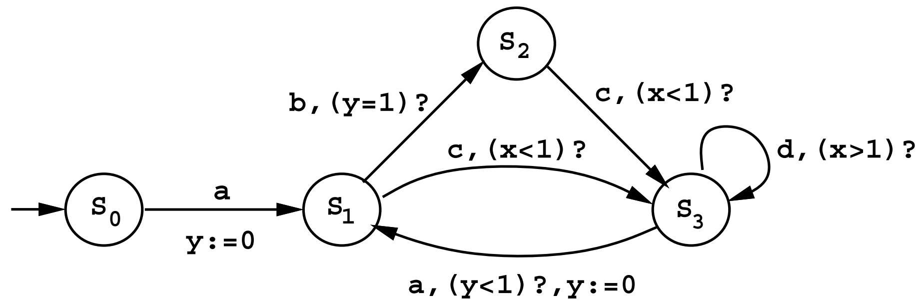
- The states of $R(\mathcal{A})$ are of the form $\langle s, \alpha\rangle$ where $s \in \mathrm{S}$ and α is a clock region.
- The initial states are of the form $\langle s_0, [\nu_0]\rangle$ where $s_0 \in \mathrm{S}_0$ and $\nu_0(x) = 0$ for all $x \in \mathrm{C}$.
- $R(\mathcal{A})$ has an edge $\langle\langle s, \alpha\rangle, \langle s', \alpha'\rangle, a\rangle$ iff there is an edge $\langle s, s', a, \lambda, \delta\rangle \in \mathrm{E}$ and a region α'' such that (1) α'' is a time-successor of α, (2) α'' satisfies δ, and (3) $\alpha' = [\lambda \mapsto 0]\alpha''$.

■

Example 3.31 Consider the timed automaton $\mathcal{A}_{\prime}$ shown in Figure 3.10. The alphabet is $\{a, b, c, d\}$. Every state of the automaton is an accepting state. The corresponding region automaton $R(\mathcal{A}_{\prime})$ is also shown. Only the regions reachable from the initial region $\langle s_0, [x = y = 0]\rangle$ are shown. Note that $c_x = 1$ and $c_y = 1$. The timing constraints of the automaton ensure that the transition from s_2 to s_3 is never taken. The only reachable region with state component s_2 satisfies the constraints $[y = 1, x > 1]$, and this region has no outgoing edges. Thus the region automaton helps us in concluding that no transitions can follow a b-transition. ■

From the bound on the number of regions, it follows that the number of states in $R(\mathcal{A})$ is $O[|\mathrm{S}| \cdot 2^{|\delta(\mathcal{A})|}]$. An inspection of the definition of the time-successor relation shows that every region has at most $\Sigma_{x\in \mathrm{C}}[\,2c_x + 2\,]$ successor regions. The region automaton has at most one edge out of $\langle s, \alpha\rangle$ for every edge out of s and every time-successor of α. It follows that the number of edges in $R(\mathcal{A})$ is $O[|\mathrm{E}| \cdot 2^{|\delta(\mathcal{A})|}]$. Note that computing the time-successor relation is easy, and can be done in time linear in the length of the representation of the region. Constructing the edge relation for the region automaton is also relatively easy; in addition to computing the time-successors, we also need to determine whether the clock constraint labeling a particular A-transition is satisfied by a clock region. The region graph can be constructed in time $O[(|\mathrm{S}| + |\mathrm{E}|) \cdot 2^{|\delta(\mathcal{A})|}]$.

Now we proceed to establish a correspondence between the runs of A and the runs of $R(\mathcal{A})$.



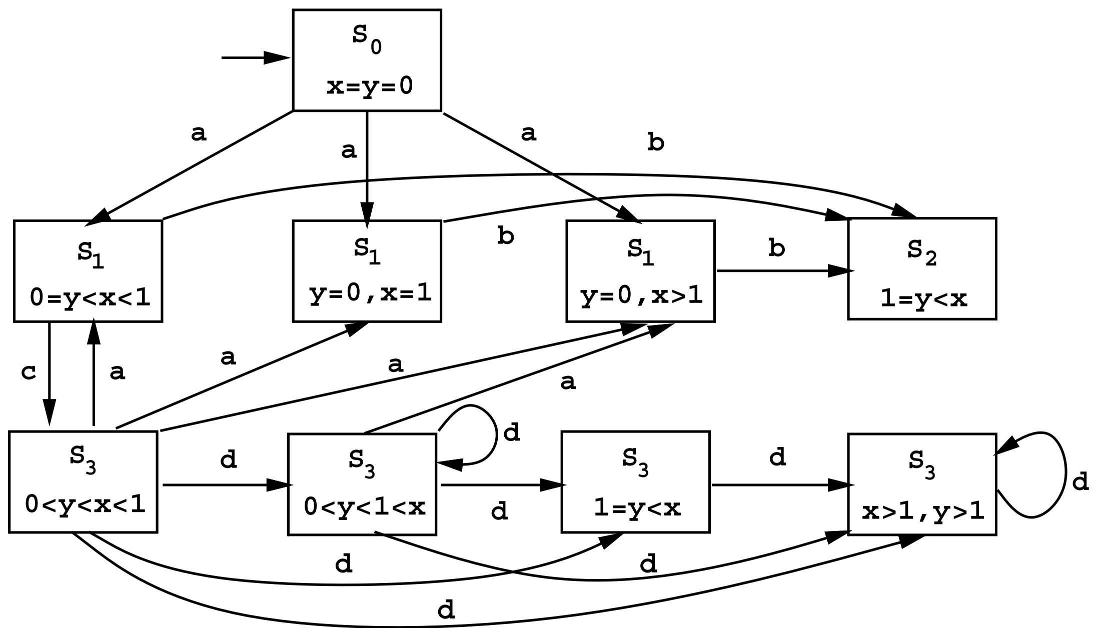


Figure 3.10: Automaton $\mathcal{A}$, and its region automaton

Definition 3.32 For a run $r = (\overline{s}, \overline{\nu})$ of A of the form

$$r \;:\; \langle s_0, \nu_0 \rangle \;\xrightarrow[\tau_1]{\sigma_1}\; \langle s_1, \nu_1 \rangle \;\xrightarrow[\tau_2]{\sigma_2}\; \langle s_2, \nu_2 \rangle \;\xrightarrow[\tau_3]{\sigma_3}\; \cdots$$

define its projection $[r] = (\overline{s}, [\overline{\nu}])$ to be the sequence

$$[r] \;:\; \langle s_0, [\nu_0] \rangle \;\xrightarrow{\sigma_1}\; \langle s_1, [\nu_1] \rangle \;\xrightarrow{\sigma_2}\; \langle s_2, [\nu_2] \rangle \;\xrightarrow{\sigma_3}\; \cdots$$

■

From the definition of the edge relation for $R(\mathcal{A})$, it follows that $[r]$ is a run of $R(\mathcal{A})$ over σ. Since time progresses without bound along r, every clock $x \in$ C is either reset infinitely often, or from a certain time onwards it increases without bound. Hence, for all $x \in$ C, for infinitely many $i \geq 0$, $[\nu_i]$ satisfies $[(x = 0) \vee (x > c_x)]$. This prompts the following definition:

Definition 3.33 A run $r = (\overline{s}, \overline{\alpha})$ of the region automaton $R(\mathcal{A})$ of the form

$$r \,:\, \langle s_0, \alpha_0 \rangle \;\xrightarrow{\sigma_1}\; \langle s_1, \alpha_1 \rangle \;\xrightarrow{\sigma_2}\; \langle s_2, \alpha_2 \rangle \;\xrightarrow{\sigma_3}\; \cdots$$

is *progressive* iff for each clock $x \in$ C, there are infinitely many $i \geq 0$ such that α_i satisfies $[(x = 0) \vee (x > c_x)]$. ■

Thus for a run r of A over (σ, τ), $[r]$ is a progressive run of $R(\mathcal{A})$ over σ. From Lemma 3.35 it follows that progressive runs of $R(\mathcal{A})$ precisely correspond to the projected runs of $\mathcal{A}$. Before we prove the lemma let us consider the region automaton of Example 3.31 again.

Example 3.34 Consider the region automaton $R(\mathcal{A}_\prime)$ of Figure 3.10. Every run r of $R(\mathcal{A}_\prime)$ has a suffix of one of the following three forms: (i) the automaton loops between the regions $\langle s_1, [y = 0 < x < 1] \rangle$ and $\langle s_3, [0 < y < x < 1] \rangle$, (ii) the automaton stays in the region $\langle s_3, [0 < y < 1 < x] \rangle$ using the self-loop, or (iii) the automaton stays in the region $\langle s_3, [x > 1, y > 1] \rangle$.

Only the case (iii) corresponds to the progressive runs. For runs of type (i), even though y gets reset infinitely often, the value of x is always less than 1. For runs of type (ii), even though the value of x is not bounded, the clock y is reset only finitely often, and yet, its value is bounded. Thus every progressive run of $\mathcal{A}_\prime$ corresponds to a run of $R(\mathcal{A}_\prime)$ of type (iii). ■

Lemma 3.35 If r is a progressive run of $R(\mathcal{A})$ over σ then there exists a time sequence τ and a run r' of A over (σ, τ) such that r equals $[r']$.

PROOF. Consider a progressive run $r = (\overline{s}, \overline{\alpha})$ of $R(\mathcal{A})$ over σ. We construct the run r' and the time sequence τ step by step. As usual, r' starts with $\langle s_0, \nu_0 \rangle$. Now suppose that the extended state of A is $\langle s_i, \nu_i \rangle$ at time τ_i with $\nu_i \in \alpha_i$. There is an edge in $R(\mathcal{A})$ from $\langle s_i, \alpha_i \rangle$ to $\langle s_{i+1}, \alpha_{i+1} \rangle$ labeled with σ_{i+1}. From the definition of the region automaton it follows that there is an edge $\langle s_i, s_{i+1}, \sigma_{i+1}, \lambda_{i+1}, \delta_{i+1} \rangle \in$ E and a time-successor α'_{i+1} of α_i such that α'_{i+1} satisfies δ_{i+1} and $\alpha_{i+1} = [\lambda_{i+1} \mapsto 0]\alpha'_{i+1}$. From the definition of time-successor, there exists a time τ_{i+1} such that $(\nu_i + \tau_{i+1} - \tau_i) \in \alpha'_{i+1}$. Now it is clear the next transition of A can be at time τ_{i+1} to an extended state $\langle s_{i+1}, \nu_{i+1} \rangle$ with $\nu_{i+1} \in \alpha_{i+1}$. Using this construction repeatedly we get a run $r' = (\overline{s}, \overline{\nu})$ over (σ, τ) with $[r'] = r$.

The only problem with the above construction is that τ may not satisfy the progress condition. Suppose that τ is a converging sequence. We use the fact that r is a progressive run to construct another time sequence τ' satisfying the progress requirement and show that the automaton can follow the same sequence of transitions as r' but at times τ'_i.

Let C_0 be the set of clocks reset infinitely often along r. Since τ is a converging sequence, after a certain position onwards, every clock in C_0 gets reset before it reaches the value 1. Since r is progressive, every clock x not in C_0, after a certain position onwards, never gets reset, and continuously satisfies $x > c_x$. This ensures that there exists $j \geq 0$ such that (1) after the j-th transition point each clock $x \notin C_0$ continuously satisfies $(x > c_x)$, and each clock $x \in C_0$ continuously satisfies $(x < 1)$, and (2) for each $k > j$, $(\tau_k - \tau_j)$ is less than 0.5.

Let $j < k_1 < k_2, \ldots$ be an infinite sequence of integers such that each clock x in C_0 is reset at least once between the k_i-th and k_{i+1}-th transition points along r. Now we construct another sequence $r'' = (\overline{s}, \overline{\nu}')$ with the sequence of transition times τ' as follows. The sequence of transitions along r'' is same as that along r'. If $i \notin \{k_1, k_2 \ldots\}$ then we require the $(i+1)$-th transition to happen after a delay of $(\tau_{i+1} - \tau_i)$, otherwise we require the delay to be 0.5. Observe that along r'' the delay between the k_i-th and k_{i+1}-th transition points is less than 1. Consequently, in spite of the additional delays, the value of every clock in C_0 remains less than 1 after the j-th transition point. So the truth of all the clock constraints and the clock regions at the transition points remain unchanged (as compared to r'). From this we conclude that r'' satisfies the consecution requirement, and is a run of A. Furthermore, $[r''] = [r'] = r$.

Since τ' has infinitely many jumps each of duration 0.5, it satisfies the progress requirement. Hence r'' is the run required by the lemma. ■

3.3.4 The untiming construction

For a timed automaton A, its region automaton can be used to recognize $Untime[L(\mathcal{A})]$. The following theorem is stated for TBAs, but it also holds for TMAs.

Theorem 3.36 Given a TBA $\mathcal{A} = \langle \pm, \mathrm{S}, \mathrm{S}_{,}, \mathrm{C}, \mathrm{E}, \mathrm{F} \rangle$, there exists a Büchi automaton over Σ which accepts $Untime[L(\mathcal{A})]$.

Proof. We construct a Büchi automaton $\mathcal{A}'$ as follows. Its transition table is $R(\mathcal{A})$, the region automaton corresponding to the timed transition table $\langle \Sigma, \mathrm{S}, \mathrm{S}_0, \mathrm{C}, \mathrm{E} \rangle$. The accepting set of $\mathcal{A}'$ is $\mathrm{F}' = \{\langle s, \alpha \rangle \mid s \in \mathrm{F}\}$.

If r is an accepting run of A over (σ, τ), then $[r]$ is a progressive and accepting run of $\mathcal{A}'$ over σ. The converse follows from Lemma 3.35. Given a progressive run r of $\mathcal{A}'$ over σ, the lemma gives a time sequence τ and a run r' of A over (σ, τ) such that r equals $[r']$. If r is an accepting run, so is r'. It follows that $\sigma \in Untime[L(\mathcal{A})]$ iff $\mathcal{A}'$ has a progressive, accepting run over it.

For $x \in \mathrm{C}$, let $\mathrm{F}_x = \{\langle s, \alpha \rangle \mid \alpha \models [(x = 0) \vee (x > c_x)]\}$. Recall that a run of $\mathcal{A}'$ is progressive iff some state from each F_x repeats infinitely often. It is straightforward to construct another Büchi automaton $\mathcal{A}''$ such that $\mathcal{A}'$ has a progressive and accepting run over σ iff $\mathcal{A}''$ has an accepting run over σ.

The automaton $\mathcal{A}''$ is the desired automaton; $L(\mathcal{A}'')$ equals $Untime[L(\mathcal{A})]$. ■

Example 3.37 Let us consider the region automaton $R(\mathcal{A}_{,})$ of Figure 3.31 again. Since all states of $\mathcal{A}_{,}$ are accepting, from the description of the progressive runs in Example 3.34 it follows that the transition table $R(\mathcal{A}_{,})$ can be changed to a Büchi automaton by choosing the accepting set to consist of a single region $\langle s_3, [x > 1, y > 1] \rangle$. Consequently

$$Untime[L(\mathcal{A}_{,})] \;=\; \mathcal{L}[\mathcal{R}(\mathcal{A}_{,})] \;=\; \dashv\rfloor\,(\dashv\rfloor)^{*}\,\lceil^{\omega}.$$

■

Theorem 3.36 says that the timing information in a timed automaton is "regular" in character; its consistency can be checked by a finite-state automaton. An equivalent formulation of the theorem is

If a timed language L is timed regular then $Untime(L)$ is ω-regular.

Furthermore, to check whether the language of a given TBA is empty, we can check for the emptiness of the language of the corresponding Büchi automaton constructed by the proof of Theorem 3.36. The next theorem follows.

Theorem 3.38 Given a timed Büchi automaton $\mathcal{A} = \langle \pm, \mathrm{S}, \mathrm{S}_{\prime}, \mathrm{C}, \mathrm{E}, \mathrm{F} \rangle$ the emptiness of $L(\mathcal{A})$ can be checked in time $O[(|\mathrm{S}| + |\mathrm{E}|) \cdot 2^{|\delta(\mathcal{A})|}]$.

PROOF. Let $\mathcal{A}'$ be the Büchi automaton constructed as outlined in the proof of Theorem 3.36. Recall that in Section 3.3.3 we had shown that the number of states in $\mathcal{A}'$ is $O[|\mathrm{S}| \cdot 2^{|\delta(\mathcal{A})|}]$, the number of edges is $O[|\mathrm{E}| \cdot 2^{|\delta(\mathcal{A})|}]$.

The language $L(\mathcal{A})$ is nonempty iff there is a cycle C in $\mathcal{A}'$ such that C is accessible from some start state of $\mathcal{A}'$ and C contains at least one state each from the set F' and each of the sets F_x. This can be checked in time linear in the size of $\mathcal{A}'$ [SVW87]. The complexity bound of the theorem follows. ■

Recall that if we start with an automaton A whose clock constraints involve rational constants, we need to apply the above decision procedure on $\mathcal{A}_{\sqcup}$ for the least common denominator t of all the rational constants (see Section 3.3.1). This involves a blow-up in the size of the clock constraints; we have $\delta[\mathcal{A}_{\sqcup}] = \mathcal{O}[\delta(\mathcal{A})^{\in}]$.

The above method can be used even if we change the acceptance condition for timed automata. In particular, given a timed Muller automaton A we can effectively construct a Muller (or, Büchi) automaton which accepts $Untime[L(\mathcal{A})]$, and use it to check for the emptiness of $L(\mathcal{A})$.

3.3.5 Complexity of checking emptiness

The complexity of the algorithm for deciding emptiness of a TBA is exponential in the number of clocks and the length of the constants in the timing constraints. This blow-up in complexity seems unavoidable; we reduce the acceptance problem for linear bounded automata, a known PSPACE-complete problem [HU79], to the emptiness question for TBAs to prove the PSPACE lower bound for the emptiness problem. We also show the problem to be PSPACE-complete by arguing that the algorithm of Section 3.3.4 can be implemented in polynomial space.

Theorem 3.39 The problem of deciding the emptiness of the language of a given timed automaton A, is PSPACE-complete.

PROOF. [PSPACE-membership] First we show that the problem is in PSPACE. Since the number of states of the region automaton is exponential in the number of clocks of A, we cannot construct the entire transition table. We give a nondeterministic version of the algorithm which uses only polynomial space; the trick involved is fairly standard.

Let l be the length of the representation of A. As observed earlier, each state of the region automaton can be represented in space $O(l)$, and all its successors in the region automaton can be generated easily. Recall that the language $L(\mathcal{A})$ is nonempty iff the region automaton has a cycle that is accessible from some start state and meets all the acceptance criteria. The procedure nondeterministically guesses an initial region v_1, another region v_n, and a path

$$v_1 \rightarrow \cdots \rightarrow v_n \rightarrow v_{n+1} \rightarrow \cdots \rightarrow v_m = v_n.$$

The path is guessed vertex by vertex, at each step checking that the newly guessed state is connected by an edge from the previous one. In addition, the procedure checks that the cycle $v_n \rightarrow \cdots \rightarrow v_m$ satisfies all the acceptance criteria. If $L(\mathcal{A})$ is nonempty then this nondeterministic algorithm succeeds. The algorithm only uses space $O(l)$; hence checking nonemptiness of $L(\mathcal{A})$ requires nondeterministic polynomial space. It follows that the emptiness can be checked in PSPACE from Savitch's theorem.

[PSPACE-hardness] The question of deciding whether a given linear bounded automaton accepts a given input string is PSPACE-complete [HU79]. A linear bounded automaton M is a nondeterministic Turing machine whose tape head cannot go beyond the end of the input markers. We construct a TBA A such that its language is nonempty iff the machine M halts on a given input.

Let Γ be the tape alphabet of M and let Q be its states. Let $\Sigma = \Gamma \cup (\Gamma \times Q)$, and let $a_1, a_2, \ldots a_k$ denote the elements of Σ. A configuration of M in which the tape reads $\gamma_1\gamma_2 \ldots \gamma_n$, and the machine is in state q reading the i-th tape symbol, is represented by the string $\sigma_1, \ldots \sigma_n$ over Σ such that $\sigma_j = \gamma_j$ if $j \neq i$ and $\sigma_i = \langle \gamma_i, q \rangle$.

The acceptance corresponds to a special state q_f; after which the configuration stays unchanged. The alphabet of $\mathcal{A}$ includes Σ, and in addition, has a symbol a_0. A computation of M is encoded by the word

$$\sigma_1^1 a_0 \ldots \sigma_n^1 a_0 \sigma_1^2 a_0 \ldots \sigma_n^2 a_0 \ldots \sigma_1^j a_0 \ldots \sigma_n^j a_0 \ldots$$

such that $\sigma_1^j \ldots \sigma_n^j$ encodes the j-th configuration according to the above scheme. The time sequence associated with this word also encodes the computation: we require the time difference between successive a_0's to be $k+1$, and if $\sigma_i^j = a_l$ then we require its time to be l greater than the time of the previous a_0. The encoding in the time sequence is used to enforce the consecution requirement.

We want to construct $\mathcal{A}$ which accepts precisely the timed words encoding the halting computations of M according to the above scheme. We only sketch the construction. $\mathcal{A}$ uses $2n+1$ clocks. The clock x is reset with each a_0. While reading a_0 we require $(x = k+1)$ to hold, and while reading a_i we require $(x = i)$ to hold. These conditions ensure that the encoding in the time sequence is consistent with the word. For each tape cell i, we have two clocks x_i and y_i. The clock x_i is reset with σ_i^j, for odd values of j, and the clock y_i is reset with σ_i^j, for even values of j. Assume that the automaton has read the first j configurations, with j odd. The value of the clock x_i represents the i-th cell of the j-th configuration. Consequently, the possible choices for the values of σ_i^{j+1} are determined by examining the values of x_{i-1}, x_i and x_{i+1} according to the transition rules for M. While reading the $(j+1)$-th configuration, the y-clocks get set to appropriate values; these values are examined while reading the $(j+2)$-th configuration. This ensures proper consecution of configurations. Proper initialization and halting can be enforced in a straightforward way. The size of $\mathcal{A}$ is polynomial in n and the size of M. ■

Note that the source of this complexity is not the choice of R to model time. The PSPACE-hardness result can be proved if we leave the syntax of timed automata unchanged, but use the discrete domain N to model time. Also this complexity is insensitive to the encoding of the constants; the problem is PSPACE-complete even if we encode all constants in unary.

3.4 Intractable problems

In this section we show the universality problem for timed automata to be undecidable. The universality problem is to decide whether the language of a given automaton over Σ comprises of all the timed words over Σ. Specifically, we show that the problem is Π_1^1-hard by reducing a Π_1^1-hard problem of 2-counter machines. The class Π_1^1 consists of highly undecidable problems, including some nonarithmetical sets (for an exposition of the analytical hierarchy consult, for instance, [Rog67]). Note that the universality problem

is same as deciding emptiness of the complement of the language of the automaton. The undecidability of this problem has several implications such as nonclosure under complement and undecidability of testing for language inclusion.

3.4.1 A Σ_1^1-complete problem

A *nondeterministic* 2*-counter machine* M consists of two counters C and D, and a sequence of n instructions. Each instruction may increment or decrement one of the counters, or jump, conditionally upon one of the counters being zero. After the execution of a nonjump instruction, M proceeds nondeterministically to one of the two specified instructions.

We represent a configuration of M by a triple $\langle i, c, d\rangle$, where $1 \leq i \leq n$, $c \geq 0$, and $d \geq 0$ give the values of the location counter and the two counters C and D, respectively. The consecution relation on configurations is defined in the obvious way. A *computation* of M is an infinite sequence of related configurations, starting with the initial configuration $\langle 1, 0, 0\rangle$. It is called *recurring* iff it contains infinitely many configurations in which the location counter has the value 1.

The problem of deciding whether a nondeterministic Turing machine has, over the empty tape, a computation in which the starting state is visited infinitely often, is known to be Σ_1^1-complete [HPS83]. Along the same lines we obtain the following result.

Lemma 3.40 The problem of deciding whether a given nondeterministic 2-counter machine has a recurring computation, is Σ_1^1-hard.

Proof. Every Σ_1^1-formula is equivalent to a Σ_1^1-formula χ of the form

$$\exists f.\, (f(0) = 1 \,\wedge\, \forall x.\, g(f(x), f(x+1))),$$

for a recursive predicate g [HPS83]. For any such χ we can construct a nondeterministic 2-counter machine M that has a recurring computation iff χ is true.

Let M start by computing $f(0) = 1$, and proceed, indefinitely, by nondeterministically guessing the next value of f. At each stage, M checks whether $f(x)$ and $f(x+1)$ satisfy g, and if (and only if) so, it jumps to instruction 1. Such an M exists, because 2-counter machines can, being universal, compute the recursive predicate g. It executes the instruction 1 infinitely often iff a function f with the desired properties exists. ■

3.4.2 Undecidability of the universality problem

Now we proceed to encode the computations of 2-counter machines using timed automata, and use the encoding to prove the undecidability result.

Theorem 3.41 Given a timed automaton over an alphabet Σ the problem of deciding whether it accepts all timed words over Σ is Π_1^1-hard.

PROOF. We encode the computations of a given 2-counter machine M with n instructions using timed words over the alphabet $\{b_1, \ldots b_n, a_1, a_2\}$. A configuration $\langle i, c, d \rangle$ is represented by the sequence $b_i a_1^c a_2^d$. We encode a computation by concatenating the sequences representing the individual configurations. We use the time sequence associated with σ to express that the successive configurations are related as per the requirements of the program instructions. We require that the subsequence of σ corresponding to the time interval $[j, j+1)$ encodes the j-th configuration of the computation. Note that the denseness of the underlying time domain allows the counter values to get arbitrarily large. To express that the number of a_1 (or a_2) symbols in two intervals encoding the successive configurations is the same (or that the number is one less or one greater) we require that every a_1 in the first interval has a matching a_1 at distance 1 and vice versa.

Define a timed language L_{undec} as follows. (σ, τ) is in L_{undec} iff

- $\sigma = b_{i_1} a_1^{c_1} a_2^{d_1} b_{i_2} a_1^{c_2} a_2^{d_2} \cdots$ such that $\langle i_1, c_1, d_1 \rangle, \langle i_2, c_2, d_2 \rangle \cdots$ is a recurring computation of M.
- For all $j \geq 1$, the time of b_{i_j} is j.
- For all $j \geq 1$,
  - if $c_{j+1} = c_j$ then for every a_1 at time t in the interval $(j, j+1)$ there is an a_1 at time $t+1$.
  - if $c_{j+1} = c_j + 1$ then for every a_1 at time t in the interval $(j+1, j+2)$ except the last one, there is an a_1 at time $t-1$.
  - if $c_{j+1} = c_j - 1$ then for every a_1 at time t in the interval $(j, j+1)$ except the last one, there is an a_1 at time $t+1$.

  Similar requirements hold for a_2's.

Clearly, L_{undec} is nonempty iff M has a recurring computation. We will construct a timed automaton $\mathcal{A}_{undec}$ which accepts the complement of L_{undec}. Hence $\mathcal{A}_{undec}$ accepts every timed word iff M does not have a recurring computation. The theorem follows from Lemma 3.40.

The desired automaton $\mathcal{A}_{undec}$ is a disjunction of several TBAs.

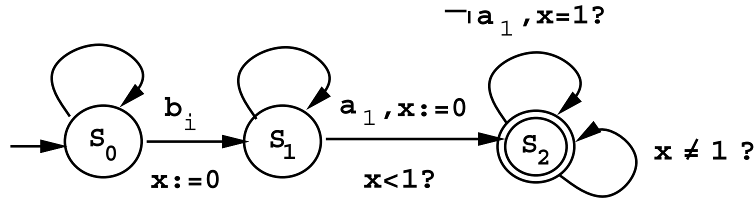
Let $\mathcal{A}_{/}$ be the TBA which accepts (σ, τ) iff for some integer $j \geq 1$, either there is no b symbol at time j, or the subsequence of σ in the time interval $(j, j+1)$ is not of the form $a_1^* a_2^*$. It is easy to construct such a timed automaton.

The subsequence of a timed word (σ, τ) in L_{undec} should encode the initial configuration over the interval $[1, 2)$. Let $\mathcal{A}_{init}$ be the TBA which requires that the subsequence of σ corresponding to the interval $[1, 2)$ is not b_1; it accepts the language $\{(\sigma, \tau) \mid (\sigma_1 \neq b_1) \vee (\tau_1 \neq 1) \vee (\tau_2 < 2)\}$.

For each instruction $1 \leq i \leq n$ we construct a TBA $\mathcal{A}_\rangle$. $\mathcal{A}_\rangle$ accepts (σ, τ) iff the timed word has b_i at some time t, and the configuration corresponding to the subsequence in $[t+1, t+2)$ does not follow from the configuration corresponding to the subsequence in $[t, t+1)$ by executing the instruction i. We give the construction for a sample instruction, say, "increment the counter D and jump nondeterministically to instruction 3 or 5". The automaton $\mathcal{A}_\rangle$ is the disjunction of the following six TBAs $\mathcal{A}_\rangle^\infty, \ldots \mathcal{A}_\rangle^/$.

Let $\mathcal{A}_\rangle^\infty$ be the automaton which accepts (σ, τ) iff for some $j \geq 1$, $\sigma_j = b_i$, and at time $\tau_j + 1$ there is neither b_3 nor b_5. It is easy to construct this automaton.

Let $\mathcal{A}_\rangle^\in$ be the following TBA:



In this figure, an edge without a label means that the transition can be taken on every input symbol. While in state s_2, the automaton cannot accept a symbol a_1 if the condition $(x = 1)$ holds. Thus $\mathcal{A}_\rangle^\in$ accepts (σ, τ) iff there is some b_i at time t followed by an a_1 at time $t' < (t+1)$ such that there is no matching a_1 at time $(t'+1)$.

Similarly we can construct $\mathcal{A}_\rangle^\ni$ which accepts (σ, τ) iff there is some b_i at time t, and for some $t' < (t+1)$ there is no a_1 at time t' but there is an a_1 at time $(t'+1)$. The

complements of $\mathcal{A}_{\rangle}^{\in}$ and $\mathcal{A}_{\rangle}^{\ni}$ together ensure proper matching of a_1's.

Along similar lines we ensure proper matching of a_2 symbols. Let $\mathcal{A}_{\rangle}^{\triangle}$ be the automaton which requires that for some b_i at time t, there is an a_2 at some $t' < (t+1)$ with no match at $(t'+1)$. Let $\mathcal{A}_{\rangle}^{\nabla}$ be the automaton which says that for some b_i at time t there are two a_2's in $(t+1, t+2)$ without matches in $(t, t+1)$. Let $\mathcal{A}_{\rangle}^{/}$ be the automaton which requires that for some b_i at time t the last a_2 in the interval $(t+1, t+2)$ has a matching a_2 in $(t, t+1)$. Now consider a word (σ, τ) such that there is b_i at some time t such that the encoding of a_2's in the intervals $(t, t+1)$ and $(t+1, t+2)$ do not match according to the desired scheme. Let the number of a_2's in $(t, t+1)$ and in $(t+1, t+2)$ be k and l respectively. If $k > l$ then the word is accepted by $\mathcal{A}_{\rangle}^{\triangle}$. If $k = l$, then either there is no match for some a_2 in $(t, t+1)$, or every a_2 in $(t, t+1)$ has a match in $(t+1, t+2)$. In the former case the word is accepted by $\mathcal{A}_{\rangle}^{\triangle}$, and in the latter case it is accepted by $\mathcal{A}_{\rangle}^{/}$. If $k < l$ the word is accepted by $\mathcal{A}_{\rangle}^{\nabla}$.

The requirement that the computation be not recurring translates to the requirement that b_1 appears only finitely many times in σ. Let $\mathcal{A}_{recur}$ be the Büchi automaton which expresses this constraint.

Putting all the pieces together we claim that the language of the disjunction of $\mathcal{A}_{/}$, $\mathcal{A}_{init}$, $\mathcal{A}_{recur}$, and each of $\mathcal{A}_{\rangle}$, is the complement of L_{undec}. ■

This result is not unusual for systems for reasoning about dense real-time. Later we will show the undecidability of certain real-time logics with dense semantics. Obviously, the universality problem for TMAs is also undecidable. We have not been able to show that the universality problem is Π_1^1-complete, an interesting problem is to locate its exact position in the analytical hierarchy. In the following subsections we consider various implications of the above undecidability result.

3.4.3 Inclusion and equivalence

Recall that the language inclusion problem for Büchi automata can be solved in PSPACE. However, it follows from Theorem 3.41 that there is no decision procedure to check whether the language of one TBA is a subset of the other. This result is an obstacle in using timed automata as a specification language for automatic verification of finite-state real-time systems.

Corollary 3.42 Given two TBAs $\mathcal{A}_{\infty}$ and $\mathcal{A}_{\in}$ over an alphabet Σ, the problem of checking $L(\mathcal{A}_{\infty}) \subseteq \mathcal{L}(\mathcal{A}_{\in})$ is Π_1^1-hard.

PROOF. We reduce the universality problem for a given timed automaton A over Σ to the language inclusion problem. Let $\mathcal{A}_{univ}$ be an automaton which accepts every timed word over Σ. The automaton A is universal iff $L(\mathcal{A}_{univ}) \subseteq \mathcal{L}(\mathcal{A})$. ■

Now we consider the problem of testing equivalence of two automata. A natural definition for equivalence of two automata uses equality of the languages accepted by the two. However alternative definitions exist. We will explore one such notion.

Definition 3.43 For timed Büchi automata $\mathcal{A}_\infty$ and $\mathcal{A}_\in$ over an alphabet Σ, define $\mathcal{A}_\infty \sim_\infty \mathcal{A}_\in$ iff $L(\mathcal{A}_\infty) = \mathcal{L}(\mathcal{A}_\in)$. Define $\mathcal{A}_\infty \sim_\in \mathcal{A}_\in$ iff for all timed automata A over Σ, $L(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_\infty)$ is empty precisely when $L(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_\in)$ is empty. ■

For a class of automata closed under complement the above two definitions of equivalence coincide. However, these two equivalence relations differ for the class of timed regular languages because of the nonclosure under complement (to be proved shortly). In fact, the second notion is a weaker notion: $\mathcal{A}_\infty \sim_\infty \mathcal{A}_\in$ implies $\mathcal{A}_\infty \sim_\in \mathcal{A}_\in$, but not vice versa. The motivation behind the second definition is that two automata (modeling two finite-state systems) should be considered different only when a third automaton (modeling the observer or the environment) composed with them gives different behaviors: in one case the composite language is empty, and in the other case there is a possible joint execution. The proof of Theorem 3.41 can be used to show undecidability of this equivalence also. Note that the problems of deciding the two types of equivalences lie at different levels of the hierarchy of undecidable problems.

Theorem 3.44 For timed Büchi automata $\mathcal{A}_\infty$ and $\mathcal{A}_\in$ over an alphabet Σ,

1. The problem of deciding whether $\mathcal{A}_\infty \sim_\infty \mathcal{A}_\in$ is Π^1_1-hard.

2. The problem of deciding whether $\mathcal{A}_\infty \sim_\in \mathcal{A}_\in$ is complete for the co-r.e. class.

PROOF. The language of a given TBA A is universal iff $\mathcal{A} \sim_\infty \mathcal{A}_{univ}$. Hence the Π^1_1-hardness of the universality problem implies Π^1_1-hardness of the first type of equivalence.

Now we show that the problem of deciding nonequivalence, by the second definition, is recursively enumerable. If the two automata are inequivalent then there exists an automaton $\mathcal{A}$ over Σ such that only one of $L(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_\infty)$ and $L(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_\in)$ is empty. Consider the following procedure P: P enumerates all the TBAs over Σ one by one. For each TBA $\mathcal{A}$,

it checks for the emptiness of $L(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_\infty)$ and the emptiness of $L(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_\in)$. If P ever finds different answers in the two cases, it halts saying that $\mathcal{A}_\infty$ and $\mathcal{A}_\in$ are not equivalent.

Finally we prove that the problem of deciding the second type of equivalence is unsolvable. We use the encoding scheme used in the proof of Theorem 3.41. The only difference is that we use the *halting* problem of a *deterministic* 2-counter machine M instead of the recurring computations of a nondeterministic machine. Recall that the halting problem for deterministic 2-counter machines is undecidable. Assume that the n-th instruction is the halting instruction. We obtain $\mathcal{A}'_{undec}$ by replacing the disjunct $\mathcal{A}_{recur}$ by an automaton which accepts (σ, τ) iff b_n does not appear in σ. The complement of $L(\mathcal{A}'_{undec})$ consists of the timed words encoding the halting computation.

We claim that $\mathcal{A}_{univ} \sim_\in \mathcal{A}'_{undec}$ iff the machine M does not halt. If M does not halt then $\mathcal{A}'_{undec}$ accepts all timed words, and hence, its language is the same as that of $\mathcal{A}_{univ}$. If M halts, then we can construct a timed automaton $\mathcal{A}_{halt}$ which accepts a particular timed word encoding the halting computation of M. If M halts in k steps, then $\mathcal{A}_{halt}$ uses k clocks to ensure proper matching of the counter values in successive configurations. The details are very similar to the PSPACE-hardness proof of Theorem 3.39. $L(\mathcal{A}_{halt}) \cap \mathcal{L}(\mathcal{A}_{univ})$ is nonempty whereas $L(\mathcal{A}_{halt}) \cap \mathcal{L}(\mathcal{A}'_{undec})$ is empty, and thus $\mathcal{A}_{univ}$ and $\mathcal{A}'_{undec}$ are inequivalent in this case. This completes the proof. ■

3.4.4 Nonclosure under complement

The Π^1_1-hardness of the inclusion problem implies that the class of TBAs is not closed under complement.

Corollary 3.45 The class of timed regular languages is not closed under complementation.

PROOF. Given TBAs $\mathcal{A}_\infty$ and $\mathcal{A}_\in$ over an alphabet Σ, $L(\mathcal{A}_\infty) \subseteq \mathcal{L}(\mathcal{A}_\in)$ iff the intersection of $L(\mathcal{A}_\infty)$ and the complement of $L(\mathcal{A}_\in)$ is empty. Assume that TBAs are closed under complement. Consequently, $L(\mathcal{A}_\infty) \not\subseteq \mathcal{L}(\mathcal{A}_\in)$ iff there is a TBA A such that $L(\mathcal{A}_\infty) \cap \mathcal{L}(\mathcal{A})$ is nonempty, but $L(\mathcal{A}_\in) \cap \mathcal{L}(\mathcal{A})$ is empty. That is, $L(\mathcal{A}_\infty) \not\subseteq \mathcal{L}(\mathcal{A}_\in)$ iff $\mathcal{A}_\infty$ and $\mathcal{A}_\in$ are inequivalent according to $\sim_2$. From Theorem 3.44 it follows that the complement of the inclusion problem is recursively enumerable. This contradicts the Π^1_1-hardness of the inclusion problem. ■

The following example provides some insight regarding the nonclosure under complementation.

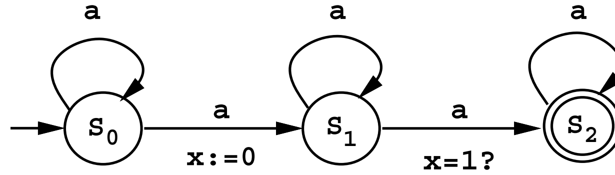


Figure 3.11: Noncomplementable automaton

Example 3.46 The language accepted by the automaton of Figure 3.11 over $\{a\}$ is

$$\{(a^\omega, \tau) \mid \exists i \geq 1.\, \exists j > i.\, (\tau_j = \tau_i + 1)\}.$$

The complement of this language cannot be characterized using a TBA. The complement needs to make sure that no pair of a's is separated by distance 1. Since there is no bound on the number of a's that can happen in a time period of length 1, keeping track of the times of all the a's within past 1 time unit, would require an unbounded number of clocks. ■

3.5 Deterministic timed automata

The results of Section 3.4 show that the class of timed automata is not closed under complement, and one cannot automatically compare the languages of two automata. In this section we define deterministic timed automata, and show that the class of deterministic timed Muller automata (DTMA) is closed under all the Boolean operations.

3.5.1 Definition

Recall that in the untimed case a deterministic transition table has a single start state, and from each state, given the next input symbol, the next state is uniquely determined. We want a similar criterion for determinism for the timed automata: given an extended state and the next input symbol *along with its time of occurrence*, the extended state after the next transition should be uniquely determined. So we allow multiple transitions starting at the same state with the same label, but require their clock constraints to be *mutually exclusive* so that at any time only one of these transitions is enabled.

Definition 3.47 A timed transition table $\langle \Sigma, S, S_0, C, E \rangle$ is called *deterministic* iff

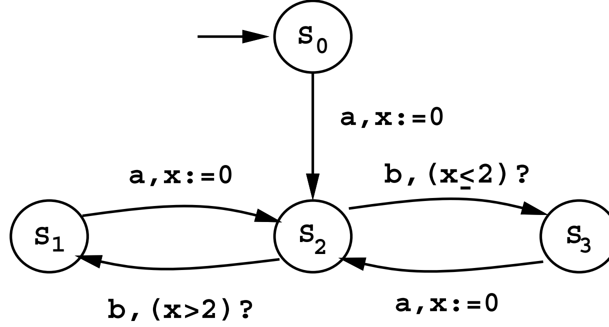


Figure 3.12: Deterministic timed Muller automaton

1. it has only one start state, $|S_0| = 1$, and
2. for all $s \in S$, for all $a \in \Sigma$, for every pair of edges of the form $\langle s, -, a, -, \delta_1\rangle$ and $\langle s, -, a, -, \delta_2\rangle$, the clock constraints δ_1 and δ_2 are mutually exclusive (i.e., $\delta_1 \wedge \delta_2$ is unsatisfiable).

A timed automaton is deterministic iff its timed transition table is deterministic. ■

Note that in absence of clocks the above definition matches with the definition of determinism for transition tables. Thus every deterministic transition table is also a deterministic timed transition table. Let us consider an example of a DTMA.

Example 3.48 The DTMA of Figure 3.12 accepts the language L_{crt} of Example 3.15

$$L_{\mathrm{crt}} = \{((ab)^\omega, \tau) \mid \exists i. \forall j \geq i. (\tau_{2j+2} \leq \tau_{2j+1} + 2)\}$$

The Muller acceptance family is given by $\{\{s_2, s_3\}\}$. The state s_2 has two mutually exclusive outgoing transitions on b. The acceptance condition requires that the transition with the clock constraint $(x > 2)$ is taken only finitely often. ■

Deterministic timed automata can be easily complemented because of the following property:

Lemma 3.49 A deterministic timed transition table has at most one run over a given timed word.

PROOF. Consider a deterministic timed transition table A, and a timed word (σ, τ). The run starts at time 0 with the extended state $\langle s_0, \nu_0 \rangle$ where s_0 is the unique start state. Suppose the extended state of A at time τ_{j-1} is $\langle s, \nu \rangle$, and the run has been constructed up to $(j-1)$ steps. By the deterministic property of A, at time τ_j there is at most one transition $\langle s, s', \sigma_j, \delta, \lambda \rangle$ such that the clock interpretation at time τ_j, $\nu + \tau_j - \tau_{j-1}$, satisfies δ. If such a transition does not exist then A has no run over (σ, τ). Otherwise, this choice of transition uniquely extends the run to the j-th step, and determines the extended state at time τ_j. The lemma follows by induction. ■

3.5.2 Closure properties

Now we consider the closure properties for deterministic timed automata. Like deterministic Muller automata, DTMAs are also closed under all Boolean operations.

Theorem 3.50 The class of timed languages accepted by deterministic timed Muller automata is closed under union, intersection, and complementation.

PROOF. We define a transformation on DTMAs to make the proofs easier; for every DTMA $\mathcal{A} = \langle \pm, \mathrm{S}, \int_{/}, \mathrm{C}, \mathrm{E}, \mathcal{F} \rangle$ we construct another DTMA $\mathcal{A}^*$ by *completing* $\mathcal{A}$ as follows. First we add a dummy state q to the automaton. From each state s, for each symbol a, we add an a-labeled edge from s to q. The clock constraint for this edge is the negation of the disjunction of the clock constraints of all the a-labeled edges starting at s. We leave the acceptance condition unchanged. This construction preserves determinism as well as the set of accepted timed words. The new automaton $\mathcal{A}^*$ has the property that for each state s and each input symbol a, the disjunction of the clock constraints of the a-labeled edges starting at s is a valid formula. Observe that $\mathcal{A}^*$ has precisely one run over any timed word. We call such an automaton *complete*. In the remainder of the proof we assume each DTMA to be complete.

Let $\mathcal{A}_\rangle = \langle \pm, \mathrm{S}_\rangle, \int_{/_\rangle}, \mathrm{C}_\rangle, \mathrm{E}_\rangle, \mathcal{F}_\rangle \rangle$, for $i = 1, 2$, be two complete DTMAs with disjoint sets of clocks. First we construct a timed transition table A using a product construction. The set of states of A is $\mathrm{S}_1 \times \mathrm{S}_2$. Its start state is $\langle s_{0_1}, s_{0_2} \rangle$. The set of clocks is $\mathrm{C}_1 \cup \mathrm{C}_2$. The transitions of A are defined by coupling the transitions of the two automata having the same label. Corresponding to an $\mathcal{A}_\infty$-transition $\langle s_1, t_1, a, \lambda_1, \delta_1 \rangle$ and an $\mathcal{A}_\in$-transition $\langle s_2, t_2, a, \lambda_2, \delta_2 \rangle$, A has a transition $\langle \langle s_1, s_2 \rangle, \langle t_1, t_2 \rangle, a, \lambda_1 \cup \lambda_2, \delta_1 \wedge \delta_2 \rangle$. It is easy to check that A is also deterministic. A has a unique run over each (σ, τ), and this run can be

obtained by putting together the unique runs of $\mathcal{A}_\rangle$ over (σ, τ).

Let $\mathcal{F}^1$ consist of the sets $\mathrm{F} \subseteq \mathrm{S}_1 \times \mathrm{S}_2$ such that the projection of F onto the first component is an accepting set of $\mathcal{A}_\infty$; that is,

$$\mathcal{F}^1 \;=\; \{\mathrm{F} \subseteq \mathrm{S}_1 \times \mathrm{S}_2 \mid \{s \in \mathrm{S}_1 \mid \exists s' \in \mathrm{S}_2.\, \langle s, s' \rangle \in \mathrm{F}\} \in \mathcal{F}_1\}.$$

Hence a run r of A is an accepting run for $\mathcal{A}_\infty$ iff $\mathit{inf}(r) \in \mathcal{F}^1$. Similarly define $\mathcal{F}^2$ to consist of the sets F such that $\{s' \mid \exists s \in \mathrm{S}_1.\, \langle s, s' \rangle \in \mathrm{F}\}$ is in $\mathcal{F}_2$. Now coupling A with the Muller acceptance family $\mathcal{F}^1 \cup \mathcal{F}^2$ gives a DTMA accepting $L(\mathcal{A}_\infty) \cup \mathcal{L}(\mathcal{A}_\in)$, whereas using the acceptance family $\mathcal{F}^1 \cap \mathcal{F}^2$ gives a DTMA accepting $L(\mathcal{A}_\infty) \cap \mathcal{L}(\mathcal{A}_\in)$.

Finally consider complementation. Let A be a complete DTMA $\langle \Sigma, \mathrm{S}, s_0, \mathrm{C}, \mathrm{E}, \mathcal{F} \rangle$. A has exactly one run over a given timed word. Hence, (σ, τ) is in the complement of $L(\mathcal{A})$ iff the run of A over it does not meet the acceptance criterion of A. The complement language is, therefore, accepted by a DTMA which has the same underlying timed transition table as A, but its acceptance condition is given by $2^{\mathrm{S}} - \mathcal{F}$. ■

Now let us consider the closure properties of DTBAs. Recall that deterministic Büchi automata (DBA) are not closed under complement. The property that "there are infinitely many a's" is specifiable by a DBA, however, the complement property, "there are only finitely many a's" cannot be expressed by a DBA. Consequently we do not expect the class of DTBAs to be closed under complementation. However, since every DTBA is also a DTMA, the complement of a DTBA-language is accepted by a DTMA. The next theorem states the closure properties.

Theorem 3.51 The class of timed languages accepted by DTBAs is closed under union and intersection, but not closed under complement. The complement of a DTBA language is accepted by some DTMA.

Proof. For the case of union, we construct the product transition table as in case of DTMAs (see proof of Theorem 3.50). The accepting set is $\{\langle s, s' \rangle \mid s \in \mathrm{F}_1 \vee s' \in \mathrm{F}_2\}$.

A careful inspection of the product construction for TBAs (see proof of Theorem 3.17) shows that it preserves determinism. The closure under intersection for DTBAs follows.

The nonclosure of deterministic Büchi automata under complement leads to the nonclosure for DTBAs under complement. The language $\{(\sigma, \tau) \mid \sigma \in (b^* a)^\omega\}$ is specifiable by a DBA. Its complement language $\{(\sigma, \tau) \mid \sigma \in (a + b)^* b^\omega\}$ is not specifiable by a DTBA.

This claim follows from Lemma 3.53 (to be proved shortly), and the fact that the language $(a+b)^*b^\omega$ is not specifiable by a DBA.

Let $\mathcal{A} = \langle \pm, \mathrm{S}, \int_\prime, \mathrm{C}, \mathrm{E}, \mathrm{F} \rangle$ be a complete deterministic automaton. (σ, τ) is in the complement of $L(\mathcal{A})$ iff the (unique) run of A over it does not meet the acceptance criterion of A. The complement language is, therefore, accepted by a DTMA with the same underlying timed transition table as A, and the acceptance family $2^{\mathrm{S}-\mathrm{F}}$. ∎

3.5.3 Decision problems

In this section we examine the complexity of the emptiness problem and the language inclusion problem for deterministic timed automata.

The emptiness of a timed automaton does not depend on the symbols labeling its edges. Consequently, checking emptiness of deterministic automata is no simpler; it is PSPACE-complete.

Since deterministic automata can be complemented, checking for language inclusion is decidable. In fact, while checking $L(\mathcal{A}_\infty) \subseteq \mathcal{L}(\mathcal{A}_\in)$, only $\mathcal{A}_\in$ need be deterministic, $\mathcal{A}_\infty$ can be nondeterministic. The problem can be solved in PSPACE:

Theorem 3.52 For a timed automaton $\mathcal{A}_\infty$ and a deterministic timed automaton $\mathcal{A}_\in$, the problem of deciding whether $L(\mathcal{A}_\infty)$ is contained in $L(\mathcal{A}_\in)$ is PSPACE-complete.

PROOF. PSPACE-hardness follows, even when $\mathcal{A}_\infty$ is deterministic, from the fact that checking for the emptiness of the language of a deterministic timed automaton is PSPACE-hard. Let $\mathcal{A}_{empty}$ be a deterministic automaton which accepts the empty language. Now for a deterministic time automaton A, $L(\mathcal{A})$ is empty iff $L(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_{empty})$.

Observe that $L(\mathcal{A}_\infty) \subseteq \mathcal{L}(\mathcal{A}_\in)$ iff the intersection of $L(\mathcal{A}_\infty)$ with the complement of $L(\mathcal{A}_\in)$ is empty. Recall that complementing the language of a deterministic automaton corresponds to complementing the acceptance condition. First we construct a timed transition table A from the timed transition tables of $\mathcal{A}_\infty$ and $\mathcal{A}_\in$ using the product construction (see proof of Theorem 3.50). The size of $\mathcal{A}$ is proportional to the product of the sizes of $\mathcal{A}_\rangle$. Then we construct the region automaton $R(\mathcal{A})$. $L(\mathcal{A}_\infty) \not\subseteq \mathcal{L}(\mathcal{A}_\in)$ iff $R(\mathcal{A})$ has a cycle which is accessible from its start state, meets the progressiveness requirement, the acceptance criterion for $\mathcal{A}_\infty$, and the complement of the acceptance criterion for $\mathcal{A}_\in$. The existence of such a cycle can be checked in space polynomial in the size of A, as in the proof of PSPACE-solvability of emptiness (Theorem 3.39). ∎

3.5.4 Expressiveness

In this section we compare the expressive power of the various types of timed automata.

Every DTBA can be expressed as a DTMA simply by rewriting its acceptance condition. However the converse does not hold. First observe that every ω-regular language is expressible as a DMA, and hence as a DTMA. On the other hand, since deterministic Büchi automata are strictly less expressive than deterministic Muller automata, certain ω-regular languages are not specifiable by DBAs. The next lemma shows that such languages cannot be expressed using DTBAs either. It follows that DTBAs are strictly less expressive than DTMAs. In fact, DTMAs are closed under complement, whereas DTBAs are not.

Lemma 3.53 For an ω-language L, the timed language $\{(\sigma, \tau) \mid \sigma \in L\}$ is accepted by some DTBA iff L is accepted by some DBA.

PROOF. Clearly if L is accepted by a DBA, then $\{(\sigma, \tau) \mid \sigma \in L\}$ is accepted by the same automaton considered as a timed automaton.

Now suppose that the language $\{(\sigma, \tau) \mid \sigma \in L\}$ is accepted by some DTBA A. We construct another DTBA $\mathcal{A}'$ such that $L(\mathcal{A}') = \{(\sigma, \tau) \mid (\sigma \in \mathcal{L}) \wedge \forall \rangle. (\tau_\rangle = \rangle)\}$. $\mathcal{A}'$ requires time to increase by 1 at each transition. The automaton $\mathcal{A}'$ can be obtained from $\mathcal{A}$ by introducing an extra clock x. We add the conjunct $x = 1$ to the clock constraint of every edge in A and require it to be reset on every edge. $\mathcal{A}'$ is also deterministic.

The next step is the untiming construction for $\mathcal{A}'$. Observe that $Untime(L(\mathcal{A}')) = \mathcal{L}$. While constructing $R(\mathcal{A}')$ we need to consider only those clock regions which have all clocks with zero fractional parts. Since the time increase at every step is predetermined, and $\mathcal{A}'$ is deterministic, it follows that $R(\mathcal{A}')$ is a deterministic transition table. We need not check the progressiveness condition also. It follows that the automaton constructed by the untiming procedure is a DBA accepting L. ■

From the above discussion one may conjecture that a DTMA language L is a DTBA language if $Untime(L)$ is a DBA language. To answer this let us consider the convergent response property L_{crt} specifiable using a DTMA (see Example 3.48). This language involves a combination of liveness and timing. We conjecture that no DTBA can specify this property.

Along the lines of the above proof we can also show that for an ω-language L, the timed language $\{(\sigma, \tau) \mid \sigma \in L\}$ is accepted by some DTMA (or TMA, or TBA) iff L is accepted by some DMA (or MA, or BA, respectively).

Class of timed languages	Operations closed under
TMA = TBA	union, intersection
$\cup$	
DTMA	union, intersection, complement
$\cup$	
DTBA	union, intersection

Figure 3.13: Classes of timed automata

Class of ω-languages	Operations closed under
MA = BA = DMA	union, intersection, complement
$\cup$	
DBA	union, intersection

Figure 3.14: Classes of ω-automata

Since DTMAs are closed under complement, whereas TMAs are not, it follows that the class of languages accepted by DTMAs is strictly smaller than that accepted by TMAs. In particular, the language of Example 3.46, ("some pair of a's is distance 1 apart") is not representable as a DTMA; it relies on nondeterminism in a crucial way.

We summarize the discussion on various types of automata in the table of Figure 3.13 which shows the inclusions between various classes and the closure properties of various classes. Compare this with the corresponding results for the various classes of ω-automata shown in Figure 3.14.

3.6 Variants of timed automata

In this section we consider some of the ways to modify our definition of timed automata and indicate how these decisions affect the expressiveness and complexity of different problems.

3.6.1 Clock constraints

Recall that our definition of the clock constraints allows Boolean combinations of atomic formulas which compare clock values with (rational) constants. With this vocabulary, timed

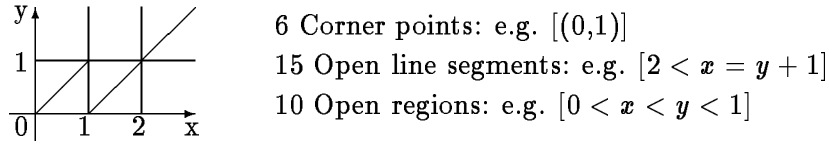


Figure 3.15: Clock regions for extended clock constraints

automata can express only constant bounds on the delays between transitions. We consider two extensions; one allows comparisons between two clock values, and the other allows clock values to be added.

Comparing two clocks

First we extend the definition of clock constraints to allow subformulas involving two clocks such as $(x \leq y + c)$. In particular, in Definition 3.8 of the set $\Phi(X)$ of clock constraints, we allow, as atomic constraints, the conditions $(x \leq y + c)$ and $(x + c \leq y)$, for $x, y \in X$ and $c \in \mathsf{Q}$. Thus the allowed clock constraints are quantifier-free formulas using the primitives of comparison $(\leq)$ and addition by rational constants $(+c)$.

The untiming construction can handle this extension very easily. We need to refine the equivalence relation on clock interpretations. Now, in addition to the previous conditions, we require that two equivalent clock interpretations agree on all the subformulas appearing in the clock constraints. Instead of giving the details we only consider the regions in Example 3.26 again.

Example 3.54 As before the timed transition table has two clocks x and y with $c_x = 2$ and $c_y = 1$. Suppose the formula $(x \leq y + 1)$ appears as a clock constraint labeling one of the edges, and this is the only condition involving both the clocks. The clock regions are shown in Figure 3.15. Comparing it to the clock regions in Figure 3.9 notice that the region $[(x > 2), (y > 1)]$ is split into three regions according to the relationship between x and $(y + 1)$. ■

The region graph is constructed as before. The complexity of the algorithm for testing emptiness remains the same.

This extension of clock constraints does not add to the expressiveness of timed automata. We can get rid of the constraints of the form $(x \leq y + c)$ or $(x + c \leq y)$ in a systematic fashion. For instance, consider the constraint $(x \leq y + 1)$. First we tag each state of the automaton with the truth value of this constraint. In the initial states, the constraint is true. Now consider an edge e from state s to s'. If x gets reset along e, then $(x \leq y + 1)$ must be true in s'. If only y gets reset along e, then we add one of the conjuncts $(x \leq 1)$ and $(x > 1)$ to the clock constraint of e depending on whether $(x \leq y + 1)$ is true or false in s'. If none of the clocks x and y are reset along e, then the tag of s' must be the same as that of s. Once all the states are tagged in the above manner, we can simplify the clock constraint of every edge e by replacing the formula $(x \leq y + 1)$ by true or false depending upon the tag of the source state of e.

Introducing the addition primitive

Next we allow the primitive of addition in the clock constraints. Now we can write clock constraints such as $(x + y \leq x' + y')$ which allow the automaton to compare various delays. This greatly increases the expressiveness of the formalism. The language of the automaton in the following example is not timed regular.

Example 3.55 Consider the automaton of Figure 3.16 with the alphabet $\{a, b, c\}$. It expresses the property that the symbols a, b, and c alternate, and the delay between b and c is always twice the delay between the last pair of a and b. The language is defined by

$$\{((abc)^\omega, \tau) \mid \forall j. [(\tau_{3j} - \tau_{3j-1}) = 2(\tau_{3j-1} - \tau_{3j-2})]\}.$$

■

Intuitively, the constraints involving addition are too powerful and cannot be implemented by finite-state systems. Even if we constrain all events to occur at integer time values (i.e., discrete-time model), to check that the delay between first two symbols is same as the delay between the next two symbols, an automaton would need an unbounded memory. Thus with finite resources, an automaton can compare delays with constants, but cannot remember delays. In fact, we can show that introducing addition in the syntax of clock constraints makes the emptiness problem for timed automata undecidable.

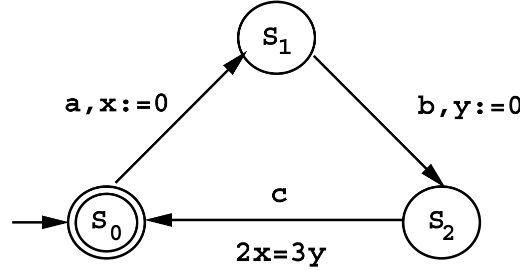


Figure 3.16: Automaton with clock constraints using +

Theorem 3.56 Allowing the addition primitive in the syntax of clock constraints makes the emptiness problem for timed automata Π_1^1-hard.

PROOF. As in the proof of Theorem 3.41 we reduce the problem of recurring computations of nondeterministic 2-counter machines to the emptiness problem for time automata using the primitive $+$. The alphabet is $\{a, b_1, \ldots b_n\}$. We say that a timed word (σ, τ) encodes a computation $\langle i_1, c_1, d_1\rangle, \langle i_2, c_2, d_2\rangle \cdots$ of the 2-counter machine iff $\sigma = b_{i_1} a b_{i_2} a b_{i_3} \cdots$ with $\tau_{2j} - \tau_{2j-1} = c_j$, and $\tau_{2j+1} - \tau_{2j} = d_j$ for all $j \geq 1$. Thus the delay between b and the following a encodes the value of the counter C, and the delay between a and the following b encodes the value of D. We construct a timed automaton which accepts precisely the timed words encoding the recurring computations of the machine. The primitive of $+$ is used to express a consecution requirement such as the value of the counter C remains unchanged. The details of the proof are quite straightforward. ■

3.6.2 Timed automata with ϵ-transitions

The definition of nondeterministic finite-state automata sometimes allows the automaton to make transitions without consuming any input (see, for instance, [HU79]). We can allow such silent transitions for timed automata also. Thus now we allow the edges to be labeled with ϵ.

Before we define these automata precisely, let us consider an example.

Example 3.57 The automaton of Figure 3.17 accepts a timed word over $\{a\}$ iff there is some a at time t with no a at time $t+1$. The language is given by

$$\{(a^\omega, \tau) \mid \exists i. \forall j. (\tau_j \neq \tau_i + 1)\}.$$

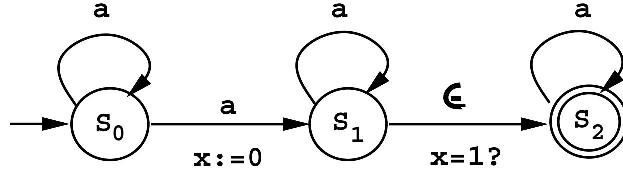


Figure 3.17: Timed automaton with ϵ-transitions

■

If we view the automaton of Figure 3.17 as accepting timed words over the alphabet $\{a, \epsilon\}$, no new definitions are required. However, we want to eliminate ϵ's from the timed words accepted by the automaton. Consequently we need to account for the possibility that there may be only finitely many visible transitions along the run. This calls for changing the definition of the timed languages to allow timed words of finite length also. In fact, it is useful to develop the entire theory of timed languages allowing both finite and infinite words. We avoided this for the sake of simplicity of presentation; redefining all the constructions to handle such languages poses no extra problems.

Definition 3.58 The pair (σ, τ) is a finite timed word over Σ iff σ is a finite sequence over Σ, and τ is finite sequence over positive real numbers such that

- both σ and τ are of the same length,
- τ satisfies monotonicity; for each $i < |\tau|$, $\tau(i) < \tau(i+1)$.

A *mixed timed language* over Σ consists of both infinite and finite timed words over Σ.

■

The Boolean operations are defined on mixed timed languages the usual way. In addition we can define a projection operation also. The result of projecting a timed word (σ, τ), finite or infinite, over Σ, onto a subset Σ' of Σ, is obtained by deleting from σ the symbols not in Σ', and discarding the associated times from the time sequence.

Definition 3.59 Let (σ, τ) be a finite or infinite timed word over Σ, and let $\Sigma' \subseteq \Sigma$. Let $\pi = i_1, i_2, \ldots$ be the (finite or infinite) increasing sequence of integers such that $j \in \pi$ iff $\sigma_j \in \Sigma'$ for all $j \leq |\tau|$.

The *projection* of (σ, τ) on Σ', denoted by $(\sigma, \tau)\lceil\Sigma'$, is the (finite or infinite) timed word (σ', τ') with $|\sigma'| = |\tau'| = |\pi|$, such that $\sigma'_j = \sigma_{i_j}$ and $\tau'_j = \tau_{i_j}$ for all $j \leq |\pi|$.

For a mixed timed language L over Σ, the projection is defined by

$$L\lceil\Sigma' = \{(\sigma, \tau)\lceil\Sigma' \mid (\sigma, \tau) \in L\}.$$

■

Observe that from an infinite timed word we may get a finite one using projection. For instance, after projecting the timed word

$$(a, 1) \rightarrow (b, 2.3) \rightarrow (b, 4) \rightarrow (a, 5) \rightarrow (a, 6) \rightarrow (a, 7) \rightarrow \cdots$$

onto $\{b\}$ we get the finite timed word $(b, 2.3)(b, 4)$.

The timed automata with ϵ-transitions accept mixed timed languages.

Definition 3.60 An ϵ-TBA over Σ is a timed Büchi automaton A over the alphabet $\Sigma \cup \{\epsilon\}$.

The mixed timed language accepted by A is defined to be the projection onto Σ of the language of infinite timed words over $\Sigma \cup \{\epsilon\}$ accepted by A. ■

If an ϵ-TBA has no cycles consisting entirely of ϵ-labeled edges then its language has only infinite words. Observe that in an ϵ-TBA, putting a self-loop with the ϵ-label (and no resets or clock constraints) on a nonaccepting state does not change its language.

Similarly we can define ϵ-TMAs. Along the lines of Theorem 3.22 we can show that ϵ-TMAs and ϵ-TBAs accept the same class of mixed timed languages. This is the class of *mixed timed regular languages*. The next theorem considers the closure properties for this class.

Theorem 3.61 The class of mixed timed languages accepted by ϵ-TBAs is closed under union, intersection, and projection.

PROOF. The case of union is obvious.

Closure under intersection can be shown by modifying the product construction for TBAs (see proof of Theorem 3.17). As before, while constructing the intersection of ϵ-TBAs $\mathcal{A}_\rangle$, $i = 1, 2 \ldots n$, the product automaton has states of the form $\langle s_1, \ldots s_n, k\rangle$, where each s_i is a state of $\mathcal{A}_\rangle$ and k is the counter handling the acceptance criteria. For $a \in \Sigma$, all the a-labeled transitions of the product automaton are obtained as before by coupling

a-labeled transitions of the individual automata. However not all the component automata need to participate in the ϵ-labeled transitions. Let J be some subset of $\{1, 2 \ldots n\}$, and let $\{\langle s_i, s_i', \epsilon, \lambda_i, \delta_i \rangle \in \mathrm{E}_i \mid i \in J\}$ be a set of transitions. Then the product automaton A has an ϵ-transition out of $\langle s_1, \ldots s_n, k \rangle$ to $\langle s_1', \ldots s_n', k' \rangle$ with $s_i' = s_i$ if $i \notin J$. The new counter value k' is $(k+1) \bmod n$ if $s_k \in \mathrm{F}_k$ and $k \in J$, otherwise k' equals k.

Given an ϵ-TBA A over Σ, to project its language onto Σ', we simply change every edge label not in Σ' to ϵ. ■

It should be obvious that testing for emptiness is no harder for ϵ-TBAs, and the problem is PSPACE-complete. The lower bound of Π_1^1-hardness for checking universality of TBAs applies to ϵ-TBAs also. Also like TBAs, these automata are not closed under complement.

In the untimed case, it is known that allowing ϵ-transitions does not add to the expressive power of the automata. The same question can be asked for timed automata also, namely, given an ϵ-TBA A does there always exist a TBA that accepts all the infinite timed words in $L(\mathcal{A})$? We conjecture that this indeed is the case, however, we do not have a construction to eliminate ϵ-transitions from a given timed automaton.

3.7 Verification

In this section we discuss how to use the theory of timed automata to prove correctness of finite-state real-time systems. We start with an overview of the application of Büchi automata to verify untimed processes [VW86, Var87].

3.7.1 ω-automata and verification

We review verification in the untimed case to set a base for the extension to the timed case. Recall, from Section 2.1, that an untimed process is a pair (A, X), where A is the set of its observable events and X is the set of its possible traces. Observe that L is an ω-language over the alphabet $\mathcal{P}^+(A)$. If it is a regular language it can be represented by a Büchi automaton.

We model a finite-state (untimed) process P with event set A using a Büchi automaton $\mathcal{A}_\mathcal{P}$ over the alphabet $\mathcal{P}^+(A)$. The states of the automaton correspond to the internal states of the process. The automaton $\mathcal{A}_\mathcal{P}$ has a transition $\langle s, s', a \rangle$, with $a \subseteq A$, if the process can change its state from s to s' participating in the events from a. The acceptance conditions

of the automaton correspond to the fairness constraints on the process. The automaton $\mathcal{A}_{\mathcal{P}}$ accepts (or generates) precisely the traces of P; that is, the process P is given by $(A, L(\mathcal{A}_{\mathcal{P}}))$. Such a process P is called an *ω-regular process*.

The user describes a system consisting of various components by specifying each individual component as a Büchi automaton. In particular, consider a system I comprising of n components, where each component is modeled as an ω-regular process $P_i = (A_i, L(\mathcal{A}_\rangle))$. The implementation process is $[\|_i \ P_i]$. We can automatically construct the automaton for I using the construction for language intersection for Büchi automata. Since the event sets of various components may be different, before we apply the product construction, we need to make the alphabets of various automata identical. Let $A = \cup_i A_i$. From each $\mathcal{A}_\rangle$, we construct an automaton $\mathcal{A}'_\rangle$ over the alphabet $\mathcal{P}^+(A)$ such that $L(\mathcal{A}'_\rangle) = \{\sigma \in \mathcal{P}^+(A)^\omega \mid \sigma\lceil\mathcal{A}_\rangle \in \mathcal{L}(\mathcal{A}_\rangle)\}$. Now the desired automaton $\mathcal{A}_{\mathcal{I}}$ is the product of the automata $\mathcal{A}'_\rangle$. The details of the construction will be given only for the timed case.

The specification is given as an ω-regular language S over $\mathcal{P}^+(A)$. The implementation meets the specification iff $L(\mathcal{A}_{\mathcal{I}}) \subseteq \mathcal{S}$. The property S can presented as a Büchi automaton $\mathcal{A}_{\mathcal{S}}$. In this case, the verification problem reduces to checking emptiness of $L(\mathcal{A}_{\mathcal{I}}) \cap \mathcal{L}(\mathcal{A}_{\mathcal{S}})^{\rfloor}$.

The verification problem is PSPACE-complete. The size of $\mathcal{A}_{\mathcal{I}}$ is exponential in the description of its individual components. If $\mathcal{A}_{\mathcal{S}}$ is nondeterministic, taking the complement involves an exponential blow-up, and thus the complexity of verification problem is exponential in the size of the specification also. However, if $\mathcal{A}_{\mathcal{S}}$ is deterministic, then the complexity is only polynomial in the size of the specification. Specifications can be written as formulas of linear temporal logic PTL also. If the formula ϕ of PTL gives the specification, we construct a Büchi automaton $\mathcal{A}_{\neg\phi}$ which accepts all traces that do not satisfy ϕ. The next step is checking for the emptiness of $L(\mathcal{A}_{\mathcal{I}}) \cap \mathcal{L}(\mathcal{A}_{\neg\phi})$. This method is also exponential in the size of the formula.

Even if the size of the specification and the sizes of the automata for the individual components are small, the number of components in most systems of interest is large, and in the above method the complexity is exponential in this number. Thus the product automaton $\mathcal{A}_{\mathcal{I}}$ has prohibitively large number of states, and this limits the applicability of this approach. Alternative methods which avoid enumeration of all the states in $\mathcal{A}_{\mathcal{I}}$ have been proposed, and shown to be applicable to verification of some moderately sized systems [BCD+90, GW91].

3.7.2 Verification using timed automata

Recall the definition of a timed process from Section 2.2.1. We use the dense-time model with $TIME = \mathsf{R}$. Thus, a timed process is a pair (A, L) where A is a finite set of events, and L is a timed language over $\mathcal{P}^+(A)$. A *timed regular process* is the one for which the set L is a timed regular language, and can be represented by a timed automaton.

Finite-state systems are modeled by TBAs. The underlying transition table gives the state-transition graph of the system. We have already seen how the clocks can be used to represent the timing delays of various physical components. As before, the acceptance conditions correspond to the fairness conditions. Notice that the progress requirement imposes certain fairness requirements implicitly. Thus, with a finite-state process P, we associate a TBA $\mathcal{A}_\mathcal{P}$ such that $L(\mathcal{A}_\mathcal{P})$ consists of precisely the timed traces of P.

Typically, an implementation is described as a composition of several components. Each component should be modeled as a timed regular process $P_i = (A_i, L(\mathcal{A}_\rangle))$. The first step in the verification process is to construct a TBA $\mathcal{A}_\mathcal{I}$ which represents the composite process $[\|_i\ P_i]$. To implement this, first we need to make the alphabets of various automata identical, and then take the intersection. Combining the two steps, however, reduces the size of the implementation automaton.

Theorem 3.62 Given timed processes $P_i = (A_i, L(\mathcal{A}_\rangle))$, $i = 1, \ldots n$, represented by timed automata $\mathcal{A}_\rangle$, there is a TBA A over the alphabet $\mathcal{P}^+(\cup_i A_i)$ which represents the timed process $[\|_i\ P_i]$.

PROOF. The construction is very similar to the one for Theorem 3.17. The main difference is that for a joint transition of the product automaton, the event sets labeling the transitions of the individual automata need not be the same. In fact, for an event set a such that $a \cap A_i = \emptyset$, the i-th automaton does not participate at all in a transition labeled with a. In addition to checking the acceptance criteria of all the automata, we need to ensure that every automaton participates in infinitely many transitions along an accepting run of A. This calls for some modifications in the strategy for the counter update.

The states of the product automaton A are of the form $\langle s_1, \ldots s_n, k\rangle$, where $s_i \in \mathrm{S}_i$, and $1 \leq k \leq (n+1)$. Initially the counter value is 1, and it is incremented from k to $(k+1)$ when $\mathcal{A}_{\|}$ participates in a transition to one of its accepting states. When the counter reaches $(n+1)$ it is reset to 1. A state is an accepting state iff the counter value is $(n+1)$. Now we define the transitions of A labeled with an event set $a \subseteq \cup_i A_i$. Let $J = \{i \mid A_i \cap a \neq \emptyset\}$.

Consider a family of transitions $\{\langle s_i, s'_i, a \cap A_i, \lambda_i, \delta_i\rangle \in \mathrm{E}_i \mid i \in J\}$. Corresponding to this family, there is a joint transition of A labeled with a out of each state of the form $\langle s_1, \ldots s_n, k\rangle$. The new state is $\langle s'_1, \ldots s'_n, j\rangle$, where, for $i \notin J$, $s'_i = s_i$. If $k = (n+1)$ then $j = 1$. Otherwise if $k \in J$ and $s'_k \in \mathrm{F}_k$ then $j = (k+1)$ else $j = k$. The set of clocks to be reset is $\cup_{i \in J} \lambda_i$, and the clock constraint is $\wedge_{i \in J} \delta_i$. ■

The number of states in $\mathcal{A}_\mathcal{I}$ is $[(n+1) \cdot \Pi_i |\mathrm{S}_i|]$. The number of clocks for $\mathcal{A}_\mathcal{I}$ is $\Sigma_i |\mathrm{C}_i|$, and for all clocks x, the value of c_x, the largest constant it gets compared with, remains the same.

The specification of the system is given as another timed regular language S over the alphabet $\mathcal{P}^+(A)$. The system is *correct* iff $L(\mathcal{A}_\mathcal{I}) \subset \mathcal{S}$. If S is given as a TBA, then in general, it is undecidable to test for correctness. However, if S is given as a DTMA $\mathcal{A}_\mathcal{S}$, then we can solve this as outlined in Section 3.5.3.

Putting together all the pieces, we conclude:

Theorem 3.63 Given timed regular processes $P_i = (A_i, L(\mathcal{A}_\rangle))$, $i = 1, \ldots n$, modeled by timed automata $\mathcal{A}_\rangle$, and a specification as a deterministic timed automaton $\mathcal{A}_\mathcal{S}$, the inclusion of the trace set of $[\|_i P_i]$ in $L(\mathcal{A}_\mathcal{S})$ can be checked in PSPACE. ■

The verification algorithm checks for a cycle with several desired properties in the region graph of the product of all the automata. The number of states in the product automaton is $O[|\mathcal{A}_\mathcal{S}| \cdot \Pi_i |\mathcal{A}_\rangle|]$. The number of clock regions for the product is exponential in the total number of clocks and linear in product of all the constants. Thus the number of vertices in this region graph is $O[|\mathcal{A}_\mathcal{S}| \cdot \Pi_i |\mathcal{A}_\rangle| \cdot 2^{|\delta(\mathcal{A}_\mathcal{S})| + \Sigma_i |\delta(\mathcal{A}_\rangle)|}]$.

There are mainly three sources of exponential blow-up:

1. The complexity is proportional to the number of states in the global timed automaton describing the implementation $[\|_i P_i]$. This is exponential in the number of components.

2. The complexity is proportional to the product of the constants c_x, the largest constant x is compared with, over all the clocks x involved.

3. The complexity is proportional to the number of permutations over the set of all clocks.

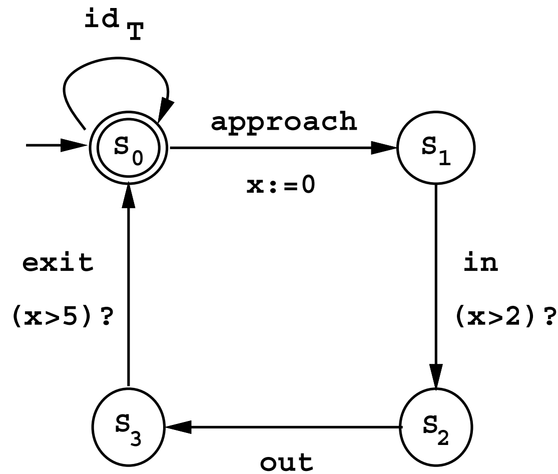


Figure 3.18: TRAIN

The first factor is present in the simplest of verification problems, even in the untimed case. Since the number of components is typically large, this exponential factor has been a major obstacle in implementing model-checking algorithms.

The second factor is typical of any formalism to reason about quantitative time. The blow-up by actual constants is observed even for simpler, discrete models. Note that if the bounds on the delays of different components are relatively prime then this factor leads to a major blow-up in the complexity.

Lastly, in constructing the regions, we need to account for all the possible orderings of the fractional parts of different clocks, and this is the source of the third factor. We remark that switching to a simpler, say discrete-time, model will avoid this blow-up in complexity. However since the total number of clocks is linear in the number of independent components, this blow-up is same as contributed by the first factor, namely, exponential in the number of components.

3.7.3 Verification example

We consider an example of a gate controller at a railroad crossing. The system is composed of three components: TRAIN, GATE and CONTROLLER.

The automaton modeling the train is shown in Figure 3.18. The event set is {*approach,*

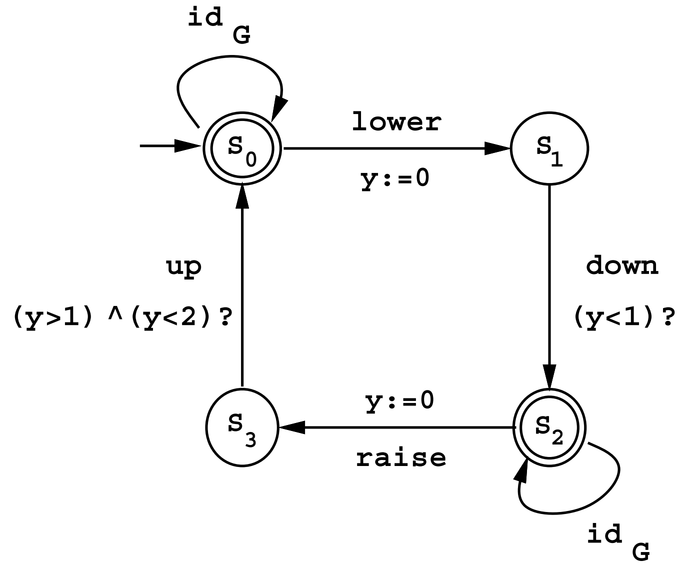


Figure 3.19: GATE

exit, in, out, id_T}. The train starts in state s_0. The event id_T represents its idling event; the train is not required to enter the gate. The train communicates with the controller with two events *approach* and *exit*. The events *in* and *out* mark the events of entry and exit of the train from the railroad crossing. The train is required to send the signal *approach* at least 2 minutes before it enters the crossing. Thus the minimum delay between *approach* and *in* is 2 minutes. Furthermore, we know that the maximum delay between the signals *approach* and *exit* is 5 minutes. This is a liveness requirement on the train. Both the timing requirements are expressed using a single clock x.

The automaton modeling the gate component is shown in Figure 3.19. The event set is {*raise, lower, up, down*, id_G}. The gate is open in state s_0 and closed in state s_2. It communicates with the controller through the signals *lower* and *raise*. The events *up* and *down* denote the opening and the closing of the gate. The gate responds to the signal *lower* by closing within 1 minute, and responds to the signal *raise* within 1 to 2 minutes. The gate can take its idling transition id_G in states s_0 or s_2 forever.

Finally, Figure 3.20 shows the automaton modeling the controller. The event set is {*approach, exit, raise, lower*, id_C}. The controller idle state is s_0. Whenever it receives the signal *approach* from the train, it responds by sending the signal *lower* to the gate. The response time is 1 minute. Whenever it receives the signal *exit*, it responds with a signal

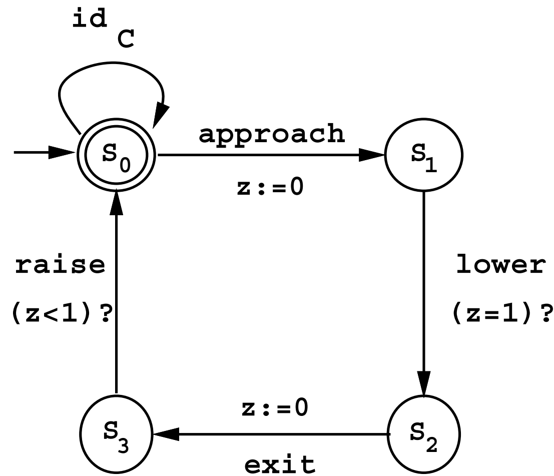


Figure 3.20: CONTROLLER

raise to the gate within 1 minute.

The entire system is then

$$[\,\text{TRAIN} \parallel \text{GATE} \parallel \text{CONTROLLER}\,].$$

The event set is the union of the event sets of all the three components. In this example, all the automata are particularly simple; they are deterministic, and do not have any fairness constraints (every run is an accepting run). The timed automaton $\mathcal{A}_{\mathcal{I}}$ specifying the entire system is obtained by composing the above three automata.

The correctness requirements for the system are the following:

1. *Safety:* Whenever the train is inside the gate, the gate should be closed.

2. *Real-time Liveness:* The gate is never closed at a stretch for more than 10 minutes.

The specification refers to only the events *in, out, up, down.* The safety property is specified by the automaton of Figure 3.21. An edge label *in* stands for any event set containing *in*, and an edge label "*in*, $\sim$ *out*" means any event set not containing *out*, but containing *in*. The automaton disallows *in* before *down*, and *up* before *out*. All the states are accepting states.

The real-time liveness property is specified by the timed automaton of Figure 3.22. The automaton requires that every *down* be followed by *up* within 10 minutes.

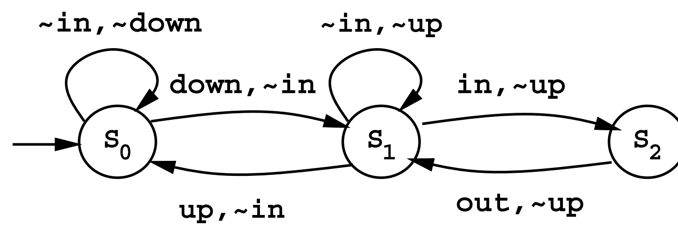


Figure 3.21: Safety property

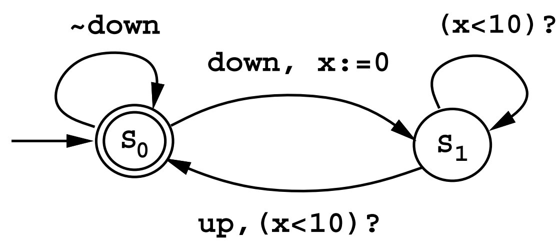


Figure 3.22: Real-time liveness property

Note that the automaton is deterministic, and hence can be complemented. Furthermore, observe that the acceptance condition is not necessary; we can include state s_1 also in the acceptance set. This is because the progress of time ensures that the self-loop on state s_1 with the clock constraint $(x < 10)$ cannot be taken indefinitely, and the automaton will eventually visit state s_0.

The correctness of $\mathcal{A}_{\mathcal{I}}$ against the two specifications can be checked separately as outlined in Section 3.7. Observe that though the safety property is purely a qualitative property, it does not hold if we discard the timing requirements.

3.7.4 Implementation

We have implemented the verification algorithm in LISP. We briefly discuss some aspects of the implementation.

The input to the program consists of a list of timed Büchi automata $\mathcal{A}_{/}$ through $\mathcal{A}_{\backslash}$. The automaton $\mathcal{A}_{/}$ is the specification automaton, and is required to be deterministic. The implementation automaton is the product of the automata $\mathcal{A}_{\infty}, \ldots \mathcal{A}_{\backslash}$. The output of the program is "yes" or "no" depending on whether or not the language of the implementation automaton is contained in that of the specification automaton.

Recall that the verification procedure involves three steps: (1) constructing the product automaton $\mathcal{A}$ from the automata $\mathcal{A}_{/}, \ldots \mathcal{A}_{\backslash}$, (2) constructing the region automaton $R(\mathcal{A})$, and (3) searching for a cycle meeting all the desired acceptance requirements. Step (3) can be implemented by finding all the strongly connected components of $R(\mathcal{A})$, and testing each component for the desired properties. Because of the constraints on the available memory space, the above three steps cannot be carried out separately. In the implemented procedure, neither the product automaton nor the region automaton is constructed explicitly, and all the three steps are performed simultaneously.

Each vertex is represented by a list

$$(s_0, \ldots s_n, (x_1, x_2, \ldots), ((y_1, n_1), (y_2, n_2), \ldots), ((z_1^1, m_1^1), (z_2^1, m_2^1), \ldots), ((z_1^2, m_1^2), \ldots), \ldots)$$

Each s_i gives the state of the automaton $\mathcal{A}_{)}$, and the remaining components represent the clock region. In the region corresponding to the above vertex the value of each clock y_i equals n_i. The value of each clock x_i is greater than its maximum value of interest, that is, c_{x_i}. The value of each clock z_j^i is between m_j^i and $m_j^i + 1$ such that (i) the fractional parts of z_j^i and z_k^i are the same, and (ii) the fractional part of z_j^i is less than that of z_k^{i+1}.

Given such a representation, all its successors in the region automaton are obtained in a straightforward way by examining the transitions starting at each of the states s_i's.

The procedure performs a depth first search starting from the initial vertex of the region automaton. Whenever a new vertex is visited, it is assigned its own depth first number (DFN). The search for strongly connected components uses Tarjan's algorithm [Tar72]. The global data structures include

1. a hash-table mapping vertex representations to their respective DFNs,
2. an array mapping DFNs to the representations of the corresponding vertices,
3. an array storing the DFN of the parent for each vertex,
4. an array which gives for every vertex the least vertex that can be reached using forward edges and at most one back edge,
5. a stack storing the strongly connected component currently being explored, and
6. an array giving the number of already explored edges from each vertex.

Observe that we do not store the edges of the region automaton. Consequently, the program needs to generate all the edges starting from a vertex, not only the first time the vertex is visited, but every time the depth first search backtracks to it.

Whenever a strongly connected component is found, the procedure checks if it contains an accepting state of each of the automata $\mathcal{A}_{\infty}, \ldots \mathcal{A}_{\backslash}$ and contains no accepting state of $\mathcal{A}_{/}$ and meets the progress requirement. If so, the program halts saying that the implementation does not meet the specification. Otherwise when the depth first search terminates, the program halts saying that the implementation meets the specification.

The example of railroad crossing of Section 3.7.3 was given as an input to our program. The region automaton has about 8000 vertices and about 50000 edges. The program terminated giving the correct answer "yes" (the running time was approximately 2 minutes on DEC station 3100). Thus the program can handle small examples. The state-explosion problem prevents its application to bigger systems. In future, we intend to try out some heuristics to improve its efficiency.

Chapter 4

Linear Temporal Logic

In this chapter we study real-time extensions of linear temporal logics as specification formalisms for real-time systems. We introduce the specification language MITL, and study its complexity and expressiveness. We define *interval automata*, a variant of timed automata, to model finite-state systems, and develop an algorithm for model checking.

4.1 Propositional temporal logic: PTL

In this section we will briefly review the definition and the complexity results about PTL.

PTL is a modal logic interpreted over infinite state sequences. The modal operator $\Box$ is interpreted as "henceforth", and its dual $\Diamond$ is interpreted as "eventually". We consider the logic with the *next* operator $\bigcirc$ and the *until* operator $\mathcal{U}$.

We assume that we are given a set of atomic propositions AP. The formulas ϕ of PTL are built from the atomic propositions using logical connectives and temporal operators as defined below:

$$\phi := p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \bigcirc \phi \mid \phi_1 \mathcal{U} \phi_2.$$

Intuitively, a formula $\bigcirc\phi$ holds at some state iff ϕ holds in the next state. $\phi_1 \mathcal{U} \phi_2$ holds at some state iff ϕ_2 holds at some later state and ϕ_1 holds at all intermediate states. We will use $\Diamond\phi$ as an abbreviation for $\mathbf{true}\, \mathcal{U}\, \phi$, and $\Box\phi$ for $\neg\Diamond\neg\phi$.

The semantics of PTL is defined with respect to state sequences. A *state sequence,* denoted by σ, is an infinite sequence $\sigma_0, \sigma_1, \ldots$ of states, where each state σ_i is a subset of AP. Thus each state assigns truth values to all the atomic propositions: p is true in state σ_i iff $p \in \sigma_i$.

Now we formally define the semantics of PTL. For a state sequence $\sigma = \sigma_0, \sigma_1, \ldots$ let σ^j denote its suffix $\sigma_j, \sigma_{j+1}, \ldots$ The satisfaction relation $\sigma \models \phi$ (that is, σ is a *model* of ϕ) is defined inductively below:

$\sigma \models p$ iff $p \in \sigma_0$

$\sigma \models \neg\phi$ iff $\sigma \not\models \phi$

$\sigma \models \phi_1 \wedge \phi_2$ iff $\sigma \models \phi_1$ and $\sigma \models \phi_2$

$\sigma \models \bigcirc\phi$ iff $\sigma^1 \models \phi$

$\sigma \models \phi_1 \mathcal{U} \phi_2$ iff $\sigma^j \models \phi_2$ for some $j \geq 0$, and $\sigma^k \models \phi_1$ for all $0 \leq k < j$.

The formula ϕ is *satisfiable* iff some state sequence is a model of ϕ.

The set of models of a formula ϕ can be viewed as a *property* of state sequences specified by ϕ; let $L(\phi)$ denote the set of state sequences σ such that $\sigma \models \phi$. PTL-formulas can specify several interesting properties such as termination, fairness, mutual exclusion, guaranteed response, precedence [OL82, Pnu86, MP89]. A typical response property that "every p-state is followed by some q-state," is specified by the following PTL-formula:

$$\Box [p \rightarrow \Diamond q].$$

There is a tableau-based decision procedure for testing satisfiability of PTL-formulas [BMP81]; the problem is known to be PSPACE-complete [SC85].

In one approach to verification using PTL, finite-state systems are modeled as finite Kripke structures. A Kripke structure is given as $\mathcal{M} = (\mathrm{S}, \mathrm{S}_0, \mu, \mathrm{E})$, where S is a finite set of states, $\mathrm{S}_0 \subseteq \mathrm{S}$ is a set of initial states, $\mu : \mathrm{S} \rightarrow 2^{\mathrm{AP}}$ gives an assignment of truth values to propositions in each state, and E is a binary relation over S giving the possible transitions. The infinite paths through $\mathcal{M}$ give the set $L(\mathcal{M})$ of state sequences generated by $\mathcal{M}$. This set consists of state sequences $(\mu(s_0), \mu(s_1) \ldots)$ such that $s_0 \in \mathrm{S}_0$, and $\langle s_i, s_{i+1} \rangle \in \mathrm{E}$ for all $i \geq 0$. A system modeled as a Kripke structure $\mathcal{M}$ satisfies the PTL-specification ϕ iff every state sequence generated by the system satisfies the formula ϕ, that is, $\mathcal{M} \models \phi$ iff $L(\mathcal{M}) \subseteq L(\phi)$ [1].

The model-checking problem is to decide whether a Kripke structure $\mathcal{M}$ satisfies a PTL-formula ϕ. The model-checking algorithm first constructs a tableau $\mathcal{M}_{\neg\phi}$ which generates precisely the models of $\neg\phi$, and then uses the product construction to check if the intersection of $L(\mathcal{M})$ and $L(\mathcal{M}_{\neg\phi})$ is empty [LP85, VW86]. The complexity of the algorithm

[1] Be warned that the notation $\mathcal{M} \models \phi$ is somewhat nonstandard in logic; $\mathcal{M}$ is not a single model for the PTL-formula ϕ, but represents a set of models for ϕ.

is exponential in the size of the PTL-specification, and linear in the size of the Kripke structure. The problem is PSPACE-complete.

Notice the similarity between the verification of (untimed) processes using ω-automata of Chapter 3 and using temporal logic. A process is modeled as a set of traces in Chapter 3, whereas now a process is modeled as a set of state sequences. The basic difference is only in the notion of what is *observable*. In trace semantics, only the events are observable, and thus we focus only on the transition points without any reference to the state of the system between the transition points. In semantics of state sequences, the emphasis is on the *state* of the system, and the observable aspect of the state is modeled using a set of propositions. Thus in one case a system is an ω-language over the alphabet $\mathcal{P}^+(A)$, where A is the set of events, and in the other case it is an ω-language over the alphabet 2^{AP}. The two approaches are very similar; the only reason we are using both of them is that it is conventional and technically convenient to develop automata theory using an event-based model, and temporal logics using a state-based model.

In Chapter 3 we used ω-automata to model finite-state systems, now we use Kripke structures. Kripke structures are similar to transition tables; in transition tables edges are labeled with alphabet symbols, whereas in Kripke structures states are labeled with propositions. The language $L(\mathcal{M})$, for a Kripke structure $\mathcal{M}$, is an ω-regular language. As in case of transition tables, we can couple Kripke structures with acceptance conditions.

There is a close connection between Büchi automata and PTL. For every PTL-formula ϕ, the language $L(\phi)$ is an ω-regular language over the alphabet 2^{AP}, and hence, can be recognized by a Büchi automaton. On the other hand, PTL does not have the full expressive power of ω-regular languages; it needs to be extended with right-linear grammar operators or automata connectives to attain the full expressive power of Büchi automata [Wol83].

Along the same lines, for modeling and verifying timed processes we will shift from the semantics of timed traces to that of *timed state sequences*. Accordingly we will modify the definition of timed automata so that they generate timed state sequences.

4.2 Metric interval temporal logic

To reason about quantitative time requirements we need to add time explicitly in the syntax and semantics of PTL. To define the syntax and semantics we use intervals of the real line.

4.2.1 Intervals

An interval is a convex subset of nonnegative real numbers R. Intervals may be open, half-open, or closed; bounded or unbounded. Each interval is of one of the following forms: $[a, b]$, $[a, b)$, $[a, \infty)$, $(a, b]$, (a, b), (a, ∞), where $a \leq b$, $a, b \in \mathsf{R}$. For an interval of the above form, a is its left end-point, and b is its right end-point. The left end-point of I is denoted by $l(I)$ and the right end-point, for bounded I, is denoted by $r(I)$.

An interval I is *singular* iff it is of the form $[a, a]$; that is, I is closed and $l(I) = r(I)$.

Two intervals I and I' are called *adjacent* iff

1. the right end-point of I is the same as the left end-point of I', and

2. either I is right open and I' is left closed, or I is right closed and I' is left open.

For instance, the intervals $(1, 2]$ and $(2, 2.5)$ are adjacent.

We will freely use intuitive pseudo-arithmetic expressions to denote intervals. For example, the expressions such as $\leq b$ and $> a$ denote the intervals $[0, b]$ and (a, ∞), respectively, and the expression $< I$ denotes the interval $\{t \mid \forall t' \in I. (t < t')\}$. The expression $I + t$, for $t \in \mathsf{R}$, denotes the interval $\{t' + t \mid t' \in I\}$. Similarly, $I - t$ and $t{\cdot}I$ stand for the intervals $\{t' - t \mid t' \in I \text{ and } t' \geq t\}$ and $\{t{\cdot}t' \mid t' \in I\}$, respectively.

4.2.2 Timed state sequences

Let AP be a set of atomic propositions. We assume that, at any point in time, the global state of a (finite-state) system can be modeled by an interpretation (or truth-value assignment) for AP. We therefore identify states s with subsets of AP; that is, $s \models p$ iff $p \in s$ (for $p \in$ AP).

We add time to state sequences by associating an interval with each state; this gives us *timed state sequences*. If the interval associated with state s is I, then the state at each time $t \in I$ is s. We require that the intervals associated with the consecutive states in a state sequence be adjacent.

Definition 4.1 A *state sequence* $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ is an infinite sequence of states, each state σ_i is a subset of AP.

An *interval sequence* $\overline{I} = I_0I_1I_2\ldots$ is an infinite sequence of intervals that partitions R such that

(i) [*adjacency*] the intervals I_i and I_{i+1} are adjacent for all $i \geq 0$, and

(ii) [*progress*] every time value $t \in \mathsf{R}$ belongs to some interval I_i.

A ***timed state sequence*** $\rho = (\sigma, \overline{I})$ is a pair consisting of a state sequence σ and an interval sequence $\overline{I}$. ■

Sometimes we will denote a timed state sequence $\rho = (\sigma, \overline{I})$ by the infinite sequence of pairs of states and intervals:

$$(\sigma_0, I_0) \to (\sigma_1, I_1) \to (\sigma_2, I_2) \to \cdots$$

A timed state sequence $\rho = (\sigma, \overline{I})$ can be viewed as a map ρ^* from the time domain R to the states 2^{AP} (let $\rho^*(t) = \sigma_i$ if $t \in I_i$). Thus a timed state sequence provides complete information about the global state of a system at each time instant. Timed state sequences obey the ***finite-variability*** condition: between any two points in time there are only finitely many state changes. This assumption is adequate for modeling *discrete* systems.

Given a timed state sequence $(\sigma, \overline{I})$, the i-th transition point, denoted by t_i, is defined to be the left end-point of the interval I_i; that is, $t_i = l(I_i)$. Note that the state at time t_i is σ_{i-1} if I_i is left-open, and is σ_i if I_i is left-closed.

Our definition allows *transient* states, which occur only at isolated points in time. If I_i is a singular interval $[t_i, t_i]$, then the state at time t_i is σ_i, but the state just before t_i is σ_{i-1}, and the state just after t_i is σ_{i+1}. Observe that in such a case neither σ_{i-1} nor σ_{i+1} can be transient, because the interval I_{i-1} must be right-open and the interval I_{i+1} must be left-open. The transient states are useful for modeling the truth of propositions that are true only at the transition points.

We define the ***suffix*** of a timed state sequence at every $t \in \mathsf{R}$ below:

Definition 4.2 For a timed state sequence $\rho = (\sigma, \overline{I})$, and time $t \in I_j$, the suffix ρ^t is defined to be the timed state sequence

$$(\sigma_j, I_j - t) \to (\sigma_{j+1}, I_{j+1} - t) \to (\sigma_{j+2}, I_{j+2} - t) \to \cdots$$

■

The suffix operator has been defined such that $(\rho^t)^*(t') = \rho^*(t + t')$ for all $t' \in \mathsf{R}$. In particular $\rho^0 = \rho$.

Another useful operation is the refinement operation.

Definition 4.3 A *refinement* of a timed state sequence $\rho = (\sigma, \overline{I})$ is a timed state sequence obtained by replacing each pair (σ_i, I_i) by a finite sequence

$$(\sigma_i, I_i^1) \rightarrow (\sigma_i, I_i^2) \rightarrow \cdots (\sigma_i, I_i^{n_i})$$

such that $\cup_{1 \leq j \leq n_i} I_i^j = I_i$, and I_i^j and I_i^{j+1} are adjacent for $1 \leq j < n_i$. ■

Observe that if ρ' is a refinement of ρ then the associated maps ρ^* and ρ'^* are identical.

4.2.3 Syntax and semantics of MITL

We introduce an extension of linear temporal logic, *metric interval temporal logic* (or MITL), that is interpreted over *timed* state sequences. A fairly standard way of introducing real-time in the syntax is to replace the unrestricted temporal operators of PTL by their time bounded versions. Thus we introduce operators such as $\Diamond_{(2,4)}$ meaning "eventually within 2 to 4 time units" [EMSS89, AH90, Koy90].

As before let AP be a set of atomic propositions. The formulas of MITL are built from propositions by boolean connectives and time-bounded versions of the *until* operator $\mathcal{U}$. The until operator may be subscripted with any nonsingular interval with integer end-points.

Definition 4.4 The formulas of MITL are inductively defined as follows:

$$\phi := p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \, \mathcal{U}_I \, \phi_2$$

where $p \in$ AP, and I is a *nonsingular* interval with integer end-points. ■

Notice that we require the end-points of intervals subscripting MITL-operators to be integers. We will later show that all our results easily extend to the case when rational end-points are allowed; the restriction to integer end-points is mainly to simplify the presentation. The formulas of MITL are interpreted over timed state sequences, which provide an interpretation for the atomic propositions at each time instant. Informally $\phi_1 \, \mathcal{U}_I \, \phi_2$ holds at time t in a timed state sequence iff there is a later time instant $t' \in I + t$ such that ϕ_2 holds at time t' and ϕ_1 holds throughout (t, t'). The formal definition follows.

Definition 4.5 For an MITL-formula ϕ, and a timed state sequence $\rho = (\sigma, \overline{I})$, the satisfaction relation $\rho \models \phi$ is defined inductively as follows:

$\rho \models p$ iff $p \in \sigma_0$.
$\rho \models \neg\phi$ iff $\rho \not\models \phi$.
$\rho \models (\phi_1 \wedge \phi_2)$ iff $\rho \models \phi_1$ and $\rho \models \phi_2$.
$\rho \models \phi_1 \mathcal{U}_I \phi_2$ iff for some $t \in I$, $\rho^t \models \phi_2$, and $\rho^{t'} \models \phi_1$ for all $t' \in (0, t)$.

For an MITL-formula ϕ, the set $L(\phi)$ consists of timed state sequences ρ such that $\rho \models \phi$. An MITL-formula ϕ is called *satisfiable* iff $L(\phi)$ is nonempty. ■

Note that MITL has no *next-time* operator $\bigcirc$, because with the dense time domain there is no unique next time. Also observe that the *until* operator is *strict* in its first argument; for $\phi_1 \mathcal{U}_I \phi_2$ to hold in the current state ϕ_1 need not hold now.

4.2.4 Defined operators

Now let us introduce some standard abbreviations for additional temporal operators. The defined operators $\Diamond_I \phi$ (constrained *eventually*) and $\Box_I \phi$ (constrained *always*) stand for $\mathbf{true}\,\mathcal{U}_I \phi$ and $\neg\Diamond_I\neg\phi$, respectively. It follows that the formula $\Box_I \phi$ (or $\Diamond_I \phi$) holds at time $t \in \mathsf{R}$ of a timed state sequence iff ϕ holds at all times (at some time, respectively) within the interval $t + I$.

We usually suppress the interval $(0, \infty)$ as a subscript. Thus the MITL-operators $\Diamond$, $\Box$, and $\mathcal{U}$ coincide with the conventional unconstrained *strict eventually*, *strict always*, and *strict until* operators of temporal logic. This is because the *until* operator of MITL is implicitly strict in its first argument. The corresponding nonstrict operators are definable in MITL. For instance,

$$\phi_2 \vee (\phi_1 \wedge \phi_1 \mathcal{U} \phi_2)$$

corresponds to the conventional unconstrained non-strict *until* operator $\phi_1 \mathcal{U} \phi_2$. Note that, on the other hand, the MITL-operator $\mathcal{U}_I$ cannot be defined in terms of an *until* operator that is not strict in its first argument; this is why we have chosen the *strict* versions of temporal operators to be primitive.

Example 4.6 In MITL, we can specify that a particular proposition always holds only instantaneously. The following formula states that the proposition p is true in infinitely many transient states and nowhere else;

$$p \wedge \Box_{\geq 0}(p \rightarrow (\neg p)\mathcal{U} p).$$

Such formulas can be used to distinguish events from state constraints. Such a property cannot be expressed if the until-operator is defined to be nonstrict in its left argument. ■

We also define a constrained *unless* operator as the dual of the *until* operator:

$$\phi_1 \,{}_I\mathrm{U}\, \phi_2 \text{ stands for } \neg((\neg\phi_2)\,\mathcal{U}_I\,(\neg\phi_1)).$$

It follows that the formula $\phi_1 \,{}_I\mathrm{U}\, \phi_2$ holds in a timed state sequence iff either ϕ_1 is true throughout the interval I, or there is a time instant $t > 0$ such that ϕ_2 is true at time t and ϕ_1 holds at all instants $t' \leq t$ within the interval I.

Let us consider a few examples of MITL properties now. Observe that all the qualitative temporal properties specifiable in PTL are already definable in MITL.

Example 4.7 A typical bounded response property that "every p-state is followed by some q-state within time 3," is expressed by the MITL-formula

$$\Box[\, p \rightarrow \Diamond_{[0,3)}\, q\,].$$

■

Example 4.8 We consider a *time-out* property now. Suppose p is some state constraint, and q is the time-out event. The property we want to specify is that whenever p ceases to hold, either within next 5 time units it becomes true again, or at time 5 the time-out event q happens. This property is specified by the formula:

$$\Box_{\geq 0}[\,(p \wedge \Box_{(0,5)}\neg p) \rightarrow (\Box_{(0,5)}\neg q \wedge \Diamond_{(0,5]} q\,)].$$

■

More examples of specifying interesting systems using real-time temporal logics can be found in [Koy90] and [Ost90b].

4.2.5 Refining the models

In this section, we present some results that will be useful in developing the decision procedure for MITL.

Observe that the logic MITL is insensitive to *stuttering*. For a timed state sequence $\rho = (\sigma, \overline{I})$, and an MITL-formula ϕ, the satisfaction relation $\rho \models \phi$ depends on the map ρ^*, and not on the particular choice of the interval sequence $\overline{I}$. This follows trivially from the semantic definition. In particular, the next lemma holds:

Lemma 4.9 If a timed state sequence ρ' is a refinement of a timed state sequence ρ then for every MITL-formula ϕ, $\rho \models \phi$ iff $\rho' \models \phi$. ■

Hence, if for a timed state sequence $\rho = (\sigma, \overline{I})$, the state does not change after the j-th transition point, that is, $\sigma_i = \sigma_j$ for all $i \geq j$, then we represent ρ by the finite sequence

$$(\sigma_0, I_0) \rightarrow (\sigma_1, I_1) \rightarrow \cdots \rightarrow (\sigma_{j-1}, I_{j-1}) \rightarrow (\sigma_j, \cup_{i \geq j} I_i).$$

Also the satisfaction relation has another nice property; the truth value of a given MITL-formula does not change more than ω times over ρ. Thus the models satisfy the finite-variability property, not only with respect to the truth of atomic propositions, but also with respect to more complex MITL properties.

Definition 4.10 For an MITL-formula ϕ, a timed state sequence $\rho = (\sigma, \overline{I})$ is called *ϕ-fine* iff for all $i \geq 0$, for all subformulas ψ of ϕ, for all $t, t' \in I_i$, $\rho^t \models \psi$ iff $\rho^{t'} \models \psi$. ■

Thus in a ϕ-fine timed state sequence $(\sigma, \overline{I})$ the truth of all the subformulas of ϕ stays invariant over every interval of $\overline{I}$. The following lemma states that every timed state sequence can be refined into a ϕ-fine timed state sequence. It follows that a formula ϕ is satisfiable iff it has some ϕ-fine model.

Lemma 4.11 Let ϕ be an MITL-formula and ρ be a timed state sequence. There exists a ϕ-fine timed state sequence ρ_ϕ which refines ρ.

PROOF. The proof is by induction on the structure of the formula. For an atomic proposition p, take ρ_p to be the same as ρ. For a negated formula $\neg\phi$, take $\rho_{\neg\phi}$ to be the same as ρ_ϕ.

In case of the conjunction $\phi_1 \wedge \phi_2$, the sequence $\rho_{\phi_1 \wedge \phi_2}$ is constructed by refining the sequence ρ_{ϕ_1}. Let $\overline{I}_{\phi_1} = J_0 J_1 \ldots$ We split each interval J_i into a finite sequence $J_i^1, \ldots J_i^{n_i}$ so that each J_i^j is fully contained in some interval of $\overline{I}_{\phi_2}$.

Now let us consider the case $\phi = \phi_1 \mathcal{U}_I \phi_2$. First we construct the refinement $\rho_{\phi_1 \wedge \phi_2}$. Let $\overline{I}_{\phi_1 \wedge \phi_2} = J_0 J_1 \ldots$ We construct a refinement $\overline{I}_\phi = J_0' J_1' \ldots$ of $\overline{I}_{\phi_1 \wedge \phi_2}$ such that whenever t and t' are in the same interval J_i', then both $t + l(I)$ and $t' + l(I)$ belong to the same interval J_k for some $k \geq i$, and, if I is bounded, both $t + r(I)$ and $t' + r(I)$ belong to the same interval J_l for some $l \geq k$. It is clear that such a sequence can be constructed by a finite splitting of each interval J_i. Furthermore, it is easy to check that $\rho^t \models \phi$ iff $\rho^{t'} \models \phi$ whenever t and t' are in the same interval J_i'. ■

4.2.6 Real versus rational time

While timed state sequences are defined by choosing the set of (nonnegative) reals to model time, for interpreting formulas of MITL, the crucial property of the time domain R is not its continuity, but only its denseness. Choosing some other dense linear order to define semantics leaves the complexity and expressiveness results unchanged. In particular, we show that replacing the time domain R with the set of nonnegative rational numbers, Q, when defining the semantics of MITL does not change the satisfiability (and validity) of any MITL-formula.

Definition 4.12 A timed state sequence $(\sigma, \overline{I})$ is called *rational* iff the end-points of all intervals in $\overline{I}$ are rational.

Let ϕ be an MITL-formula, and let ρ be a rational timed state sequence. ρ Q-satisfies ϕ iff $\rho \models \phi$, where the satisfaction relation $\models$ (Definition 4.5) is redefined so that all time quantifiers range over Q only.

A formula ϕ is called Q-satisfiable iff ρ Q-satisfies ϕ for some rational timed state sequence ρ. ■

We show that this new notion of satisfiability is the same as the old one. In other words, MITL-formulas cannot distinguish the time domain R from the time domain Q. This equivalence of real and rational models follows from the following two lemmas.

Lemma 4.13 Let ϕ be an MITL-formula and ρ a rational ϕ-fine timed state sequence. Then ρ Q-satisfies ϕ iff ρ satisfies ϕ.

PROOF. We use induction on the structure of ϕ. Let us consider only the interesting case, that ϕ has the form $\phi_1 \, \mathcal{U}_I \, \phi_2$.

Suppose $\rho = (\sigma, \overline{I})$ is rational, and ϕ-fine timed state sequence. Suppose ρ Q-satisfies ϕ; that is, ρ^t Q-satisfies ϕ_2 for some rational $t \in I$, and $\rho^{t'}$ Q-satisfies ϕ_1 for all rationals $0 < t' < t$. By the induction hypothesis, we may conclude that $\rho^t \models \phi_2$ and $\rho^{t'} \models \phi_1$ for all rationals $0 < t' < t$. Hence, to show that $\rho \models \phi_1 \, \mathcal{U}_I \, \phi_2$, it suffices to show that $\rho^{t''} \models \phi_1$ for all reals $0 < t'' < t$. Consider an arbitrary real $0 < t'' < t$, and assume that $t'' \in I_i$. If I_i is singular then, since ρ is rational, t'' must be rational. Otherwise, I_i is nonsingular, and there is also a rational $t' \in I_i$ with $0 < t' < t$. We know that $\rho^{t'} \models \phi_1$ and, since ρ is ϕ-fine, it follows that $\rho^{t''} \models \phi_1$.

The second direction, that every rational ϕ-fine model of ϕ Q-satisfies ϕ, follows by a similar argument. ■

Lemma 4.14 Let $\rho = (\sigma, \overline{I})$ and $\rho' = (\sigma, \overline{I}')$ be two timed state sequences such that for all $t \in \mathsf{R}$, if $t = t_i + m$ for some left end-point t_i of an interval I_i in $\overline{I}$ and some $m \in \mathsf{N}$, then $t \in I_j$ iff $t \in I'_j$. For all MITL-formulas $\rho \models \phi$ iff $\rho' \models \phi$.

Proof. The proof is by induction on the structure of ϕ. The only interesting case is $\phi = \phi_1 \mathcal{U}_I \phi_2$. Suppose $\rho \models \phi_1 \mathcal{U}_I \phi_2$. Let $t \in I_i$ be such that $\rho^t \models \phi_2$ and $\rho^{t''} \models \phi_1$ for all $0 < t'' < t$. We can find $t' \in I'_i$ such that for all $t'' \in \mathsf{R}$, if $t'' = t + m$ for some $m \in \mathsf{N}$, then $t'' \in I_j$ iff $t'' \in I'_j$. The integer part of t' should be the same as that of t, and for all i, $\mathit{fract}(t) \leq \mathit{fract}(t_i)$ iff $\mathit{fract}(t') \leq \mathit{fract}(t'_i)$. Now by applying the induction hypothesis to ρ^t and $\rho'^{t'}$, we get that $\rho'^{t'} \models \phi_2$. By a similar argument, we can show that for all $0 < t'' < t'$, $\rho'^{t''} \models \phi_1$, and hence, $\rho' \models \phi_1 \mathcal{U}_I \phi_2$. ■

Lemma 4.14 classifies timed state sequences into equivalence classes such that the members of a class cannot be distinguished by formulas of MITL. It implies, in particular, the following theorem:

Theorem 4.15 A formula ϕ of MITL is Q-satisfiable iff it is satisfiable.

Proof. Suppose that ϕ is Q-satisfiable in a rational model ρ. By Lemmas 4.9 and 4.11, there is a rational ϕ-fine refinement of ρ that Q-satisfies ϕ. By Lemma 4.13, this refinement is a (real) model of ϕ.

The proof of the second direction uses Lemma 4.14. Consider a (real) model $\rho = (\sigma, \overline{I})$ of ϕ. We construct another timed state sequence $\rho' = (\sigma, \overline{I}')$ as follows. We adjust the interval boundaries in $\overline{I}$ so that no interval is adjusted across integers, and the ordering of the fractional parts of all interval boundaries is not altered. The denseness of Q allows us to adjust all boundaries to be rational numbers. In particular, for all $i \geq 0$, the interval I'_i is left-open iff I_i is left-open, and I'_i is right-open iff I_i is right-open. Let $\{t'_i \in [0,1) \mid i \geq 0\}$ be a sequence of rational numbers such that for every pair i, j, $t'_i \leq t'_j$ iff $\mathit{fract}(t_i) \leq \mathit{fract}(t_j)$. Set the left end-point of I'_i to be the integral part of t_i plus t'_i. The timed state sequences ρ and ρ' satisfy the requirements of Lemma 4.14, and hence $\rho' \models \phi$. By Lemma 4.9 and Lemma 4.13 the ϕ-refinement of ρ' Q-satisfies ϕ. ■

4.2.7 Allowing rational constants

While defining the syntax of MITL we required the boundaries of the intervals subscripting the until operator to be integers. We can relax this condition, and allow intervals with rational end-points. For example, we can allow formulas such as $\Box_{(0.5,0.6)} p$, which says that p holds over the interval $(0.5, 0.6)$.

All our results can be easily adopted to accommodate this extension. This is because of the following obvious property of the models of MITL.

Lemma 4.16 Let ϕ be a formula of MITL possibly with arbitrary rational constants, and let $\rho = (\sigma, \overline{I})$ be a timed state sequence. For $c \in \mathbb{Q}$, let ϕ_c denote the formula ϕ obtained by replacing every interval subscript I by $c{\cdot}I$, similarly, let ρ_c denote the timed state sequence obtained from ρ by replacing every interval I_i in $\overline{I}$ by $c{\cdot}I_i$. $\rho \models \phi$ iff $\rho_c \models \phi_c$. ■

The lemma can be proved by a straightforward induction on the structure of ϕ. Consequently, there is an isomorphism between the models of ϕ and the models of ϕ_c.

In particular, to test the satisfiability of ϕ with rational constants, we test the formula ϕ_c for satisfiability, where c is the least common multiple of all the constants appearing in ϕ. Note that ϕ_c has only integer constants, and the size of ϕ_c is bounded by $|\phi|^2$.

4.2.8 Avoiding undecidability

In MITL, we cannot write formulas such as

$$\Box(p \rightarrow \Diamond_{=5}\, q);$$

since intervals such as $[5, 5]$ are not allowed as subscripts. We will show that there is, in fact, no MITL-formula that expresses this condition, and that the restriction of MITL to *nonsingular* intervals is essential for decidability. Note that some forms of equality are expressible in MITL; we define $(\neg\phi)\,\mathcal{U}_{=t}\,\phi$ as an abbreviation for

$$(\Box_{(0,t)} \neg\phi) \wedge (\Diamond_{(0,t]}\phi).$$

Thus the stronger condition that "for every p-state the next following q-state is after exactly 5 time units,"

$$\Box(p \rightarrow (\neg q)\,\mathcal{U}_{=5}\, q),$$

is expressible in MITL.

We show that allowing singular intervals as subscripts for the temporal operators makes MITL undecidable. The denseness of the underlying time domain allows us to encode Turing machine computations. Notice that the undecidability result depends on the choice of dense-time semantics; the problem is decidable for the discrete models [AH90]. The proof of the next theorem is similar to the proof of Theorem 3.41.

Theorem 4.17 If singular intervals are allowed as subscripts for temporal operators in the syntax of MITL then the satisfiability problem is Σ_1^1-complete.

PROOF. [Σ_1^1-hardness]: Recall that the problem of deciding whether a nondeterministic 2-counter machine has a recurring computation is Σ_1^1-hard (Lemma 3.40). We construct a formula ϕ such that ϕ is satisfiable iff the given machine has a recurring computation.

Let A be a 2-counter machine with counters C and D, and n program instructions. As before, a configuration of the machine is represented by a triple $\langle i, c, d \rangle$, where i gives the instruction to be executed next and c, d give the contents of the counters. A computation of the machine is an infinite sequence of triples starting at $\langle 1, 0, 0 \rangle$.

We encode the computations of A in the logic using the propositions p_C, p_D, and $p_1, \ldots p_n$. First we require that all propositions are true only in singular intervals of a timed state sequence. The following MITL-formula specifies this constraint for the proposition p_C.

$$\Box_{\geq 0}(p_C \rightarrow (\neg p_C)\,\mathcal{U}\, p_C).$$

Let us say that a timed state sequence ρ encodes a configuration $\langle i, c, d \rangle$ over the interval $[a, b)$, $a < b$, iff following hold:

1. The proposition p_C holds at exactly c time instants in the interval $[a, b)$ along ρ.
2. The proposition p_D holds at exactly d time instants in the interval $[a, b)$ along ρ.
3. Each proposition p_j, $j \neq i$, is false everywhere in the interval $[a, b)$ along ρ.
4. The proposition p_i is true over $[a, a]$ and false during the interval (a, b) along ρ.

We construct a formula ϕ such that $\rho \models \phi$ iff there exists a computation $\{\langle i_j, c_j, d_j \rangle : j \geq 0\}$ of A such that ρ encodes the j-th configuration over the interval $[j, j+1)$ for all $j \geq 0$.

The nature of the initial configuration is expressed by the formula;

$$p_1 \wedge \Box_{(0,1)} \neg p_1 \wedge \Box_{[0,1)}[\neg p_C \wedge \neg p_D \wedge \bigwedge_{2 \leq i \leq n} \neg p_i].$$

To relate one configuration to the next, we establish a one-to-one correspondence between the states separated by distance 1. The formula ϕ has one conjunct for each program instruction. For instance, if the second instruction is to increment D (and goto instruction 3), ϕ has the following conjunct:

$$\Box_{\geq 0}\left[p_2 \rightarrow \left(\begin{array}{l} \Diamond_{=1} p_3 \wedge \Box_{(1,2)} \neg p_3 \wedge \Box_{[1,2)}(\wedge_{1 \leq i \leq n, i \neq 3} \neg p_i) \wedge \\ \Box_{[0,1)} \, copy(p_C) \wedge \\ copy(p_D)\, \mathcal{U}_{<1}(\neg p_D \wedge \Diamond_{=1} p_D \wedge copy(p_D)\, \mathcal{U}\, p_3) \end{array}\right)\right]$$

We have used the abbreviation $copy(p)$ for $(p \leftrightarrow \Diamond_{=1} p)$. The first conjunct requires the propositions representing the instruction counter change according to the desired scheme. The second conjunct makes sure that the number of p_C-states is the same in the next configuration as in the current one. The third conjunct holds at time t, iff the number of p_D-states in the interval $[t+1, t+2)$ is precisely one greater than the corresponding number for the interval $[t, t+1)$.

The recurrence requirement is expressed by the conjunct $\Box\Diamond p_1$. It follows that the satisfiability question for the extended logic is Σ_1^1-hard.

[Σ_1^1-containment]: Let ϕ be a formula of MITL possibly using singular interval subscripts also. First observe that Theorem 4.15 holds even in presence of singular interval subscripts. Hence if ϕ has a model, then it has a model with rational interval end-points. Now the satisfiability of ϕ can be phrased as a Σ_1^1-sentence, asserting that some timed state sequence with rational transition times is a model of ϕ. It is routine to encode a rational model by a set of natural numbers, and to express the satisfaction relation in first-order arithmetic. ■

Another possible extension of the syntax is to permit time bounds for *both* arguments of the until operator. The intended semantics of $(\phi_1 \,{}_{I'}\mathcal{U}_I\, \phi_2)$ is that it holds in a timed state sequence ρ provided ϕ_2 holds at some time instant $t \in I$ and ϕ_1 holds throughout $(0, t) \cap I'$. However such an extension leads to undecidability. A close inspection of the proof of Theorem 4.17 shows that the only operator using the equality subscript is $\Diamond_{=1}$. The formula $\Diamond_{=1}\phi$ can be replaced by $\mathbf{false}\,{}_{\geq 1}\mathcal{U}_{\geq 1}\, \phi$.

4.3 Interval automata

In this section we define *interval automata* as a model for finite-state real-time systems. Just as Kripke structures are the finite models for PTL-specifications, interval automata

will be the finite models for MITL-specifications.

Interval automata are similar to timed transition tables. The main difference is that in a timed transition table, the transitions are labeled with events and clock constraints, whereas in a timed states are labeled with atomic propositions and clock constraints. With this modification interval automata now generate timed state sequences, instead of timed traces.

A interval automaton operates with finite control — a finite set of states and a finite set of real-valued clocks. All clocks proceed at the same rate and measure the amount of time that has elapsed since they were started (or reset). Each transition of the automaton may reset some of the clocks, and each state of the automaton puts certain constraints on the values of the atomic propositions as well as on the values of the clocks: the control of the automaton can reside in a particular state only if the values of the propositions and clocks satisfy the corresponding constraints.

Recall that for a set X of clocks, $\Phi(X)$ denotes the set of allowed clock constraints. Each clock constraint is a Boolean combination of atomic conditions which compare clock values with constants.

Definition 4.18 A *interval automaton* is a tuple $\mathcal{M} = \langle \mathrm{S}, \mathrm{S}_0, \mu, \mathrm{C}, \Delta, \mathrm{E} \rangle$, where

- S is a finite set of states.
- $\mathrm{S}_0 \subseteq \mathrm{S}$ is a set of initial states.
- $\mu : \mathrm{S} \rightarrow 2^{\mathrm{AP}}$ is a labeling function assigning to each state the set of atomic propositions true in that state.
- C is a finite set of clocks.
- $\Delta : \mathrm{S} \rightarrow \Phi(\mathrm{C})$ is a labeling function assigning to each state the clock constraint that should hold in that state.
- $\mathrm{E} \subseteq \mathrm{S} \times \mathrm{S} \times 2^{\mathrm{C}}$ is a set of edges. An edge $\langle s, s', \lambda \rangle$, also denoted as $s \xrightarrow{\lambda} s'$, represents a transition from state s to state s', and λ gives the set of clocks to be reset with this transition.

■

The *runs* of an interval automaton define timed state sequences. At any time instant during a run, the configuration of the system is completely determined by the state in which the control resides and the values of all clocks. The values of all clocks are given by a *clock interpretation* ν, which is a map from C to R: for any clock $x \in \mathrm{C}$, the value of x under the interpretation ν is $\nu(x) \in \mathsf{R}$.

Assume that, at time $t \in \mathsf{R}$, the interval automaton is in state s and the clock values are given by the clock interpretation ν. Suppose the state of the automaton remains unchanged during the time interval I with $l(I) = t$. All clocks proceed at the same rate as time elapses; at any time $t' \in I$ the value of any clock x is $\nu(x) + t' - t$, so the clock interpretation is $\nu + t' - t$. During all this time the value of the clock interpretation satisfies the clock constraint that is associated with s; that is, $(\nu + t' - t) \models \Delta(s)$. Now suppose that the automaton changes its state at time $r(I) = t''$ via the transition $s \xrightarrow{\lambda} s'$. This state change happens in one of the possible two ways. If I is right-closed, then the state at time t'' is still s, otherwise the state at time t'' is s'. The clocks in λ get reset to 0 at time t''. Let ν'' be the clock interpretation $[\lambda \mapsto 0](\nu + t'' - t)$, which gives the valuation for the clocks at the beginning of the next interval. The clock interpretation at the transition time t'', however, depends on whether the state at time t'' is s or s'. If I is right-closed, then the clock interpretation at time t'' is $(\nu + t'' - t)$ and should satisfy $\Delta(s)$. If I is right-open then the clock values at the transition point are given by ν'', and should satisfy $\Delta(s')$. The state s' stays unchanged over some interval adjacent to I, and the same cycle repeats.

Let $\Gamma(\mathcal{M})$ denote $[\mathrm{C} \mapsto \mathsf{R}]$, the set of clock interpretations for the clocks of M. The behavior of the system is formally defined below:

Definition 4.19 A run r of an interval automaton $\mathcal{M} = \langle \mathrm{S}, \mathrm{S}_0, \mu, \mathrm{C}, \Delta, \mathrm{E} \rangle$ is an infinite sequence

$$r: \quad \xrightarrow[\nu_0]{} (s_0, I_0) \xrightarrow[\nu_1]{\lambda_1} (s_1, I_1) \xrightarrow[\nu_2]{\lambda_2} (s_2, I_2) \xrightarrow[\nu_3]{\lambda_3} \cdots$$

of states $s_i \in \mathrm{S}$, intervals I_i, clock sets $\lambda_i \subseteq \mathrm{C}$, and clock interpretations $\nu_i \in \Gamma(\mathcal{M})$ satisfying the following constraints:

- *Initiality:* $s_0 \in \mathrm{S}_0$,
- *Consecution:*
  - the sequence $\overline{I} = I_0 I_1 I_2 \ldots$ forms an interval sequence,
  - for all $i \geq 0$ either $\langle s_i, s_{i+1}, \lambda_i \rangle \in \mathrm{E}$ or $s_{i+1} = s_i$ with $\lambda_i = \emptyset$,

- $\nu_{i+1} = [\lambda_{i+1} \mapsto 0](\nu_i + r(I_i) - l(I_i))$ for all $i \geq 0$,

- *Timing:* for all $t \in I_i$, the clock interpretation $\nu_i + t - l(I_i)$, denoted by $\gamma_r(t)$, satisfies $\Delta(s_i)$.

■

Note that, according to this definition, the clocks may start at any real values that satisfy the clock constraints of an initial state. Requiring the initial values of all the clocks to be 0 would complicate the presentation of the model-checking algorithm for MITL (to be presented in Section 4.4). A run r gives a map γ_r from R to $\Gamma(\mathcal{M})$ giving the clock values at every instant of time.

Note that the above definition assumes that there are implicit self-loops on every state of the automaton, that is, introducing an edge $s \stackrel{\emptyset}{\longrightarrow} s$ does not change the set of runs of the interval automaton.

We can associate timed state sequences with the runs, and use them to interpret MITL-formulas.

Definition 4.20 The timed state sequence ρ_r associated with a run r is

$$\rho_r: \quad (\mu(s_0), I_0) \rightarrow (\mu(s_1), I_1) \rightarrow (\mu(s_2), I_2) \rightarrow (\mu(s_3), I_3) \rightarrow \cdots$$

$L(\mathcal{M})$ denotes the set of all timed state sequences ρ_r that correspond to the runs of the interval automaton M.

An interval automaton M satisfies an MITL-specification ϕ iff $L(\mathcal{M}) \subseteq L(\phi)$. ■

We say that $\mathcal{M}$ generates (or accepts) the timed state sequences in $L(\mathcal{M})$. The set $L(\mathcal{M})$ gives the set of all possible behaviors of the real-time system modeled by M.

Observe that the above definition allows transient or instantaneous states along a run. This corresponds to conditions (such as propositions denoting occurrence of events) that hold only during state-transitions. Such states can be forced along a run by associating clock constraints involving equality with the state. Also there is no commitment to defining the state at the point of transition: we only require that the clock interpretation consistent with this choice should satisfy the corresponding clock constraint.

Let us consider some examples.

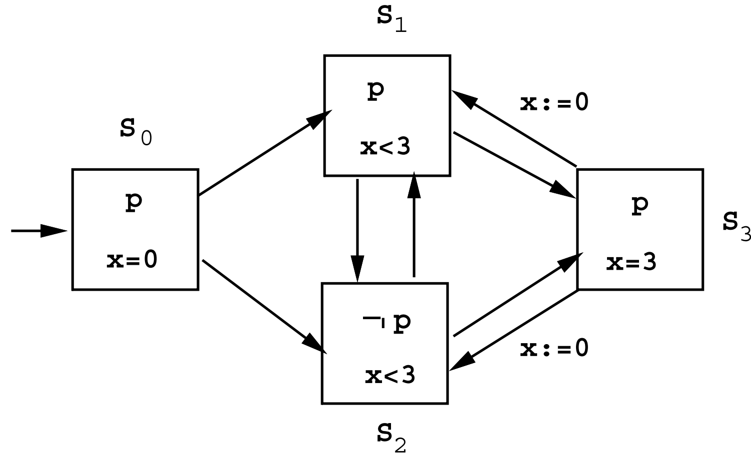


Figure 4.1: Interval automaton $\mathcal{M}_\infty$

Example 4.21 The interval automaton $\mathcal{M}_\infty$ of Figure 4.1 has four states s_0 to s_3, a single clock x, and a single proposition p.

The automaton starts in state s_0 with the clock x set to 0 and the proposition p true. During the time interval $(0, 3)$ the automaton loops between states s_1 and s_2, and thus, the proposition p may have any value. At time 3 the automaton moves to state s_3 to check that p holds. The clock is reset at this point, and the whole cycle repeats. Thus $\mathcal{M}_\infty$ requires p to hold at all time values that are integer multiples of 3. A prefix of a possible run of the automaton is shown below. Since there is a single clock, the clock interpretation is given by a real value.

$$\xrightarrow[0]{} (s_0, [0,0]) \xrightarrow[0]{\emptyset} (s_1, (0, 1.1)) \xrightarrow[1.1]{\emptyset} (s_2, [1.1, 3)) \xrightarrow[3]{\emptyset} (s_3, [3,3]) \xrightarrow[0]{\{x\}} (s_2, (3, 4]) \cdots$$

The set of timed state sequences generated by $\mathcal{M}_\infty$ is

$$L(\mathcal{M}_\infty) = \{\rho \mid \forall n \in \mathsf{N}.\, p \in \rho^*(3 \cdot n)\}.$$

■

Example 4.22 Consider the interval automaton $\mathcal{M}_\in$, shown in Figure 4.2 with seven states, s_0 to s_6, and uses two clocks, x and y.

The automaton starts in the initial state s_0 with the clock y initialized to 0. At time 40 the automaton moves to state s_6, and simply loops there. The proposition p denotes an external event which is true only at instantaneous points $t < 40$ in time (and no more than

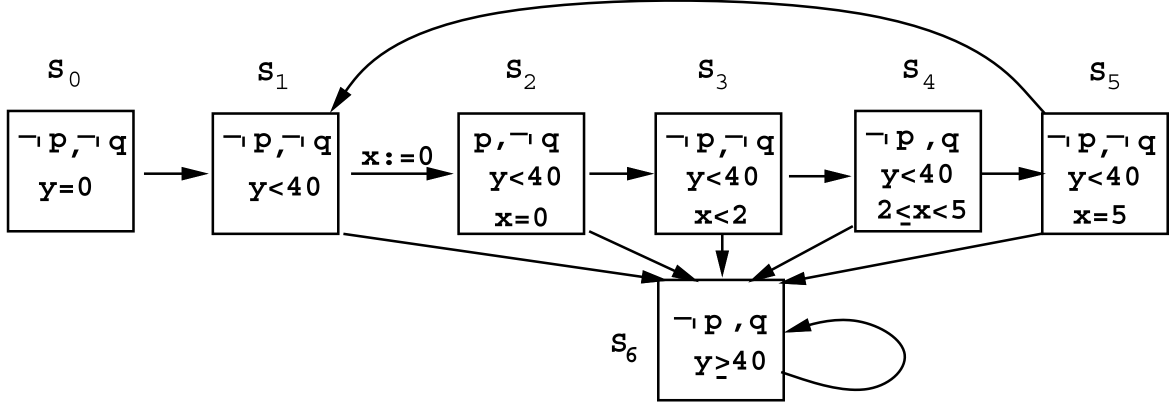


Figure 4.2: Interval automaton $\mathcal{M}_{\in}$

once every 5 time units), namely, whenever $\mathcal{M}_{\in}$ is in state s_2. The automaton responds to p by resetting the clock x, and then it requires that the proposition q holds over the interval $t + [2, 5)$. Thus the automaton $\mathcal{M}_{\in}$ models a system which responds, until time 40, to the event p by setting q to true for the interval $[2, 5)$ following p. A possible timed state sequence generated by $\mathcal{M}_{\in}$ is

$$(\emptyset, [0, 13)) \to (\{p\}, [13, 13]) \to (\emptyset, (13, 15)) \to (\{q\}, [15, 20)) \to (\emptyset, [20, 40)) \to (\{q\}, [40, \infty)).$$

The interval automaton satisfies the MITL-formula

$$\Box_{[0,40)}[p \ \to \ \Box_{[2,5)} q] \ \wedge \ \Box_{\geq 40}\, q.$$

■

Checking emptiness

The emptiness problem for interval automata is to decide whether $L(\mathcal{M})$ is empty for a given automaton M. This problem can be solved using the same techniques that we used to solve the emptiness problem for timed automata in Section 3.3. Also the model-checking algorithm for TCTL (to be defined in Chapter 5) can be used to check for the emptiness of an interval automaton. It follows that the problem can be solved in PSPACE; specifically, in time $O[(|\mathrm{S}| + |\mathrm{E}|) \cdot 2^{|\Delta|}]$, where $|\Delta|$ denotes the length of clock constraints.

The proof of PSPACE-hardness of the emptiness problem for timed automata (see Theorem 3.39) is easily modified to show PSPACE-hardness for interval automata also. The next theorem follows.

Theorem 4.23 Given an interval automaton M, the problem of deciding whether $L(\mathcal{M})$ is empty, is PSPACE-complete. ■

Composing interval automata

To describe any real-time system it is useful to describe individual components separately and then combine the different descriptions. We discussed this operation, called parallel composition, in context of timed automata. Interval automata describing behaviors of different components can be put together in a similar way. The next theorem gives the construction for composing interval automata.

Theorem 4.24 Given interval automata $\mathcal{M}_\rangle = \langle S_\rangle, S_{\prime_\rangle}, \mu_\rangle, C_\rangle, \cdot_\rangle, E_\rangle \rangle$, $1 \leq i \leq 2$, there exists an interval automaton $\mathcal{M}$ such that $L(\mathcal{M}) = L(\mathcal{M}_\infty) \cap L(\mathcal{M}_\in)$.

Proof. We assume that the clock sets C_i are disjoint. The states of $\mathcal{M}$ are of the form $\langle s_1, s_2 \rangle$, where s_1 is a state of $\mathcal{M}_\infty$ and s_2 is a state of $\mathcal{M}_\in$ such that both s_1 and s_2 agree on the assignment of truth values to propositions, that is, $\mu_1(s_1) = \mu_2(s_2)$. The clock constraint for $\langle s_1, s_2 \rangle$ is the conjunction of the clock constraints for s_1 and s_2. For any pair of transitions $s_1 \xrightarrow{\lambda_1} s_1'$ and $s_2 \xrightarrow{\lambda_2} s_2'$ in $\mathcal{M}_\infty$ and $\mathcal{M}_\in$, respectively, the product automaton has *three* transitions, $\langle s_1, s_2 \rangle \xrightarrow{\lambda_1 \cup \lambda_2} \langle s_1', s_2' \rangle$, $\langle s_1, s_2 \rangle \xrightarrow{\lambda_1} \langle s_1', s_2 \rangle$, and $\langle s_1, s_2 \rangle \xrightarrow{\lambda_2} \langle s_1, s_2' \rangle$. Thus the transitions of $\mathcal{M}$ simulate the joint behavior of the two component automata. ■

Introducing fairness

In verifying concurrent systems, we are generally interested only in correctness along the *fair* computation paths [LPS82]. For example, in a system with two processes, we may wish to consider only those computation sequences in which each process executes infinitely often. For a Kripke structure $\mathcal{M} = (S, S_0, \mu, E)$, a standard way to introduce fairness requirements uses a *fairness family* $\mathcal{F} \subseteq 2^S$. A path $(s_0, s_1, \ldots)$ is fair with respect to $\mathcal{F}$ iff for each $F \in \mathcal{F}$, $s_i \in F$ for infinitely many $i \geq 0$. Note that the fairness constraints for Kripke structures are similar to the acceptance conditions for ω-automata. In the automata framework, the fairness requirement of the above form has been called *generalized Büchi* condition. We introduce fairness in interval automata in a similar way.

Definition 4.25 An interval automaton with fairness is an automaton $\langle S, S_0, \mu, C, \Delta, E\rangle$ with a fairness family $\mathcal{F} \subseteq 2^S$.

A run r of the form

$$r: \quad \xrightarrow[\nu_0]{} (s_0, I_0) \xrightarrow[\nu_1]{\lambda_1} (s_1, I_1) \xrightarrow[\nu_2]{\lambda_2} (s_2, I_2) \xrightarrow[\nu_3]{\lambda_3} \cdots$$

is called $\mathcal{F}$-fair iff for every $F \in \mathcal{F}$, for infinitely many $i \geq 0$, $s_i \in F$.

For an interval automaton $\mathcal{M}$ with a fairness family $\mathcal{F}$, the set $L(\mathcal{M})$ of timed state sequences generated by M consists of all timed state sequences ρ_r corresponding to the $\mathcal{F}$-fair runs r of M. ■

The algorithm to check emptiness of $L(\mathcal{M})$ can be generalized to handle fairness constraints in a straightforward way. Similarly, the product construction for interval automata is easily extended to handle fairness constraints.

4.4 Deciding MITL

We solve the satisfiability problem for MITL by reducing it to the emptiness problem for interval automata. Our main result is that, given an MITL-formula ϕ, we can construct an interval automaton $\mathcal{M}_\phi$ such that the runs of $\mathcal{M}_\phi$ that meet certain fairness requirements correspond precisely to the timed state sequences that satisfy ϕ.

4.4.1 Restricting the problem

To simplify the exposition of the decision procedure, we restrict the satisfiability question for MITL to formulas and models of a specific form and show that this can be done without loss of generality.

Definition 4.26 An MITL-formula ϕ is said to be in *normal form* iff it is built from atomic propositions and negated atomic propositions using conjunctions, disjunctions, and temporal subformulas ψ of the following six types:

1. $\psi_1 \, \mathcal{U}_I \, \psi_2$ with bounded I and $l(I) > 0$.

2. $\psi_1 \, {}_I\mathrm{U} \, \psi_2$ with bounded I and $l(I) > 0$.

3. $\Diamond_I \, \psi'$ with $I = (0, n)$ or $I = (0, n]$.

4. $\Box_I \psi'$ with $I = (0, n)$ or $I = (0, n]$.

5. $\psi_1 \mathcal{U} \psi_2$.

6. $\Box \psi'$.

■

Next we give a series of transformations that allow us to rewrite any formula ϕ into an equivalent formula ϕ^* in normal form. Thus we restrict ourselves to test the satisfiability of MITL-formulas each of whose temporal subformulas are, according to the above classification, of one of six types, *type-1* to *type-6*.

First, we require that no interval in ϕ contains 0. This can be achieved by applying the following equivalence:

$$\psi_1 \mathcal{U}_I \psi_2 \leftrightarrow (\psi_2 \vee \psi_1 \mathcal{U}_{I \cap (0,\infty)} \psi_2)$$

provided that $0 \in I$.

Secondly, we require that the only unbounded intervals in ϕ are of the form $(0, \infty)$. This can be achieved by applying the following two equivalences:

$$\begin{aligned} \psi_1 \mathcal{U}_{(n,\infty)} \psi_2 &\leftrightarrow \Box_{(0,n]} (\psi_1 \wedge \psi_1 \mathcal{U} \psi_2) \\ \psi_1 \mathcal{U}_{[n,\infty)} \psi_2 &\leftrightarrow \Box_{(0,n)} \psi_1 \wedge \Box_{(0,n]} (\psi_2 \vee (\psi_1 \wedge \psi_1 \mathcal{U} \psi_2)) \end{aligned}$$

provided that $n > 0$.

Thirdly, we require that only the *eventually* and the *always* operators are constrained with bounded intervals I such that $l(I) = 0$. This can be achieved by applying the following equivalence:

$$\psi_1 \mathcal{U}_I \psi_2 \leftrightarrow \Diamond_I \psi_2 \wedge \psi_1 \mathcal{U} \psi_2$$

provided that $l(I) = 0$.

Finally, we push all negations in ϕ to the inside and use the following equivalence to eliminate each subformula of the form $\psi_1 \mathrm{U} \psi_2$:

$$\psi_1 \mathrm{U} \psi_2 \leftrightarrow \Box \psi_1 \vee \psi_1 \mathcal{U} (\psi_1 \wedge \psi_2).$$

It is easy to check that the resulting formula is in normal form.

It may appear that these rewritings blow up the size of the formula ϕ, but observe that the number of distinct subformulas of ϕ^* is only linear in the length of ϕ. This is because

at each step of rewriting, only a constant number of new subformulas are created. In fact, if the formula is represented as a directed acyclic graph, thus avoiding the duplication of shared subformulas, then the size of the formula blows up only by a constant factor. The next lemma follows.

Lemma 4.27 For every MITL-formula ϕ there exists an equivalent formula ϕ^* in normal form such that

- the largest (integer) constant appearing as an interval end-point in ϕ^* is the same as the largest constant appearing in ϕ, and
- if $|\phi|$ is the number of atomic propositions, Boolean connectives, and temporal operators in ϕ, then the number of distinct syntactic subformulas of ϕ^* is $O(|\phi|)$.

PROOF. For each step of rewriting, we can show that if a formula ψ is rewritten to ψ' then the number of syntactic subformulas in ψ' is bounded by a constant multiple of the number of subformulas in ψ by induction on the automaton of ψ. Also note that the rewriting steps do not introduce any new constants. ■

Henceforth we assume that the MITL-formulas under consideration are in normal form. Let $K \in \mathbb{N}$ be such that $K - 1$ is the largest constant appearing as an interval end-point in the formula ϕ to be checked.

To check the satisfiability of an MITL-formula ϕ, by Lemmas 4.9 and 4.11 we can confine ourselves to the question if ϕ has a ϕ-fine model. Therefore we consider, throughout this section, only ϕ-fine timed state sequences $\rho = (\sigma, \overline{I})$. It follows that, if ψ is a subformula of ϕ, we may write $\rho^i \models \psi$ for "$\rho^t \models \psi$ for all $t \in I_i$." In addition, we assume that all intervals in $\overline{I}$ are either singular or open. This is sufficient, because any model of ϕ can be brought into this form by splitting all nonsingular closed intervals; for instance, the interval $[a, b)$ can be split into the two intervals $[a, a]$ and (a, b).

Let us introduce a new atomic proposition p_{sing} such that $\rho^i \models p_{sing}$ iff the i-th interval I_i of $\rho = (\sigma, \overline{I})$ is singular. Hence the proposition p_{sing} holds exactly in every other interval. For a timed state sequence ρ that satisfies these conditions, Then:

- $\rho \models \psi_1 \mathcal{U}_I \psi_2$ iff for some i with $I_i \cap I \neq \emptyset$, (1) both $\rho^i \models \psi_2$ and $\rho^i \models \psi_1 \vee p_{sing}$, and (2) $\rho^j \models \psi_1$ for all $0 < j < i$, and (3) $\rho^0 \models \psi_1 \vee p_{sing}$.

- $\rho \models \psi_1 \, {}_I\mathrm{U} \, \psi_2$ iff $\rho^0 \models \psi_1$ if $I_0 \cap I \neq \emptyset$, and either (1) $\rho^0 \models \psi_2 \wedge \neg p_{sing}$, or (2) $\rho^i \models \psi_2$ for some $i > 0$ and $\rho^j \models \psi_1$ for all $0 < j \leq i$ with $I_j \cap I \neq \emptyset$, or (3) $\rho^j \models \psi_1$ for all $j > 0$ with $I_j \cap I \neq \emptyset$.

The different types of temporal subformulas of ϕ are handled differently by our algorithm. The simplest case is that of type-5 and type-6 formulas; they are treated essentially in the same way in which tableau decision procedures for linear temporal logic handle unconstrained temporal operators. The most interesting case is that of type-1 and type-2 formulas. We concentrate first on this case. The case of type-3 and type-4 formulas will be considered later.

4.4.2 Intuition for the algorithm

Consider the MITL-formula

$$\Box_{[0,1)} (p \rightarrow \Diamond_{[1,2]} q).$$

Let us assume, for simplicity, that both p and q are true only in singular intervals and let us try to build an interval automaton that accepts precisely the models of this formula.

Whenever the automaton visits a p-state, it needs to make sure that within 1 to 2 time units a q-state is visited. This can be done by setting a clock x to 0 when the p-state is visited, and demanding that some q-state with the clock constraint $1 \leq x \leq 2$ is visited later. This strategy requires a clock per visit to a p-state within the interval $[0,1)$. However, the number of such visits is potentially unbounded and, hence, any automaton with a fixed number of clocks cannot reset a new clock for every visit. That is why this simple strategy cannot be made to work.

An alternative approach is to guess the times for future q-states in advance. The automaton nondeterministically guesses two time values t_1 and t_2 within the interval $[0,1)$; this is done by resetting a clock x at time t_1 and another clock y at time t_2. The guess is that the *last* q-state within the interval $[1,2)$ is at time $t_1 + 1$, and that the *first* q-state within the interval $[2,3)$ is at time $t_2 + 2$. If the guesses are correct, then the formula $\Diamond_{[1,2]} q$ holds during the intervals $[0, t_1]$ and $[t_2, 1)$, and does not hold during the interval (t_1, t_2). Consequently, the automaton requires that every p-state within the interval $[0,1)$ lies either within $[0, t_1]$ or within $[t_2, 1)$. It also needs to make sure that the guesses are right; that is, whenever either $x = 1$ or $y = 2$, the automaton must be in a q-state. This strategy requires only two clocks for the interval $[0,1)$ of length 1, irrespective of the number of p-states

within $[0, 1)$.

We say that the guessed times $t_1 + 1$ and $t_2 + 2$ *witness* the formula $\Diamond_{[1,2]} q$ throughout the intervals $[0, t_1]$ and $[t_2, 1)$, respectively. In general, the witnesses need not be singular intervals, they can be open intervals. To see this, let us relax the assumption that q holds only in the singular intervals. Let $0 < t_1 < t_1' < 1$ be such that q is true during $I_1 = (t_1+1, t_1'+1)$ and false during $[t_1'+1, 2]$, and let $0 < t_2 < t_2' < 1$ be such that q is false during $[2, t_2+2]$ and true over $I_2 = (t_2+2, t_2'+2)$. Thus I_1 is the last q-interval within $[1, 2)$, and I_2 is the first q-interval within $[2, 3)$. The formula $\Diamond_{[1,2]} q$ holds during the intervals $[0, t_1')$ and $(t_2, 1)$, and does not hold during the interval $[t_1', t_2]$. To check the formula, the automaton nondeterministically guesses four time values t_1, t_1', t_2, and t_2'. It requires that no p-state lies within $[t_1', t_2]$, and also it needs to ensure that the guesses are right. In this case we say that the intervals I_1 and I_2 witness the formula $\Diamond_{[1,2]} q$ throughout the intervals $[0, t_1')$ and $(t_2, 1)$, respectively. Notice that we could not have chosen a particular time instant from I_1 as a witness for $[0, t_1')$; if I_1 were right-closed we could have chosen its right end-point as the witness.

In the following we develop an algorithm based on this idea of guessing, in advance, time intervals that witness temporal formulas and, later, checking the correctness of these guesses. The crucial fact that makes this strategy work, with a finite number of clocks, is that the *same* interval may serve as a witness for many points in time. We warn the reader that the details of this algorithm are quite tedious compared to the other parts of the thesis.

4.4.3 Witnessing intervals

Definition 4.28 The interval I' is a *witnessing interval* for the MITL-formula $\psi_1 \mathcal{U}_I \psi_2$ under ρ^t, for a timed state sequence ρ and $t \in \mathsf{R}$, iff $I' \cap (t + I) \neq \emptyset$ and $\rho^t \models \psi_1 \mathcal{U}_{J-t} \psi_2$ for every nonempty interval $J \subseteq I'$.

The interval I' is a witnessing interval for the MITL-formula $\psi_1 {}_I\mathrm{U}\, \psi_2$ under ρ^t iff $t + I \subseteq I'$ and $\rho^t \models \psi_1 {}_{I'-t}\mathrm{U}\, \psi_2$. ■

Observe that if I' witnesses $\psi_1 \mathcal{U}_I \psi_2$ under ρ^t, then ψ_1 holds throughout $(t, r(I'))$, and ψ_2 holds throughout the interval I'. Witnessing intervals are defined such that the following property holds:

Lemma 4.29 Let ψ be an MITL-formula of the form $\psi_1 \mathcal{U}_I \psi_2$ or $\psi_1 {}_I\mathrm{U}\, \psi_2$, let ρ be a timed state sequence and $t \in \mathsf{R}$. There is a witnessing interval for ψ under ρ^t iff $\rho^t \models \psi$.

PROOF. If $\rho^t \models \psi$ for the formula $\psi = \psi_1 \mathcal{U}_I \psi_2$, then $\rho^{t'} \models \psi_2$ for some $t' \in t + I$ and the singular interval $[t', t']$ witnesses ψ under ρ^t. If $\rho^t \models \psi$ for the formula $\psi = \psi_1 {}_I\mathrm{U}\, \psi_2$, then the interval $t + I$ witnesses ψ under ρ^t.

The other direction of the lemma follows from the semantic clauses for the *until* and *unless* operators. ■

Now we show that the same interval may serve as a witnessing interval for a temporal formula under (infinitely) many suffixes of a timed state sequence.

Example 4.30 Consider the timed state sequence ρ over two propositions p and q:

$$(\{p\}, [0, 1.2]) \rightarrow (\{p, q\}, (1.2, 1.6)) \rightarrow (\{p\}, [1.6, \infty)).$$

Thus along ρ the proposition p is always true, but the proposition q is true only during the interval $I_q = (1.2, 1.6)$. The interval I_q witnesses the formula $p \mathcal{U}_{(1,2)} q$ under ρ^t for every $t \in [0, 0.6)$. On the other hand, the interval $[1.6, 3]$ witnesses the formula $\Box_{(1,2)} (\neg q)$ under ρ^t for every $t \in [0.6, 1]$. ■

Lemma 4.31 Let ψ be the type-1 formula $\psi_1 \mathcal{U}_I \psi_2$. For every timed state sequence ρ, there are two bounded, open or closed, intervals I' and I'' such that, for every $t \in [0, 1)$, the formula ψ is satisfied by ρ^t iff either I' or I'' witnesses ψ under ρ^t. Furthermore, $r(I_i) \leq r(I) + 1$ for $i = 1, 2$.

PROOF. Let $\rho = (\sigma, \overline{I})$ be a ψ-fine timed state sequence with only singular and open intervals, including the singular interval $[r(I) + 1, r(I) + 1]$ (split intervals if necessary). We choose two witnessing intervals I' and I'' as follows:

- Let $\hat{\imath}$ be the *maximal* $i \geq 0$ such that $I_i \cap I \neq \emptyset$, both $\rho^i \models \psi_2$ and $\rho^i \models \psi_1 \vee p_{sing}$, and $\rho^k \models \psi_1$ for all $0 \leq k < i$ with $I_k \cap I \neq \emptyset$. If no such i exists, let $I' = \emptyset$; otherwise, let $I' = I_{\hat{\imath}}$.
- Let $\hat{\jmath}$ be the *minimal* $j \geq 0$ such that $I_j \cap (I + 1) \neq \emptyset$, both $\rho^j \models \psi_2$ and $\rho^j \models \psi_1 \vee p_{sing}$, and $\rho^k \models \psi_1$ for all $0 \leq k < j$ with $I_k \cap (I \cup I + 1) \neq \emptyset$. If no such j exists, let $I'' = \emptyset$; otherwise, let $I'' = I_{\hat{\jmath}}$.

Consider $0 \leq t < 1$. Assume ρ^t satisfies ψ. Clearly $\rho^{t'} \models \psi_1$ for all $t' < I$. If I' exists and $I' \cap (t + I) \neq \emptyset$ then I' witnesses ψ under ρ^t. Otherwise, let $t' \in (t + I)$ be such that

$\rho^{t'} \models \psi_2$ and $\rho^{t''} \models \psi_1$ for all $t'' \in (t, t')$. In this case I'' exists, if $t' \in I_k$ then $\hat{\jmath} \leq k$. Hence $I_{\hat{\jmath}} \cap (t + I) \neq \emptyset$, and I'' witnesses ψ under ρ^t. ■

In the case of type-2 formulas, a single witness per unit interval suffices to reduce the problem to type 3:

Lemma 4.32 Let ψ be the type-2 formula $\psi_1 \,{}_I\mathrm{U}\, \psi_2$. For every timed state sequence ρ, there is a bounded interval I' such that, for every $t \in [0, 1)$, the formula ψ is satisfied by ρ^t iff either ρ^t satisfies the type-3 formula $\Diamond_{(0,\infty)\cap(<I)} \psi_2$ or I' witnesses ψ under ρ^t. Furthermore, $r(I') \leq r(I) + 1$.

Proof. Let $\rho = (\sigma, \overline{I})$ be a ψ-fine timed state sequence with only singular and open intervals, including the singular interval $I_n = [r(I) + 1, r(I) + 1]$. We choose witnessing interval I' as follows:

- Let $\hat{\imath}$ be the minimal $i \geq 0$ such that $I_i \cap I \neq \emptyset$ and either
  1. $\rho^k \models \psi_1$ for all $k \geq i$ with $I_k \cap I \neq \emptyset$, or
  2. there is some $i \leq j \leq n$ such that $\rho^j \models \psi_1 \wedge \psi_2$ and $\rho^k \models \psi_1$ for all $i \leq k < j$.
- Given $\hat{\imath}$, let $\hat{\jmath}$ be the maximal $\hat{\imath} \leq j \leq n$ such that either $\rho^k \models \psi_1$ for all $\hat{\imath} \leq k \leq j$, or $\rho^k \models \psi_1 \wedge \psi_2$ for some $\hat{\imath} \leq k \leq j$. Note that if $\hat{\imath}$ exists, then so does $\hat{\jmath}$; in particular, if $\hat{\imath}$ exists because of clause 2, then $\hat{\jmath} = n$.

If no appropriate $\hat{\imath}$ exists, let $I' = \emptyset$; otherwise, let I' be the union of all I_k for $\hat{\imath} \leq k \leq \hat{\jmath}$.

Assume that $0 \leq t < 1$; then ρ^t satisfies ψ iff either (1) $\rho^i \models \psi_1$ for all i with $I_i \cap (t+I) \neq \emptyset$, or (2) $\rho^i \models \psi_1 \wedge \psi_2$ for some i with $I_i \cap (t + I) \neq \emptyset$ and $\rho^j \models \psi_1$ for all $j < i$ with $I_j \cap (t + I) \neq \emptyset$, or (3) $\rho^{t'} \models \psi_2$ for some $t < t' < t + I$. In either of the first two cases, I' witnesses ψ under ρ^t; the third case is equivalent to ρ^t satisfying the formula $\Diamond_{(0,\infty)\cap(<I)} \psi_2$. If I' witnesses ψ under ρ^t, then $\rho^t \models \psi$ by Lemma 4.29. ■

4.4.4 Type-1 and type-2 formulas

Now we can be more precise about how we will construct the interval automaton $\mathcal{M}_\phi$ that accepts exactly the models of ϕ. To check the truth of type-1 and type-2 subformulas of ϕ, the automaton guesses corresponding witnessing intervals. The boundaries of a witnessing interval are marked by clocks: a *clock interval* is a bounded interval that is defined by

its *type* (e.g., left-closed and right-open) and a pair of clocks. Given a time t and a clock interpretation ν, the clock interval $C = [x, y]$, for two clocks x and y, stands for the closed witnessing interval $[t + K - \nu(x), t + K - \nu(y)]$; the clock interval $C = [x, y)$ stands for the corresponding half-open interval, etc. (recall that $K - 1$ is the largest constant appearing in the formula). We write $K - C$ for the interval $\{K - \nu(x), K - \nu(y)\}$, for any type of clock interval $C = \{x, y\}$.

For simplicity, let us consider a type-1 subformula ψ of the form $\Diamond_I \psi'$. The automaton resets, nondeterministically, any of its clocks at any time. When guessing a witnessing interval I', it writes the prediction that "the clock interval $C = \{x, y\}$ witnesses the formula ψ" into its memory. If the clock x was reset at time t_1, and y was reset at time $t_2 \geq t_1$, then the witnessing interval guessed is $I' = \{t_1 + K, t_2 + K\}$. To check the the truth of the temporal formula ψ at time $t \geq t_2$, the automaton needs to verify that its guess I' is indeed a witness. The condition $I' \cap (t + I) \neq \emptyset$ translates to verifying the clock constraint $(K - C) \cap I \neq \emptyset$. It remains to be checked that ψ' is satisfied throughout the witnessing interval I'; that is, the automaton needs to verify that ψ' holds at all states with the clock constraint $0 \in (K - C)$.

The Lemmas 4.31 and 4.32 are the key to constructing an automaton that needs only *finitely* many clocks. For the type-1 formula $\psi_1 \mathcal{U}_I \psi_2$, at most 2 witnessing intervals need to be guessed per interval of unit length. Furthermore, the fact that the right end-point of a witnessing interval is bounded allows the automaton to reuse every clock after a period of length $r(I)+1$. Thus we need, at any point in time, at most $2r(I)+2$ *active* clock intervals; that is, clock intervals that stand for a guess of a witnessing interval and, therefore, have to be verified later. Similarly, to check a type-2 formula $\psi_1 \, {}_I\mathrm{U} \, \psi_2$, we need, at any point in time, no more than $r(I)+1$ active clock intervals. Consequently, $4K$ clocks suffice to check any type-1 subformula of ϕ, and $2K$ clocks suffice for any type-2 subformula of ϕ.

4.4.5 Type-3 and type-4 formulas

Now let us move to formulas of the form $\Diamond_I \psi'$ and $\Box_I \psi'$ with $I = (0, n)$ or $I = (0, n]$. Checking the truth of such a formula is much easier and can be done using a single clock.

Consider the type-3 formula $\psi = \Diamond_I \psi'$. Whenever the automaton needs to check that ψ holds, say at time t, it starts a clock x and writes the corresponding proof obligation into its memory — to verify that ψ' holds at some later state with the clock constraint $x \in I$. The obligation is discharged as soon as an appropriate ψ'-state is found. If the

automaton encounters another ψ-state in the meantime, at time $t' > t$ before the obligation is discharged, it does not need to check the truth of ψ separately for this state. This is because if there is a ψ'-state after time t' within the interval $t + I$, then both $\rho^t \models \Diamond_I \psi'$ and $\rho^{t'} \models \Diamond_I \psi'$. Once the proof obligation is discharged, the clock x can be used again. Thus one clock suffices to check the formula ψ as often as necessary.

The described strategy works for checking the truth of ψ at singular intervals. There is, however, a subtle problem with this method when the truth of ψ during open intervals needs to be checked, as is illustrated by the following example. Consider the timed state sequence

$$(\{\}, [0,0]) \rightarrow (\{\}, (0,1)) \rightarrow (\{p\}, [1,\infty]);$$

it satisfies the formula $\Diamond_{(0,1)} p$ at all times $t \in (0,1)$. To check the truth of $\Diamond_{(0,1)} p$ during the open interval $(0,1)$, the automaton starts a clock x upon entry, at time 0. However, the proof obligation that p holds at some later state with the clock constraint $x \in (0,1)$ can never be verified. On the other hand, if the automaton were to check, instead, the truth of the formula $\Diamond_{(0,1]} p$ during the interval $(0,1)$, then our strategy works and the corresponding proof obligation can be verified, because there is a p-state while $x \in (0,1]$ holds. Furthermore, observe that the validity of $\Diamond_{(0,1]} p$ throughout the open interval $(0,1)$ implies that $\Diamond_{(0,1)} p$ is also true throughout $(0,1)$.

In general, the following lemma holds:

Lemma 4.33 Let ψ and $\hat{\psi}$ be the type-3 MITL-formulas $\Diamond_I \psi'$ and $\Diamond_{I \cup \{r(I)\}} \psi'$, respectively. For every timed state sequence $\rho = (\sigma, \overline{I})$ and open interval I_i in $\overline{I}$, $\rho^i \models \psi$ iff $\rho^i \models \hat{\psi}$.

PROOF. First note that, for all $t \geq 0$, if ψ is satisfied by ρ^t, then $\hat{\psi}$ is also satisfied by ρ^t. This is because $I \subseteq I \cup \{r(I)\}$.

Now consider an open interval I_i and assume that $\rho^i \models \hat{\psi}$. If I is right-closed, then $\psi = \hat{\psi}$. So suppose that I is right-open, and let $t \in I_i$. Since I_i is open, there exists some $t' \in I_i$ with $t' < t$. Since $\rho^{t'} \models \hat{\psi}$, there exists some $j \geq i$ such that $I_j \cap (t' + (I \cup \{r(I)\})) \neq \emptyset$ and $\rho^j \models \psi'$. It follows that $I_j \cap (t + I) \neq \emptyset$ and, hence, that $\rho^t \models \psi$. ■

Consequently, to check the truth of a type-3 formula ψ during an open interval, it suffices to check the truth of the weaker formula $\hat{\psi}$. Accordingly, the automaton we construct writes only the proof obligation that corresponds to checking $\hat{\psi}$ into its memory.

For checking a type-4 formula of the form $\psi = \Box_I \psi'$, the situation is symmetric. The automaton uses also a single clock x to check this formula. Whenever the formula ψ needs to be verified, say at time t, the automaton starts the clock x with the proof obligation that as long as the clock constraint $x \in I$ holds, so does ψ'. The obligation is discharged as soon as $x > I$. If the automaton encounters another ψ-state within the interval $t + I$, say at time t', it simply resets the clock x, and thus overwrites the previous proof obligation. This strategy is justified by the observation that if ψ' holds throughout the interval $(t, t']$ and $\rho^{t'} \models \Box_I \psi'$, then also $\rho^t \models \Box_I \psi'$. Once the proof obligation is discharged, the clock x can be reused to check ψ again whenever necessary.

As in the case of type-3 formulas, we need to be more careful when checking ψ during open intervals. For the type-4 formula $\psi = \Box_I \psi'$, let $\hat{\psi}$ be the formula $\Box_{I-\{r(I)\}} \psi'$. From Lemma 4.33 and duality, it follows that for every timed state sequence $\rho = (\sigma, \overline{I})$, if I_i is open, then $\rho^i \models \psi$ iff $\rho^i \models \hat{\psi}$. Hence to check the truth of ψ during an open interval, it suffices again to check the truth of the weaker formula $\hat{\psi}$. Accordingly, only a proof obligation for $\hat{\psi}$ is set up. This is because the corresponding clock x is started at time $r(I_i)$, and for ψ to hold during the open interval I_i, ψ' need not hold at time $r(I_i) + r(I)$, even if I is right-closed.

4.4.6 Constructing the interval automaton

Now let us define the interval automaton $\mathcal{M}_\phi$ formally. For each temporal subformula of ϕ of type-1, the automaton $\mathcal{M}_\phi$ has $2K$ pairs of clocks. These clocks always appear in pairs, to form clock intervals. From any pair of clocks x and y, four different clock intervals can be formed: (x, y), $[x, y)$, $(x, y]$, and $[x, y]$. From Lemma 4.31 for checking type-1 formulas we need only open or closed witnessing intervals. Thus associated with each type-1 subformula ψ of ϕ we have $4K$ clock intervals; they are denoted by $C_1(\psi)$,... $C_{4K}(\psi)$. For each type-2 subformula of ϕ the automaton uses K clock pairs giving $4K$ clock intervals. For subformulas ψ of types 3 and 4, the automaton needs one clock x_ψ per formula.

In addition to these clocks, we use the clock x_{sing} to enforce that the runs of $\mathcal{M}_\phi$ have alternate singular and open intervals.

Closure set

For the given MITL-formula ϕ, we define its *closure* set $Closure(\phi)$ to consist of the following items:

1. All subformulas of ϕ; for each type-2 subformula $\psi_1 \, {}_I\mathrm{U} \, \psi_2$ of ϕ, the type-3 formula $\Diamond_{(0,\infty)\cap(<I)} \psi_2$; for each type-3 subformula $\psi = \Diamond_I \psi'$ of ϕ, the type-3 formula $\hat{\psi} = \Diamond_{I \cup \{r(I)\}} \psi'$; and for each type-4 subformula $\psi = \Box_I \psi'$ of ϕ, the type-4 formula $\hat{\psi} = \Box_{I - \{r(I)\}} \psi'$.

2. For each type-1 or type-2 formula ψ in the closure set, the clock intervals $C_1(\psi)$ through $C_{4K}(\psi)$; and for each type-3 and type-4 formula ψ in the closure set, the clock x_ψ.

3. For each clock interval $C = C_j(\psi)$ in the closure set, where ψ is $\psi_1 \mathcal{U}_I \psi_2$ or $\psi_1 \, {}_I\mathrm{U} \, \psi_2$, all clock constraints of the form $0 < (K - C)$, $0 \subset (K - C)$, $0 = (K - C)$, $(K - C) = \emptyset$, $I \subseteq (K - C)$, and $(K - C) \cap I \neq \emptyset$; and for each clock x_ψ in the closure set, where ψ is $\Diamond_I \psi'$ or $\Box_I \psi'$, the clock constraints $x \in I$ and $x > I$.

   We write $0 \subset (K - C)$ short for $\{0\} \subset (K - C)$. It should be clear that all of these conditions are indeed clock constraints. For instance, the condition $0 \subset (K - [x, y))$ stands for the clock constraint $x \leq K \wedge y > K$; the condition $0 = (K - [x, y))$ is never satisfied.

4. The clock constraint $x_{sing} = 0$.

■

Note that the number of subformulas of ϕ is $O(|\phi|)$ and the number of clocks is $O(K)$ for each subformula of ϕ. Hence the size of the closure set *Closure*(ϕ) is $O(|\phi| \cdot K)$.

The states of the desired automaton $\mathcal{M}_\phi$ will be subsets of *Closure*(ϕ). We need to consider only those subsets of *Closure*(ϕ) that satisfy certain local consistency constraints. Whenever the automaton is in state s, the formulas in s indicate which subformulas of ϕ are true. Accordingly, a state $s \subseteq$ *Closure*(ϕ) is initial iff both ϕ and $x_{sing} = 0$ are in s, and for each state s the propositional constraints $\mu(s)$ are defined such that $p \in \mu(s)$ iff $p \in s$ for all atomic propositions $p \in \mathrm{AP}$.

The clock constraints $\Delta(s)$ are the conjunction of all clock constraints in s. The clock intervals in s indicate which clock intervals are currently active and represent witnessing intervals for type-1 and type-2 formulas; the clocks in s indicate which clocks are currently active and represent proof obligations for type-3 and type-4 formulas.

The transitions of $\mathcal{M}_\phi$ are all triples $s \xrightarrow{\lambda} s'$ that satisfy certain global consistency criteria. Both the local and the global consistency conditions are defined in the following

catalog. For every state $s \subseteq \mathit{Closure}(\phi)$ and every transition $s \xrightarrow{\lambda} s'$ with source state s:

Logical consistency

- For each atomic proposition $p \in \mathrm{AP}$, precisely one of p and $\neg p$ is in s.
- If the formula $\psi_1 \wedge \psi_2$ is in s, then both ψ_1 and ψ_2 are in s.
- If the formula $\psi_1 \vee \psi_2$ is in s, then either ψ_1 or ψ_2 is in s.

These conditions ensure that no state contains subformulas of ϕ that are mutually inconsistent.

Timing consistency

- s contains at most one of the clock constraints $0 < (K - C)$, $0 \subset (K - C)$, $0 = (K - C)$, and $(K - C) = \emptyset$ for each clock interval C. Furthermore, no two clock intervals in s share clocks; for instance, s does not contain both the clock intervals (x, y) and $[x, y)$.
- s contains at most one of the clock constraints $x_\psi \in I$ and $x_\psi > I$ for each type-3 or type-4 formula ψ.
- If s contains $x_{sing} = 0$, then $x_{sing} \notin \lambda$. If s does not contain $x_{sing} = 0$, then $x_{sing} \in \lambda$ and s' contains $x_{sing} = 0$.

These conditions guarantee that no state contains clock constraints that are mutually inconsistent. We say that a state s is *singular* iff it contains $x_{sing} = 0$; otherwise s is *open*. The last clause of the above conditions ensures that singular and open states alternate along any run.

Type-1 formulas

Consider a type-1 formula $\psi = \psi_1 \mathcal{U}_I \psi_2$ in the closure set.

Firstly, if ψ is in s, then there is some clock interval $C = C_j(\psi)$ such that

- $(K - C) \cap I \neq \emptyset$ is in s, and
- either C is in s, or s is singular and C is in s' and the clocks associated with C are not in λ.

The first condition checks that the interval $K-C$ is an appropriate candidate for witnessing the formula ψ. The second condition activates the clock interval C to represent a witnessing interval for ψ. Note that if s is singular, the corresponding clock interval is activated only in the following open interval. This is because, to check that the interval $(K-C)$ is indeed a witness, no conditions are required of the current singular state.

Secondly, if some clock interval $C = C_j(\psi)$ is in s, then

- if either $0 = (K-C)$ or $0 \subset (K-C)$ is in s, then ψ_2 is in s, and
- if either $0 < (K-C)$ or $0 \subset (K-C)$ is in s, then ψ_1 is in s, and
- the clocks associated with C are not in λ and either C or $(K-C) = \emptyset$ is in s'.

The first two conditions verify that the active clock interval C represents indeed a witness for the formula ψ. The final condition keeps the clock interval C active as long as necessary.

To prove the correctness of the construction of $\mathcal{M}_\phi$, we show that along every run r of $\mathcal{M}_\phi$, if the state at time t contains the formula ψ then ρ_r^t satisfies ψ. The proof is by induction on the automaton of ψ. Let us consider the case when ψ is a type-1 formula. Suppose that the above conditions are satisfied along a run r and the formula ψ is in a state s at time t. Consider the clock interval $C = C_j(\psi)$ satisfying the consistency requirements. We show that the interval $I' = (t + K - C)$ is a witnessing interval for ψ under ρ_r^t. The constraint $(K-C) \cap I \neq \emptyset$ is satisfied, and hence $I' \cap (t+I) \neq \emptyset$ holds. If s is an open state then s contains C or if s is singular then its successor contains C. Note that until the condition $(K-C) = \emptyset$ becomes true marking the end of I', all the intermediate states contain C, and the clocks associated with C do not get reset, and hence continue to denote the same witness I'. In all states s' at time instants $t < t' < I$, the constraint $0 < (K-C)$ holds; by consistency conditions s' contains ψ_1, and hence by induction hypothesis, $\rho_r^{t'} \models \psi_1$. Similarly all states s' at time instants $t' \in I'$ contain the constraint $0 \subset (K-C)$ or $0 = (K-C)$, and hence, the formula ψ_2. Furthermore, if $t' \neq r(I')$ then $0 \subset (K-C)$ holds, and the state contains the formula ψ_1 also. Hence, by induction hypothesis, for all $t' \in I'$, $\rho_r^{t'} \models \psi_2$, and if $t' \neq r(I')$ then $\rho_r^{t'} \models \psi_1$. Thus I' satisfies all criteria to be a witness for ψ at time t. By Lemma 4.29, it follows that $\rho_r^t \models \psi$.

Conversely, if $\rho^t \models \psi$, then there is a run r that satisfies all conditions. This is because, by Lemma 4.31, the automaton can, at time t, either share an already activated clock interval $C_j(\psi)$ or has enough clocks to activate an unused clock interval $C_j(\psi)$. If C is the

activated clock interval and $K - C$ stands for the guessed witness, then it follows, from definitions, that all the consistency criteria are met. The first state in which $(K - C) = \emptyset$ holds, the automaton discards the clock interval C from the state.

Type-2 formulas

Consider a type-2 formula $\psi = \psi_1 \, {}_I\mathrm{U} \, \psi_2$ in the closure set.

Firstly, if ψ is in s, then either

- $\Diamond_{(0,\infty)\cap(<I)} \psi_2$ is in s , or
- there is some clock interval $C = C_j(\psi)$ such that
  - $I \subseteq (K - C)$ is in s, and
  - either C is in s, or s is singular and C is in s' and the clocks associated with C are not in λ.

If $\Diamond_{(0,\infty)\cap(<I)} \psi_2$ holds then so does ψ. The second clause corresponds to guessing a witness. The first condition checks that the interval $K - C$ is an appropriate candidate for witnessing the formula ψ. The second condition activates this clock interval C.

Secondly, if some clock interval $C = C_j(\psi)$ is in s, then

- if either $0 = (K - C)$ or $0 \subset (K - C)$ is in s, then ψ_1 is in s, and
- either ψ_2 is in s, or the clocks associated with C are not λ and either C or $(K - C) = \emptyset$ is in s'.

These conditions ensure that the active clock interval C represents indeed a witness for the formula ψ and that it is kept active as long as necessary.

Soundness and completeness of these conditions follow by the Lemmas 4.29 and 4.32.

Type-3 formulas

Consider a type-3 formula $\psi = \Diamond_I \psi'$ in the closure set.

Firstly, if ψ is in s, then either

- s is singular and $x_\psi \in s'$, or
- s is open and I is right-open and $\hat{\psi}$ is in s, or

- s is open and I is right-closed and x_ψ is in s.

These conditions activate a clock to represent a proof obligation. Lemma 4.33 justifies the decision to start a clock corresponding to the weaker formula $\hat{\psi}$ when s is open.

Secondly, if x_ψ is in s, then

- $x_\psi \in I$ is in s, and
- either ψ' is in s, or x_ψ is in s' and $x_\psi \notin \lambda$.

These conditions verify the proof obligation that is represented by the clock x_ψ and keep it active as long as necessary.

Type-4 formulas

Consider a type-4 formula $\psi = \Box_I \psi'$ in the closure set.

Firstly, if ψ is in s, then either

- s is singular and $x_\psi \in s'$ and $x_\psi \in \lambda$, or
- s is open and I is right-closed and $\hat{\psi}$ is in s, or
- s is open and I is right-open and $x_\psi \in s$ and $x_\psi \in s'$ and $x_\psi \in \lambda$.

These conditions activate a clock to represent a proof obligation, and reset it, as was justified in the previous subsection. Recall that if s is open then instead of checking ψ the automaton checks $\hat{\psi}$.

Secondly, if x_ψ is in s then

- ψ' is in s, and
- either x_ψ or $x > I$ is in s'.

The first condition verifies the proof obligation that is represented by the clock x_ψ, and the second condition keeps it active as long as necessary.

Type-5 formulas

Consider a subformula $\psi = \psi_1 \mathcal{U} \psi_2$ of type 5. Whenever ψ is in s, then either

- s is singular and $\psi \in s'$, or

- s is open and ψ_1 is in s, and either ψ_2 is in s or ψ_2 is in s' or both ψ_1 and ψ are in s'.

These conditions ensure that unconstrained *until* formulas are propagated correctly (remember that singular and open intervals alternate). However, these conditions admit the possibility that a run consists of states containing ψ and ψ_1 without ever visiting a ψ_2-state. Additional fairness constraints are needed to ensure that whenever a run r contains a state s with the type-5 formula ψ then some later state s' along the run contains ψ_2. For each type-5 formula $\psi = \psi_1 \mathcal{U} \psi_2$ in the closure set, we define,

$$\mathrm{F}_\psi = \{s \subseteq \mathit{Closure}(\phi) \mid \psi_2 \in s \text{ or } \psi \notin s\}.$$

It is straightforward to show that for the runs r that are fair with respect to F_ψ, if a state s at time t contains the formula ψ then $\rho_r^t \models \psi$.

Type-6 formulas

Consider a subformula $\psi = \Box\, \psi'$ of type 6. Whenever ψ is in s, then either

- s is singular and $\psi \in s'$, or
- s is open and $\psi' \in s$ and both ψ' and ψ are in s'.

These conditions guarantee that unconstrained *always* formulas are propagated forever.

This concludes the definition of the interval automaton $\mathcal{M}_\phi$. The runs of $\mathcal{M}_\phi$ are defined as before. The fairness requirements on the timed state sequences are given by the family $\mathcal{F}_\phi$ consisting of the sets F_ψ for every type-5 formula ψ in the closure set.

The following main lemma states the correctness of our construction by relating the accepting runs of $\mathcal{M}_\phi$ to the models of ϕ.

Lemma 4.34 The set $L(\phi)$ of the models of an MITL-formula ϕ equals the set $L(\mathcal{M}_\phi)$ of timed state sequences generated by $\mathcal{M}_\phi$.

PROOF. It can be shown, by induction on the automaton of ϕ, that given a $\mathcal{F}_\phi$-fair run r of $\mathcal{M}_\phi$, if a formula ψ in $\mathit{Closure}(\phi)$ is in a state s in r at time $t \in \mathsf{R}$, then $\rho_r^t \models \psi$. We have outlined the crucial arguments for the six interesting cases of temporal subformulas above.

Conversely, given a ϕ-fine model ρ of ϕ with alternating singular and open intervals, we can construct a $\mathcal{F}_\phi$-fair run r of $\mathcal{M}_\phi$ such that $\rho = \rho_r$. The Lemmas 4.31 and 4.32 instruct us how to use the available limited number of clocks to mark witnessing intervals. ■

This result yields algorithms for checking the satisfiability and validity of the given MITL-formula ϕ. To check satisfiability, we first construct the interval automaton $\mathcal{M}_\phi$, and then we use the algorithm that checks whether $L(\mathcal{M}_\phi)$ is nonempty to test if ϕ has a model. Similarly, ϕ is valid iff $L(\mathcal{M}_{\neg\phi})$ is empty.

4.4.7 Complexity of MITL

We conclude this section by showing that our decision procedure for MITL is in EXPSPACE, and that this is optimal, because the decision problem for MITL is EXPSPACE-complete.

First we show the lower bound of EXPSPACE-hardness by encoding computations of exponential-space bounded Turing machines. The proof is similar to the proof of the EXPSPACE-hardness of the real-time logic MTL [AH90].

Lemma 4.35 The satisfiability problem for MITL is EXPSPACE-hard.

PROOF. Consider a (deterministic) 2^n-space-bounded Turing machine M; for each input X of length n, we construct an MITL-formula ϕ_X of length $O(n \cdot \log n)$ that is satisfiable iff M accepts X. The EXPSPACE lower bound follows by a standard complexity-theoretic argument.

Thus it suffices to show, given X, how to construct a sufficiently succinct formula ϕ_X that describes the (unique) computation of M on X, as an infinite sequence of propositions, and requires it to be accepting.

First we require that propositions change their values only at the integer time points. This requirement is imposed by introducing a special proposition *clock*, and adding the conjunct

$$\phi_{clock} = clock \wedge \Box_{\geq 0}[\, clock \rightarrow \Box_{(0,1)}(\neg clock) \wedge \Diamond_{(0,1]}\, clock\,].$$

In every timed state sequence satisfying ϕ_{clock} the proposition *clock* is true during the interval $[j, j]$ and false during the interval $(j, j+1)$ for all $j \in \mathbb{N}$.

For every proposition p to be used in the formula, we add the requirement

$$\Box_{\geq 0}\,[\,(p \rightarrow p\,\mathcal{U}\, clock) \wedge ((\neg p) \rightarrow (\neg p)\,\mathcal{U}\, clock)\,].$$

This conjunct ensures that p can change its value only at the integer time values. With these restrictions we assume that every state stays unchanged for precisely one time unit.

We use a proposition p_i and a proposition q_j for every tape symbol i and state j of M, respectively. In particular, p_0 and q_0 correspond to the special tape symbol "blank" and

the initial state of M. Let

$$\begin{aligned}
\hat{p}_i &= p_i \wedge \bigwedge_{i' \neq i} \neg p_{i'} \wedge \bigwedge \neg q_j, \\
r_{i,j} &= p_i \wedge q_j \wedge \bigwedge_{i' \neq i} \neg p_{i'} \wedge \bigwedge_{j' \neq j} \neg q_{j'}, \\
s &= \bigwedge \neg p_{i'} \wedge \bigwedge \neg q_j .
\end{aligned}$$

We represent configurations of M by $\hat{p}$-state sequences of length 2^n, which are separated by s-states; the position of the read-write head is marked by an r-state. The computation of M on X is completely determined by the following two conditions:

(i) it starts with the initial configuration, and

(ii) every configuration follows from the previous one by a move of M.

The computation is accepting iff, furthermore,

(iii) it contains the accepting state F.

These conditions can be expressed in MITL. The initialization constraints are expressed by the formula

$$\phi_{init} = [\ s \wedge \Box_{[1,2)}\, r_{X_1,0} \wedge \bigwedge_{2 \leq i \leq n} \Box_{[i,i+1)}\, \hat{p}_{X_i} \wedge \Box_{[n+1,2^n+1)}\, \hat{p}_0\ ].$$

The first conjunct says that the first state is an s-state, the second conjunct says that the head is at the beginning of the tape reading X_1, the next conjunct sets up the input symbols in positions 2 through n, and the last conjunct says that the rest of the tape is blank.

The consecution requirement is expressed by the formula

$$\phi_{move} = \left(\begin{array}{l} \Box(\mathit{clock} \wedge s \rightarrow \Box_{[2^n+1,2^n+2)}\, s\,) \wedge \\ \bigwedge_{P,Q,R} \Box \left(\begin{array}{l} \mathit{clock} \wedge P \wedge \Box_{[1,2)}\, Q \wedge \Box_{[2,3)}\, R \rightarrow \\ \Box_{[2^n+2,2^n+3)}\, f_M(P,Q,R) \end{array} \right) \end{array} \right)$$

The first conjunct requires s-states separated by length $2^n + 1$, and the second conjunct relates the successive configurations. Note that the k-th state in the new configuration can be determined from the the $(k-1)$-th, k-th, and $(k+1)$-th states of the previous configuration. P, Q, and R represent three consecutive tape symbols, and each range over the propositions $\hat{p}_i$, $r_{i,j}$, and s. $f_M(P,Q,R)$ refers to the transition function of M. For

instance, if M writes, in state j on input i', the symbol k onto the tape, moves to the right, and enters state j', then $f_M(\hat{p}_i, r_{i',j}, \hat{p}_{i''}) = \hat{p}_k$ and $f_M(r_{i',j}, \hat{p}_{i''}, \hat{p}_{i'''}) = r_{i'',j'}$

The acceptance requirement corresponds to eventually visiting the state F; it is given by the formula $\phi_{accept} = \Diamond q_F$.

We take ϕ_X to consist of these three additional conjuncts ϕ_{init}, ϕ_{move}, and ϕ_{accept}. The lengths of ϕ_{init}, ϕ_{move}, and ϕ_{accept} are $O(n \cdot log\, n)$, $O(n)$, and $O(1)$, respectively (recall that constants are represented in binary), thus implying the desired $O(n \cdot log\, n)$-bound for ϕ_X. ■

Now we show that the time complexity of our algorithm is doubly exponential in the length of the constants in ϕ, and singly exponential in the number of logical and temporal operators in ϕ. Furthermore, the algorithm also implies an upper bound of EXPSPACE for deciding MITL.

Theorem 4.36 The decision problem for testing satisfiability of MITL-formulas is complete for EXPSPACE. In particular, the proposed algorithm checks satisfiability of an MITL-formula ϕ in time $O(2^{|\phi| \cdot K \cdot log K})$, where $K - 1$ is the largest constant appearing in ϕ.

PROOF. Given a formula ϕ the first step of the algorithm rewrites it to a normal form ϕ^*. By Lemma 4.27, the number of subformulas of ϕ^* is $O(|\phi|)$. Let $K - 1$ be the largest constant appearing in ϕ, then the size of the closure set $Closure(\phi^*)$ is $O(|\phi| \cdot K)$. Hence the number of states in $\mathcal{M}_\phi$ is $O(2^{|\phi| \cdot K})$. The number of clocks in $\mathcal{M}_\phi$ is $O(|\phi| \cdot K)$. Furthermore, for every clock x the largest constant that x is compared with in a clock constraint of $\mathcal{M}_\phi$ is bounded by K. Recall that the complexity of the emptiness test for an interval automaton is exponential in its number of clocks, and is proportional to the size of its state-transition graph and product of the timing constants for all the clocks (see Section 4.3). Consequently, the algorithm testing emptiness of $\mathcal{M}_\phi$ runs in time $O(K^{|\phi| \cdot K})$.

We have already proved EXPSPACE-hardness. For containment in EXPSPACE note that the description of $\mathcal{M}_\phi$ can be given in space polynomial in $|\phi| \cdot K$; that is, in space exponential in the length of ϕ, assuming binary encoding of all interval end-points. The emptiness problem for an interval automaton $\mathcal{M}$ is PSPACE complete (see Theorem 4.23). It follows that the validity of ϕ can be decided in space polynomial in $|\phi| \cdot K$, that is, in EXPSPACE. ■

4.5 Verification using MITL

Model checking is a powerful and well-established technique for automatic verification of finite-state systems; it compares a temporal-logic specification of a system against a state-transition description of the system.

In the qualitative case, the system is modeled by its state-transition graph, also known as Kripke structure, and the specification may be presented as a formula of the propositional linear temporal logic PTL [LP85]. Using our results about MITL, we can present a real-time verification procedure that checks MITL-specifications against descriptions given as interval automata.

We model a real-time system by an interval automaton $\mathcal{M}$ and give the specification as a formula ϕ of MITL. Hence the *model checking* problem is to decide whether or not all the timed state sequences generated by the structure $\mathcal{M}$ satisfy the specification ϕ:

$$L(\mathcal{M}) \stackrel{?}{\subseteq} L(\phi)$$

Notice the similarity between this approach to verification and the one in Chapter3. Apart from changing the specification language from automata to logic, we have shifted from the event-based semantics of timed traces to the state-based semantics of timed state sequences.

Our construction for testing the satisfiability of MITL-formulas can be used to develop an algorithm for model checking. The first step is to construct an interval automaton $\mathcal{M}_{\neg\phi}$ such that its runs precisely capture the models of the negated formula $\neg\phi$: $L(\mathcal{M}_{\neg\phi}) = L(\neg\phi)$. The model checking question can then be reformulated as follows:

$$L(\mathcal{M}) \subseteq L(\phi) \text{ iff } L(\mathcal{M}) \cap L(\mathcal{M}_{\neg\phi}) = \emptyset.$$

The next step in the model checking algorithm is to construct an interval automaton $\mathcal{M}'$ that is the product of $\mathcal{M}$ and $\mathcal{M}_{\neg\phi}$ (see Section 4.3); a timed state sequence is generated by $\mathcal{M}'$ iff it is generated by both $\mathcal{M}$ and $\mathcal{M}_{\neg\phi}$.

Hence we have reduced the model checking problem to the emptiness question for interval automata: $\mathcal{M} \models \phi$ iff $L(\mathcal{M}')$ is empty. The size of $\mathcal{M}'$ is polynomial in the sizes of $\mathcal{M}$ and $\mathcal{M}_{\neg\phi}$. Consequently, the description of $\mathcal{M}'$ is exponential in the length of ϕ, and polynomial in the length of the description of $\mathcal{M}$. Since the emptiness for interval automata can be solved in PSPACE, it follows that the model checking problem can be solved in EXPSPACE.

As for all linear temporal logics, the model checking question for MITL is no simpler than the satisfiability question; it is also EXPSPACE-hard. The following theorem follows:

Theorem 4.37 The problem of checking whether an interval automaton $\mathcal{M}$ satisfies an MITL-formula ϕ is EXPSPACE-complete.

PROOF. We have already outlined how to solve the model-checking problem in EXPSPACE.

To prove EXPSPACE-hardness we reduce the satisfiability question for MITL to model-checking question. An MITL-formula ϕ is unsatisfiable iff the universal interval automaton, which generates all possible timed state sequences, satisfies $\neg\phi$. Thus EXPSPACE-hardness of satisfiability implies EXPSPACE-hardness of model checking. ∎

The time complexity of the model checking algorithm is polynomial in the qualitative part of the system description, exponential in the qualitative part of the MITL-specification, exponential in the timing part of the system description, and doubly exponential in the timing part of the specification. Compared to this the model checking algorithm for PTL [LP85] is polynomial in the size of the Kripke structure and exponential in the size of the specification.

Thus moving to *real-time* gives an additional *exponential* blow-up. This blow-up seems, however, unavoidable for formalisms for quantitative reasoning about time. It occurs even in the discrete-time case [EMSS89, AH89, AH90].

4.6 Expressiveness

Every formula ϕ of MITL specifies a real-time property — the set of models of ϕ. The expressive power of a formalism is measured by the real-time properties that can be specified in it. First we compare the expressive power of MITL to that of MTL, a real-time logic based on the fictitious-clock model. Then we compare the expressive power of MITL to that of interval automata.

4.6.1 Comparison with fictitious-clock logics

We compare the expressive power of MITL to the use of a fictitious clock and MTL, which admits singular intervals as time bounds on temporal operators. More precisely, we show that the dense-time model without equality (MITL) is more expressive than any fictitious-clock model with equality (MTL).

The logic MTL

First let us define the syntax and semantics of the logic MTL, *metric temporal logic*. The syntax of MTL is the same as that of MITL, however, it also admits singular intervals as subscripts.

As before let AP be a set of atomic propositions. The formulas of MTL are built from propositions by Boolean connectives and time-bounded versions of the *until* operator $\mathcal{U}$.

Definition 4.38 The formulas of MTL are inductively defined as follows:

$$\phi := p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \, \mathcal{U}_I \, \phi_2$$

where $p \in \mathrm{AP}$, and I is an interval with integer end-points. ■

The semantics of MITL uses the fictitious-clock model of Section 2.2.4. Thus time is viewed as a discrete counter, which is updated with every tick of the global clock. The formulas of MTL are interpreted over observation sequences.

Definition 4.39 An *observation sequence* is a pair $\varrho = (\sigma, \overline{T})$, where $\sigma = \sigma_0\sigma_1\ldots$ is an infinite sequence of states $\sigma_i \in 2^{\mathrm{AP}}$, and $T = T_0T_1\ldots$ is an infinite sequence of corresponding time stamps $T_i \in \mathsf{N}$ satisfying

- the *initiality* condition that $T_0 = 0$,
- the *weak monotonicity* condition that $T_i \leq T_{i+1}$ for all $i \geq 0$, and
- the *progress* condition that, for all $n \in \mathsf{N}$, there is some $i \geq 0$ such that $T_i > n$.

■

We will denote the observation sequence $\varrho = (\sigma, \overline{T})$ by an infinite sequence

$$(\sigma_0, T_0) \rightarrow (\sigma_1, T_1) \rightarrow (\sigma_2, T_2) \rightarrow (\sigma_3, T_3) \rightarrow \cdots$$

of observations. Each *observation* consists of a state $\sigma_i \in 2^{\mathrm{AP}}$ and a time stamp $T_i \in \mathsf{N}$. For an observation sequence ϱ and $i \in \mathsf{N}$, the observation sequence ϱ^i is the suffix of ϱ that begins with the observation (s_i, T_i).

For an observation sequence ϱ and an MTL-formula ϕ, the satisfaction relation $\varrho \models \phi$ is defined as usual by induction on the structure of ϕ.

Definition 4.40 For an MTL-formula ϕ, and an observation sequence $\varrho = (\sigma, \overline{T})$, the satisfaction relation $\rho \models \phi$ is defined inductively as follows:

$\varrho \models p$ iff $p \in \sigma_0$.
$\varrho \models \neg\phi$ iff $\varrho \not\models \phi$.
$\varrho \models \phi_1 \wedge \phi_2$ iff $\varrho \models \phi_1$ and $\varrho \models \phi_2$.
$\varrho \models \phi_1 \mathcal{U}_I \phi_2$ iff $\varrho^i \models \phi_2$ for some $i \geq 0$ with $T_i \in I$, and $\varrho^j \models \phi_1$ for all $0 < j < i$.

An MTL-formula is called satisfiable iff $\varrho \models \phi$ for some observation sequence ϱ. ■

Note that the *until* operator is *strict* in its left argument. Also MTL has no *next-state* operator; this restriction makes MTL-formulas insensitive to stuttering.

The tableau-based decision procedure for PTL can be extended to get a decision procedure for checking satisfiability of MTL-formulas. In particular, given an MTL-formula ϕ we can construct a Kripke structure $\mathcal{M}_\phi$ with fairness constraints that characterizes the satisfying models of ϕ. Details of the decision procedure, and comparison of the expressiveness of MTL with other fictitious-clock logics can be found in [AH90]. Here we only state the result on complexity of MTL.

Theorem 4.41 The decision problem to test satisfiability of a given MTL-formula ϕ is EXPSPACE-complete. Specifically, there is an algorithm that checks satisfiability of a given MTL-formula ϕ in time exponential in $O(|\phi| \cdot K)$, where $K - 1$ is the largest constant appearing in ϕ. ■

Defining real-time properties using MTL

We need to formalize which real-time properties can be specified in MTL. To this end, let us consider how to extract an observation sequence from a timed state sequence ρ that describes the actual behavior of a real-time system. Observations are made with respect to a digital clock; the observation at time t records the state $\rho^*(t)$ and the reading of the clock at time t (recall that ρ^* is a map from R to states induced by ρ). Clearly the observations depend on how fast the clock ticks, and at what time the clock is started. This motivates the next definition.

Definition 4.42 A *digital clock* $D = (\kappa, \varepsilon)$ is a pair consisting of the distance $\kappa \in \mathsf{R}$ between two successive clock ticks and the time $\varepsilon \in \mathsf{R}$ of the first clock tick; that is, $0 \leq \varepsilon < \kappa$. At time $t \in \mathsf{R}$ the clock D shows the integer value $t_D = \lceil (t - \varepsilon)/\kappa \rceil$.

The clock D is called *rational* iff both κ and ε are rational numbers. ■

The clock D starts at time 0 with the reading $0_D = 0$. The reading of the clock during the interval $[0, \varepsilon]$ is 0, and during the interval $(\varepsilon + n\kappa, \varepsilon + (n+1)\kappa]$ its reading is $(n + 1)$ for all $n \in \mathsf{N}$.

Now we can define the observed behavior of a timed state sequence ρ with respect to a given clock D. The observation of ρ at time t consists of the state $\rho^*(t)$ and the clock reading t_D. As time increases, the D-observation stays the same until either the clock tick changes the value of t_D, or the state changes along ρ. Consequently, all possible D-observations along ρ can be described by an ω-sequence.

Definition 4.43 For a clock D, and a timed state sequence $\rho = (\sigma, \overline{I})$, the D-observation of ρ at time t is $O_t = (\rho^*(t), t_D)$.

The D-observed behavior of ρ is the observation sequence

$$\rho_D : \quad O_{t_0} \rightarrow O_{t_1} \rightarrow O_{t_2} \rightarrow \cdots,$$

for time values $t_i \in \mathsf{R}$ such that for all $i \geq 0$, (1) $t_i < t_{i+1}$, and (2) for all $t \in (t_i, t_{i+1})$, O_t equals either O_{t_i} or $O_{t_{i+1}}$. ■

Note that above properties define ρ_D uniquely modulo stuttering (i.e., duplication of neighboring observations). Furthermore, the state component of ρ_D is the state component of ρ (modulo stuttering).

Example 4.44 Consider the timed state sequence ρ:

$$(s_0, [0, 1)) \rightarrow (s_1, [1, 1]) \rightarrow (s_2, (1, 1.5]) \rightarrow (s_3, (1.5, \infty)).$$

Then the digital clock $(1, 0.5)$ observes the observation sequence $\rho_{(1,0.5)}$:

$$(s_0, 0) \rightarrow (s_0, 1) \rightarrow (s_1, 1) \rightarrow (s_2, 1) \rightarrow (s_3, 2) \rightarrow (s_3, 3) \rightarrow (s_3, 4) \rightarrow \cdots$$

■

Using the above notion of observations we can associate real-time properties with every MTL-formula for every choice of the digital clock.

Definition 4.45 For a given digital clock D, a formula ϕ of MTL specifies a real-time property $L_D(\phi)$ — the set of timed state sequences ρ such that $\rho_D \models \phi$. ■

Comparing MITL with MTL

Now we can be specific about the sense in which the dense-time model is, even without equality, more expressive than the fictitious-clock model, for any choice of digital clock.

Theorem 4.46 (*a*) For every MTL-formula ϕ and every rational clock D, the real-time property $L_D(\phi)$ equals $L(\psi)$ for some MITL-formula ψ (using intervals with rational endpoints). (*b*) There is an MITL-formula ϕ such that the real-time property $L(\phi)$ does not equal $L_D(\psi)$ for all MTL-formulas ψ and all choices of digital clock D.

PROOF. (*a*) Given a rational clock $D = (\kappa, \varepsilon)$ and a formula ϕ of MTL, we construct an MITL-formula that specifies the real-time property $L_D(\phi)$. We assume that ϕ contains only intervals of the form $[0,0]$, $[1,1]$, $[m,n]$ for $2 \leq m \leq n$, and $[m,\infty)$ for $m \geq 2$. It is trivial to convert any MTL-formula into this form; for instance, the MTL-formula $\Diamond_{<5}\,\psi$ is equivalent to the formula $\Diamond_{=0}\,\psi \vee \Diamond_{=1}\,\psi \vee \Diamond_{[2,4]}\,\psi$.

We model the ticks of the digital clock D by a new proposition r that holds only in transient states at time instants $(\varepsilon + n\kappa)$:

$$\phi_D: \quad \Box_{<\varepsilon}\,\neg r \,\wedge\, \Diamond_{\leq\varepsilon}\,r \,\wedge\, \Box_{\geq 0}\,(r \,\rightarrow\, (\neg r)\,\mathcal{U}_{=\kappa}\,r).$$

Let ϕ^* be the MITL-formula that results from ϕ by replacing every occurrence of a subformula $\psi_1\,\mathcal{U}_I\,\psi_2$ with

$$\neg r \,\wedge\, (\psi_1 \,\wedge\, \neg r)\,\mathcal{U}_{\geq 0}\,\psi_2$$

if I is $[0,0]$; with

$$(r \,\wedge\, (\psi_1 \,\wedge\, \neg r)\,\mathcal{U}_{\geq 0}\,\psi_2) \,\vee\, \psi_1\,\mathcal{U}_{(0,\kappa)}\,(r \,\wedge\, \psi_1 \,\wedge\, \psi_1\,\mathcal{U}_{(0,\kappa]}\,\psi_2)$$

if I is $[1,1]$; with

$$\psi_1\,\mathcal{U}_{[(l(I)-1)\kappa, r(I)\kappa)}\,(r \,\wedge\, \psi_1 \,\wedge\, \psi_1\,\mathcal{U}_{(0,\kappa]}\,\psi_2)$$

if I is bounded and $l(I) > 1$; and with

$$\psi_1\,\mathcal{U}_{\geq(l(I)-1)\kappa}\,(r \,\wedge\, \psi_1 \,\wedge\, \psi_1\,\mathcal{U}\,\psi_2)$$

if I is unbounded and $l(I) > 1$. It is not hard to show that $\rho_D \models \phi$ iff $\rho \models (\phi_D \wedge \phi^*)$ for every timed state sequence ρ.

For example, consider the MTL-formula

$$\Box_{\geq 0}\,(p \,\rightarrow\, \Diamond_{=5}\,q),$$

and the digital clock $D = (1,0)$. This formula specifies the property that "for every p-state there is a q-state separated from p by exactly five integer times," and is equivalent to the MITL-formula

$$\phi_{(1,0)} \wedge \Box_{\geq 0}(p \rightarrow \Diamond_{[4,5)}(r \wedge \Diamond_{(0,1]}q)).$$

(b) From the tableau decision procedure for MTL [AH90], it follows that if a formula ϕ of MTL is satisfiable, then it has a model ρ_D such that any two state changes in ρ are separated by at least some minimum time gap (which depends on D and the size of ϕ). In fact, for any digital clock D one can always construct timed state sequences in $L_D(\phi)$ that become periodic after some point in time. We show that this is not the case for MITL.

Let us construct a satisfiable MITL-formula ϕ for which every model $\rho = (\sigma, \overline{I})$ contains arbitrarily close state changes; that is, for every real $\epsilon > 0$, there is some $i \geq 1$ such that $\sigma_{i-1} \neq \sigma_i$ and $\sigma_i \neq \sigma_{i+1}$ and $t_{i+1} - t_i < \epsilon$. The set of models of ϕ can clearly not be specified in MTL, for any choice of digital clock D.

The formula ϕ uses three propositions p, q, and r. First, it requires at most one of these three propositions to be true at any state. In addition, it has the following three conjuncts. The first condition,

$$r \wedge \Box_{\geq 0}(r \rightarrow (\neg r)\mathcal{U}_{=2}\, r),$$

places transient r-states at precisely the even integers. The second condition,

$$\Box_{\geq 0}((p \vee q) \rightarrow \Diamond_{<1}\, r),$$

ensures that p and q can only hold in the second half of the intervals of length 2 separating consecutive r-states. The third condition,

$$\Diamond_{<2}\, p \wedge \Box_{\geq 0}(p \rightarrow \Diamond_{<1}\, q) \wedge \Box_{\geq 0}(q \rightarrow \Diamond_{(2,3)}\, p),$$

implies that there is a p-state, and later a q-state, between every pair of consecutive r-states, and thus between every odd integer and the subsequent even integer.

Moreover, from any model of ϕ we can extract an infinite sequence of alternating p and q states, with the q-state following a p-state guaranteed by the condition $p \rightarrow \Diamond_{<1}\, q$, and the p-state following a q-state by the condition $q \rightarrow \Diamond_{(2,3)} p$. The times that are associated with the states in this sequence, taken modulo 2, form a strictly increasing infinite sequence of reals contained in the interval $(1,2)$. Since this time sequence is bounded above, there must be arbitrarily close pairs of a p-state followed by a q-state. It follows that ϕ has no eventually periodic models.

On the other hand, the MITL-formula ϕ is satisfiable; a model for ϕ can be readily constructed by introducing, in addition to the transient r-states at all even integers, transient p-states at time $2n - 2/4^n$, and transient q-states at time $2n - 1/4^n$, for each integer $n \geq 1$. ■

Thus even though MITL disallows singular interval subscripts, it is more expressive than the fictitious-clock logics allowing the use of equality irrespective of the rate of the clock.

Recall that MITL with equality is undecidable (Theorem 4.17). We have shown two ways of achieving decidability; MITL puts a syntactic restriction disallowing subscripts such as $[2, 2]$; on the other hand, MTL weakens the expressiveness through a semantic abstraction that, at every state change only a discrete approximation to the real-time may be recorded.

4.6.2 Comparison with interval automata

An interval automaton M generates the set of timed state sequences $L(\mathcal{M})$. Thus it provides an alternative to MITL for specifying real-time properties. Now we compare the two formalisms with respect to expressiveness.

The decision procedure for MITL constructs, for a given formula ϕ, an interval automaton $\mathcal{M}_\phi$ with fairness conditions such that $L(\phi) = L(\mathcal{M}_\phi)$. Thus interval automata with fairness conditions are at least as expressive as MITL. A close inspection of the construction shows that fairness conditions are needed to handle unconstrained eventualities only. In particular, for an MITL-formula ϕ, if its equivalent normal form ϕ^* does not contain any type-5 subformulas $\psi_1 \mathcal{U} \psi_2$, then there exists an interval automaton $\mathcal{M}_\phi$ generating $L(\phi)$. Thus no fairness conditions are needed to handle *until* operators subscripted with bounded intervals.

However, the formalism of interval automata has strictly greater expressive power compared to MITL. We consider some examples to illustrate this point.

Example 4.47 Assume AP $= \{p, q\}$. Consider the following property of timed state sequences:

$$L = \{\rho \mid \exists t \in \mathsf{R}. (p \in \rho^*(t) \;\wedge\; q \in \rho^*(t+1))\}.$$

The property L asserts there is a p-state at some time instant t followed by a q-state at time $t + 1$. This property can easily be expressed as an interval automaton with fairness conditions. It cannot be specified using MITL. Note that it can be expressed in the extension

of MITL that allows singular intervals as subscripts:

$$\Diamond_{\geq 0}(p \wedge \Diamond_{=1} q).$$

■

Example 4.47 shows that the interval automata can express some forms of equality constraints not expressible by MITL. In addition, there are certain "qualitative" properties that can be expressed using Kripke structures, but not by linear temporal logics. This also makes MITL less expressive than interval automata.

Example 4.48 Assume AP $= \{p, q\}$. Furthermore, assume that both propositions are true only during singular intervals. Such a requirement is expressible using both formalisms. Now a timed state state sequence ρ is specified by a state sequence $\sigma = \sigma_0\sigma_1\ldots$ and an infinite sequence $\overline{I} = t_0t_1\ldots$ of time values. Consider the property that "the proposition p holds in every state σ_i with even i". This is a property of state sequences, and does not restrict the possible choices for $\overline{I}$. It is known that PTL cannot specify this property [Wol83]. It follows that MITL cannot specify this property either, even if we allow the use of singular intervals. It is straightforward to specify the property using interval automata. No fairness constraints are required. ■

The next theorem follows.

Theorem 4.49 (a) For every MITL-formula ϕ there exists an interval automaton $\mathcal{M}$ with fairness constraints such that $\mathcal{M}$ generates $L(\phi)$. (b) For every MITL-formula ϕ such that its normal form ϕ^* has no unconstrained *until* operators, there exists an interval automaton $\mathcal{M}$ that generates $L(\phi)$. (c) There exists an interval automaton $\mathcal{M}$ such that $L(\mathcal{M})$ is not specifiable by any MITL-formula, even if we allow singular intervals and intervals with rational end-points. ■

Note that the real-time properties definable by MITL are closed under complement, since the logic has the negation operator. On the other hand, interval automata are not closed under complement. For instance, the complement of the property considered in Example 4.47 is not specifiable using interval automata. In general, the negative results for timed automata in Section 3.4 can be shown for timed structures also. In particular, the language inclusion problem for interval automata is undecidable.

Chapter 5

Branching-Time Logic

This chapter is based on branching-time semantics of reactive systems. Until now, we have been using linear-time semantics where a system is modeled by a set of linear sequences. In contrast, in the branching-time framework, the system is viewed as a single tree, and the paths in the tree correspond to the possible executions [Mil80, BMP81, EC82]. In general, the branching-time semantics is finer than the linear-time semantics; that is, there are examples where the branching-time semantics distinguishes two systems which the linear-time semantics considers to be the same (see, for instance, [Mil80]). Amongst the researchers there is no consensus on the choice between the two approaches. Having presented verification methods based on the linear-time approach, now we proceed to show how real-time can be added to the branching-time framework. We define a real-time extension of the branching-time logic CTL, and develop an algorithm for checking whether a finite-state system modeled by an interval automaton satisfies its specification in this language.

5.1 Computation tree logic

Computation tree logic (CTL) was introduced by Emerson and Clarke [EC82] as a specification language for finite-state systems. Let us briefly review its syntax and semantics.

Similarly to the linear-time temporal logic PTL, the logic CTL has atomic propositions, logical connectives, and temporal operators. However, as CTL-formulas are interpreted over finitely-branching trees, each temporal operator of CTL has two types: "existential" and "universal." For instance, $\exists\Box$ means "in all states along some path," and $\forall\Box$ means "in all states along all paths." Let AP be a set of atomic propositions. The formulas of CTL are

inductively defined as follows:

$$\phi := p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists \bigcirc \phi_1 \mid \exists\, \phi_1 \mathcal{U}\, \phi_2 \mid \forall\, \phi_1 \mathcal{U}\, \phi_2$$

where $p \in \mathrm{AP}$, and ϕ_1, ϕ_2 are CTL-formulas.

Intuitively $\exists \bigcirc \phi$ means that there is an immediate successor state, reachable by executing one step, in which ϕ holds. $\exists\, \phi_1 \mathcal{U}\, \phi_2$ means that for some computation path, there exists an initial prefix of the path such that ϕ_2 holds at the last state of the prefix and ϕ_1 holds at all the intermediate states. $\forall\, \phi_1 \mathcal{U}\, \phi_2$ means that for every computation path the above property holds.

Formally, the semantics of CTL is defined with respect to a Kripke structure $\mathcal{M} = \langle \mathrm{S}, s_{init}, \mu, \mathrm{E} \rangle$, where S is a countable set of states, $s_{init} \in \mathrm{S}$ is an initial state, $\mu : \mathrm{S} \rightarrow 2^{\mathrm{AP}}$ gives an assignment of truth values to propositions in each state, and E is a binary relation over S giving the possible transitions. A *path* is an infinite sequence of states $(s_0, s_1 \ldots) \in \mathrm{S}^\omega$ such that $\langle s_i, s_{i+1} \rangle \in \mathrm{E}$ for all $i \geq 0$. Given a CTL-formula ϕ and a state $s \in \mathrm{S}$, the satisfaction relation $(\mathcal{M}, s) \models \phi$ (meaning ϕ is true in $\mathcal{M}$ at s) is defined inductively as follows (since the structure is fixed, we abbreviate $(\mathcal{M}, s) \models \phi$ to $s \models \phi$):

$s \models p$ iff $p \in \mu(s)$.

$s \models \neg\phi$ iff $s \not\models \phi$.

$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$.

$s \models \exists \bigcirc \phi$ iff $s' \models \phi$, for some state s' such that $\langle s, s' \rangle \in \mathrm{E}$.

$s \models \exists\, \phi_1 \mathcal{U}\, \phi_2$ iff for some path $(s_0, s_1, \ldots)$ with $s = s_0$, for some $i \geq 0$, $s_i \models \phi_2$ and $s_j \models \phi_1$ for $0 \leq j < i$.

$s \models \forall\, \phi_1 \mathcal{U}\, \phi_2$ iff for every path $(s_0, s_1, \ldots)$ with $s = s_0$, for some $i \geq 0$, $s_i \models \phi_2$ and $s_j \models \phi_1$ for $0 \leq j < i$.

The Kripke structure $\mathcal{M}$ satisfies ϕ iff $(\mathcal{M}, s_{init}) \models \phi$.

Some of the commonly used abbreviations are: $\exists\Diamond\, \phi$ for $\exists\, \mathbf{true}\, \mathcal{U}\, \phi$, $\forall\Diamond\, \phi$ for $\forall\, \mathbf{true}\, \mathcal{U}\, \phi$, $\exists\Box\, \phi$ for $\neg\forall\Diamond\neg\phi$, and $\forall\Box\, \phi$ for $\neg\exists\Diamond\neg\phi$.

CTL can express several interesting properties of reactive systems. The formula $\forall\Box p$ says that the property p is an invariant of the system. The formula $\forall\Diamond p$ says that p is inevitable; every possible computation path will have some p-state. The formula $\forall\Box\, \exists\Diamond\, p$ says that in every reachable state p potentially holds; that is, it is always possible to get to some p-state. Incidentally, this property cannot be expressed using a linear-time logic such

as PTL; that means there is no PTL-formula ϕ such that a Kripke structure $\mathcal{M}$ satisfies ϕ iff $\mathcal{M}$ satisfies the CTL-formula $\forall\Box\,\exists\Diamond\, p$. We remind the reader that a Kripke structure gives a set of models for a PTL-formula ϕ, and $\mathcal{M} \models \phi$ iff all the models generated by $\mathcal{M}$ satisfy ϕ. Properties such as fairness are expressible in PTL, but not in CTL. Consider the PTL-formula

$$\phi_{fair} \;=\; \Box\Diamond\, p \;\rightarrow\; \Diamond\, q.$$

The formula holds in a Kripke structure $\mathcal{M}$ iff all paths through $\mathcal{M}$ with infinitely many p-states also have some q-state. There is no CTL-formula ϕ such that $\mathcal{M} \models \phi$ iff $\mathcal{M} \models \phi_{fair}$ [EH82].

A typical response property that "every p-state is always followed by a q-state," is specified by the following CTL-formula:

$$\forall\Box[\, p \;\rightarrow\; \forall\Diamond q\,].$$

A CTL-formula ϕ is called *satisfiable* iff there is a Kripke structure $\mathcal{M}$ such that $\mathcal{M} \models \phi$. CTL has the *finite-model property*: if a CTL-formula is satisfiable, then it is satisfiable in a structure with a finite set of states (in fact, in a structure of size exponential in the size of the formula). Consequently, the satisfiability question for CTL is decidable. It has been shown that the decision problem is complete for deterministic singly-exponential-time (EXPTIME-complete). However, the model-checking problem is in PTIME — given a CTL-formula ϕ, and a finite Kripke structure $\mathcal{M} = \langle \mathrm{S}, s_{init}, \mu, \mathrm{E}\rangle$ there is an algorithm for deciding whether or not $\mathcal{M}$ satisfies ϕ with time complexity $O[|\phi|\cdot(|\mathrm{S}| + |\mathrm{E}|)]$ [CES86].

Finite-state concurrent systems can be modeled as finite Kripke structures. Given a system modeled as a Kripke structure and the specification as a CTL-formula, the model-checking algorithm can be used to decide whether or not the implementation satisfies the specification (see [CES86, BCDM86] for examples of CTL based verification).

Since CTL cannot express correctness along fair paths, Clarke et.al. have defined $\mathrm{CTL^F}$ by changing the semantics of CTL [CES86]. The syntax of $\mathrm{CTL^F}$ is same as that of CTL, however, the formulas are interpreted over Kripke structures with fairness constraints. A Kripke structure $\mathcal{M}$ now has an additional component $\mathcal{F} \subseteq 2^{\mathrm{S}}$. Recall that a path $(s_0, s_1, \ldots)$ is called *$\mathcal{F}$-fair* if for each $\mathrm{F} \in \mathcal{F}$, there are infinitely many i such that $s_i \in \mathrm{F}$. Given a $\mathrm{CTL^F}$-formula ϕ, we change the meaning of $(\mathcal{M}, s) \models \phi$ so that the path-quantifiers range only over $\mathcal{F}$-fair paths. The model-checking algorithm for CTL can be modified to handle this extension of CTL at a cost of a factor of $|\mathcal{F}|$.

5.2 The logic TCTL

To reason about quantitative time requirements, we add time explicitly to the syntax and semantics of CTL. As before, we choose the *dense-time* model, and define the logic *timed computation tree logic* or TCTL.

5.2.1 Syntax

Recall that one way to introduce real-time in the syntax of a temporal logic is to allow bounds on the temporal operators restricting their scope in time. The logic MITL uses such a notation of bounded temporal operators. The syntax of CTL can be extended in a similar way. For example, we can write $\exists\Diamond_{<5}\, p$ to say that along some computation path p becomes true within 5 time units. However, alternative notations have been proposed, and to define TCTL we use a more powerful notation.

In [AH89], we define a linear-time logic TPTL using a different syntax for writing real-time specifications. In this logic the bounded response property that "every p-state is followed by some q-state within 5 time units," is written as

$$\Box x.\,(p \rightarrow \Diamond y.\,(q \wedge y \leq x + 5)).$$

The *time quantifier* "$x.$" binds the associated variable x to the "current" time: $x.\,\phi(x)$ holds at time t iff $\phi(t)$ does. Read the above formula as "in every state with time x, if p holds then there is a later q-state with time y such that $(y \leq x + 5)$ holds". Thus in the formula $\Diamond y.\,\phi$, the time variable y is bound to the time of the state at which ϕ is "eventually" true.

We adopt a notation similar to TPTL to define the syntax of TCTL. The syntax uses the time quantifiers which allow references to the times of states, and admits the addition of timing constraints; that is, atomic formulas that relate the times of different states. As timing constraints, we permit comparisons of clock values, possibly with addition of constants. The formulas of TCTL are built from propositions and timing constraints by connectives, CTL temporal operators, and time quantifiers. As time constants we want to use the set of rational numbers, but for the sake of simplicity we define the syntax using only the integer constants. The changes necessary to handle rational constants should be obvious to the reader.

Let AP be a set of propositions, V be a set of variables, and N be the set of constants $\{0, 1, 2, \ldots\}$ denoting the natural numbers.

Definition 5.1 The formulas ϕ of TPTL are inductively defined as follows:

$$\phi := p \mid (x + c) \leq (y + d) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists\, \phi_1 \mathcal{U}\, \phi_2 \mid \forall\, \phi_1 \mathcal{U}\, \phi_2 \mid x.\, \phi$$

for $c, d \in \mathsf{N}$, $p \in \mathrm{AP}$, and $x, y \in V$. ■

5.2.2 Semantics

To define the semantics of TCTL we choose the set of nonnegative reals R to model time. Recall that an interpretation for MITL-formulas provides an assignment for atomic propositions at each time instant of R. Similarly, to define the semantics of TCTL we first generalize the notion of a computation path from an ω-sequence of states to a map from R to states.

Definition 5.2 Let S be a set of states and μ be a function from S to 2^{AP} labeling each state with atomic propositions. A *computation* over S is a map ρ from R to S satisfying the *finite-variability* condition:

> there exists an interval sequence $I_0 I_1 I_2 \ldots$ such that whenever two time values t and t' belong to the same interval I_i, $\mu(\rho(t))$ equals $\mu(\rho(t'))$.

■

The finite-variability condition says that the map $\mu \cdot \rho$ from R to 2^{AP} changes its value at atmost ω number of points. This rules out arbitrary variation of truth values of propositions along a computation. For instance, a map along which p is true at all rational time values and false at all irrational time values is not a legal computation. Notice that we have not required that the map ρ change its value only ω number of times. In fact, in the models for TCTL-formulas extracted from interval automata, a state will also encode the clock values, and thus the state set S will be an uncountable set, and ρ, by itself, will not meet the finite-variability constraint.

A structure for a branching-time logic should specify a set of labeled states and should associate a computation tree with each state. In discrete time, a structure can be described by associating a set of "next" states with each state. In dense time, there is always a third state between every pair of states on a path, so another characterization of trees is needed.

A dense tree can be considered to be a set of (dense) computations describing the paths in the tree. However, not every such collection of computations describes a reasonable tree, so some additional properties are needed. To give the definition in detail, let us define some notation.

Definition 5.3 Let S be a labeled set, ρ be a computation over S, and let $t \in \mathsf{R}$.

The *prefix* of ρ up to time t, denoted by ρ_t, is a map from $[0,t)$ to S obtained by restricting the domain of ρ. The *suffix* of ρ at time t, denoted by ρ^t, is a computation defined by $\rho^t(t') = \rho(t+t')$ for every $t' \in \mathsf{R}$.

If ρ' is some map from $[0,t)$ to S, then its concatenation with ρ, denoted by $\rho' \cdot \rho$, is defined by:

$$\text{for } t' \in \mathsf{R}:\ (\rho' \cdot \rho)(t') = \begin{cases} \rho'(t') & \text{if } t' < t \\ \rho(t'-t) & \text{otherwise} \end{cases}$$

■

Now we are ready define the notion of a TCTL-structure.

Definition 5.4 A TCTL-*structure* is a tuple $\mathcal{T} = \langle \mathrm{S}, s_{init}, \mu, f \rangle$, where

- S is a set of states.
- $s_{init} \in \mathrm{S}$ is an initial state.
- $\mu : \mathrm{S} \rightarrow 2^{\mathrm{AP}}$ is a labeling function which assigns to each state the set of atomic propositions true in that state.
- f is a collection of computations over S satisfying the following closure properties:
  1. *Suffix closure:* for all $\rho \in f$ and $t \in \mathsf{R}$, $\rho^t \in f$.
  2. *Fusion closure:* for all $\rho, \rho' \in f$ and $t \in \mathsf{R}$, if $\rho(t) = \rho'(0)$ then $\rho_t \cdot \rho' \in f$.

■

The second condition says that the behavior of the system does not depend on the past and only on the current state. If a state s appears along a computation at time t, the set of all possible computations can be obtained by concatenating the prefix up to time t with all the computations in f starting at the state s. This requirement ensures that the reachability relation over S induced by f is transitive.

We define below what it means for a TCTL-formula to be true in a state of a TCTL-structure.

Definition 5.5 For a TCTL-structure $\mathcal{T} = \langle \mathrm{S}, s_{init}, \mu, f \rangle$, a state $s \in \mathrm{S}$, an environment $\mathcal{E}$ mapping V to R, a time value $t \in \mathsf{R}$ and a TCTL-formula ϕ, the satisfaction relation $(\mathcal{T}, s, t) \models_{\mathcal{E}} \phi$ is defined inductively as follows:

$(s,t) \models_{\mathcal{E}} p$ iff $p \in \mu(s)$.

$(s,t) \models_{\mathcal{E}} (x+c) \leq (y+d)$ iff $\mathcal{E}(x) + c \leq \mathcal{E}(y) + d$.

$(s,t) \models_{\mathcal{E}} \neg\phi$ iff $(s,t) \not\models_{\mathcal{E}} \phi$.

$(s,t) \models_{\mathcal{E}} \phi_1 \wedge \phi_2$ iff $(s,t) \models_{\mathcal{E}} \phi_1$ and $(s,t) \models_{\mathcal{E}} \phi_2$.

$(s,t) \models_{\mathcal{E}} x.\,\phi$ iff $(s,t) \models_{[x \mapsto t]\mathcal{E}} \phi$.

$(s,t) \models_{\mathcal{E}} \exists\, \phi_1 \,\mathcal{U}\, \phi_2$ iff for some $\rho \in f$ with $\rho(0) = s$, for some $t' \geq 0$, $(\rho(t'), t + t') \models_{\mathcal{E}} \phi_2$ and $(\rho(t''), t + t'') \models_{\mathcal{E}} \phi_1$ for all $0 \leq t'' < t'$.

$(s,t) \models_{\mathcal{E}} \forall\, \phi_1 \,\mathcal{U}\, \phi_2$ iff for every $\rho \in f$ with $\rho(0) = s$, for some $t' \geq 0$, $(\rho(t'), t + t') \models_{\mathcal{E}} \phi_2$ and $(\rho(t''), t + t'') \models_{\mathcal{E}} \phi_1$ for all $0 \leq t'' < t'$.

A TCTL-structure $\mathcal{T}$ satisfies a TCTL-formula ϕ, written $\mathcal{T} \models \phi$, iff $(\mathcal{T}, s_{init}, 0) \models_{[V \mapsto 0]} \phi$. A TCTL-formula ϕ is called *satisfiable* iff there is a TCTL-structure $\mathcal{T}$ such that $\mathcal{T} \models \phi$. ■

In the above definition of $(s,t) \models_{\mathcal{E}} \phi$, the environment $\mathcal{E}$ gives the valuation of all the free variables of ϕ. The time value t gives the current time, and is used to bind the variable x while evaluating $(x.\,\phi)$. Consider, for instance, the formula

$$\phi \;=\; x.\,\exists\,(y.\,y \leq x+2 \vee p)\,\mathcal{U}\,(z.\,z \leq x+10 \,\wedge\, q).$$

Applying the definition of the satisfaction relation gives

$(s,t) \models_{\mathcal{E}} \phi$ iff for some $\rho \in f$ with $\rho(0) = s$, for some $t' \geq 0$, $(\rho(t'), t+t') \models_{[x \mapsto t]\mathcal{E}} (z.\,z \leq x+10 \wedge q)$ and $(\rho(t''), t+t'') \models_{[x \mapsto t]\mathcal{E}} (y.\,y \leq x+2 \vee p)$ for all $0 \leq t'' < t'$.

Unfolding the definition further gives

$(s,t) \models \phi$ iff for some $\rho \in f$ with $\rho(0) = s$, for some $t' \geq 0$, $q \in \mu(\rho(t'))$ and $(t+t' \leq t+10)$ and for all $0 \leq t'' < t'$, either $(t+t'' \leq t+2)$ or $p \in \mu(\rho(t''))$.

Thus the formula states the property that along some computation path, q holds at time $t' \leq 10$, and the proposition p holds over the interval $(2, t')$.

Note that every TCTL-formula is equivalent to its closure, in which all free variables are bound by a prefix of time quantifiers.

5.2.3 On the choice of syntax

In TCTL additional temporal operators such as $\exists\Diamond$, $\exists\Box$, $\forall\Diamond$, and $\forall\Box$ are defined using the "until" operators as in case of CTL. Furthermore, for writing timing constraints the

abbreviations such as $=$, $<$, $>$, $\geq$ are defined using the logical connectives and $\leq$. We will also use expressions such as $x \in (2,5]$. Note that TCTL has no *next-time* operator $\bigcirc$, because if time is dense then, by definition, there is no unique next time.

A typical bounded response property that "every p-state is followed by some q-state within 5 time units," is written in TCTL as

$$\forall\Box\, x.\,[\, p \rightarrow \forall\Diamond\, y.\,(q \wedge y < x+5)\,].$$

A few comments regarding the use of the notation of the time quantifier "$x.$" are in order. The variables in TCTL range over the time domain R, however, unlike the conventional first-order logics, the quantifier "$x.$" is neither existential nor universal. It is its own dual; observe the following TCTL equivalence:

$$\neg(x.\,\phi) \;\leftrightarrow\; x.\,(\neg\phi).$$

This quantifier has been called "half-order" in [Hen90].

An alternative to using the temporal quantifiers is to adopt the notation of first-order temporal logics such as RTTL [Ost90b] or XCTL [HLP90] which use a dynamic variable T denoting the current time along with the explicit quantification over the time domain. For instance, the above bounded response property is written as

$$\forall\Box\,\forall x.\,[\,(p \wedge T = x) \rightarrow \forall\Diamond\,(q \wedge T < x+5)\,].$$

In this formula, the variable x is a static variable ranging over the time domain R, and T is a dynamic variable representing the current time. We have shown that this first-order notation leads to a nonelementary blow-up in the decision procedure; for a more detailed comparison of the two notations see [AH90].

In TCTL temporal operators subscripted with time intervals are defined as abbreviations. We define $\exists\,\phi_1\, {}_{I'}\mathcal{U}_I\, \phi_2$ as an abbreviation for

$$x.\,\exists[y.\,(y \in I' + x \rightarrow \phi_1)]\,\mathcal{U}\,[z.\,(\phi_2 \wedge z \in I + x)].$$

The intervals I and I' have integer endpoints, but they may be bounded or unbounded, singular or nonsingular, closed or open or semiclosed. Recall that in the syntax of MITL singular intervals are disallowed, and so are interval subscripts in both arguments of $\mathcal{U}$. Such a restriction is not needed for the decidability of the model-checking problem for TCTL.

Consequently, the response property that "every p-state is followed by some q-state at time 2," is expressible in TCTL, we may write

$$\forall\Box[\, p \rightarrow \forall\Diamond_{[2,2]}\, q\,].$$

The abbreviation $\forall\, \phi_1\, {}_{I'}\mathcal{U}_I\, \phi_2$ is similarly defined. Notice that, unlike MITL, the *until* operator of TCTL is *nonstrict* in its both arguments. The corresponding strict versions can be defined as abbreviations. For instance, $(x.\, x > 0 \rightarrow \phi_1)\,\mathcal{U}\,\phi_2$ holds of a computation ρ iff for some $t \geq 0$, ϕ_2 holds at time t and ϕ_1 holds at all times $0 < t' < t$.

In TCTL all CTL properties can be specified with time bounds. For instance, now we can write $\exists\Diamond_{[2,5)}\phi$. It says that "$\phi$ holds at least once during the time interval $[2, 5)$ along some computation path." Similarly $\forall\Box_{\leq 2}\, p$ means that "p is an invariant until time 2," $\forall\Box\, \exists\Diamond_{<3}\, p$ says that "from every reachable state, some p-state is reachable within time 3."

For linear-time temporal logics with discrete semantics, the notation of temporal quantifiers is equally expressive as the notation of subscripted temporal operators [AH90]; though it can express certain properties more succinctly. However with the dense-time semantics, this notation seems to be strictly more expressive than the notation of subscripted operators. For example, consider the formula:

$$\exists\Diamond x.\, [p \wedge \exists\Diamond\, (q \wedge \exists\Diamond z.\, (r \wedge z < x + 5))]$$

It says that "there exists a computation with a p-state followed by a q-state, followed by an r-state, which is within 5 time units from the p-state." The notation of subscripted operators allows us to relate only the adjacent temporal contexts, and we conjecture that this property cannot be expressed by subscripted CTL operators.

5.2.4 Undecidability

The denseness of the underlying time domain allows us to encode Turing machine computations in TCTL-formulas. Consequently, unlike other logics with similar syntax but discrete-time semantics, the satisfiability question for TCTL is undecidable — Σ_1^1-hard. The proof of the next theorem is similar to the proof of undecidability of MITL with singular intervals (Theorem 4.17).

Theorem 5.6 The satisfiability question for TCTL is Σ_1^1-hard.

PROOF. We reduce the recurrence problem for nondeterministic 2-counter machines to TCTL-satisfiability. The encoding of a configuration $\langle i, c, d\rangle$ by an interval $[a, b)$ of

a computation ρ is as in the proof of Theorem 4.17. For example, we require that the proposition p_C holds at exactly c time instants in the interval $[a, b)$ along ρ.

We construct a TCTL-formula ϕ such that $(\mathcal{T}, s, 0) \models_{[V \mapsto 0]} \phi$ iff there exists a computation $\{\langle i_j, c_j, d_j \rangle : j \geq 0\}$ of the 2-counter machine A such that every computation ρ starting at s encodes the j-th configuration over the interval $[j, j+1)$ for all $j \geq 0$.

The initiation, consecution, and recurrence requirements are expressed as in the proof of Theorem 4.17; we replace each temporal operator by its universal version. For instance, if the second instruction is to increment D (and proceed to instruction 3), ϕ has the following conjunct:

$$\forall\Box_{\geq 0}\left[p_2 \rightarrow \begin{pmatrix} \forall\Diamond_{=1}\, p_3 \ \wedge\ \forall\Box_{(1,2)}\, \neg p_3 \wedge \\ \forall\Box_{[1,2)}\, (\wedge_{1 \leq i \leq n, i \neq 3}\, \neg p_i) \ \wedge \\ \forall\Box_{[0,1)}\ copy(p_C) \ \wedge \\ \forall\ copy(p_D)\, \mathcal{U}_{<1}(\neg p_D \ \wedge\ \forall\Diamond_{=1}\, p_D \ \wedge\ \forall\ copy(p_D)\, \mathcal{U}\, p_3) \end{pmatrix}\right]$$

The abbreviation $copy(p)$ stands for the conjunction

$$(p \rightarrow \forall\Diamond_{=1}\, p) \wedge (\neg p \rightarrow \forall\Diamond_{=1}\, \neg p).$$

The conjunct $\forall\Box\forall\Diamond\, p_1$ expresses the recurrence requirement. If ϕ is the formula constructed this way then ϕ is satisfiable iff A has a recurring computation. Consequently, TCTL-satisfiability is Σ_1^1-hard. ■

5.2.5 Interval automata as TCTL-structures

We defined interval automata to model finite-state real-time systems in Section 4.3. We will interpret TCTL-formulas over interval automata. For technical reasons, we need to modify the definition of interval automata slightly, so we define *timed structures*. The timed structures differ from the interval automata in two aspects. Firstly, to associate a unique computation tree with a given timed structure we assume that timed structures have a single initial state denoted by s_{init}. Secondly, we need to consider runs that start at arbitrary states, and hence, we do not impose the initiality requirement on the runs. The definition of a timed structure and its runs is given below:

Definition 5.7 A *timed structure* is a tuple $\mathcal{M} = \langle \mathrm{S}, s_{init}, \mu, \mathrm{C}, \cdot, \mathrm{E} \rangle$, where S is a finite set of states, $s_{init} \in \mathrm{S}$ is an initial state, $\mu : \mathrm{S} \rightarrow 2^{\mathrm{AP}}$ is an assignment of atomic propositions

to states, C is a finite set of clocks, $\Delta : \mathrm{S} \rightarrow \Phi(\mathrm{C})$ is an assignment of clock constraints to states, and $\mathrm{E} \subseteq \mathrm{S} \times \mathrm{S} \times 2^{\mathrm{C}}$ is a set of edges.

A run r of a timed structure $\mathcal{M}$ starting in state s_0 is an infinite sequence

$$r: \quad \xrightarrow[\nu_0]{} (s_0, I_0) \xrightarrow[\nu_1]{\lambda_1} (s_1, I_1) \xrightarrow[\nu_2]{\lambda_2} (s_2, I_2) \xrightarrow[\nu_3]{\lambda_3} \cdots$$

of states $s_i \in \mathrm{S}$, intervals I_i, clock sets $\lambda_i \subseteq \mathrm{C}$, and clock interpretations $\nu_i \in \Gamma(\mathcal{M})$ satisfying the following constraints:

- *Consecution:*
  - the sequence $I_0 I_1 I_2 \ldots$ forms an interval sequence,
  - for all $i \geq 0$ either $\langle s_i, s_{i+1}, \lambda_i \rangle \in \mathrm{E}$ or $s_{i+1} = s_i$ with $\lambda_i = \emptyset$,
  - $\nu_{i+1} = [\lambda_{i+1} \mapsto 0](\nu_i + r(I_i) - l(I_i))$ for all $i \geq 0$.
- *Timing:* for all $t \in I_i$, the clock interpretation at time t, $\gamma_r(t) = \nu_i + t - l(I_i)$, satisfies $\Delta(s_i)$.

■

Observe that the future behavior of the system is fully determined by its current state and the current clock interpretation. This motivates the following definition:

Definition 5.8 For a timed structure $\mathcal{M} = \langle \mathrm{S}, s_{init}, \mu, \mathrm{C}, \cdot, \mathrm{E} \rangle$, an *extended state* is a pair $\langle s, \nu \rangle$ where $s \in \mathrm{S}$ and ν is a clock interpretation over C such that $\nu \models \Delta(s)$. ■

Given an extended state we can evaluate TCTL-formulas by considering all the runs starting at it. Hence to obtain a TCTL-structure $\mathcal{T}_{\mathcal{M}}$ from a timed structure M we take all the extended states of M as the states of $\mathcal{T}_{\mathcal{M}}$. The initial state of $\mathcal{T}_{\mathcal{M}}$ is $\langle s_{init}, [\mathrm{C} \mapsto 0] \rangle$. Runs of M give maps from R to extended states of M, that is, a computation over the states of $\mathcal{T}_{\mathcal{M}}$.

Definition 5.9 Consider a run r of M of the form

$$r: \quad \xrightarrow[\nu_0]{} (s_0, I_0) \xrightarrow[\nu_1]{\lambda_1} (s_1, I_1) \xrightarrow[\nu_2]{\lambda_2} (s_2, I_2) \xrightarrow[\nu_3]{\lambda_3} \cdots$$

The computation ρ_r associated with the run r is a map from R to $\mathrm{S} \times \Gamma(\mathcal{M})$ defined by, for each $t \in I_j$, $\rho_r(t) = \langle s_j, \gamma_r(t) \rangle$ with $\gamma_r(t) = \nu_j + t - l(I_j)$. ■

Notice that if two time instants t and t' belong to the same time interval I_i along a run r then $\mu(\rho_r(t)) = \mu(\rho_r(t'))$. It follows that ρ_r satisfies the finite-variability constraint. Furthermore, the following lemma also holds; it can be proved by a straightforward application of the definitions.

Lemma 5.10 For a timed structure M the collection of all the computations corresponding to the runs of M satisfies both the suffix closure and fusion closure requirements of Definition 5.4. ■

Now we can formally associate a TCTL-structure with M and use it to interpret TCTL-formulas.

Definition 5.11 Given a timed structure M, the corresponding TCTL-structure is $\mathcal{T}_{\mathcal{M}} = \langle \mathrm{S} \times \Gamma(\mathcal{M}), \langle s_{init}, [\mathrm{C} \mapsto 0] \rangle, \mu', f_{\mathcal{M}} \rangle$, where the labeling function is defined by $\mu'(\langle s, \nu \rangle) = \mu(s)$, and $f_{\mathcal{M}}$ consists of precisely the computations corresponding to all the runs of M.

For a TCTL-formula ϕ, $\mathcal{M} \models \phi$ precisely when $\mathcal{T}_{\mathcal{M}} \models \phi$. A TCTL-formula ϕ is called *finitely satisfiable* iff there exists a timed structure M such that $\mathcal{M} \models \phi$. ■

Theorem 5.12 shows that the finite-satisfiability question is also undecidable. For the conventional temporal logics such CTL or PTL, the algorithm for constructing finite models is used for automatic synthesis of a synchronization skeleton meeting the constraints specified by the temporal logic specification [EC82, MW84]. The undecidability of the finite-satisfiability question for TCTL implies that this approach cannot be used for automatic synthesis from TCTL-specifications. Later, we will show that the set of satisfiable TCTL-formulas differs from the set of finitely-satisfiable TCTL-formulas; that is, unlike CTL the two notions of satisfiability are different for TCTL.

Theorem 5.12 The set of finitely-satisfiable TCTL-formulas is not recursive.

PROOF. We reduce the halting problem for 2-counter machines to the finite-satisfiability question.

While proving the undecidability of TCTL-satisfiability (see proof of Theorem 5.6), we encoded the computations of a given 2-counter machine A using a formula ϕ. Let us assume that A is deterministic and its halting corresponds to the location counter taking a specific value, say n. Let ψ be the conjunction of ϕ and the halting requirement $\forall\Diamond\, p_n$.

If A does not halt, then ψ is not satisfiable, and hence, not finitely satisfiable.

If A halts, then ψ is satisfiable. Suppose A stops after m steps. Clearly both the counters never exceed m. Hence according to our encoding scheme the truth assignment need not change more than $2m$ times during an interval of unit length. Consequently we can construct a finite prefix of state-interval pairs $\rho = (s_0, I_0)(s_1, I_1)\ldots(s_k, I_k)$ such that the transition times (the boundaries of I_js) are multiples of $(1/2m)$ and ρ encodes the halting computation of A. It is straightforward to construct a finite timed structure that has a single run which starts with the prefix ρ. Consequently ψ is finitely satisfiable.

Thus, ψ is finitely satisfiable iff A halts. The theorem follows. ∎

5.3 Model checking

In this section we develop an algorithm for deciding whether a finite-state real-time system presented as a timed structure M meets its specification given as a TCTL-formula ϕ. We will also study the complexity of the model-checking problem. Throughout this section we will assume M and ϕ are fixed.

5.3.1 Introducing formula clocks

A state of the TCTL-structure $\mathcal{T}_{\mathcal{M}}$ is a pair $\langle s, \nu \rangle$ where ν is a clock interpretation. Let ψ be a subformula of ϕ. Recall that the satisfaction relation is defined as $(\langle s, \nu \rangle, t) \models_{\mathcal{E}} \psi$. Thus to evaluate the truth of a formula, in addition to the state and the clock values, one also needs the current time t and the environment $\mathcal{E}$ giving values for the free variables in ψ. It is convenient to reformulate the semantic definition by introducing a clock for each variable appearing in ϕ.

Definition 5.13 Let C_ϕ be the set of variables appearing in ϕ, let C^* be $\mathrm{C} \cup \mathrm{C}_\phi$. The timed structure $\mathcal{M}_\phi$ is defined to be the tuple $\langle \mathrm{S}, s_{init}, \mu, \mathrm{C}^*, \Delta, \mathrm{E} \rangle$. ∎

An extended state of $\mathcal{M}_\phi$ is a pair $\langle s, \nu \rangle$ where $s \in \mathrm{S}$ and ν is a clock interpretation over C^*. The runs of $\mathcal{M}_\phi$ give computations which are maps from R to its extended states. Note that the extra clocks introduced are not reset along the runs of $\mathcal{M}_\phi$, they only record the elapsed time and are used to interpret the variables in ϕ.

Consider a clock interpretation ν for $\mathcal{M}_\phi$. The time assignment for the clocks of M is interpreted as before. In addition, ν also gives an assignment from C_ϕ to R. This is viewed as an interpretation for the variables in ϕ: the intended meaning is that at time t, for each

variable x, $\nu(x)$ should equal $t - \mathcal{E}(x)$. Now given an extended state $\langle s, \nu\rangle$, the formula ψ can be interpreted; the clock interpretation ν contains the information about t and $\mathcal{E}$. We redefine the semantics as follows:

Definition 5.14 For an extended state $\langle s, \nu\rangle$ of $\mathcal{M}_\phi$, and a subformula ψ of ϕ, the satisfaction relation $\langle s, \nu\rangle \models \psi$ is defined inductively as follows:

$\langle s, \nu\rangle \models p$ iff $p \in \mu(s)$.
$\langle s, \nu\rangle \models (x + c) \leq (y + d)$ iff $\nu(x) + d \geq \nu(y) + c$.
$\langle s, \nu\rangle \models \neg\phi$ iff $\langle s, \nu\rangle \not\models \phi$.
$\langle s, \nu\rangle \models \phi_1 \wedge \phi_2$ iff $\langle s, \nu\rangle \models \phi_1$ and $\langle s, \nu\rangle \models \phi_2$.
$\langle s, \nu\rangle \models x.\,\phi$ iff $(s, [x \mapsto 0]\nu) \models \phi$.
$\langle s, \nu\rangle \models \exists\, \phi_1 \,\mathcal{U}\, \phi_2$ iff for some run r of $\mathcal{M}_\phi$ starting at $\langle s, \nu\rangle$, $\rho_r(t) \models \phi_2$ for some $t \geq 0$, and $\rho_r(t') \models \phi_1$ for all $0 \leq t' < t$.
$\langle s, \nu\rangle \models \forall\, \phi_1 \,\mathcal{U}\, \phi_2$ iff for every run r of $\mathcal{M}_\phi$ starting at $\langle s, \nu\rangle$, $\rho_r(t) \models \phi_2$ for some $t \geq 0$, and $\rho_r(t') \models \phi_1$ for all $0 \leq t' < t$.

■

The advantage of using the revised definition is that environments give static interpretations to variables, whereas the clock interpretations for variables integrate conveniently with the clock interpretations of the timed structure. The following lemma states the equivalence of the two semantic definitions.

Lemma 5.15 Consider a subformula ψ of ϕ, and $t \in \mathsf{R}$. Let $\mathcal{E}$ be an environment and ν' be a clock interpretation for C_ϕ such that $\nu'(x) = t - \mathcal{E}(x)$ for all variables $x \in \mathrm{C}_\phi$. Then $(\langle s, \nu\rangle, t) \models_{\mathcal{E}} \psi$ iff $\langle s, \nu\cdot\nu'\rangle \models \psi$.

PROOF. The proof is by a straightforward induction on the structure of ϕ. Observe that $[\mathcal{E}(x) + c \leq \mathcal{E}(y) + d]$ iff $[t - \mathcal{E}(x) + d \geq t - \mathcal{E}(y) + c]$. ■

It follows that to check the truth of ϕ with respect to M, it suffices to check whether the initial extended state $\langle s_{init}, [\mathrm{C}^* \mapsto 0]\rangle$ satisfies ϕ.

5.3.2 Clock regions

The task of the model-checking algorithm is to evaluate the truth of a formula in the initial extended state. To this end the algorithm needs to evaluate the truth of all the subformulas

in all the extended states. There are infinitely many (in fact, uncountable) extended states, but, not all of these extended states are distinguishable by our logic. If two extended states corresponding to the same state agree on the integral parts of all clock values, and also on the ordering of the fractional parts of all clock values, then the computation trees rooted at these two extended states cannot be distinguished by TCTL-formulas. We define an equivalence relation over the set of all clock interpretations as in Section 3.3.2 and show that TCTL-formulas cannot distinguish between equivalent clock interpretations.

Definition 5.16 For $x \in \mathrm{C}$, let c_x denote the largest constant d such that $x \leq d$ is a subformula of some clock constraint appearing in E. For $x \in \mathrm{C}_\phi$, let c_x denote the largest constant d such that $(x + c \leq y + d)$ or $(x + d \leq y + c)$ is a subformula of ϕ.

Given clock interpretations ν and ν' over C^*, define $\nu \sim \nu'$ iff all of the following conditions are met:

1. For each $x \in \mathrm{C}^*$, either $\lfloor \nu(x) \rfloor$ and $\lfloor \nu'(x) \rfloor$ are the same, or both $\nu(x)$ and $\nu'(x)$ are greater than c_x.
2. For every $x, y \in \mathrm{C}^*$ such that $\nu(x) \leq c_x$ and $\nu(y) \leq c_y$, $\mathit{fract}(\nu(x)) \leq \mathit{fract}(\nu(y))$ iff $\mathit{fract}(\nu'(x)) \leq \mathit{fract}(\nu'(y))$.
3. For each $x \in \mathrm{C}^*$ such that $\nu(x) \leq c_x$, $\mathit{fract}(\nu(x)) = 0$ iff $\mathit{fract}(\nu'(x)) = 0$.
4. For a subformula $(x + c \leq y + d)$ of ϕ, $(\nu(x) + d \geq \nu(y) + c)$ iff $(\nu'(x) + d \geq \nu'(y) + c)$.

A *clock region* is an equivalence class of clock interpretations induced by $\sim$.

An *end region* is a clock region satisfying $x > c_x$ for all $x \in \mathrm{C}^*$. A *boundary region* is a clock region satisfying $x = c$ for some $x \in \mathrm{C}^*$ and for some $c \leq c_x$. ■

Example 5.17 Consider a system with $\mathrm{C}^* = \{x, y\}$ with $c_x = 2$ and $c_y = 1$. For a clock interpretation ν with $\nu(x) = 0.3$ and $\nu(y) = 0.7$, the clock region $[\nu]$ consists of all clock interpretations satisfying $(0 < x < y < 1)$ (see Section 3.3.2 for greater details).

The clock region $[0 < y < x = 1]$ is a boundary region. The end region is given by $[(x > 2), (y > 1)]$. If x and y are clocks corresponding to ϕ and $x \leq y$ is a subformula of ϕ, then this end region is split into two end regions: $[2 < x \leq y]$ and $[(1 < y < x), (2 < x)]$. ■

As a corollary to the model-checking algorithm it will follow that

> *For two extended states $\langle s, \nu \rangle$ and $\langle s, \nu' \rangle$ of $\mathcal{M}_\phi$, if $\nu \sim \nu'$ then for every subformula ψ of ϕ, $\langle s, \nu \rangle \models \psi$ iff $\langle s, \nu' \rangle \models \psi$.*

5.3.3 The region graph

The first step in the model-checking algorithm is to construct a region graph similar to the one in Section 3.3.3.

Definition 5.18 A *region* is a pair $\langle s, \alpha \rangle$, where $s \in \mathrm{S}$, and α is a clock region satisfying $\Delta(s)$. ■

We want to label the regions with all the subformulas of ϕ such that $\langle s, [\nu] \rangle$ is labeled with ψ iff $\langle s, \nu \rangle \models \psi$.

Suppose we want to determine whether $\langle s, [\nu] \rangle$ should be labeled with $\exists\, \phi_1 \,\mathcal{U}\, \phi_2$. We try to find a run starting at $\langle s, \nu \rangle$ such that the associated path satisfies $\phi_1 \,\mathcal{U}\, \phi_2$. As time progresses, the extended state of the system changes, but the truth of the subformulas ϕ_1 and ϕ_2 changes only when it moves to a new region. Hence, instead of the desired run we search for a finite sequence of regions starting at $\langle s, [\nu] \rangle$ such that each region can be reached from the previous one either by a state-transition of M or by increase in the values of clocks. Furthermore, for the desired sequence, ϕ_2 should be true over its last region, and ϕ_1 should be true over all the intermediate regions.

The edge relation over the regions captures two different types of events: (1) transitions in M, and (2) moving into a new clock region because of the passage of time. We define a *time-successor* function over clock regions to capture the second type of transitions.

Definition 5.19 Let α and β be distinct clock regions. $succ(\alpha) = \beta$ iff for each $\nu \in \alpha$, there exists a positive $t \in \mathsf{R}$ such that $\nu + t \in \beta$, and for all $t' < t$, $\nu + t' \in \alpha \cup \beta$. ■

Successor is defined for every clock region except the end region.

Example 5.20 Let us consider an example with two clocks x and y with $c_x = 2$ and $c_y = 1$. The clock regions are shown in Figure 5.1. Note that the regions which lie on either horizontal or vertical lines are boundary regions.

The successor of a clock region α is the class to be hit first by a line drawn from some point in α in the diagonally upwards direction. For example, the successor of the clock region $[x = 1, y = 0]$ is the clock region $[(1 < x < 2), (y = x - 1)]$. The successor of $[0 < y < x < 1]$ is the clock region $[0 < y < x = 1]$. ■

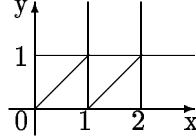


Figure 5.1: Clock regions

Notice that the definition of successor here is somewhat different from the one used while checking emptiness of timed automata (see Section 3.3.3). Consider a system with one clock x with $c_x = 1$. In case of timed automata, the region $[0 < x < 1]$ has two time-successors; the region $[x = 1]$ and the region $[x > 1]$. On the other hand, by the above definition, the successor of $[0 < x < 1]$ is $[x = 1]$, and the successor of $[x = 1]$ is $[x > 1]$. This is because for TCTL model-checking all the regions appearing along the path are of importance. Consider a transition from state s to state s' with the clock constraint $(x > 1)$. The single transition from $\langle s, [x < 1]\rangle$ to $\langle s', [x > 1]\rangle$ in case of the region automaton for timed automata, now needs to be modeled by a sequence of transitions: $\langle s, [x < 1]\rangle$ to $\langle s, [x = 1]\rangle$ to $\langle s, [x > 1]\rangle$ to $\langle s', [x > 1]\rangle$.

We can represent each clock region as in Section 3.3.3 by recording (i) for every clock x a single constraint of the form $x = c$, or $x \in (c, c+1)$, or $x > c_x$, and (ii) the ordering of the fractional parts of all the clocks. The time-successor of every clock region can be computed (in time linear in the number of clocks) as in Section 3.3.3. Let us define the region graph.

Definition 5.21 The region graph $\mathrm{R}(\mathcal{M}, \phi)$ is defined to be the graph $\langle V_1, E_1\rangle$. The vertex set V_1 is the set of all regions. The edge set E_1 consists of two types of edges:

1. Edges representing the passage of time: every vertex $\langle s, \alpha\rangle$ has an edge to $\langle s, succ(\alpha)\rangle$.

2. Edges representing the transitions of M:

   - Consider a vertex $\langle s, \alpha\rangle$, where α is not a boundary region. For every edge $s \xrightarrow{\lambda} s'$, it has an edge to each of the vertices $\langle s', [\lambda \mapsto 0]\alpha\rangle$, $\langle s', succ([\lambda \mapsto 0]\alpha)\rangle$, and $\langle s', [\lambda \mapsto 0]succ(\alpha)\rangle$.
   - Consider a vertex $\langle s, \alpha\rangle$, where α is a boundary region. For every edge $s \xrightarrow{\lambda} s'$, it has an edge to the vertex $\langle s', succ([\lambda \mapsto 0]\alpha)\rangle$.

■

In the above definition, the edges are defined only if the target is a legal region. For example, in the first clause there is an implicit assumption that $\langle s, succ(\alpha)\rangle$ is a region, that is, α is not an end-region, and $succ(\alpha)$ satisfies $\Delta(s)$. The second clause considers different cases, and can be best understood through an example.

Example 5.22 Consider a system with two clocks x and y. Assume that there is an edge from s to s' which resets x. Also assume that $\Delta(s)$ and $\Delta(s')$ are always true.

Suppose the current region is $v = [s, 0.4, 0.8]$. The first clause gives an edge to $[s, 0.6, 1]$. This corresponds to the case when the transition to state s' does not happen before the clock y reaches 1.

Next consider the edges representing the transition from s to s' assuming that the transition happens before the equivalence class changes. From v there is an edge to $[s', 0, 0.9]$: this corresponds to the case when the state at the transition point is s'; that is, current interval is right-open, and the value of x at the transition point is 0. The prefix of such a run looks like

$$\xrightarrow[{[0.4,0.8]}]{} (s, [0, 0.1)) \xrightarrow[{[0,0.9]}]{\{x\}} (s', [0.1, -)) \cdots$$

The vertex v also has an edge to $[s', 0.05, 0.95]$, this corresponds to the case when the current interval is right-closed, and hence the region $[s', 0, 0.9]$ does not appear along the run, only its successor appears. This case is shown in the following prefix:

$$\xrightarrow[{[0.4,0.8]}]{} (s, [0, 0.1]) \xrightarrow[{[0,0.9]}]{\{x\}} (s', (0.1, -)) \cdots$$

Now let us consider the case when the transition from s to s' happens exactly when the clock region changes. In this case v has an edge to $[s', 0, 1]$ corresponding to the case that the current interval is right-open. If the current interval is right-closed then a region change precedes the state-change.

Finally let us consider the edges from the boundary region $[s, 0.6, 1]$. The system can stay in this region only instantaneously. The first clause gives an edge to $[s, 0.7, 1.1]$. For the edges representing the state-transitions of M, the only relevant case is if the state changes before the region $[s, 0.7, 1.1]$ appears on the run. Since the current interval has to be right-closed, there is only one case to be considered, and the region has an edge to $[s', 0.1, 1.1]$. ■

There is a simple correspondence between the runs of M and infinite paths through $R(\mathcal{M}, \phi)$. To formalize this correspondence, we define the notion of a *refined run* as follows:

Definition 5.23 Let r be a run of the timed structure $\mathcal{M}_\phi$:

$$r: \quad \xrightarrow[\nu_0]{} (s_0, I_0) \xrightarrow[\nu_1]{\lambda_1} (s_1, I_1) \xrightarrow[\nu_2]{\lambda_2} (s_2, I_2) \xrightarrow[\nu_3]{\lambda_3} \cdots$$

It is called a *refined run* iff whenever two time instants t and t' are in the same interval I_j, the clock interpretations at time t and t' are equivalent, that is, $\gamma_r(t) \sim \gamma_r(t')$. ■

Thus along a refined run all the clock interpretations in the same interval belong to the same clock region. The next lemma asserts that from any given run we can obtain a refined run by splitting each interval.

Lemma 5.24 For every run r of $\mathcal{M}_\phi$ there exists a refined run r' such that the computations ρ_r and $\rho_{r'}$ associated with them are the same.

PROOF. Consider a run r of the form

$$r: \quad \xrightarrow[\nu_0]{} (s_0, I_0) \xrightarrow[\nu_1]{\lambda_1} (s_1, I_1) \xrightarrow[\nu_2]{\lambda_2} (s_2, I_2) \xrightarrow[\nu_3]{\lambda_3} \cdots$$

To obtain the corresponding refined run, we split each interval I_j into a finite sequence of adjacent intervals. Given ν_j, let $t_1, \ldots t_k$, each $t_i \in I_j$, be the sequence of time values at which the clock region changes as time increases. The region does not change during the intervals (t_i, t_{i+1}), and during $I_{j_1} = \{t \in I_j \mid t < t_1\}$, and during $I_{j_2} = \{t \in I_j \mid t > t_n\}$. The successive regions along this chain are related by the successor function. We replace the element (s_j, I_j) by the sqeuence

$$(s_j, I_{j_1}) \xrightarrow[\nu_j + t_1]{\emptyset} (s_j, [t_1, t_1]) \xrightarrow[\nu_j + t_1]{\emptyset} (s_j, (t_1, t_2)) \xrightarrow[\nu_j + t_2]{\emptyset} \cdots (s_j, [t_n, t_n]) \xrightarrow[\nu_j + t_n]{\emptyset} (s_j, I_{j_2})$$

For example, let $\nu_j = [0.3, 0.7]$, and $I_j = [1, 2.4)$. Then the desired sequence of transition times is $1.3, 1.7, 2.3$, and the sequence of intervals is

$$[1, 1.3) \rightarrow [1.3, 1.3] \rightarrow (1.3, 1.7) \rightarrow [1.7, 1.7] \rightarrow (1.7, 2.3) \rightarrow [2.3, 2.3] \rightarrow (2.3, 2.4)$$

Check that if t and t' belong to the same interval in the above sequence then $\nu_j + t - 1$ and $\nu_j + t' - 1$ are equivalent. ■

Consequently, while interpreting TCTL-formulas at an extended state of $\mathcal{M}_\phi$ one can restrict attention only to the refined runs starting at that state.

Observe that a refined run can be characterized by an ω-sequence of regions:

Definition 5.25 Let r be a refined run of $\mathcal{M}_\phi$ with states s_i and intervals I_i. The projection of r, denoted *project*(r), is an infinite sequence of regions

$$\langle s_0, \alpha_0 \rangle \rightarrow \langle s_1, \alpha_1 \rangle \rightarrow \langle s_2, \alpha_2 \rangle \rightarrow \cdots$$

such that for each $t \in I_j$, the clock interpretation $\gamma_r(t) \in \alpha_j$. ■

The edges in the region graph are defined so that if two regions v and v' appear adjacent along the projection of some refined run then there is an edge from v to v'. Thus if $v_0 v_1 v_2 \ldots$ is the projection of a refined run then it is also a path through $\mathrm{R}(\mathcal{M}, \phi)$.

Now let us see if the converse of this holds: does every infinite path through the region graph correspond to the projection of some run? This "almost" holds, but to ensure that the runs satisfy the *progress* requirement, we need to impose certain fairness constraints on the region graph. For example, if a vertex whose clock region satisfies $(x < 1)$ has a self loop, then the progress of time requires that no run can correspond to looping infinitely on this vertex.

Consider the projection *project*$(r) = v_0 v_1 \ldots$ Since time progresses without bound along r, every clock $x \in \mathrm{C}^*$ is either reset infinitely often or eventually it always increases. Hence, for each $x \in \mathrm{C}^*$, along *project*(r), infinitely many regions satisfy either $(x = 0)$ or $(x > c_x)$. This is because if a clock is never reset from a certain position onwards then its value will eventually cross every time value.

This motivates the following definition:

Definition 5.26 Define the set $\mathrm{F}_x = \{\langle s, [\nu] \rangle \mid (s \in \mathrm{S}) \wedge (\nu(x) = 0 \vee \nu(x) > c_x)\}$ for $x \in \mathrm{C}^*$. A path $v_0 v_1 v_2 \ldots$ through $\mathrm{R}(\mathcal{M}, \phi)$ is called $\mathcal{F}_p$-fair iff for each $x \in \mathrm{C}^*$, for infinitely many i's, v_i is in F_x. ■

Thus the projections of the runs are $\mathcal{F}_p$-fair paths in the region graph. The following lemma states this property:

Lemma 5.27 For every refined run r of $\mathcal{M}_\phi$, the sequence of regions *project*(r) is an $\mathcal{F}_p$-fair path through $\mathrm{R}(\mathcal{M}, \phi)$. ■

Conversely, from an $\mathcal{F}_p$-fair path it is possible to construct a run which projects onto this path, and also satisfies the progress requirement.

Lemma 5.28 For every $\mathcal{F}_p$-fair path π through $\mathrm{R}(\mathcal{M}, \phi)$, there exists a refined run r of $\mathcal{M}_\phi$ such that π equals *project*(r).

Proof. Consider an infinite path $\pi = v_0 v_1 \ldots$ through $\mathrm{R}(\mathcal{M}, \phi)$ with $v_i = \langle s_i, \alpha_i \rangle$. Choose some $\nu_0 \in \alpha_0$. The interval I_0 is left-closed and $l(I_0) = 0$. Now we construct the desired (refined) run r inductively. Assume ν_i and the left boundary for I_i is chosen. We show how to choose the right-boundary of I_i and the clock interpretation ν_{i+1}. Repeating this process defines the desired run r.

First consider the case when α_i is a boundary region. In this case I_i is a singular interval, that is, set $r(I_i) = l(I_i)$, and I_i is right-closed. If $\alpha_{i+1} = succ(\alpha_i)$ then $\lambda_{i+1} = \emptyset$, and $\nu_{i+1} = \nu_i$. Otherwise, the edge from v_i to v_{i+1} corresponds to a M-transition $s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$; choose $\nu_{i+1} = [\lambda_{i+1} \mapsto 0]\nu_i$.

Now consider the case when α_i is not a boundary region. If α_i is not the end-region then there is some positive ϵ_i such that $\nu_i + \epsilon_i \in succ(\alpha_i)$, and $\nu_i + \epsilon \in \alpha_i$ for all $\epsilon \in (0, \epsilon_i)$. That is, the clock interpretation ν_i changes its equivalence class after ϵ_i time elapses. There are several cases to consider.

(1) Suppose the edge from v_i to v_{i+1} indicates passage of time, that is, $s_{i+1} = s_i$ and $\alpha_{i+1} = succ(\alpha_i)$. In this case, I_i is right-open. Choose $r(I_i) = l(I_i) + \epsilon_i$, $\nu_{i+1} = \nu_i + \epsilon_i$, and $\lambda_{i+1} = \emptyset$.

(2) This case corresponds to a state-transition $s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$ happening before the current clock region changes. We can choose the length of I_i in this case. If α_i is the end-region let $\epsilon = 1$ else $\epsilon = \epsilon_i/2$. Choose $r(I_i) = l(I_i) + \epsilon$, and $\nu_{i+1} = [\lambda_{i+1} \mapsto 0](\nu_i + \epsilon')$. If $\alpha_{i+1} = [[\lambda_{i+1} \mapsto 0]\nu_i]$ then I_i is right-open, and if $\alpha_{i+1} = succ([[\lambda_{i+1} \mapsto 0]\nu_i])$ then I_i is right-closed.

(3) Let us consider the case when the state change occurs after time ϵ_i according the M-edge $s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$. The current interval I_i is right-open. Choose $r(I_i) = l(I_i) + \epsilon_i$, and $\nu_{i+1} = [\lambda_{i+1} \mapsto 0](\nu_i + \epsilon_i)$.

It is easy to check that the run r satisfies the consecution and timing requirements. Also π equals *project*(r). Furthermore, since π is $\mathcal{F}_p$-fair, r satisfies the progress requirement. Suppose r does not meet the progress requirement. The run contains infinitely many transitions corresponding to the edges in M. From the above construction it follows that the sequence of values ϵ_i converges to 0. This implies that there exists $j \geq 0$ such that for all $i \geq j$ the regions α_i satisfy $(d < x < d+1)$ for some clock x and some integer $d < c_x$. This implies that π is not fair with respect to F_x. ■

From the above two lemmas it follows that the behavior of the timed structure $\mathcal{M}_\phi$ can be analyzed by examining the $\mathcal{F}_p$-fair paths in the region graph. Notice that by introducing the fairness requirements, we are essentially treating the region graph as a Kripke structure for CTL$^{\mathrm{F}}$ with the fairness family given by $\mathcal{F}_p = \{\mathrm{F}_x : x \in \mathrm{C}^*\}$.

5.3.4 Labeling algorithm

We label the regions with subformulas of ϕ starting from the subformulas of length 1, then of length 2, and so on. We introduce a special proposition p_b, and label a region $v = \langle s, \alpha \rangle$ with p_b iff α is a boundary region.

Let ψ be a subformula of ϕ. Assume that all the regions are already labeled with each subformula of ψ. Let $v = \langle s, [\nu] \rangle$ be a region.

- If ψ is an atomic proposition, label v with ψ iff $\psi \in \mu(s)$.
- If ψ is the timing constraint $(x + c \leq y + d)$, then label v with ψ iff $\nu(x) + d \geq \nu(y) + c$.
- If ψ is $\neg\psi'$, then label v with ψ iff v is not labeled with ψ'.
- If ψ is the conjunction $\psi_1 \wedge \psi_2$, then label v with ψ iff v is labeled with both ψ_1 and ψ_2.
- If ψ is $x.\psi'$, then label v with ψ iff the region $\langle s, [x \mapsto 0]\nu \rangle$ is labeled with ψ'.
- If ψ is the temporal subformula $\exists\, \psi_1 \mathcal{U}\, \psi_2$ ($\forall\, \psi_1 \mathcal{U}\, \psi_2$, respectively), then label v with ψ iff some (every, respectively) $\mathcal{F}_p$-*fair* path through $\mathrm{R}(\mathcal{M}, \phi)$ starting at v has a prefix $v_1, v_2, \ldots v_n$ such that each v_i, $1 \leq i < n$, is labeled with ψ_1, and v_n is labeled with ψ_2 and with either ψ_1 or p_b.

■

The last condition can be tested using conventional model-checking algorithms for CTL$^{\mathrm{F}}$ [CES86]. The following lemma states the correctness of the above labeling procedure.

Lemma 5.29 Let ψ be a subformula of ϕ. The above labeling algorithm labels $\langle s, [\nu] \rangle$ with ψ iff $\langle s, \nu \rangle \models \psi$.

Proof. We assume that the CTL labeling algorithm works correctly. The proof is by induction on the structure of ψ. For satisfaction of TCTL-formulas, we use Definition 5.14.

The cases $\psi \in$ AP, $\psi = (x + c \leq y + d)$, $\psi = \neg\psi'$, $\psi = \psi_1 \wedge \psi_2$, and $\psi = x.\psi'$ follow trivially from definitions. We prove that $\langle s, \nu\rangle \models \psi$ iff the algorithm labels $\langle s, [\nu]\rangle$ with ψ, where $\psi = \exists\, \phi_1 \mathcal{U}\, \phi_2$. The other case $\psi = \forall\, \phi_1 \mathcal{U}\, \phi_2$ is similar.

First assume that the algorithm labels $\langle s, [\nu]\rangle$ with ψ. Hence there exists an $\mathcal{F}_p$-fair path π with prefix $v_0v_1 \ldots v_n$ such that v_n is labeled with ψ_2 and each v_i, $0 \leq i < n$, is labeled with ψ_1. Let $v_i = \langle s_i, \alpha_i\rangle$. By Lemma 5.28 there exists a refined run r with states s_i and intervals I_i such that *project*$(r) = \pi$. Consider some $t \in I_n$. We know that $\gamma_r(t) \in \alpha_n$. Since v_n is labeled with ψ_2 it follows from the induction hypothesis that $\rho_r(t) \models \psi_2$. Consider $t' < t$, and let $t' \in I_j$. If $j < n$ then v_j is labeled with ψ_1. If $j = n$, the clocks at two distinct time values, t and t', belong to the same clock region α_n, and hence, α_n cannot be a boundary region. So even in this case, the vertex v_j is labeled with ψ_1. Now we can use the induction hypothesis to conclude that $\rho_r(t') \models \psi_1$. Hence, $\langle s, \nu\rangle$ satisfies ψ.

Now let us assume that $\langle s, \nu\rangle \models \psi$. Hence there exists a run starting at $\langle s, \nu\rangle$ that satisfies $\psi_1 \mathcal{U}\, \psi_2$. By Lemma 5.24 there exists a refined run r with states s_i and intervals I_i, satisfying $\psi_1 \mathcal{U}\, \psi_2$. Suppose $\rho_r(t) \models \psi_2$, and $\rho_r(t') \models \psi_1$ for some $t \in I_n$ and for all $t' < t$. By Lemma 5.27 the projection $v_0v_1 \ldots$ of r is an $\mathcal{F}_p$-fair path in R$(\mathcal{M}, \phi)$. Let $v_i = \langle s_i, \alpha_i\rangle$. We know that $\gamma_r(t) \in \alpha_n$. Hence, by the induction hypothesis, the algorithm labels v_n with ψ_2. For each $0 \leq i < n$, there is some t' in I_i with $t' < t$, and $\rho_r(t') \models \psi_1$. Again, by similar reasoning, each v_i should be labeled with ψ_1. Also if α_n is not a boundary region then there is some $t' \in I_n$ with $t' < t$, and hence in this case, v_n also should be labeled with ψ_1. Thus the region graph has a path of the desired form, and hence, $\langle s, [\nu]\rangle$ gets labeled with ψ. ■

This gives a decision procedure for model-checking:

> *Given a timed structure M and a* TCTL*-formula ϕ, first construct the region graph* R$(\mathcal{M}, \phi)$. *Then label all the regions with the subformulas of ϕ using the labeling procedure. The structure $\mathcal{M}$ satisfies the specification ϕ iff $\langle s_{init}, [\mathrm{C}^* \mapsto 0]\rangle$ is labeled with ϕ.*

5.3.5 Complexity of the algorithm

Using the ideas discussed above, one can implement an algorithm for model-checking which runs in time linear in the qualitative part, and exponential in the timing part of the input.

As in Lemma 3.27 of Section 3.3.2, we can show that the the number of clock regions of $[\mathrm{C}^* \mapsto \mathsf{R}]$ induced by $\sim$ is bounded by $[|\mathrm{C}^*|! \cdot 2^{|\mathrm{C}^*|} \cdot \Pi_{x \in \mathrm{C}^*}(2 \cdot c_x + 2)]$. Thus the number

of regions is $O[2^{|\Delta(\phi)|+|\Delta(\mathcal{M})|}]$; it is exponential in $|\Delta(\phi)|$, the length of the atomic timing formulas in ϕ, and also exponential in $|\Delta(\mathcal{M})|$, the length of the clock constraints (we assume binary encoding for the constants).

Now from the definition of the region graph, it follows that $|V_1| = O[|\mathrm{S}| \cdot 2^{|\Delta(\phi)|+|\Delta(\mathcal{M})|}]$. The first clause in the definition of E_1 contributes at most one edge for every vertex, and the second clause contributes at most three edges for every edge in E and a clock region of $[\mathrm{C}^* \mapsto \mathsf{R}]$. Hence, $|E_1| = O[(|\mathrm{S}| + |\mathrm{E}|) \cdot 2^{|\Delta(\phi)|+|\Delta(\mathcal{M})|}]$.

Thus the size of the region graph is exponential in the length of timing constraints of the given timed structure and the formula, but linear in the size of the state-transition graph.

Theorem 5.30 Given a timed structure M and a TCTL-formula ϕ, there is a decision procedure for checking whether or not $\mathcal{M} \models \phi$ which runs in time $O[|\phi| \cdot (|\mathrm{S}| + |\mathrm{E}|) \cdot 2^{|\Delta(\phi)|+|\Delta(\mathcal{M})|}]$.

PROOF. First construct the region graph $\mathrm{R}(\mathcal{M}, \phi) = \langle V_1, E_1 \rangle$. The successor class of any class can be computed in time $O[|\mathrm{C}^*|]$. Hence, $\mathrm{R}(\mathcal{M}, \phi)$ can be constructed in time $O[|V_1| + |E_1|]$. Then run the labeling algorithm on the subformulas of ϕ. The number of fairness constraints is $|\mathrm{C}^*|$. The vertices of $\mathrm{R}(\mathcal{M}, \phi)$ can be marked with a formula ψ in time $O[(|V_1| + |E_1|) \cdot |\mathrm{C}^*|]$, assuming they are already marked with the subformulas of ψ, using the labeling algorithm for $\mathrm{CTL}^{\mathrm{F}}$ [CES86]. So the labeling algorithm takes time $O[|\phi| \cdot |\mathrm{C}^*| \cdot (|V_1| + |E_1|)]$. The complexity bound follows from the bounds on $|V_1|$ and $|E_1|$. ■

Since we have shown that the model-checking problem is decidable, we can also characterize the complexity class of deciding finite satisfiability of TCTL-formulas.

Corollary 5.31 The problem of deciding whether a given TCTL-formula is finitely satisfiable, is complete for the class of recursively-enumerable problems (Σ_1-complete).

PROOF. The set of timed structures is enumerable. For any given timed structure M one can find whether or not M satisfies the given TCTL-formula. Consequently, the set of finitely-satisfiable TCTL-formulas is recursively enumerable. Undecidability was proved earlier in Theorem 5.12. ■

Note that since the set of satisfiable TCTL-formulas is not recursively enumerable, there must be some formulas which are satisfiable but not finitely satisfiable. Thus TCTL does not have the finite-model property.

5.3.6 Complexity of model-checking

The model-checking algorithm we considered, requires time exponential in the length of the timing constraints. Now we establish a lower bound for the problem, and show the problem to be PSPACE-complete.

Theorem 5.32 Given a timed structure M and a TCTL-formula ϕ, the problem of deciding whether or not $\mathcal{M} \models \phi$, is PSPACE-complete.

PROOF. [PSPACE-hardness] Theorem 4.23 asserts that the problem of testing whether an interval automaton has an infinite run is PSPACE-hard. This emptiness question for a timed structure can be phrased as a model-checking problem: a timed structure M has an infinite run iff $\mathcal{M} \models \exists\Box\,\mathbf{true}$. It follows that model-checking problem is PSPACE-hard.

[PSPACE-membership] We show that given a timed structure M and a TCTL-formula ϕ, the problem of deciding whether or not $\mathcal{M} \models \phi$ can be solved using space polynomial in the length of the input.

The region graph has size exponential in the length l of the input, and hence if we construct it fully, and label its vertices with the subformulas of ϕ, the algorithm will need space exponential in l. The standard way to save on the work-space is to compute the labels of the vertices as they are required. We sketch out another version of the labeling algorithm using this idea.

The main procedure of the algorithm is *label*(v, ψ), which returns *true* if v should be labeled with ψ else returns *false*. Let n be the maximum depth of the nesting of the path-quantifiers in ψ. We claim that a non-deterministic version of *label* can be implemented so as to use space $O[l \cdot n]$. This can be proved by an induction on the structure of ψ.

The cases $\psi \in$ AP, $\psi = (x + c \leq y + d)$, $\psi = \neg\psi'$, and $\psi = \psi_1 \wedge \psi_2$ are straightforward.

For $\psi = \exists\, \phi_1 \mathcal{U}\, \phi_2$, the procedure nondeterministically guesses a path $v \rightarrow v_1 \rightarrow \cdots v_m$. The path is guessed vertex by vertex, at each step checking that the newly guessed vertex is connected by an edge from the previous one. Furthermore, some $\mathcal{F}_p$-fair path should be accessible from v_m. The procedure checks that *label*(v_i, ψ_1) returns *true* for each $i < m$, *label*(v_m, ψ_2) returns *true*, and if v_m does not correspond to a boundary region then *label*(v_m, ψ_1) returns *true*. Notice that the procedure just needs to remember the current guess and the previous guess. This does involve recursive calls to *label*, so the total space required is $O[l]$ plus the space for *label* when called with ψ_1 or ψ_2 as the second argument. The claim follows by the inductive hypothesis.

Now consider the case $\psi = \forall\, \phi_1 \mathcal{U}\, \phi_2$. First observe that the negation of ψ can be written using only existential path-quantifiers,

$$\forall\, \phi_1 \mathcal{U}\, \phi_2 \;\leftrightarrow\; \neg[\; \exists(\neg\psi_2)\mathcal{U}\,(\neg\psi_1 \wedge \neg\psi_2) \;\vee\; \exists\Box\, \neg\psi_2 \;]$$

The procedure *label* is called recursively on each subformula of the above translation. Note that the rewriting does not increase the depth of the nesting of path quantifiers. The case $\psi = \exists\Box\, \psi'$ is handled similarly to the previous case.

By Savitch's theorem the deterministic version can be implemented in space $O[l^2 \cdot |\phi|^2]$.

■

5.3.7 TCTL with fairness

The model-checking algorithm can be modified to handle timed structures with fairness in a straightforward way. Recall that a timed structure M with fairness has an associated fairness family $\mathcal{F}$. A run r of M is $\mathcal{F}$-fair iff for every F $\in \mathcal{F}$, infinitely many states of r are in F. For the TCTL-structure $\mathcal{T}_{\mathcal{M}}$ associated with such a timed structure, the set of computations $f_{\mathcal{M}}$ consists only of those computations that correspond to the $\mathcal{F}$-fair runs of M.

The region graph R$(\mathcal{M}, \phi)$ is constructed as before. We only change the definition of fair paths through the region graph. A path $v_0 v_1 v_2 \ldots$, with $v_i = \langle s_i, \alpha_i \rangle$ for all $i \geq 0$, in the region graph is called $\mathcal{F}$-fair iff for every F $\in \mathcal{F}$, for infinitely many $i \geq 0$, $s_i \in$ F. Now while labeling a vertex with $\forall\, \phi_1 \mathcal{U}\, \phi_2$ or $\exists\, \phi_1 \mathcal{U}\, \phi_2$, we consider only those paths that are both $\mathcal{F}_p$-fair and $\mathcal{F}$-fair. The algorithm for CTL$^{\mathrm{F}}$ is used to compute the labels.

Chapter 6

Concluding Remarks

Summary

We have presented a theory for modeling, specifying, and verifying finite-state real-time systems. The focus of the thesis has been on decidability, complexity, and expressiveness issues. It shows that automated reasoning is plausible only if we restrict timing constraints to those comparing delays with constants. With this restriction, the simplifying assumption of discreteness is not required for checking correctness, we can use the more appealing dense-time model. The thesis also shows that the additional cost of introducing real-time in finite-state reasoning is a factor proportional to the magnitudes of the constants bounding the delays.

We have considered three different styles of specifications. The automata-theoretic approach uses an event-based model, and regards the verification problem as an inclusion problem between two descriptions of the system. In the temporal logic based approach, the process is modeled by an automaton generating timed state sequences. The logic MITL uses a linear-time semantics, whereas the logic TCTL uses a branching-time semantics. All three formalisms can express most of the interesting properties of real-time systems, but differ from each other in expressiveness. We have outlined an algorithm for verification in each case.

Towards "efficient" verification

We have tested toy examples of circuits and traffic controllers using our implementation of one of the verification algorithms. However, all algorithms presented in the thesis are of

exponential time complexity, and we have not suggested ways to cope with the PSPACE-hardness of verification problems. Now that we have a reasonable theory for real-time reasoning, and we understand the basic complexity issues related to automated reasoning about timing constraints, we should proceed to find ways to do verification more efficiently.

Search for heuristics to cope with the state-explosion problem has been an active area of research in automatic verification. Recently heuristics to implement model-checking without explicitly enumerating all the states have been proposed. For instance, [BCD$^+$90] proposes the use of binary decision diagrams to represent large state sets symbolically. Godefroid has proposed a scheme to avoid the state-explosion due to the modeling of concurrency by interleaving [God90, GW91]. Whether our methods can be applied in practice to verify complex systems, largely depends on the success of such attempts. We feel hopeful that the techniques designed for the qualitative case can be generalized to handle timing constraints also.

Apart from an efficient implementation of the verification algorithms, the other important aspect is to provide tools to specify complex systems. Timed automata is a fairly low-level representation, and automatic translations from more structured representations such as process algebras, timed Petri nets, or high-level real-time programming languages, should exist. Recently, Sifakis et.al. have shown how to translate a term of the real-time process algebra ATP to a timed automaton [NSY91].

Probabilistic verification

One promising direction of extending the work reported here is to incorporate probabilistic information in the process model. This is particularly relevant for systems that control and interact with physical processes. The computational models used in the theory of stochastic processes are physically realistic, but formal methods for verification of stochastic real-time systems have not been studied much.

In the simplest type of probabilistic model-checking, the state-transition graph is converted into a finite-state Markov chain by placing probabilities on the transitions [Var85, PZ86, VW86, CY88]. If the specification is presented as a formula of PTL or as a Büchi automaton, then the verification problem is to decide whether the sample paths of the Markov chain satisfy the specification property with probability 1. For CTL specifications, the semantics is redefined so that the path quantifiers in the logic, which previously meant "for all paths" and "there exists a path," are reinterpreted to mean "with probability one"

and "with positive probability," respectively.

We add probabilities to our model of timed automata by associating fixed distributions with the delays. Now we can express constraints like "the delay between the request and the response is distributed uniformly between 2 to 4 seconds". This extension makes our processes *generalized semi-Markov processes* (see [She87] for an introduction to the GSMP model). Our technique of constructing a region automaton by grouping the uncountably many configurations of the system into a finite number of equivalence classes, can be used to analyze the behavior of GSMPs.

In [ACD91a], we present an algorithm that combines model-checking for TCTL with model-checking for discrete-time Markov chains. The method can be adopted to check properties specified using deterministic timed automata also [ACD91b]. However, the problem of checking specifications presented as formulas of MITL or as nondeterministic timed automata is still open. A more ambitious problem is to compute estimates on the probability that a given process satisfies its logical specification.

Beyond verification

In this thesis we have addressed only the verification problem for systems modeled as timed automata. Clearly, questions other than verification can be studied using timed automata. For example, Wong-Toi and Hoffmann study the problem of supervisory control of discrete event systems when the plant and specification behaviors are represented by timed automata [WH91]. The problem of synthesizing schedulers from timed automata specifications is addressed in [DW90]. Courcoubetis and Yannakakis use timed automata to solve certain minimum and maximum delay problems for real-time systems [CY91]. For instance, they show how to compute the earliest and the latest time a target state can appear along the runs of an automaton given an initial state and an interpretation for clock values. A generalized version of this problem is to synthesize time bounds on delays so that a particular specification is met. Specifically, consider a timed automaton $\mathcal{A}$ whose timing constraints involve comparisons of clock values with unknown constants or parameters. The problem is to compute the set of values (or the extremal values with respect to some optimizing function) of these parameters for which $L(\mathcal{A})$ is empty. Several interesting questions about computing least restrictive bounds on delays, preserving some property of interest, can be formulated within this framework.

Bibliography

[ACD90] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990.

[ACD91a] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for probabilistic real-time systems. In *Automata, Languages and Programming: Proceedings of the 18th ICALP*, Lecture Notes in Computer Science 510, 1991.

[ACD91b] Rajeev Alur, Costas Courcoubetis, and David Dill. Verifying automata specifications of probabilistic real-time systems. In *Proccedings of REX workshop "Real-time: theory in practice"*, 1991.

[AD90] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Automata, Languages and Programming: Proceedings of the 17th ICALP*, Lecture Notes in Computer Science 443, pages 322–335. Springer-Verlag, 1990.

[AFH91] Rajeev Alur, Tomás Feder, and Thomas Henzinger. The benefits of relaxing punctuality. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 139–152, 1991.

[AH89] Rajeev Alur and Thomas Henzinger. A really temporal logic. In *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[AH90] Rajeev Alur and Thomas Henzinger. Real-time logics: complexity and expressiveness. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 390–401, 1990.

[AK83] S. Aggarwal and R.P. Kurshan. Modeling elapsed time in protocol specification. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification*, volume III, pages 51–62, 1983.

[AKS83] S. Aggarwal, R.P. Kurshan, and K. Sabnani. A calculus for protocol specification and validation. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification*, volume III, pages 19–34, 1983.

[AS89] Bowen Alpern and Fred Schneider. Verifying temporal properties without using temporal logic. *ACM Transactions on Programming Languages and Systems*, 11(1):147–167, 1989.

[BB91] J.C.M. Baeten and J.A. Bergstra. Real-time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[BCD+90] J.R. Burch, E.M. Clarke, D.L. Dill, L.J. Hwang, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[BCDM86] Michael Browne, Edmund Clarke, David Dill, and Bud Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time-dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, SE-17(3):259–273, 1991.

[BH81] Arthur Bernstein and Paul Harter. Proving real-time properties of programs with temporal logic. In *Proceedings of the Eighth ACM Symposium on Operating System Principles*, 1981.

[BKP86] Howard Barringer, Ruard Kuiper, and Amir Pnueli. A really abstract concurrent model and its temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 173–183, 1986.

[BMP81] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *Proceedings of the Eighth ACM Symposium on Principles of Programming Languages*, 1981.

[BS91] J.A. Brzozowski and C.J.H. Seger. Advances in asynchronous circuit theory, Part II: Bounded inertial delay models, MOS circuits, design techniques. 1991.

[Büc62] Richard Büchi. On a decision method in restricted second-order arithmetic. In *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science 1960*, pages 1–12. Stanford University Press, 1962.

[CDK89] E.M. Clarke, I.A. Draghicescu, and R.P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. Technical report, Carnegie Mellon University, 1989.

[CES86] Edmund Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal-logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[Cho74] Yaacov Choueka. Theories of automata on ω-tapes: a simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[CR83] J.E. Coolahan and N. Roussopoulos. Timing requirements for time-driven systems using augmented Petri-nets. *IEEE Transactions on Software Engineering*, SE-9(9):603–616, 1983.

[CY88] Costas Courcoubetis and Mihalis Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pages 338–345, 1988.

[CY91] Costas Courcoubetis and Mihalis Yannakakis. Minimum and maximum delay problems in real-time systems. In *Proceedings of the Third Workshop on Computer-Aided Verification, Aalborg University, Denmark*, 1991.

[DHW91] David Dill, Alan Hu, and Howard Wong-Toi. Checking for language inclusion using simulation relations. In *Proceedings of the Third Workshop on Computer-aided Verification, Aalborg University, Denmark*, 1991.

[Dil89] David Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407. Springer–Verlag, 1989.

[dR91] Willem-Paul de Roever, editor. *Real Time: Theory in Practice.* Lecture Notes in Computer Science. Springer-Verlag, 1991. To appear.

[DW90] David Dill and Howard Wong-Toi. Synthesizing processes and schedulers from temporal specifications. In *Proceedings of the Second Workshop on Computer-Aided Verification, Rutgers University*, 1990.

[EC82] E. Allen Emerson and Edmund M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EH82] E. Allen Emerson and Joseph Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 169–179, 1982.

[EL85] E.A. Emerson and C.L. Lei. Modalities for model-checking: Branching time logic strikes back. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 84–96, 1985.

[Eme90] E. Allen Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publishers, 1990.

[EMSS89] E. Allen Emerson, Aloysius Mok, A. Prasad Sistla, and Jai Srinivasan. Quantitative temporal reasoning. Presented at the First Workshop on Computer-aided Verification, Grenoble, France, 1989.

[God90] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the Second Workshop on Computer-Aided Verification, Rutgers University*, 1990.

[GW91] Patrice Godefroid and Pierre Wolper. A partial approach to model-checking. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science*, pages 406–415, 1991.

[Har88] Eyal Harel. Temporal analysis of real-time systems. Master's thesis, The Weizmann Institute of Science, Rehovot, Israel, 1988.

[Hen90] Thomas Henzinger. Half-order modal logic: how to prove real-time properties. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 281–296, 1990.

[Hen91] Thomas Henzinger. *Temporal Specification and Verification of Real-Time systems.* PhD thesis, Stanford University, 1991.

[HJ89] Hans Hansson and Bengt Jonsson. A framework for reasoning about time and reliability. In *Proceedings of the Tenth IEEE Real-Time Systems Symposium*, pages 102–111, 1989.

[HK90] Zvi Har'El and Robert Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 1990.

[HLP90] Eyal Harel, Orna Lichtenstein, and Amir Pnueli. Explicit-clock temporal logic. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 402–413, 1990.

[HMP91] Thomas Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for real-time systems. In *Proceedings of the 18th ACM Symposium on Principles of Programming Languages*, pages 353–366, 1991.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HPS83] David Harel, Amir Pnueli, and Jonathan Stavi. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 26:222–243, 1983.

[HU79] John Hopcroft and Jeff Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[JM86] Farnam Jahanian and Aloysius Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE–12(9):890–904, 1986.

[JM87] Farnam Jahanian and Aloysius Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8):961–975, 1987.

[JS88] Farnam Jahanian and Douglas Stuart. A method for verifying properties of modechart specifications. In *Proceedings of the Ninth IEEE Real-Time Systems Symposium*, pages 12–21, 1988.

[Koy90] Ron Koymans. Specifying real-time properties with metric temporal logic. *Journal of Real-Time Systems*, 2:255–299, 1990.

[Kur87] Robert Kurshan. Complementing deterministic Büchi automata in polynomial time. *Journal of Computer and System Sciences*, 35:59–71, 1987.

[KVdR83] Ron Koymans, Jan Vytopil, and Willem-Paul de Roever. Real-time programming and asynchronous message passing. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 187–197, 1983.

[LA90] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 265–280, 1990.

[Lam83] Leslie Lamport. What good is temporal logic? In R.E.A. Mason, editor, *Information Processing 83: Proceedings of the Ninth IFIP World Computer Congress*, pages 657–668. Elsevier Science Publishers, 1983.

[Lam91] Leslie Lamport. The temporal logic of actions. Technical report, DEC Systems Research Center, Palo Alto, California, 1991.

[Lew89] Harry Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical Report TR-15-89, Harvard University, 1989.

[Lew90] Harry Lewis. A logic of concrete time intervals. In *Proceedings of the Fifth IEEE Symposium on Logic in Computer Science*, pages 380–389, 1990.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, 1985.

[LPS82] Daniel Lehman, Amir Pnueli, and Jonathan Stavi. Impartiality, justice, and fairness: The ethics of concurrent termination. In *Automata, Languages and Programming: Proceedings of the Ninth ICALP*, Lecture Notes in Computer Science 115, pages 264–277. Springer-Verlag, 1982.

[LT87] Nancy Lynch and Mark Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 137–151, 1987.

[McN66] Robert McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92. Springer-Verlag, 1980.

[Mil83] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

[MP81] Zohar Manna and Amir Pnueli. The temporal framework for concurrent programs. In R.S. Boyer and J.S. Moore, editors, *The correctness problem in Computer science*, pages 215–274. Academic Press, 1981.

[MP89] Zohar Manna and Amir Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, Lecture Notes in Computer Science 354. Springer-Verlag, 1989.

[MT90] F. Moller and C. Tofts. A temporal calculus of communicating processes. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR 90: Theories of Concurrency*, Lecture Notes in Computer Science 458, pages 401–415. Springer-Verlag, 1990.

[Mul63] David Muller. Infinite sequences and finite machines. In *Proceedings of the Fourth IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 3–16, 1963.

[MW84] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.

[NRSV90] Xavier Nicollin, Jean-Luc Richier, Joseph Sifakis, and Jacques Voiron. ATP: an algebra for timed processes. In *Proceedings of the IFIP TC2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, 1990.

[NSY91] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. From ATP to timed graphs and hybrid systems. In *Proccedings of REX workshop "Real-time: theory in practice"*, 1991.

[OL82] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–595, 1982.

[Ost90a] Jonathan Ostroff. Deciding properties of timed transition models. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):17–183, 1990.

[Ost90b] Jonathan Ostroff. *Temporal Logic of Real-time Systems.* Research Studies Press, 1990.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[PH88] Amir Pnueli and Eyal Harel. Applications of temporal logic to the specification of real-time systems. In *Formal Techniques in Real-time and Fault-tolerant Systems*, Lecture Notes in Computer Science 331, pages 84–98. Springer-Verlag, 1988.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.

[Pnu86] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, Lecture Notes in Computer Science 224, pages 510–584. Springer-Verlag, 1986.

[PZ86] Amir Pnueli and Lenore Zuck. Probabilistic verification by tableaux. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.

[Ram74] Chander Ramchandani. Analysis of asynchronous concurrent systems by Petri nets. Technical Report MAC TR-120, Massachusetts Institute of Technology, 1974.

[Rog67] Hartley Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[RR88] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[Saf88] Shmuel Safra. On the complexity of ω-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pages 319–327, 1988.

[SC85] A. Prasad Sistla and Edmund Clarke. The complexity of propositional linear temporal logics. *The Journal of the ACM*, 32:733–749, 1985.

[She87] Gerald Shedler. *Regeneration and Networks of Queues.* Springer-Verlag, 1987.

[SVW87] A. Prasad Sistla, Moshe Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49, 1987.

[Tar72] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–170, 1972.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier Science Publishers, 1990.

[Var85] Moshe Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science*, pages 327–338, 1985.

[Var87] Moshe Vardi. Verification of concurrent programs – the automata-theoretic framework. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 167–176, 1987.

[VW86] Moshe Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science*, pages 332–344, 1986.

[Wan90] Yi Wang. Real time behavior of asynchronous agents. In *CONCUR 90: Theories of Concurrency*, Lecture Notes in Computer Science 458, pages 502–520. Springer-Verlag, 1990.

[WH91] Howard Wong-Toi and Girard Hoffmann. The control of dense real-time discrete event systems. 1991.

[Wol83] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[WVS83] Pierre Wolper, Moshe Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.

[YKT91] Tomohiro Yoneda, Yutaka Kondo, and Yoshihiro Tohma. On the acceleration of timing verification method based on time Petri nets. 1991.

[Zwa88] Amy Zwarico. *Timed Acceptances: An Algebra of Time Dependent Computing.* PhD thesis, University of Pennsylvania, 1988.