The Theory of Timed I/O Automata

The Theory of Timed I/O Automata

Dilsun K. Kaynar and Nancy Lynch, Roberto Segala and Frits Vaandrager

www.morganclaypool.com

159829010X paper Kaynar/Lynch/Segala/Vaandrager
1598290118 ebook Kaynar/Lynch/Segala/Vaandrager

DOI 10.2200/S00006ED1V01Y200508CSL001

A Publication in the Morgan & Claypool Publishers' series
SYNTHESIS LECTURES ON COMPUTER SCIENCE
Lecture #1

First Edition

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

The Theory of Timed I/O Automata

Dilsun K. Kaynar and Nancy Lynch
MIT Computer Science and Artificial Intelligence Laboratory

Roberto Segala
Dipartimento di Informatica
Università di Verona

Frits Vaandrager
Institute for Computing and Information Sciences
Radboud University Nijmegen

SYNTHESIS LECTURES ON COMPUTER SCIENCE #1

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

This monograph presents the *timed input/output automaton (TIOA)* modeling framework, a basic mathematical framework to support description and analysis of timed (computing) systems. Timed systems are systems in which desirable correctness or performance properties of the system depend on the timing of events, not just on the order of their occurrence. Timed systems are employed in a wide range of domains including communications, embedded systems, real-time operating systems, and automated control. Many applications involving timed systems have strong safety, reliability, and predictability requirements, which makes it important to have methods for systematic design of systems and rigorous analysis of timing-dependent behavior.

An important feature of the TIOA framework is its support for decomposing timed system descriptions. In particular, the framework includes a notion of *external behavior* for a TIOA, which captures its discrete interactions with its environment. The framework also defines what it means for one TIOA to *implement* another, based on an inclusion relationship between their external behavior sets, and defines notions of *simulations*, which provide sufficient conditions for demonstrating implementation relationships. The framework includes a *composition* operation for TIOAs, which respects external behavior, and a notion of *receptiveness*, which implies that a TIOA does not block the passage of time.

KEYWORDS

Formal modeling and verification, I/O automata, Timed computing systems.

Contents

The Authors

DILSUN KAYNAR

Dilsun Kaynar is a Postdoctoral Research Associate in the Theory of Distributed Systems Group at MIT's Computer Science and Artificial Intelligence Laboratory. She received the PhD degree from the University of Edinburgh at the Laboratory for Foundations of Computer Science and the BSc degree in computer engineering from METU in Turkey. The broad area of her research is the specification, programming, and verification of distributed computing systems. Her PhD work focused on the design of functional programming languages that support mobile computation. She investigated the application of type-based analysis in this context, in particular to improve safety and security of systems. In her postdoctoral research she has been working on the development of I/O automata-based formal modeling frameworks for distributed systems, with collaborators including Nancy Lynch, Roberto Segala, and Frits Vaandrager.

NANCY LYNCH

Nancy Lynch is a Professor in the Department of Electrical Engineering and Computer Science at MIT and heads the Theory of Distributed Systems research group in MIT's Computer Science and Artificial Intelligence Laboratory. Prior to joining MIT in 1981, she served on the faculty at Tufts University, the University of Southern California, Florida International University, and Georgia Tech. She received the BS degree in mathematics from Brooklyn College, and the PhD degree in mathematics from MIT. She has written numerous research articles about distributed algorithms and impossibility results, and about formal modeling and verification of distributed systems. Her notable research contributions include the well-known "FLP" impossibility result for distributed consensus in the presence of process failures (with Fischer and Paterson), the "DLS" algorithms for stabilizing fault-tolerant consensus (with Dwork and Stockmeyer), and the I/O automata mathematical modeling frameworks (with Tuttle, Vaandrager, Segala, and Kaynar). Prior to this monograph, she has written two books: *Atomic Transactions* (with Merritt, Weihl, and Fekete) and *Distributed Algorithms*. She is a member of the National Academy of Engineering and an ACM Fellow.

ROBERTO SEGALA

Roberto Segala is a Professor at the University of Verona, Italy, and heads the Formal Models and Verification group at the Department of Computer Science. Prior to joining the University of Verona in 2001, he was a Research Associate at the University of Bologna. He received his Laurea in computer science from the University of Pisa as a student of the Scuola Normale Superiore, and the Master's and PhD degrees in computer science from MIT. As part of his PhD work, he made contributions to the theory of liveness and receptiveness for real-time systems and he designed the model of Probabilistic Automata for the formal analysis of randomized distributed algorithms. After that, he worked with Lynch, Kaynar, Vaandrager, and others on the hybrid extension of the I/O automata framework. He also worked on model checking of probabilistic real-time systems, contributing to the design of some of the algorithms used in the PRISM model checker. One of his long-term goals is to design a general mathematical model that can be used for the description and analysis of systems that exhibit stochastic hybrid behavior.

FRITS VAANDRAGER

Frits Vaandrager is a Professor at the Radboud University Nijmegen, the Netherlands, where he heads the Informatics for Technical Applications Group at the Institute of Computing and Information Sciences. Prior to joining the Radboud University in 1995, he was a group leader at the CWI in Amsterdam and held postdoctoral positions at MIT in the group of Nancy Lynch and in the group of Gérard Berry at the École Nationale Supérieure des Mines in Sophia-Antipolis. He received the M.S. degree in mathematics from the University of Leiden, and the PhD degree in computer science from the University of Amsterdam. As part of his PhD work, he made major contributions to the general theory of structural operational semantics. After that he worked with Lynch, Segala, Kaynar, and others on the theory and applications of the I/O automata framework. He also has a strong interest in model checking techniques for timed systems and coordinates a European project (AMETIST) in this area. One of his long-term research objectives is to help to give the new discipline of *(computer-based) system engineering* a sound mathematical basis.

Acknowledgments

Dilsun Kaynar and Nancy Lynch were supported by DARPA/AFOSR MURI contract F49620-02-1-0325, DARPA SEC contract F33615-01-C-1850, NSF ITR contract CCR-0121277, and Air Force Aerospace Research-OSR contract F49620-00-1-0097. Frits Vaandrager was supported by EU IST project IST-2001-35304 (Advanced Methods for Timed Systems, AMETIST) and PROGRESS project TES4999 (Verification of Hard and Softly Timed Systems, HaaST).

Notations

a, b	action
f, g, h	function
i, j	index
l	locally controlled action
t	time point
v, x	variable
A	set of actions
C	task
E	set of external actions
F	set of functions
H	set of internal (hidden) actions
I	set of input actions
J	interval
K	set of time points
L	set of locally controlled actions
O	set of output actions
P	set of elements in cpo
Q	set of automaton states
R	(simulation) relation
S	set
T	set of trajectories
V	set of variables
X	set of internal variables
$\mathbf{x}$	state
$\mathbf{v}$	valuation
$\mathcal{A}, \mathcal{B}, \mathcal{C}$	timed (I/O) automaton
$\mathcal{D}$	set of discrete transitions
$\mathcal{T}$	set of trajectories
N	natural numbers
R	real numbers
T	time axis

Z	integers
V	universe of variables
α, β, δ	(A, V)-sequence
γ	sequence
λ	the empty sequence
π	projection function
σ, ρ	sequence
τ, υ	trajectory
Θ	set of start states

CHAPTER 1
Introduction

1.1 OVERVIEW

Timed computing systems are systems in which desirable correctness or performance properties of the system depend on the timing of events, not just on the order of their occurrence. A typical timed system consists of computer components, which operate in discrete steps, and timing-related components such as physical or logical clocks, whose behavior involve continuous transformation over time. Timed systems are employed in a wide range of domains including communications, embedded systems, real-time operating systems, and automated control. Many applications involving timed systems have strong safety, reliability, and predictability requirements, which makes it important to have methods for systematic design of systems and rigorous analysis of timing-dependent behavior.

Modeling plays a key role in all stages in the design and analysis of systems. Models represent system designs at a level of abstraction that is suitable for isolating and focusing on their most crucial aspects. They can be modified and experimented with more easily than real implementations. Moreover, if the modeling is performed using the concepts provided by a formal framework, the modeling can be done more precisely, and analysis and verification methods supported by that framework can be applied. Timed systems, which combine discrete steps with continuous evolution of state over time, exhibit complex behaviors that are typically hard to describe and analyze in the absence of a carefully developed modeling framework [1–3].

A modeling framework must support designing systems in structured ways, viewing them at multiple levels of abstraction, and as compositions of interacting components. If a framework is to provide flexibility and generality, it must also support nondeterminism. A system designer might wish to allow several potential behaviors at certain points in the computation of a system, for example, to avoid making assumptions about how the environment will behave, or to allow several correct implementations for the same design. Such liberty in specification would not be possible to accommodate without nondeterminism. In addition to supporting all of these features, modeling frameworks for timed systems must provide mechanisms for representing continuously evolving components such as clocks and timers.

An interesting complication that arises in modeling timed systems is that time can progress in ways that conflict with our intuition about physical time. For example, we may force time to stop entirely to "urge" some discrete action to happen, or schedule infinitely many discrete actions to happen in a finite amount of time. A framework needs to provide concepts that identify the conditions under which a timed system behaves according to our intuitions, that is, the conditions under which time diverges as the system continues to run.

In this monograph, we introduce a basic mathematical framework—the *timed input/output automaton* modeling framework—to support description and analysis of timed systems. In this framework, a system is represented as a *timed I/O automaton (TIOA)*, which is a kind of nondeterministic, possibly infinite-state, state machine. The state of a TIOA is described by a valuation of state variables that are internal to the automaton. The state of a TIOA can change in two ways: instantaneously by the occurrence of a *discrete transition*, which is labeled by a discrete action, or according a *trajectory*, which is a function that describes the evolution of the state variables over intervals of time. Trajectories may be continuous or discontinuous functions.

The TIOA framework supports decomposition of system description and analysis. A key to this decomposition is the rigorously defined notion of *external behavior* for timed I/O automata. The external behavior of each TIOA is defined by a simple mathematical object called a *trace*—essentially, a sequence of actions interspersed with time-passage steps. *Abstraction* and *parallel composition* are other important notions for decomposition of system description and analysis.

For abstraction, the framework includes notions of *implementation* and *simulation*, which can be used to view timed systems at multiple levels of abstraction, starting from a high-level version that describes required properties and ending with a low-level version that describes a detailed design or implementation. In particular, the TIOA framework defines what it means for one TIOA, $\mathcal{A}$, to *implement* another TIOA, $\mathcal{B}$, namely, any trace that can be exhibited by A is also allowed by B. In this case, $\mathcal{A}$ might be more deterministic than $\mathcal{B}$, in terms of either discrete transitions or trajectories. For instance, $\mathcal{B}$ might be allowed to perform an output action at an arbitrary time before noon, whereas $\mathcal{A}$ produces the same output sometime between 10 and 11 A.M. The notion of a *simulation relation* from $\mathcal{A}$ to $\mathcal{B}$ provides a sufficient condition for demonstrating that $\mathcal{A}$ implements $\mathcal{B}$. A simulation relation is defined to satisfy three conditions: one relating start states, one relating discrete transitions, and one relating trajectories of $\mathcal{A}$ and $\mathcal{B}$.

For parallel composition, the framework provides a *composition operation*, by which TIOAs modeling individual timed system components can be combined to produce a model for a larger timed system. The model for the composed system can describe interactions among the components, which involves joint participation in discrete transitions. Composition requires certain "compatibility" conditions, namely, that each output action be controlled by at most one automaton, and that internal actions of one automaton cannot be shared by any other automaton. The composition operation respects traces, for example, if $\mathcal{A}_1$ implements $\mathcal{A}_2$, then the composition of $\mathcal{A}_1$ and $\mathcal{B}$ implements the composition of $\mathcal{A}_2$ and $\mathcal{B}$. Composition also satisfies *projection* and *pasting* results, which are fundamental for compositional design and verification of systems: a trace of a composition of TIOAs "projects" to give traces of the individual TIOAs and traces of components are "pastable" to give behaviors of the composition.

If a TIOA approaches a finite point in time without quite reaching it, or by scheduling infinitely many discrete actions to happen in a finite amount of time, it is said to exhibit *Zeno behavior*, in reference to Zeno's paradox [4]. The TIOA framework includes a notion of *receptiveness*, which is used to classify automata that do not contribute to producing behavior and which is preserved by composition. Receptiveness of a TIOA, $\mathcal{A}$, in the TIOA framework is defined in terms of the existence of a strategy, which is defined as a subautomaton of $\mathcal{A}$ that chooses some of the evolutions from each state of $\mathcal{A}$.

The TIOA framework presented in this work is purely mathematical. However, it constitutes a natural basis for computer support tools, which are currently under development [5].

1.2 EVOLUTION OF THE TIOA FRAMEWORK

The TIOA modeling framework presented in this work has evolved from the *hybrid input/output automaton (HIOA)* modeling framework for hybrid systems by Lynch *et al.* [6]. Our approach is based on the assumption that a timed system can be viewed as a special kind of a hybrid system where the continuous transformation is limited to internal system components that determine the timing of events. Therefore, we define a TIOA as a restricted HIOA where the only essential difference between an HIOA and a TIOA is that an HIOA may have *external variables* to model the continuous information flowing into and out of the system, in addition to state variables. A major consequence of this definition is that the communication between TIOAs is restricted to shared-action communication only. The TIOA model does not impose any further restrictions on the expressive power of the HIOA model.

We have undertaken the project of developing this new modeling framework even though there are several timed automaton models that extend the basic I/O automaton model [7–10], because we have observed that the new HIOA modeling framework offered a way of improving and simplifying previous work on TIOA models [8–10]. For example, the use of trajectories as first-class objects to represent the external behavior of a timed automaton, the definition of a strategy as an automaton rather than a two-player game, and the variable structure on states are all new features that were motivated by what we learned in developing the HIOA framework and that gave rise to more elegant definitions and simpler proofs for timed automata.

We intend the TIOA model to serve as a general semantic framework in which previous results for TIOA [7–10] and other related models [11–14] can be recast in a style that is upwardly compatible with the new HIOA model. Limiting the communication to discrete interactions is an apt choice since the previous TIOA automation models also adopt this type of communication. On the other hand, by avoiding any further restrictions on the general hybrid model, we obtain an expressive model suitable for specifying complex timing behavior. For example, our model does not require variables to be either discrete or to evolve at the same rate as real-time as in some other models [11, 13]. Consequently, algorithms such as clock synchronization algorithms that use local clocks evolving at different and varying rates can be formalized naturally in our framework.

The fact that HIOAs subsume TIOAs as a special class does not eliminate the need for having a separate modeling, framework for timed systems. First, having no external variables in the TIOA model gives rise to considerable simplifications in the theory. For example, proving that the composition of two timed automata is a well-defined automaton becomes simpler in the absence of external variables; no extra compatibility conditions as in the general HIOA framework are needed to obtain the desirable composition theorems for TIOAs.

Second, we believe that focusing on the TIOA model presented in this monograph is compatible with our long-term goal of developing a unified I/O automaton model that can address timing-dependent, probabilistic, and general hybrid behavior in a common framework. We are planning to start out with a probabilistic model with discrete interactions only, and then extend the model to handle timing-dependent behavior, and only at later stages consider continuous interactions. It would be harder to integrate probabilistic mechanisms into the full hybrid model than it would be to integrate them into the TIOA model presented here.

1.3 RELATED WORK

There are several formalisms and tools for timed systems that are based on automata and state transition models. In this section, we briefly introduce those lines of work that we think are most closely related to ours. Note that we do not focus on the toolsets and their capabilities, but rather on the underlying formal models and languages.

One of the widely used formal frameworks for timed systems is that of Alur–Dill timed automata [11, 15]. An Alur–Dill automaton is a finite directed multigraph augmented with a finite set of clock variables. The semantics of such a timed automaton are defined as a state transition system in which each state consists of a location and a clock valuation. Clocks are assumed to change with the same rate as real-time, that is, with rate 1. Timed automata accept timed languages consisting of sequences of events tagged with their occurrence times. Decision problems such as universality and language inclusion are undecidable for timed automata. Recently, a version of timed automata called perturbed automata has been presented [16]. The clocks in perturbed timed automata can change at a rate within the interval $[1 - \epsilon, 1 + \epsilon]$, where ϵ is a given perturbation error. It has been shown that the language inclusion problem is decidable for systems modeled as products of perturbed automata each of which has a single clock.

The aim of facilitating automated verification seems to have motivated the restrictions on the expressive power of the model. The timed automaton model presented in this work is more expressive than the model of Alur–Dill automata. In our model, there are no finiteness assumptions and no restrictions imposed on the dynamic types of variables. Alur–Dill timed automata have been extensively studied with a formal language theoretic-view [17]. Our focus, on the other hand, has been to develop a general formal framework with a well-defined notion of external behavior, parallel composition, and abstraction that supports reasoning with simulation relations.

Uppaal [13, 18] is a widely used modeling and verification tool for timed systems. It supports the description of systems as a network of Alur–Dill timed automata and enhances that model with CCS-style communication [19] along with other notions such as committed and urgent locations. Uppaal also supports (synchronous) broadcast communication and communication via shared variables. Uppaal has a sophisticated model-checker that explores the whole state space of the modeled system to verify timing properties. Therefore, finiteness assumptions are built into the model to make such verification possible and the operations on clocks are restricted. Uppaal can be used as a model-checker for restricted TIOAs. We have done some preliminary work in this direction [20].

It would be interesting to work on formal semantics for Uppaal based on some variation of our restricted HIOA model. There are several small mismatches due to the style of communication and notions such as committed locations. It remains to be seen to what extent we can use the communication mechanisms of our automata to model these formally. We could, for example, allow a nonempty set of external variables with restricted dynamic types and seek restrictions on the use of shared variables in Uppaal, which would allow us to view these variables as external variables in the HIOA sense.

Kronos [21, 22] is another verification tool for timed systems that uses Alur–Dill automata. This tool requires systems to be represented as timed automata and the correctness conditions to be expressed in the real-time temporal logic TCTL [23]. Kronos, as Uppaal, can perform model-checking using a symbolic representation of the infinite state space by sets of linear constraints. Kronos can model check full TCTL and implements the symbolic algorithm developed by [24]. It would be possible to use Kronos as a model-checker for restricted TIOAs.

The IF notation, which is the intermediate representation used in the IF toolset [25], is based on Alur–Dill automata extended with discrete data variables, communication primitives, dynamic process creation, and destruction. This notation has been designed such that it can serve as a target for the translation of higher level modeling languages, such as real-time extensions of SDL and UML. The support for dynamic process creation and destruction appears to be a distinguishing feature of the IF notation.

A slight generalization of Alur–Dill timed automata are the linear hybrid automata of [26]. In this model, apart from clocks that progress with rate 1, one can also use continuous variables whose derivatives are contained in some arbitrary interval. A well-known model checking tool for linear hybrid automata is HyTech [27], which uses symbolic manipulation techniques as in Uppaal and Kronos. The input language of HyTech can be translated into our TIOA model, to apply TIOA verification methods. Likewise, TIOAs whose continuous variables conform to the linearity conditions of HyTech could be verified using model-checking capabilities of HyTech.

The TIOA modeling framework presented in this monograph can be used to express models that use lower and upper time bounds on tasks or actions [7, 12]. Our framework includes an operation for adding time bounds on a subset of the actions of a timed automaton. As a result of this operation, lower bounds are transformed to appropriate preconditions for transitions and upper bounds are transformed to stopping conditions for trajectories.

An interesting timed automaton model called "Clock GTA" has been introduced in [14]. The model was used for describing algorithms that behave in accordance with their timing constraints in certain intervals but may exhibit timing failures for some other intervals. The possibility of expressing such an ability turns out to be crucial for performance and fault-tolerance analysis for practical algorithms [14, 28]. We are interested in finding a systematic way of describing such behavior with our new TIOA model.

1.4 ORGANIZATION OF THE BOOK

The rest of this book is organized as follows. Chapter 2 contains mathematical preliminaries. Chapter 3 defines notions that are useful for describing the behavior of timed systems, most importantly, trajectories and timed sequences. Chapter 4 defines *timed automata (TAs)*, which contain all of the structure of TIOAs except for the classification of external actions as inputs or outputs. It also defines external behavior for TAs and implementation and simulation relationships between TAs. Chapter 5 presents composition and hiding operations for TAs, along with operations for adding bounds that relate TAs to other timed automaton models. Chapter 6 defines TIOAs by adding an input/output classification to TAs and extends the theory of TAs to TIOAs. It also defines special kinds of TIOAs such as progressive and receptive TIOAs. Chapter 7 presents compositionality results for TIOAs in general, and for the special classes of progressive and receptive TIOAs. Finally, Chapter 8 presents some conclusions and discusses future work. Examples are included throughout.

CHAPTER 2

Mathematical Preliminaries

In this chapter, we present the basic mathematical definitions and notation that will be used as a foundation for our definitions of timed automata and TIOA. These definitions involve functions, sequences, partial orders, and untimed automata.

2.1 FUNCTIONS AND RELATIONS

If f is a function, then we denote the domain and range of f by $dom(f)$ and $range(f)$, respectively. If S is a set, then we write $f \lceil S$ for the restriction of f to S, that is, the function g with $dom(g) = dom(f) \cap S$ such that $g(c) = f(c)$ for each $c \in dom(g)$.

We say that two functions f and g are *compatible* if $f \lceil dom(g) = g \lceil dom(f)$. If f and g are compatible functions, then we write $f \cup g$ for the unique function h with $dom(h) = dom(f) \cup dom(g)$ satisfying the condition: for each $c \in dom(h)$, if $c \in dom(f)$ then $h(c) = f(c)$ and if $c \in dom(g)$ then $h(c) = g(c)$. More generally, if F is a set of pairwise compatible functions, then we write $\bigcup F$ for the unique function h with $dom(h) = \bigcup\{dom(f) \mid f \in F\}$ satisfying the condition: for each $f \in F$ and $c \in dom(f)$, $h(c) = f(c)$.

If f is a function whose range is a set of functions and S is a set, then we write $f \downarrow S$ for the function g with $dom(g) = dom(f)$ such that $g(c) = f(c) \lceil S$ for each $c \in dom(g)$. The restriction operation $\downarrow$ is extended to sets of functions by pointwise extension. Also, if f is a function whose range is a set of functions, all of which have a particular element d in their domain, then we write $f \downarrow d$ for the function g with $dom(g) = dom(f)$ such that $g(c) = f(c)(d)$ for each $c \in dom(g)$.

We say that two functions f and g whose ranges are sets of functions are *pointwise compatible* if for each $c \in dom(f) \cap dom(g)$, $f(c)$ and $g(c)$ are compatible. If f and g have the same domain and are pointwise compatible, then we denote by $f \dot{\cup} g$ the function h with $dom(h) = dom(f)$ such that $h(c) = f(c) \cup g(c)$ for each c.

A relation over sets X and Y is defined to be any subset of $X \times Y$. If R is a relation, then we denote the domain and range of R by $dom(R)$ and $range(R)$, respectively. A relation over X and Y is *total* over X if $dom(R) = X$. If R is a relation over X and Y, and $x \in X$, we define $R(x) = \{y \in Y \mid (x, y) \in R\}$. We say that a relation R over X and Y is *image-finite* if for each $x \in X$, $R(x)$ is finite.

2.2 SEQUENCES

Let S be any set. A *sequence* σ over S is a function from a downward-closed subset of $\mathsf{Z}^{>0}$ to S. Thus, the domain of a sequence is either the set of all positive integers, or is of the form $\{1, \dots, k\}$ for some k. In the first case we say that the sequence is infinite, and in the second case it is finite. We use $|\sigma|$ to denote the cardinality of $dom(\sigma)$. The sets of finite and infinite sequences over S are denoted by S^* and S^ω, respectively. *Concatenation* of a finite sequence ρ with a finite or infinite sequence σ is denoted by $\rho \frown \sigma$. The *empty sequence*, that is, the sequence with the empty domain is denoted by λ. The sequence containing one element $c \in S$ is abbreviated as c. We say that a sequence σ is a *prefix* of a sequence ρ, denoted by $\sigma \leq \rho$, if $\sigma = \rho \lceil dom(\sigma)$. Thus, $\sigma \leq \rho$ if either $\sigma = \rho$ or σ is finite and $\rho = \sigma \frown \sigma'$ for some sequence σ'. If σ is a nonempty sequence, then $head(\sigma)$ denotes the first element of σ and $tail(\sigma)$ denotes σ with its first element removed. Moreover, if σ is finite, then $last(\sigma)$ denotes the last element of σ and $init(\sigma)$ denotes σ with its last element removed. Let σ and σ' be sequences over S. Then σ' is a *subsequence* of σ provided that there exists a monotone increasing function $f\colon dom(\sigma') \to dom(\sigma)$ such that $\sigma'(i) = \sigma(f(i))$ and $f(i+1) = f(i) + 1$ for all $i \in dom(\sigma')$. If $1 \leq j_1 \leq j_2 \leq |\sigma|$, then we define $\sigma(j_1, \dots, j_2)$ to be the subsequence of σ obtained by extracting the elements in positions $j_1, \dots, j_2$; that is, σ' is the subsequence obtained from function f of length $j_2 - j_1 + 1$, where $f(i) = i + j_1 - 1$ for all $i \in dom(\sigma')$.

2.3 PARTIAL ORDERS

We recall some basic definitions and results regarding partial orders and, in particular, complete partial orders (cpos) from [29, 30]. A *partial order* is a set S together with a binary relation $\sqsubseteq$ that is reflexive, antisymmetric, and transitive. In the sequel, we usually denote posets by the set S without explicit mention to the binary relation $\sqsubseteq$.

A subset $P \subseteq S$ is *bounded (above)* if there is a $c \in S$ such that $d \sqsubseteq c$ for each $d \in P$; in this case, c is an *upper bound* for P. A *least upper bound (lub)* for a subset $P \subseteq S$ is an upper bound c for P such that $c \leq d$ for every upper bound d for P. If P has a lub, then it is necessarily unique, and we denote it by $\bigsqcup P$. A subset $P \subseteq S$ is *directed* if every finite subset Q of P has an upper bound in P. A poset S is *complete*, and hence is a *complete partial order (cpo)* if every directed subset P of S has a lub in S.

We say that $P' \subseteq S$ *dominates* $P \subseteq S$, denoted by $P \sqsubseteq P'$, if for every $c \in P$ there is some $c' \in P'$ such that $c \sqsubseteq c'$. We use the following two simple lemmas, adapted from [30], Lemmas 3.1.1 and 3.1.2].

Lemma 2.1 *If P, P' are directed subsets of a cpo S and $P \sqsubseteq P'$, then $\bigsqcup P \sqsubseteq \bigsqcup P'$.*

Lemma 2.2 *Let $P = \{c_{ij} \mid i \in I, j \in J\}$ be a doubly indexed subset of a cpo S. Let P_i denote the set $\{c_{ij} \mid j \in J\}$ for each $i \in I$. Suppose*

1. *P is directed,*
2. *each P_i is directed with lub c_i, and*
3. *the set $\{c_i \mid i \in I\}$ is directed.*

Then $\sqcup P = \sqcup\{c_i \mid i \in I\}$.

A finite or infinite sequence of elements, $c_0 \, c_1 \, c_2 \cdots$, of a partially ordered set $(S, \sqsubseteq)$ is called a *chain* if $c_i \sqsubseteq c_{i+1}$ for each nonfinal index i. We define the *limit* of the chain, $\lim_{i\to\infty} c_i$, to be the lub of the set $\{c_0, c_1, c_2, \ldots\}$ if S contains such a bound; otherwise, the limit is undefined. Since a chain is a special case of a directed set, each chain of a cpo has a limit.

A function $f : S \to S'$ between posets S and S' is *monotone* if $f(c) \sqsubseteq f(d)$ whenever $c \sqsubseteq d$. If f is monotone and P is a directed set, then the set $f(P) = \{f(c) \mid c \in P\}$ is directed as well. If f is monotone and $f(\bigsqcup P) = \bigsqcup f(P)$ for every directed P, then f is said to be *continuous*.

An element c of a cpo S is *compact* if, for every directed set P such that $c \sqsubseteq \bigsqcup P$, there is some $d \in P$ such that $c \sqsubseteq d$. We define $K(S)$ to be the set of compact elements of S. A cpo S is *algebraic* if every $c \in S$ is the lub of the set $\{d \in K(S) \mid d \sqsubseteq c\}$. A simple example of an algebraic cpo is the set of finite or infinite sequences over some given domain, equipped with the prefix ordering. Here the compact elements are the finite sequences.

2.4 A BASIC GRAPH LEMMA

We require the following lemma, a slight generalization of König's Lemma [31]. If G is a directed graph, then a *root* of G is defined to be a node with no incoming edges.

Lemma 2.3 *Let G be an infinite directed graph that satisfies the following properties:*

1. *G has finitely many roots.*
2. *Each node of G has finite outdegree.*
3. *Each node of G is reachable from some root of G.*

Then, there is an infinite path in G starting from some root.

Proof: An extension of the usual proof of König's Lemma [31]. □

CHAPTER 3

Describing Timed System Behavior

In this chapter, we present the basic definitions that are useful for describing discrete and continuous changes to the system's state. The key notions are *static* and *dynamic types* for variables, *trajectories*, and *hybrid sequences*. Most of the material in this chapter comes from the paper on the HIOA modeling framework [6]. The reader may refer [6] for the proofs that are not included here.

3.1 TIME

Throughout this chapter, we fix a *time axis* T, which is a subgroup of $(R, +)$, the real numbers with addition. We assume that every infinite, monotone, bounded sequence of elements of T has a limit in T. The reader may find it convenient to think of T as the set R of real numbers, but the set Z of integers and the singleton set $\{0\}$ are also examples of allowed time axes. We define $\mathsf{T}^{\geq 0} \triangleq \{t \in \mathsf{T} \mid t \geq 0\}$.

An *interval* J is a nonempty, convex subset of T. We denote intervals as usual: $[t_1, t_2] = \{t \in \mathsf{T} \mid t_1 \leq t \leq t_2\}$, $[t_1, t_2) = \{t \in \mathsf{T} \mid t_1 \leq t < t_2\}$, etc. An interval J is *left-closed* (*right-closed*) if it has a minimum (resp., maximum) element and is *left-open* (*right-open*) otherwise. It is *closed* if it is both left-closed and right-closed. We write $\min(J)$ and $\max(J)$ for the minimum and maximum elements, respectively, of an interval J (if they exist), and $\inf(J)$ and $\sup(J)$ for the infimum and supremum, respectively, of J in $\mathsf{R} \cup \{-\infty, \infty\}$. For $K \subseteq \mathsf{T}$ and $t \in \mathsf{T}$, we define $K + t \triangleq \{t' + t \mid t' \in K\}$. Similarly, for a function f with domain K, we define $f + t$ to be the function with domain $K + t$ satisfying, for each $t' \in K + t$, $(f + t)(t') = f(t' - t)$.

In some definitions and theorems in this chapter when we use R as the time domain we assume that the relation $\leq$ on R extends to a relation on $\mathsf{R} \cup \{\infty\}$ such that $\infty \leq \infty$ and for all $t \in \mathsf{R}$, $t < \infty$.

3.2 STATIC AND DYNAMIC TYPES

We assume a universal set V of *variables*. A variable represents a location within the state of a system. For each variable v, we assume both a *(static) type*, which gives the set of values it may take on, and a *dynamic type*, which gives the set of trajectories it may follow. Formally, for each variable v we assume the following:

1. $type(v)$, the *(static) type* of v. This is a nonempty set of values.
2. $dtype(v)$, the *dynamic type* of v. This is a set of functions from left-closed intervals of T to $type(v)$ that satisfies the following properties.

   a) *Closure under time shift*: For each $f \in dtype(v)$ and $t \in \mathsf{T}$, $f + t \in dtype(v)$.
   b) *Closure under subinterval*: For each $f \in dtype(v)$ and each left-closed interval $J \subseteq dom(f)$, $f \lceil J \in dtype(v)$.
   c) *Closure under pasting*: Let $f_0\, f_1\, f_2 \cdots$ be a sequence of functions in $dtype(v)$ such that, for each nonfinal index i, $dom(f_i)$ is right-closed and $\max(dom(f_i)) = \min(dom(f_{i+1}))$. Then the function f defined by $f(t) \triangleq f_i(t)$, where i is the smallest index such that $t \in dom(f_i)$, is in $dtype(v)$.

Example 3.1 (Discrete variables). Let v be any variable and let *Constant* be the set of constant functions from a left-closed interval of T to $type(v)$. Then *Constant* is closed under time shift and subinterval. If the dynamic type of v is obtained by closing *Constant* under the pasting operation, then v is called a *discrete* variable. This is essentially the same as the definition of a discrete variable in [12].

Example 3.2 (Analog variables). Assume that $\mathsf{T} = \mathsf{R}$. Let v be any variable whose static type is an interval of R and *Continuous* be the set of continuous functions from a left-closed interval of T to $type(v)$. Then *Continuous* is closed under time shift and subinterval. If the dynamic type of v is obtained by closing *Continuous* under the pasting operation, then v is called an *analog* variable. Fig. 3.1 shows an example of a function f in the dynamic type of an analog

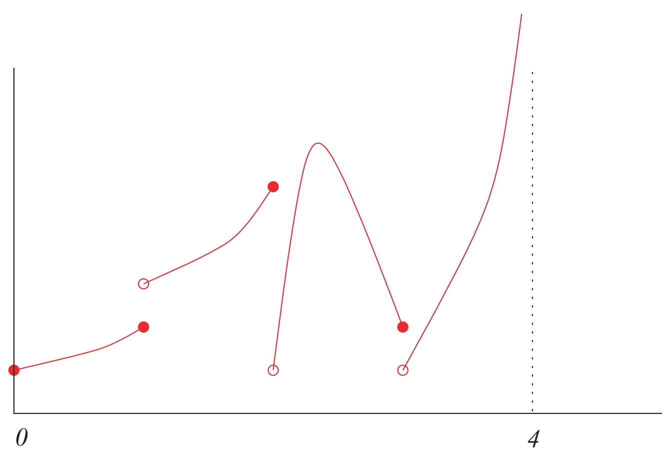

FIGURE 3.1: Example of a function in the dynamic type of an analog variable.

variable. Function f is defined on the interval $[0, 4)$ and is obtained by pasting together four pieces. At the boundary points between these pieces, f takes the value specified by the leftmost piece, which makes f continuous from the left. Note that f is undefined at time 4.

Example 3.3 (Standard real-valued function classes). If we take $\mathsf{T} = \mathsf{R}$ and $type(v) = \mathsf{R}$, then other examples of dynamic types can be obtained by taking the pasting closure of standard function classes from real analysis, the set of differentiable functions, the set of functions that are differentiable k times (for any k), the set of smooth functions, the set of integrable functions, the set of L^p functions (for any p), the set of measurable locally essentially bounded functions [32], or the set of all functions.

Standard function classes are closed under time shift and subinterval, but not under pasting. A natural way of defining a dynamic type is as the pasting closure of a class of functions that is closed under time shift and subinterval. In such a case, it follows that the new class is closed under all three operations.

3.3 TRAJECTORIES

In this section, we define the notion of a *trajectory*, define operations on trajectories, and prove simple properties of trajectories and their operations. A trajectory is used to model the evolution of a collection of variables over an interval of time.

3.3.1 Basic Definitions

Let V be a set of variables, that is, a subset of V. A *valuation* $\mathbf{v}$ for V is a function that associates with each variable $v \in V$ a value in $type(v)$. We write $val(V)$ for the set of valuations for V. Let J be a left-closed interval of T with left endpoint equal to 0. Then a J*-trajectory* for V is a function $\tau : J \rightarrow val(V)$, such that for each $v \in V$, $\tau \downarrow v \in dtype(v)$. A *trajectory* for V is a J-trajectory for V, for any J. We write $trajs(V)$ for the set of all trajectories for V. If Q is a set of valuations for some set V of variables, we write $trajs(Q)$ for the set of all trajectories whose range is a subset of Q.

A trajectory for V where $V = \emptyset$ is simply a function from a time interval to the special function with the empty domain. Thus, the only interesting information represented by such a trajectory is the length of the time interval that constitutes the domain of the trajectory. We use trajectories over the empty set of variables when we wish to capture the amount of time-passage, but abstract away the evolution of variables.

A trajectory for V with domain $[0, 0]$ is called a *point* trajectory for V. If $\mathbf{v}$ is a valuation for V then $\wp(\mathbf{v})$ denotes the point trajectory for V that maps 0 to $\mathbf{v}$. We say that a J-trajectory is *finite* if J is a finite interval, *closed* if J is a (finite) closed interval, *open* if J is a right-open interval, and *full* if $J = \mathsf{T}^{\geq 0}$. If T is a set of trajectories, then $finite(T)$, $closed(T)$, $open(T)$, and $full(T)$ denote the subsets of T consisting of all the finite, closed, open, and full trajectories in T, respectively.

If τ is a trajectory then $\tau.ltime$, the *limit time* of τ, is the supremum of $dom(\tau)$. We define $\tau.fval$, the *first valuation* of τ, to be $\tau(0)$, and if τ is closed, we define $\tau.lval$, the *last valuation* of τ, to be $\tau(\tau.ltime)$. For τ a trajectory and $t \in \mathsf{T}^{\geq 0}$, we define

$$\tau \trianglelefteq t \triangleq \tau \lceil [0, t],$$
$$\tau \triangleleft t \triangleq \tau \lceil [0, t),$$
$$\tau \trianglerighteq t \triangleq (\tau \lceil [t, \infty)) - t.$$

Note that, since dynamic types are closed under time shift and subintervals, the result of applying the above operations is always a trajectory, except when the result is a function with an empty domain. By convention, we also write $\tau \trianglelefteq \infty \triangleq \tau$ and $\tau \triangleleft \infty \triangleq \tau$.

3.3.2 Prefix Ordering

Trajectory τ is a *prefix* of trajectory υ, denoted by $\tau \leq \upsilon$, if τ can be obtained by restricting υ to a subset of its domain. Formally, if τ and υ are trajectories for V, then $\tau \leq \upsilon$ iff $\tau = \upsilon \lceil dom(\tau)$. Alternatively, $\tau \leq \upsilon$ iff there exists a $t \in \mathsf{T}^{\geq 0} \cup \{\infty\}$ such that $\tau = \upsilon \trianglelefteq t$ or $\tau = \upsilon \triangleleft t$. If $\tau \leq \upsilon$, then clearly $dom(\tau) \subseteq dom(\upsilon)$. If T is a set of trajectories for V, then $pref(T)$ denotes the *prefix closure* of T, defined by

$$pref(T) \triangleq \{\tau \in trajs(V) \mid \exists \upsilon \in T : \tau \leq \upsilon\}.$$

We say that T is *prefix closed* if $T = pref(T)$.

The following lemma gives a simple domain-theoretic characterization of the set of trajectories over a given set V of variables:

Lemma 3.4 *Let V be a set of variables. The set $trajs(V)$ of trajectories for V, together with the prefix ordering $\leq$, is an algebraic cpo. Its compact elements are the closed trajectories.*

3.3.3 Concatenation

The concatenation of two trajectories is obtained by taking the union of the first trajectory and the function obtained by shifting the domain of the second trajectory until the start time agrees with the limit time of the first trajectory; the last valuation of the first trajectory, which may not be the same as the first valuation of the second trajectory, is the one that appears in the concatenation. Formally, suppose τ and τ' are trajectories for V, with τ closed. Then the *concatenation* $\tau \frown \tau'$ is the function given by

$$\tau \frown \tau' \triangleq \tau \cup (\tau' \lceil (0, \infty) + \tau.ltime).$$

Because dynamic types are closed under time shift and pasting, it follows that $\tau \frown \tau'$ is a trajectory for V. Observe that $\tau \frown \tau'$ is finite (resp., closed, full) iff τ' is finite (resp., closed, full). Observe also that concatenation is associative.

The following lemma, which is easy to prove, shows the close connection between concatenation and the prefix ordering.

Lemma 3.5 *Let τ and υ be trajectories for V with τ closed. Then*

$$\tau \leq \upsilon \quad \Leftrightarrow \quad \exists \tau' : \upsilon = \tau \frown \tau'.$$

Note that if $\tau \leq \upsilon$, then the trajectory τ' such that $\upsilon = \tau \frown \tau'$ has an arbitrary value for $\tau'.\mathit{fval}$ and the remainder of the trajectory is unique. Note also that the $\Leftarrow$ implication in Lemma 3.5 would not hold if the first valuation of the second argument, rather than the last valuation of the first argument, were used in the concatenation.

We extend the definition of concatenation to any (finite or countably infinite) number of arguments. Let $\tau_0\, \tau_1\, \tau_2 \cdots$ be a (finite or infinite) sequence of trajectories such that τ_i is closed for each nonfinal index i. Define trajectories $\tau'_0, \tau'_1, \tau'_2, \ldots$ inductively by

$$\begin{aligned} \tau'_0 &\triangleq \tau_0, \\ \tau'_{i+1} &\triangleq \tau'_i \frown \tau_{i+1} \text{ for nonfinal } i. \end{aligned}$$

Lemma 3.5 implies that for each nonfinal i, $\tau'_i \leq \tau'_{i+1}$. We define the *concatenation* $\tau_0 \frown \tau_1 \frown \tau_2 \cdots$ to be the limit of the chain $\tau'_0\, \tau'_1\, \tau'_2 \cdots$; existence of this limit follows from Lemma 3.4.

3.4 HYBRID SEQUENCES

In this section, we introduce the notion of a *hybrid sequence*, which is used to model a combination of changes that occur instantaneously and changes that occur over intervals of time. Our definition is parameterized by a set A of *actions*, which are used to model instantaneous changes and instantaneous synchronizations with the environment, and a set V of *variables*, which are used to model changes over intervals of time. We also define some special kinds of hybrid sequences and some operations on hybrid sequences, and give basic properties.

3.4.1 Basic Definitions

Fix a set A of actions and a set V of variables. An (A, V)-*sequence* is a finite or infinite alternating sequence $\alpha = \tau_0\, a_1\, \tau_1\, a_2\, \tau_2 \cdots$, where

1. each τ_i is a trajectory in $\mathit{trajs}(V)$,
2. each a_i is an action in A,

3. if α is a finite sequence, then it ends with a trajectory, and
4. if τ_i is not the last trajectory in α, then τ_i is closed.

A *hybrid sequence* is an (A, V)-sequence for some A and V.

Since the trajectories in a hybrid sequence can be point trajectories our notion of hybrid sequence allows a sequence of discrete actions to occur at the same real time, with corresponding changes of variable values. An alternative approach is described in [33], where state changes at a single real time are modeled using a notion of "superdense time." Specifically, hybrid behavior is modeled in [33] using functions from an extended time domain, which includes countably many elements for each real time, to states.

If α is a hybrid sequence, with notation as above, then we define the *limit time* of α, $\alpha.ltime$, to be $\sum_i \tau_i.ltime$. A hybrid sequence α is defined to be

- *time bounded* if $\alpha.ltime$ is finite.
- *admissible* if $\alpha.ltime = \infty$.
- *closed* if α is a finite sequence and its final trajectory is closed.
- *Zeno* if α is neither closed nor admissible, that is, if α is time bounded and is either an infinite sequence, or else a finite sequence ending with a trajectory whose domain is right-open.
- *non-Zeno* if α is not Zeno.

For any hybrid sequence α, we define the *first valuation* of α, $\alpha.fval$, to be $head(\alpha).fval$. Also, if α is closed, we define the *last valuation* of α, $\alpha.lval$, to be $last(\alpha).lval$, that is, the last valuation in the final trajectory of α.

If α is a closed (A, V)-sequence, where $V = \emptyset$ and $\beta \in trajs(\emptyset)$, we call $\alpha \frown \beta$ a *time-extension* of α.

3.4.2 Prefix Ordering

We say that (A, V)-sequence $\alpha = \tau_0\, a_1\, \tau_1 \cdots$ is a *prefix* of (A, V)-sequence $\beta = \upsilon_0\, b_1\, \upsilon_1 \cdots$, denoted by $\alpha \leq \beta$, provided that (at least) one of the following holds:

1. $\alpha = \beta$.
2. α is a finite sequence ending in some τ_k; $\tau_i = \upsilon_i$ and $a_{i+1} = b_{i+1}$ for every i, $0 \leq i < k$; and $\tau_k \leq \upsilon_k$.

Similar to the set of trajectories over V, the set of (A, V)-sequences is also an algebraic cpo.

Lemma 3.6 *Let V be a set of variables and A a set of actions. The set of (A, V)-sequences, together with the prefix ordering $\leq$, is an algebraic cpo. Its compact elements are the closed (A, V)-sequences.*

3.4.3 Concatenation

Suppose α and α' are (A, V)-sequences with α closed. Then the *concatenation* $\alpha \frown \alpha'$ is the (A, V)-sequence given by

$$\alpha \frown \alpha' \triangleq \mathit{init}(\alpha)\ (\mathit{last}(\alpha) \frown \mathit{head}(\alpha'))\ \mathit{tail}(\alpha').$$

(Here, *init*, *last*, *head*, and *tail* are ordinary sequence operations.)

Lemma 3.7 *Let α and β be (A, V)-sequences with α closed. Then*

$$\alpha \leq \beta \quad \Leftrightarrow \quad \exists \alpha' : \beta = \alpha \frown \alpha'.$$

Note that if $\alpha \leq \beta$, then the (A, V)-sequence α', such that $\beta = \alpha \frown \alpha'$, is unique except that it has an arbitrary value in $\mathit{val}(V)$ for $\alpha'.\mathit{fval}$.

As we did for trajectories, we extend the concatenation definition for (A, V)-sequences to any finite or infinite number of arguments. Let $\alpha_0\, \alpha_1\, \ldots$ be a finite or infinite sequence of (A, V)-sequences such that α_i is closed for each nonfinal index i. Define (A, V)-sequences $\alpha'_0, \alpha'_1, \ldots$ inductively by

$$\begin{aligned} \alpha'_0 &\triangleq \alpha_0, \\ \alpha'_{i+1} &\triangleq \alpha'_i \frown \alpha_{i+1} \text{ for nonfinal } i. \end{aligned}$$

Lemma 3.7 implies that for each nonfinal i, $\alpha'_i \leq \alpha'_{i+1}$. We define the *concatenation* $\alpha_0 \frown \alpha_1 \cdots$ to be the limit of the chain $\alpha'_0\, \alpha'_1\, \cdots$; existence of this limit is ensured by Lemma 3.6.

3.4.4 Restriction

Let A and A' be sets of actions and let V and V' be sets of variables. The (A', V')-*restriction* of an (A, V)-sequence α, denoted by $\alpha \lceil (A', V')$, is obtained by first projecting all trajectories of α on the variables in V', then removing the actions not in A', and finally concatenating all adjacent trajectories. Formally, we define the (A', V')-restriction first for closed (A, V)-sequences and then extend the definition to arbitrary (A, V)-sequences using a limit construction. The definition for closed (A, V)-sequences is by induction on the length of those sequences:

$$\tau \lceil (A', V') = \tau \downarrow V' \text{ if } \tau \text{ is a single trajectory,}$$

$$\alpha\ a\ \tau \lceil (A', V') = \begin{cases} (\alpha \lceil (A', V'))\ a\ (\tau \downarrow V') & \text{if } a \in A', \\ (\alpha \lceil (A', V')) \frown (\tau \downarrow V') & \text{otherwise.} \end{cases}$$

It is easy to see that the restriction operator is monotone on the set of closed (A, V)-sequences. Hence, if we apply this operation to a directed set, the result is again a directed set. Together with Lemma 3.6, this allows us to extend the definition of restriction to arbitrary (A, V)-sequences by

$$\alpha \lceil (A', V') = \sqcup\{\beta \lceil (A', V') \mid \beta \text{ is a closed prefix of } \alpha\}.$$

The next four lemmas state some basic properties of the restriction operation.

Lemma 3.8 *(A', V')-restriction is a continuous operation.*

Lemma 3.9 $(\alpha_0 \frown \alpha_1 \frown \cdots) \lceil (A, V) = \alpha_0 \lceil (A, V) \frown \alpha_1 \lceil (A, V) \frown \cdots$.

Lemma 3.10 $(\alpha \lceil (A, V)) \lceil (A', V') = \alpha \lceil (A \cap A', V \cap V')$.

Lemma 3.11 *Let α be a hybrid sequence, A a set of actions and V a set of variables.*

1. *α is time bounded iff $\alpha \lceil (A, V)$ is time bounded.*
2. *α is admissible iff $\alpha \lceil (A, V)$ is admissible.*
3. *If α is closed, then $\alpha \lceil (A, V)$ is closed.*
4. *If α is non-Zeno, then $\alpha \lceil (A, V)$ is non-Zeno.*

Example 3.12 (A Zeno execution with a closed (A, V)-restriction). In order to understand why in Lemma 3.11 we have an implication in only one direction in points 3 and 4, consider the Zeno sequence α of the form $\wp(\mathbf{v})\ a\ \wp(\mathbf{v})\ a\ \wp(\mathbf{v}) \cdots$. Let A be a set such that $a \notin A$ and let V consist of the variables in $dom(\mathbf{v})$. Obviously, $\alpha \lceil (A, V)$, which is $\wp(\mathbf{v})$, is closed, and hence also non-Zeno. This shows that the fact that $\alpha \lceil (A, V)$ is closed (resp., non-Zeno) does not imply that α is closed (resp., non-Zeno).

CHAPTER 4

Timed Automata

In this chapter, as a preliminary step toward defining TIOA, we define a slightly more general *timed automaton* model. In timed automata, actions are classified as external or internal, but external actions are not further classified as input or output (the input/output distinction is discussed in Chapter 6). We define how timed automata execute and define implementation and simulation relations between timed automata.

4.1 DEFINITION OF TIMED AUTOMATA

A timed automaton is a state machine whose states are divided into *variables* and that has a set of discrete *actions*, some of which may be internal and some external. The state of a timed automaton may change in two ways: by *discrete transitions*, which change the state atomically, and by *trajectories*, which describe the evolution of the state over intervals of time. The discrete transitions are labeled with actions; this will allow us to synchronize the transitions of different timed automata when we compose them in parallel. The evolution described by a trajectory may be described by continuous or discontinuous functions.

Formally, a *timed automaton (TA)* $\mathcal{A} = (X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$ consists of the following:

- A set X of *internal variables*.
- A set $Q \subseteq val(X)$ of *states*.
- A nonempty set $\Theta \subseteq Q$ of *start states*.
- A set E of *external actions* and a set H of *internal actions*, disjoint from each other. We write $A \triangleq E \cup H$.
- A set $\mathcal{D} \subseteq Q \times A \times Q$ of *discrete transitions*. We use $\mathbf{x} \xrightarrow{a}_{\mathcal{A}} \mathbf{x}'$ as short for $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}$. We sometimes drop the subscript and write $\mathbf{x} \xrightarrow{a} \mathbf{x}'$, when we think $\mathcal{A}$ should be clear from the context. We say that a is *enabled* in $\mathbf{x}$ if $\mathbf{x} \xrightarrow{a} \mathbf{x}'$ for some $\mathbf{x}'$. We say that a set C of actions is enabled in a state $\mathbf{x}$ if some action in C is enabled in $\mathbf{x}$.
- A set $\mathcal{T} \subseteq trajs(Q)$ of trajectories. Given a trajectory $\tau \in \mathcal{T}$ we denote $\tau.fval$ by $\tau.fstate$ and, if τ is closed, we denote $\tau.lval$ by $\tau.lstate$. When $\tau.fstate = \mathbf{x}$ and

$\tau.lstate = \mathbf{x}'$, we write $\mathbf{x} \xrightarrow{\tau}_{\mathcal{A}} \mathbf{x}'$. We require that the following axioms hold:

T0 (Existence of point trajectories). If $\mathbf{x} \in Q$, then $\wp(\mathbf{x}) \in \mathcal{T}$.
T1 (Prefix closure). For every $\tau \in \mathcal{T}$ and every $\tau' \leq \tau$, $\tau' \in \mathcal{T}$.
T2 (Suffix closure). For every $\tau \in \mathcal{T}$ and every $t \in dom(\tau)$, $\tau \trianglerighteq t \in \mathcal{T}$.
T3 (Concatenation closure). Let $\tau_0\, \tau_1\, \tau_2\, \cdots$ be a sequence of trajectories in $\mathcal{T}$ such that, for each nonfinal index i, τ_i is closed and $\tau_i.lstate = \tau_{i+1}.fstate$. Then $\tau_0 \frown \tau_1 \frown \tau_2 \cdots \in \mathcal{T}$.

A timed automaton is essentially a hybrid automaton in the sense of [6] in which W, the set of external variables, is empty. Apart from that, the only difference is the addition of Axiom **T0**, a small restriction that does not affect any of the results of [6] but that we need to prove Theorem 7.7. Axioms **T1–T3** express some natural further conditions on the set of trajectories that we need to construct our theory. A key part of this theory is a parallel composition operation for timed automata. In a composed system, any trajectory of any component automaton may be interrupted at any time by a discrete transition of another (possibly independent) component automaton. Axiom **T1** ensures that the part of the trajectory up to the discrete transition is a trajectory, and Axiom **T2** ensures that the remainder is a trajectory. Axiom **T3** is required because the environment of a timed automaton, as a result of its own internal discrete transitions, may change its dynamics repeatedly, and the automaton must be able to follow this behavior.

Our definition of a timed automaton differs from previous definitions of timed automata [8, 10] in two major respects. First, the states are structured using variables, which have dynamic types with specific closure properties. The variable structure is convenient for writing specifications and the dynamic types are useful in analyzing continuous evolution of the state. Second, the set of trajectories is defined as an explicit component of an automaton. In the previous definitions, time-passage was represented by special time-passage actions and trajectories were defined implicitly, as auxiliary functions describing the effects of time-passage actions on states.

Notation: We often denote the components of a TA $\mathcal{A}$ by $X_{\mathcal{A}}$, $Q_{\mathcal{A}}$, $\Theta_{\mathcal{A}}$, $E_{\mathcal{A}}$, etc., and the components of a TA $\mathcal{A}_i$ by X_i, Q_i, Θ_i, E_i, etc. We sometimes omit these subscripts, where no confusion seems likely. In examples we typically specify sets of trajectories using differential and algebraic equations and inclusions. Below we explain a few notational conventions that help us in doing this. Suppose the time domain T is R, τ is a (fixed) trajectory over some set of variables V, and $v \in V$. With some abuse of notation, we use the variable name v to denote the function $\tau \downarrow v$ in $dom(\tau) \rightarrow type(v)$, which gives the value of v at all times during trajectory τ. That is, for all $t \in dom(\tau)$, we have $v(t) = (\tau \downarrow v)(t) = \tau(t)(v)$. Similarly, we view any expression e containing variables from V as a function with domain $dom(\tau)$. Suppose that v is a variable and e is a real-valued expression containing variables from V. Using these conventions we can say, for example, that τ satisfies the algebraic equation

$$v = e$$

which means that, for every $t \in dom(\tau)$, $v(t) = e(t)$, that is, the constraint on the variables expressed by the equation $v = e$ holds for each state on trajectory τ. Now suppose that e, when viewed as a function, is integrable. Then we say that τ satisfies

$$d(v) = e$$

if, for every $t \in dom(\tau)$, $v(t) = v(0) + \int_0^t e(t')\,dt'$. Equivalently, for every $t_1, t_2 \in dom(\tau)$ such that $t_1 \leq t_2$, $v(t_2) = v(t_1) + \int_{t_1}^{t_2} e(t')\,dt'$. Note that this interpretation of the differential equation makes sense even at points where v is not differentiable. A similar interpretation of differential equations is used by Polderman and Willems [34], who call functions defined in this way as "weak solutions."

We generalize this notation to handle inequalities as well as equalities. Suppose that v is a variable and e is a real-valued expression containing variables from V. The inequality

$$e \leq v$$

means that, for every $t \in dom(\tau)$, $e(t) \leq v(t)$. That is, the constraint expressed by the inequality $e \leq v$ holds for each state of trajectory τ. Similarly, the inequality

$$v \leq e$$

means that, for every $t \in dom(\tau)$, $v(t) \leq e(t)$. Now suppose that e is integrable when viewed as a function. Then we say that τ satisfies

$$e \leq d(v)$$

if, for every $t_1, t_2 \in dom(\tau)$ such that $t_1 \leq t_2$, $v(t_1) + \int_{t_1}^{t_2} e(t')\,dt' \leq v(t_2)$, and τ satisfies

$$d(v) \leq e$$

if, for every $t_1, t_2 \in dom(\tau)$ such that $t_1 \leq t_2$, $v(t_2) \leq v(t_1) + \int_{t_1}^{t_2} e(t')\,dt'$.

Conventions for automata specifications: In all the examples given in this monograph we assume the time axis T to be R and specify timed automata by using a variant of the TIOA language presented in [35–38].

An automaton specification consists of four main parts: a signature, which lists the actions along with their kinds (external or internal) and parameter types, a state variables list, which declares the names and types of state variables, a collection of transition definitions, and a trajectories definition.

Unless specified otherwise, the set of states of an automaton equals the set of all valuations of its state variables. Static types of variables are always declared explicitly in the state variables list. For example, we write `v:t` for a variable `v` of static type `t`. Moreover, a variable can be initialized to a specific value allowed by its type. For example, in order to initialize the variable `v` above to the value `val`, we write `v:t := val`. If no initial value is specified, it is assumed to be arbitrary. The state variables list in an automaton specification can be followed by an **initially** clause, which consists of a predicate that constrains the automaton parameters and initial values of state variables. All of the static types used in the examples have standard interpretations, except possibly for the type **AugmentedReal**, which denotes $\mathsf{R} \cup \{\infty\}$.

The dynamic types of variables are specified implicitly. By default, variables of type `Real` are assumed to be analog and variables of types other than `Real` are assumed to be discrete. The definition of what it means for a variable to be discrete or analog is given in Examples 1 and 2. The keyword **discrete** is used to qualify a discrete variable of type `Real`. Although timed automata may contain variables that are neither discrete nor analog, none of our examples use such variables.

The transitions are specified in precondition-effect style. A **pre** clause specifies the enabling condition for an action. An **eff** clause contains a list of statements that specify the effect of performing that action on the state. All the statements in an effect clause are assumed to be executed sequentially in a single indivisible step. The absence of a specified precondition for an action means that the action is always enabled and the absence of a specified effect means that performing the action does not change the state.

The trajectories are specified using a combination of algebraic and differential equations and inequalities, and stopping conditions. A trajectory belongs to the set of legal trajectories of an automaton if it satisfies the stopping condition expressed by the **stop when** clause and the equations or inequalities in the **evolve** clause. The stopping condition is satisfied by a trajectory if the only state in which the condition holds is the last state of that trajectory. That is, time cannot advance beyond the point where the stopping condition is true. The **evolve** clause specifies the algebraic and differential equations that must be satisfied by the trajectories. We write $\mathbf{d}(v) = e$ for $d(v) = e$, $\mathbf{d}(v) \leq e$ for $d(v) \leq e$, and $e \leq \mathbf{d}(v)$ for $e \leq d(v)$. We assume that the evolution of each variable follows a continuous function throughout a trajectory. This implies that the value of a discrete variable is constant throughout a trajectory: time-passage does not change the value of discrete variables.

Example 4.1 (Time-bounded channel). The automaton `TimedChannel(b, M)` in Fig. 4.1 is the specification of a reliable FIFO channel that delivers its messages within a certain time

```
automaton TimedChannel(b: Real, M: Type) where b ≥ 0
type Packet = tuple of message: M, deadline: Real
  signature
     external send(m: M), receive(m: M)
  states
     queue: Queue[Packet] := {},
     now: Real := 0
  transitions
     external send(m)
       eff
         queue := append([m,now+b],queue)
     external receive(m)
       pre
         head(queue).message = m
       eff
         queue := tail(queue)
  trajectories
     stop when
       ∃p: Packet p ∈ queue ∧ (now = p.deadline)
     evolve
       d(now) = 1
```

FIGURE 4.1: Time-bounded channel

bound, represented by the automaton parameter `b` of type `Real`, which is nonnegative. The other automaton parameter `M` is an arbitrary type parameter that represents the type of messages communicated by the channel.

The variable `queue` is used to hold a sequence of pairs consisting of a message that has been sent and its delivery deadline. The variable `now` is used to describe real time. Every `send(m)` transition adds to the queue a new pair whose first component is `m` and second component is the deadline `now` + `b`. A `receive(m)` transition can occur only when `m` is the first message in the queue and it results in the removal of the first message from the queue.

The trajectory specification shows that the variable `now` increases with rate `1`, that is, at the same rate as real time. The stopping condition implies that, within a trajectory, time cannot pass beyond the point where `now` becomes equal to the delivery deadline of some message in the queue.

Example 4.2 (Periodic sending process). The automaton `PeriodicSend(u, M)` in Fig. 4.2 is the specification of a process that sends messages periodically, every `u` time units, where `u` is an automaton parameter of type `Real`, which is nonnegative. The type parameter `M` represents the type of the messages sent by the process.

```
automaton PeriodicSend(u: Real, M: Type) where u ≥ 0
  signature
    external send(m: M)
  states
    clock: Real := 0
  transitions
    external send(m)
      pre
        clock = u
      eff
        clock := 0
  trajectories
    stop when
      clock = u
    evolve
      d(clock) = 1
```

FIGURE 4.2: Periodic sending process

The analog variable `clock` is a timer whose value records the amount of time that has elapsed since it was last reset to `0`. A `send(m)` transition can occur only when `clock` = `u`, and it causes `clock` to be reset. The trajectory specification says that `clock` increases at the same rate as real time and time cannot pass beyond the point where `clock` = `u`.

Example 4.3 (Periodic sending process with failures). The specification of the `PeriodicSend` process from Example 4.2 does not model failures. We now consider a variant of `PeriodicSend` where the process may fail and stop doing any discrete actions. The specification of this new automaton is given in Fig. 4.3.

The discrete variable `failed` in automaton `PeriodicSend2` is a boolean flag that records whether the process fails. It is initialized to `false` and is set to `true` when a `fail` action occurs. The trajectory specification of `PeriodicSend2` shows that time can advance without any bound when the process fails.

Example 4.4 (Timeout process). The automaton `Timeout` in Fig. 4.4 is the specification of a process that awaits the receipt of a message from another process. If `u` time units elapse without such a message arriving, `Timeout` performs a `timeout` action, thereby "suspecting" the other process. When a message arrives it "unsuspects" the other process. `Timeout` may suspect and unsuspect repeatedly.

The discrete variable `suspected` is a flag that shows whether `Timeout` suspects that the other process fails. The variable `clock` is a timer that records the amount of time that has elapsed

```
automaton PeriodicSend2(u: Real,M: Type) where u ≥ 0
  signature
    external send(m: M), fail
  states
    failed: Bool := false,
    clock: Real := 0
  transitions
    external send(m)
      pre
        ¬failed ∧ clock = u
      eff
        clock := 0
    external fail
      eff
        failed:= true
  trajectories
    stop when
      ¬failed ∧ clock = u
    evolve
      d(clock) = 1
```

FIGURE 4.3: Periodic sending process with failures

```
automaton Timeout(u:Real, M: Type) where u > 0
  signature
    external receive(m: M), timeout
  states
    suspected: Bool := false,
    clock Real := 0
  transitions
    external receive(m)
      eff
        clock:=0;
        suspected:= false
    external timeout
      pre
        ¬suspected ∧ clock = u
      eff
        suspected := true
  trajectories
    stop when
      clock = u and ¬suspected
    evolve
      d(clock) = 1
```

FIGURE 4.4: Timeout

```
type Index = enumeration of p1, p2, p3, p4

type PcValue =  enumeration of rem, test, set, check,
                               leavetry, crit, reset, leaveexit

automaton FischerME(u_set, l_check: Real)
 where u_set ≥ 0 ∧ l_check ≥ 0 ∧ u_set < l_check
 signature
  external try(i:Index), crit(i:Index), exit(i:Index), rem(i:Index)
  internal test(i:Index), set(i:Index),
           check(i:Index), reset(i:Index)

 states
   x: Null[Index] := nil,
   pc: Array[Index,PcValue] := constant(rem),
   lastset: Array[Index,discrete AugmentedReal] := constant(infty),
   firstcheck: Array[Index,discrete AugmentedReal] := constant(0),
   now: Real:=0
```

FIGURE 4.5: Fischer's mutual exclusion algorithm: Signature and states

since the receipt of the last message. A `receive(m)` transition can occur at any time; this causes the variable `clock` to be reset and the flag `suspected` to be set to `false`. If `clock` reaches `u` before the arrival of a message, then the `timeout` action becomes enabled. The process sets `suspected` to `true` as a result of a `timeout`.

The trajectory specification shows that `clock` increases at the same rate as real time and if `suspected` = `false`, then time cannot go beyond the point where `clock` = `u`. Note that if `suspected` = `true`, there is no restriction on the amount of time that can elapse.

Example 4.5 (Fischer's algorithm). The timed automaton `FischerME` presented in Figs. 4.5 and 4.6 is the specification of a shared memory mutual exclusion algorithm that uses a single shared variable that can be read and written by all the participants. We fix here the number of participants to be four, by defining `Index` to be an enumeration consisting of four elements. Note, however, that this specification can be generalized to any finite number of participants.

The automaton parameters `u_set` and `l_check` represent upper and lower time bounds for the `set(i)` and `check(i)` actions, respectively. We assume that `u_set` < `l_check`.

The shared variable `x` can be assigned any value of type `Index` plus one additional special value `nil`. If a process is in the critical region, then the variable `x` contains the index of that process. If all users are in the remainder region, then the variable `x` contains the value `nil`. The array variable `pc` records the program counters of all processes. The array variable `lastset` keeps track of the deadlines by which the processes' `set` actions must occur. Similarly, the array

```
transitions
  external try(i)
    pre
      pc[i] = rem
    eff
      pc[i]:= test
  internal test(i)
    pre
      pc[i] = test
    eff
      if x = nil then
         pc[i] := set;
         lastset[i]:=now+u_set
  internal set(i)
    pre
      pc[i] = set
    eff
      x := embed(i);
      pc[i] := check;
      lastset[i] := infty;
      firstcheck[i]:= now + l_check
  internal check(i)
    pre
      pc[i] = check ∧
            now ≥ firstcheck[i]
    eff
      if x = embed(i) then pc[i] := leavetry
      else pc[i] := test

  external crit(i)
    pre
      pc[i] = leavetry
    eff
      pc[i] := crit
  external exit(i)
    pre
      pc[i] = crit
    eff
      pc[i] := reset

  internal reset(i)
    pre
      pc[i] = reset
    eff
      x := nil;
      pc[i] := leaveexit

  external rem(i)
    pre
      pc[i] = leaveexit
    eff
      pc[i] := rem
 trajectories
   stop when
     ∃ i: Index now= lastset[i]
   evolve
     d(now) = 1
```

FIGURE 4.6: Fischer's mutual exclusion algorithm: Transitions and trajectory definitions

variable `firstcheck` keeps track of the earliest time the processes' `check` actions may occur. The analog variable `now` models real time.

The transition definitions for external actions `try(i)`, `crit(i)`, `exit(i)`, and `rem(i)` are straightforward. When a process performs one of these actions, its program counter is updated to record the region entered by the process. The most interesting transition definitions are `test(i)`, `set(i)`, and `check(i)` since they are the ones that involve timing constraints of the algorithm. When a process `i` performs a `test` action and observes `x` to be `nil`, it sets `lastset[i]` to `now` + `u_set`. This sets the deadline for the performance of the `set(i)` action.

Note that this deadline is enforced through the stopping condition in the trajectory specification. The transition `set(i)` sets `firstcheck[i]` to `now` + `l_check`. The value of `firstcheck[i]` determines the earliest time `check(i)` may occur. The `check(i)` action is enabled only when the current time has at least this value.

The stopping condition implies that if the value of `now` reaches the value of `lastset[i]` for some process `i` at some point in time, then that point must be the limit time of the trajectory.

Example 4.6 (Clock synchronization). The automaton `ClockSync` in Fig. 4.7 is the specification of a single process in a clock synchronization algorithm. Each process has a physical clock and generates a logical clock. The goal of the algorithm is to achieve "agreement" and "validity" among the logical clock values. Agreement means that the logical clocks are close to one another. Validity means that the logical clocks are within the range of the physical clocks.

```
automaton ClockSync(u,r: Real, i: Index) where u > 0 ∧ (0 ≤ r < 1)
  signature
    external send(m: Real, const i: Index),
             receive(m: Real, j: Index, const i: Index) where j ≠ i
  states
     nextsend: discrete Real := 0,
     maxother: discrete Real := 0,
     physclock: Real := 0

  derived variables
     logclock = max(maxother, physclock)

  transitions
     external send(m,i)
       pre
         m = physclock ∧ physclock = nextsend
       eff
         nextsend := nextsend + u
     external receive(m,j,i)
       eff
         maxother := max(maxother,m)
  trajectories
     stop when
       physclock = nextsend
     evolve
       (1 - r) ≤ d(physclock) ≤ (1 + r)
```

FIGURE 4.7: Clock synchronization

The algorithm is based on the exchange of physical clock values between different processes in the system. The parameter `u` determines the frequency of sending messages. Processes in the system are indexed by the elements of the type `Index`, which we assume to be predefined. `ClockSync` has a physical clock `physclock`, which may drift from the real time with a drift rate bounded by `r`. It uses the variable `maxother` to keep track of the largest physical clock value of the other processes in the system. The variable `nextsend` records when it is supposed to send its physical clock to the other processes. The logical clock `logclock` is defined to be the maximum of `maxother` and `physclock`. Formally, `logclock` is a *derived variable*, which is a function whose value is defined in terms of the state variables.

A `send(m,i)` transition is enabled when `m = physclock` and `nextsend = physclock`. It causes the value of `nextsend` to be updated so that the next send can occur when `physclock` has advanced by `u` time units. The transition definition for `receive(m,j,i)` specifies the effect of receiving a message from another process `j` in the system. On receiving a message `m` from `j`, `i` sets `maxother` to the maximum of `m` and the current value of `maxother`, thereby updating its knowledge of the largest physical clock value of other processes in the system.

The trajectory specification is slightly different from that in the previous examples. In this example, the analog variable `physclock` does not change at the same rate as real time but it drifts with a rate that is bounded by `r`. The periodic sending of physical clocks to other processes is enforced through the stopping condition in the trajectory specification. Time is not allowed to pass beyond the point where `physclock = nextsend`.

4.2 EXECUTIONS AND TRACES

We now define execution fragments, executions, trace fragments, and traces, which are used to describe automaton behavior. An *execution fragment* of a timed automaton $\mathcal{A}$ is an (A, V)-sequence $\alpha = \tau_0\, a_1\, \tau_1\, a_2\, \tau_2 \cdots$, where (1) each τ_i is a trajectory in $\mathcal{T}$ and (2) if τ_i is not the last trajectory in α, then $\tau_i.lstate \stackrel{a_{i+1}}{\rightarrow} \tau_{i+1}.fstate$. An execution fragment records what happens during a particular run of a system, including all the instantaneous, discrete state changes and all the changes to the state that occur while time advances. We write $frags_{\mathcal{A}}$ for the set of all execution fragments of $\mathcal{A}$.

If α is an execution fragment, with notation as above, then we define the *first state* of α, $\alpha.fstate$, to be $\alpha.fval$. An *execution fragment* of a timed automaton $\mathcal{A}$ *from* a state $\mathbf{x}$ of $\mathcal{A}$ is an execution fragment of $\mathcal{A}$ whose first state is $\mathbf{x}$. We write $frags_{\mathcal{A}}(\mathbf{x})$ for the set of execution fragments of $\mathcal{A}$ from $\mathbf{x}$. An execution fragment α is defined to be an *execution* if $\alpha.fstate$ is a start state, that is, $\alpha.fstate \in \Theta$. We write $execs_{\mathcal{A}}$ for the set of all executions of $\mathcal{A}$. If α is a closed (A, V)-sequence, then we define the *last state* of α, $\alpha.lstate$, to be $\alpha.lval$.

A state of $\mathcal{A}$ is *reachable* if it is the last state of some closed execution of $\mathcal{A}$. A property that is true for all reachable states of an automaton is called an *invariant assertion* or *invariant* for short.

Similar to trajectories execution fragments are also closed under countable concatenation.

Lemma 4.7 *Let $\alpha_0\, \alpha_1\, \cdots$ be a finite or infinite sequence of execution fragments of $\mathcal{A}$ such that, for each nonfinal index i, α_i is closed and $\alpha_i.lstate = \alpha_{i+1}.fstate$. Then $\alpha_0 \frown \alpha_1 \frown \cdots$ is an execution fragment of $\mathcal{A}$.*

Proof: Follows easily from the definitions, using Axiom **T3**. □

The characterization of the prefix ordering on (A, V)-sequences from Lemma 3.7 carries over to execution fragments.

Lemma 4.8 *Let α and β be execution fragments of $\mathcal{A}$ with α closed. Then*

$$\alpha \leq \beta \quad \Leftrightarrow \quad \exists \alpha' \in frags_{\mathcal{A}} \,:\, \beta = \alpha \frown \alpha'.$$

Proof: Implication $\Leftarrow$ follows from the corresponding implication in Lemma 3.7. Implication $\Rightarrow$ follows from the definitions and Axiom **T2**. □

The external behavior of a timed automaton is captured by the set of "traces" of its execution fragments, which record external actions and the trajectories that describe the intervening passage of time. A trace consists of alternating external actions and trajectories over the empty set of variables, $\emptyset$; the only interesting information contained in these trajectories is the amount of time that elapses.

Formally, if α is an execution fragment, then the *trace* of α, denoted by $trace(\alpha)$, is the $(E, \emptyset)$-restriction of α, $\alpha \lceil (E, \emptyset)$. A *trace fragment* of a timed automaton $\mathcal{A}$ *from* a state $\mathbf{x}$ of $\mathcal{A}$ is the trace of an execution fragment of $\mathcal{A}$ whose first state is $\mathbf{x}$. We write $tracefrags_{\mathcal{A}}(\mathbf{x})$ for the set of trace fragments of $\mathcal{A}$ from $\mathbf{x}$. Also, we define a *trace* of $\mathcal{A}$ to be a trace fragment from a start state, that is, the trace of an execution of $\mathcal{A}$, and write $traces_{\mathcal{A}}$ for the set of traces of $\mathcal{A}$.

In the earlier timed automaton models [8,10], execution fragments were defined in a style similar to the one presented here, that is, as an alternating sequence of trajectories and actions. However, the traces were not derived from execution fragments by a simple restriction to external actions and the empty set of variables. Rather, a trace was defined as a sequence consisting of actions paired with their time of occurrence together with a limit time. The new definition increases uniformity; the definitions, results, and proof techniques for hybrid sequences apply to both execution fragments and traces.

We now revisit some of the automata presented earlier in this chapter and give sample executions and traces for these automata.

Example 4.9 (Periodic sending process). Consider the automaton `PeriodicSend(u,M)` from Example 4.2, where `u` is instantiated to the real number 3 and the message type parameter `M` is instantiated to the set {`m1`, `m2`, ...}. The following sequence is an execution of the automaton:

$$\alpha = \tau \ \texttt{send(m1)} \ \tau \ \texttt{send(m2)} \ \tau \ \texttt{send(m3)} \ \tau \cdots$$

where $\tau : [0, 3] \rightarrow val(\{\texttt{clock}\})$ is defined such that $\tau(t)(\texttt{clock}) = t$ for all $t \in [0, 3]$. The function τ is defined for closed intervals of length 3, starting at time 0. It describes the evolution of the variable `clock`, which is 0 at the start of τ and increases with rate 1 for 3 time units. The discrete `send` events occur periodically, every 3 time units and reset the `clock` variable to 0.

The trace of the above execution fragment, $trace(\alpha)$, is the sequence

$$\alpha' = \tau' \ \texttt{send(m1)} \ \tau' \ \texttt{send(m2)} \ \tau' \ \texttt{send(m3)} \ \tau' \cdots$$

where $\tau' : [0, 3] \rightarrow val(\emptyset)$. Since the range of function τ' contains only the function with the empty domain, $trace(\alpha)$ does not contain any information about what happens to the value of `clock` as time progresses. Since the domains of τ and τ' are identical, α and α' express the same information about the amount of time that elapses between discrete steps.

Example 4.10 (Timeout process). We now present an execution of the automaton `Timeout(u,M)` from Example 4.4 where the the maximum waiting time `u` for a message is 5 and the message alphabet `M` is the set {`m1`, `m2`}. The following finite sequence is an execution of `Timeout`:

$$\alpha = \tau_0 \ \texttt{receive(m1)} \ \tau_1 \ \texttt{timeout} \ \tau_2 \ \texttt{receive(m2)} \ \tau_3 \ \texttt{timeout} \ \tau_4$$

where $Val = val(\{\texttt{suspected},\texttt{clock}\})$ and the functions τ_0, τ_1, τ_2, τ_3, and τ_4 are defined as follows:

τ_0: $[0, 2] \rightarrow Val$, where $\tau_0(t)(\texttt{suspected}) = \texttt{false}$ and $\tau_0(t)(\texttt{clock}) = t$ for all $t \in [0, 2]$.
τ_1: $[0, 5] \rightarrow Val$, where $\tau_1(t)(\texttt{suspected}) = \texttt{false}$ and $\tau_1(t)(\texttt{clock}) = t$ for all $t \in [0, 5]$.
τ_2: $[0, 1] \rightarrow Val$, where $\tau_2(t)(\texttt{suspected}) = \texttt{true}$ and $\tau_2(t)(\texttt{clock}) = 5 + t$ for all $t \in [0, 1]$.
τ_3: $[0, 5] \rightarrow Val$, where $\tau_3(t)(\texttt{suspected}) = \texttt{false}$ and $\tau_3(t)(\texttt{clock}) = t$ for all $t \in [0, 5]$.
τ_4: $[0, \infty) \rightarrow Val$, where $\tau_4(t)(\texttt{suspected}) = \texttt{true}$ and $\tau_4(t)(\texttt{clock}) = 5 + t$ for all $t \in [0, \infty)$.

In this sample execution, the first awaited message arrives at time 2. Since no other message arrives within the next 5 time units, the process performs a timeout. A new message arrives 1 time unit after the timeout and the variable `clock` is reset to 0. Since no new message arrives in the next 5 time units the process performs another timeout. The time elapses forever after this timeout since no further message arrives.

This example illustrates that the automaton `Timeout` can perform multiple timeout transitions. Another point to note is that the sample execution consists of a finite (A, V)-sequence ending with a trajectory, as opposed to an infinite sequence as in Example 4.9. The final trajectory here is a trajectory whose domain is right open and the execution is admissible and non-Zeno. Replacing τ_4 with a function on a closed interval would yield a non-Zeno execution that is not admissible.

The trace of the execution α can be obtained by letting the range of τ_i be the set consisting of the function with the empty domain, as we did in the previous example. That is, by hiding the values of the internal variables `clock` and `suspected` during trajectories.

Example 4.11 (Time-bounded channel). Consider the time-bounded channel automaton from Example 4.1. It is easy to observe that time cannot pass beyond any delivery deadline recorded in the message queue and that each deadline in the queue is less than or equal to the sum of the current time and the bound `b`. This property can be stated as an invariant assertion as follows.

Invariant: In any reachable state $\mathbf{x}$ of automaton `TimedChannel`, for all $\mathtt{p} \in \mathbf{x}(\mathtt{queue})$, $\mathbf{x}(\mathtt{now}) \leq \mathtt{p.deadline} \leq \mathbf{x}(\mathtt{now}) + \mathtt{b}$.

Such an invariant can be proved by induction. Recall that reachable states are the final states of closed executions. Axioms **T1** and **T2** allow us to view any closed execution as a concatenation of closed execution fragments, $\alpha_0 \frown \alpha_1 \frown \cdots \alpha_k$, where every α_i is either a closed trajectory or a discrete action surrounded by point trajectories and where $\alpha_i.lstate = \alpha_{i+1}.fstate$ for $0 \leq i \leq k-1$. The invariant can then be proved using induction on the length k of the sequence of execution fragments α_i.

Example 4.12 (Fischer's mutual exclusion). The main safety property that needs to be satisfied by the automaton `FischerME` from Example 4.5 is mutual exclusion. This safety property can be expressed as an invariant assertion.

Invariant 1: In any reachable state $\mathbf{x}$ of `FischerME`, there do not exist `i : Index` and `j : Index` such that $\mathtt{i} \neq \mathtt{j}$, $\mathbf{x}(\mathtt{pc})[\mathtt{i}] = \mathtt{crit}$ and $\mathbf{x}(\mathtt{pc})[\mathtt{j}] = \mathtt{crit}$.

Even though the invariant does not refer to time, its proof depends on the timing constraints of the automaton. For example, the following auxiliary invariant can be used in proving Invariant 1:

Invariant 2: In any reachable state $\mathbf{x}$ of `FischerME`, if $\mathbf{x}(\mathtt{pc})[\mathtt{i}] = \mathtt{check}$, $\mathbf{x}(\mathtt{x}) = \mathtt{embed(i)}$, and $\mathbf{x}(\mathtt{pc})[\mathtt{j}] = \mathtt{set}$, then $\mathbf{x}(\mathtt{firstcheck})[\mathtt{i}]) > \mathbf{x}(\mathtt{lastset})[\mathtt{j}]$.

This invariant states that if the program counter of process `i` has the value `check`, the program counter of process `j` has the value `set`, and the variable `x` has the value `embed(i)`, then `i` will allow enough time for `j` to set `x` to `embed(j)`, before performing the check. If this timing constraint were not satisfied, it would be possible for `i` to check that `x = embed(i)` before `j` sets `x` to `embed(j)`. Both of the processes would then observe `x` to contain their own index and enter the critical region.

The following lemma states that some properties of executions carry of to their traces and vice versa.

Lemma 4.13 *If* α *is an execution of* $\mathcal{A}$ *then*

1. α *is time bounded iff* $trace(\alpha)$ *is time bounded.*
2. α *is admissible iff* $trace(\alpha)$ *is admissible.*
3. *and if* α *is closed, then* $trace(\alpha)$ *is closed.*
4. *and if* α *is non-Zeno, then* $trace(\alpha)$ *is non-Zeno.*

Proof: The proof follows directly from the corresponding properties for the restriction of (A, V)-sequences (Lemma 3.11). □

Lemma 4.14 *If* β *is a trace of* $\mathcal{A}$ *and*

1. *if* β *is closed, then there exists an execution* α *of* $\mathcal{A}$ *such that* $trace(\alpha) = \beta$ *and* α *is closed.*
2. *if* β *is non-Zeno, then there exists an execution* α *of* $\mathcal{A}$ *such that* $trace(\alpha) = \beta$ *and* α *is non-Zeno.*

Proof: For the first part of the lemma, let $\beta = trace(\alpha)$ be a closed trace of $\mathcal{A}$. By definition of a trace, we know that $\beta.ltime = \alpha.ltime$. We also know that α is either closed or has a suffix that is an infinite sequence of alternating point trajectories and internal actions. Now, let α' be the least closed prefix of α such that $\alpha'.ltime = \beta.ltime$. Clearly, α' is a closed execution of $\mathcal{A}$ and $\beta = trace(\alpha')$.

For the second part of the lemma, observe that a non-Zeno trace is either closed or admissible. Let $\beta = trace(\alpha)$. For the case where β is closed, we have already shown how we can find a closed execution. For the case where $\beta = trace(\alpha)$ is admissible, we know that $\alpha.ltime = \infty$. Hence, α is admissible, as needed. □

Example 4.15 (Constructing a closed execution from a closed trace). Consider the Zeno hybrid sequence $\alpha = \wp(\mathbf{v})\ a\ \wp(\mathbf{v})\ a\ \wp(\mathbf{v}) \ldots$ given in Example 3.12. Suppose that α is an execution of $\mathcal{A}$ and that a is an internal action of $\mathcal{A}$. Then, $trace(\alpha) = \wp(\mathbf{v}')$, where $\wp(\mathbf{v}')$ is a trajectory over the empty set of variables. However, the fact that $trace(\alpha)$ is closed does not imply that α is closed. Thus, we see why we have a one-way implication in item 3 of Lemma 4.13. On the other hand, we can construct a closed execution of $\mathcal{A}$ with trace $\wp(\mathbf{v}')$ as explained in the proof of Lemma 4.14. The execution consisting of the point trajectory $\wp(\mathbf{v})$ is a closed execution of $\mathcal{A}$ with trace $\wp(\mathbf{v}')$.

4.3 SPECIAL KINDS OF TIMED AUTOMATA

This section describes several restricted forms of timed automata and gives definitions that are needed for theorems that are presented later in this monograph.

4.3.1 Timed Automata with Finite Internal Nondeterminism

We are sometimes interested in bounding the amount of internal nondeterminism in a timed automaton. Thus, we say that a timed automaton $\mathcal{A}$ *has finite internal nondeterminism (FIN)* provided that

1. the set Θ of start states is finite and
2. for every state $\mathbf{x}$ of $\mathcal{A}$ and every trace fragment β of $\mathcal{A}$ from $\mathbf{x}$, the set $\{\alpha.lstate \mid \alpha \in frags_{\mathcal{A}}(\mathbf{x}) \wedge trace(\alpha) = \beta\}$ is finite.

Example 4.16 (Automata with FIN). It is not hard to see that the automata `TimedChannel`, `PeriodicSend`, `PeriodicSend2`, and `Timeout` given in Section 4.1 all have FIN. The first property of the definition of FIN is satisfied since each of these automata has a unique start state. The second property follows from the fact that in each automaton, for every state $\mathbf{x}$ and every trace fragment β from $\mathbf{x}$, there is a unique execution fragment α such that $trace(\alpha) = \beta$.

Example 4.17 (Automata without FIN). We show that automata `FischerME` and `ClockSync` from Section 4.1 do not have FIN. For each automaton, we specify a trace, describe the set of all executions that have the specified trace, and argue that the second property in the definition of FIN fails for the chosen trace.

Let $\mathbf{x}$ be the start state of `FischerME` and let $\beta = \tau_0$ `try(i)` τ_1 be a trace of the same automaton, where the domains of the functions τ_0 and τ_1 are, respectively, the single point interval $[0, 0]$ and the interval $[0, u]$, and the range of both functions is the set consisting of the function with the empty domain. For any execution α, $trace(\alpha) = \beta$, iff $\alpha.ltime =$ `u`, `try(i)` occurs at time 0, and all the actions in α that occur after `try(i)` are internal actions. There are infinitely many different times that the internal actions may occur, and infinitely many values `lastcheck` and `firstcheck` could have, by the time u. Therefore, the set $\{\alpha.lstate \mid \alpha \in frags_{\mathcal{A}}(\mathbf{x}) \wedge trace(\alpha) = \tau_0\ \texttt{try(i)}\ \tau_1\}$ is not finite and `FischerME` does not have FIN.

Now, let $\mathbf{x}$ be the start state of `ClockSync` where $\mathbf{x}(\texttt{physclock}) = \mathbf{x}(\texttt{nextsend}) = \mathbf{x}(\texttt{maxother}) = 0$ and let $\beta = \tau_0\ \texttt{send(0)}\ \tau_1$ be a trace of `ClockSync`, where the domains of functions τ_0 and τ_1 are, respectively, the interval $[0, 0]$ and the interval $[0, u]$, and the range of both functions is the set consisting of the function with the empty domain. For any α in which `send(0)` occurs at time 0 and is followed by a trajectory τ such that $\tau.ltime = u$, we have $trace(\alpha) = \beta$. For any such α, $\alpha.lstate(\texttt{physclock})$ can be any value in the interval `[u (1 - r), u (1 + r)]`. Therefore, the set $\{\alpha.lstate \mid \alpha \in frags_{\mathcal{A}}(\mathbf{x}) \wedge trace(\alpha) = \tau_0\ \texttt{send(0)}\ \tau_1\}$ is not finite and `ClockSync` does not have FIN.

The following lemma states that if a timed automaton has FIN, then its set of traces is limit closed.

Lemma 4.18 *Suppose that timed automaton $\mathcal{A}$ has FIN and $\mathbf{x} \in Q$. Suppose that $\beta_1\, \beta_2 \cdots$ is a chain of trace fragments of $\mathcal{A}$ from $\mathbf{x}$. Then the hybrid sequence $\lim_i \beta_i$ is a trace fragment of $\mathcal{A}$ from $\mathbf{x}$.*

Proof: This is analogous to the proof of Lemma 4.3 of [10]. Suppose that $\mathcal{A}$ is a timed automaton that has FIN, $\mathbf{x}$ is a state of $\mathcal{A}$, and $\beta_1\, \beta_2 \cdots$ is a chain of trace fragments of $\mathcal{A}$ from $\mathbf{x}$. We define a relation *after* between trace fragments from $\mathbf{x}$ and states of $\mathcal{A}$: $after = \{(\beta, \mathbf{y}) \mid \exists \alpha \in frags_{\mathcal{A}}(\mathbf{x}).\ trace(\alpha) = \beta \wedge \alpha.lstate = \mathbf{y}\}$.

We construct a directed graph G whose nodes are pairs $(\beta_i, \mathbf{y}) \in after$, where β_i is an element of the given chain. In G, there is an edge from $(\beta_i, \mathbf{y})$ to $(\beta_{i+1}, \mathbf{y}')$ exactly if $\beta_{i+1} = \beta_i \frown \gamma$ such that $\gamma = trace(\alpha)$ for some $\alpha \in frags_{\mathcal{A}}(\mathbf{y})$, and $\alpha.lstate = \mathbf{y}'$. By the definition of property FIN, there are finitely many roots of G of the form $(\beta_1, \mathbf{y})$. By the definition of FIN and the construction of G, each node of G has finite outdegree.

We claim that each node $(\beta_i, \mathbf{y})$ of G is reachable from some root $(\beta_1, \mathbf{z})$ for some $\mathbf{z}$. By definition of the node set, there exists $\alpha \in frags_{\mathcal{A}}(\mathbf{x})$ such that $trace(\alpha) = \beta_i$ and $\alpha.lstate = \mathbf{y}$. Choose $\alpha' \in frags_{\mathcal{A}}(\mathbf{x})$ to be a prefix of α such that $trace(\alpha') = \beta_1$ and let $\mathbf{z} = \alpha'.lstate$. By definition of the edge set of G, $(\beta_i, \mathbf{y})$ is reachable from $(\beta_1, \mathbf{z})$.

Hence, G satisfies the hypotheses of Lemma 2.3, which implies that there is an infinite execution fragment starting from $\mathbf{x}$ whose trace is $\lim_i \beta_i$. □

There are two references to automata with FIN later in the chapter. The first one is in Theorem 4.19, which lists some sufficient conditions for establishing an implementation relationship between two automata. The second reference appears in the discussion about the kinds of automata that satisfy the assumptions of Theorem 7.7.

4.3.2 Feasible Timed Automata

A timed automaton $\mathcal{A}$ is *feasible* provided that for every state $\mathbf{x}$ of $\mathcal{A}$ there exists an admissible execution fragment of $\mathcal{A}$ from $\mathbf{x}$.

Feasibility is a basic requirement that any "reasonable" timed automaton should satisfy. Theorem 4.19 and Lemma 6.2 establish some results about feasible automata.

4.3.3 Timing-Independent Timed Automata

A timed automaton $\mathcal{A}$ is said to be *timing-independent* provided that all its state variables are discrete variables and its set of trajectories is exactly the set of constant-valued functions over left-closed time intervals with left endpoint 0.

We refer to timing-independent automata later in Examples 5.12 and 7.9 and in our discussion about Theorem 7.7.

4.4 IMPLEMENTATION RELATIONSHIPS

Timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* if they have the same external interface, that is, if $E_1 = E_2$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, then we say that $\mathcal{A}_1$ *implements* $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \leq \mathcal{A}_2$, if the traces of $\mathcal{A}_1$ are included among those of $\mathcal{A}_2$, that is, if $traces_{\mathcal{A}_1} \subseteq traces_{\mathcal{A}_2}$.[1]

Other preorders between timed automata could also be used as implementation relationships, for example, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable timed automata, we could consider

- every closed trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$.
- every admissible trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$.
- every non-Zeno trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$.

Theorem 4.19 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be comparable TAs.*

1. *If every closed trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$ and $\mathcal{A}_2$ has FIN, then $\mathcal{A}_1 \leq \mathcal{A}_2$.*
2. *If every admissible trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$ and $\mathcal{A}_1$ is feasible, then every closed trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$.*
3. *If every admissible trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$, $\mathcal{A}_1$ is feasible, and $\mathcal{A}_2$ has FIN, then $\mathcal{A}_1 \leq \mathcal{A}_2$.*

[1] In [10, 39–41], definitions of the set of traces of an automaton and of one automaton implementing another are based on closed and admissible executions only. The results we obtain in this work by using the newer, more inclusive definition imply corresponding results for the earlier definition. For example, we have the following property: If $\mathcal{A}_1 \leq \mathcal{A}_2$, then the set of traces that arise from closed or admissible executions of $\mathcal{A}_1$ is a subset of the set of traces that arise from closed or admissible executions of $\mathcal{A}_2$. This follows from Lemmas 4.13 and 4.14.

Proof: Proof of part 1 follows from Lemma 4.18.

Proof of part 2, consider a closed trace β of $\mathcal{A}_1$. By feasibility of $\mathcal{A}_1$, we may extend β to an admissible trace β' of $\mathcal{A}_1$. Then by assumption, β' is also a trace of $\mathcal{A}_2$. By prefix closure of the set of traces, β is a trace of $\mathcal{A}_2$.

Proof of part 3 follows from parts 1 and 2. □

4.5 SIMULATION RELATIONS

In this section, we define simulation relations between timed automata. Simulation relations may be used to show that one TA implements another, in the sense of inclusion of sets of traces. We define two main types of simulation relations (forward and backward simulations) and three derived notions (refinements, history relations, and prophecy relations).

Forward simulations are more commonly used than are backward simulations because they are easier to think about and are general enough to cover most interesting situations that arise in practice. Backward simulations are sometimes necessary, in particular, when nondeterministic choices are resolved earlier in the specification than in the implementation. In proving implementation relations, we prefer to use forward simulation relations whenever they exist, since backward simulations are harder to think about.

4.5.1 Forward Simulations

Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs. A *forward simulation* from $\mathcal{A}$ to $\mathcal{B}$ is a relation $R \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ satisfying the following conditions, for all states $\mathbf{x}_{\mathcal{A}}$ and $\mathbf{x}_{\mathcal{B}}$ of $\mathcal{A}$ and $\mathcal{B}$, respectively:

1. If $\mathbf{x}_{\mathcal{A}} \in \Theta_{\mathcal{A}}$, then there exists a state $\mathbf{x}_{\mathcal{B}} \in \Theta_{\mathcal{B}}$ such that $\mathbf{x}_{\mathcal{A}} \; R \, \mathbf{x}_{\mathcal{B}}$.
2. If $\mathbf{x}_{\mathcal{A}} \; R \, \mathbf{x}_{\mathcal{B}}$ and α is an execution fragment of $\mathcal{A}$ consisting of one action surrounded by two point trajectories, with $\alpha.fstate = \mathbf{x}_{\mathcal{A}}$, then $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = \mathbf{x}_{\mathcal{B}}$, $trace(\beta) = trace(\alpha)$, and $\alpha.lstate \; R \; \beta.lstate$.
3. If $\mathbf{x}_{\mathcal{A}} \; R \, \mathbf{x}_{\mathcal{B}}$ and α is an execution fragment of $\mathcal{A}$ consisting of a single closed trajectory, with $\alpha.fstate = \mathbf{x}_{\mathcal{A}}$, then $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = \mathbf{x}_{\mathcal{B}}$, $trace(\beta) = trace(\alpha)$, and $\alpha.lstate \; R \; \beta.lstate$.

The first condition states that for each start state of $\mathcal{A}$ there exists a related start state of $\mathcal{B}$. The second and third condition, which are referred to as *transfer properties*, assert that each discrete transition respective trajectory of $\mathcal{A}$ can be simulated by a corresponding execution fragment of $\mathcal{B}$ with the same trace.

Forward simulation relations induce a preorder between timed automata.

Theorem 4.20 *Let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be comparable TAs. If R_1 is a forward simulation from $\mathcal{A}$ to $\mathcal{B}$ and R_2 is a forward simulation from $\mathcal{B}$ to $\mathcal{C}$, then $R_2 \circ R_1$ is a forward simulation from $\mathcal{A}$ to $\mathcal{C}$.*

Even though the definition of a forward simulation refers only to closed trajectories, it also yields a correspondence for open trajectories.

Lemma 4.21 *Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs and let R be a forward simulation from $\mathcal{A}$ to $\mathcal{B}$. Let $\mathbf{x}_\mathcal{A}$ and $\mathbf{x}_\mathcal{B}$ be states of $\mathcal{A}$ and $\mathcal{B}$, respectively, such that $\mathbf{x}_\mathcal{A}\ R\, \mathbf{x}_\mathcal{B}$. Let α be an execution fragment of $\mathcal{A}$ from state $\mathbf{x}_\mathcal{A}$ consisting of a single open trajectory. Then $\mathcal{B}$ has an execution fragment β with $\beta.fstate = \mathbf{x}_\mathcal{B}$ and $trace(\beta) = trace(\alpha)$.*

Proof: Let τ be the single open trajectory in α. Using Axioms **T1** and **T2**, we construct an infinite sequence $\tau_0\, \tau_1\, \cdots$ of closed trajectories of $\mathcal{A}$ such that $\tau = \tau_0 \frown \tau_1 \frown \cdots$. Then, working recursively, we construct a sequence $\beta_0\, \beta_1\, \cdots$ of closed execution fragments of $\mathcal{B}$ such that $\beta_0.fstate = \mathbf{x}_\mathcal{B}$ and, for each i, $\tau_i.lstate\ R\ \beta_i.lstate$, $\beta_i.lstate = \beta_{i+1}.fstate$, and $trace(\tau_i) = trace(\beta_i)$. This construction uses induction on i, using property 3 of the definition of a forward simulation in the induction step. Now let $\beta = \beta_0 \frown \beta_1 \frown \cdots$. By Lemma 4.7, β is an execution fragment of $\mathcal{B}$. Clearly, $\beta.fstate = \mathbf{x}_\mathcal{B}$. By Lemma 3.9 applied to both α and β, $trace(\beta) = trace(\alpha)$. Thus β has the required properties. □

Theorem 4.22 *Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs and let R be a forward simulation from $\mathcal{A}$ to $\mathcal{B}$. Let $\mathbf{x}_\mathcal{A}$ and $\mathbf{x}_\mathcal{B}$ be states of $\mathcal{A}$ and $\mathcal{B}$, respectively, such that $\mathbf{x}_\mathcal{A}\ R\, \mathbf{x}_\mathcal{B}$. Then $tracefrags_\mathcal{A}(\mathbf{x}_\mathcal{A}) \subseteq tracefrags_\mathcal{B}(\mathbf{x}_\mathcal{B})$.*

Proof: Suppose that δ is the trace of an execution fragment of $\mathcal{A}$ that starts from $\mathbf{x}_\mathcal{A}$; we prove that δ is also a trace of an execution fragment of $\mathcal{B}$ that starts from $\mathbf{x}_\mathcal{B}$. Let $\alpha = \tau_0\, a_1\, \tau_1\, a_2\, \tau_2 \cdots$ be an execution fragment of $\mathcal{A}$ such that $\alpha.fstate = \mathbf{x}_\mathcal{A}$ and $\delta = trace(\alpha)$. We consider the following cases:

1. *α is an infinite sequence.* Using Axioms **T1** and **T2**, we can write α as an infinite concatenation $\alpha_0 \frown \alpha_1 \frown \alpha_2 \cdots$, in which the execution fragments α_i with i even consist of a trajectory only, and the execution fragments α_i with i odd consist of a single discrete step surrounded by two point trajectories.

   We define inductively a sequence $\beta_0\, \beta_1\, \cdots$ of closed execution fragments of $\mathcal{B}$, such that $\beta_0.fstate = \mathbf{x}_\mathcal{B}$, and, for all i, $\beta_i.lstate = \beta_{i+1}.fstate$, $\alpha_i.lstate\ R\ \beta_i.lstate$, and $trace(\beta_i) = trace(\alpha_i)$. We use property 3 of the definition of a simulation for the construction of the β_i's with i even, and property 2 for the construction of the β_i's with i odd. Let $\beta = \beta_0 \frown \beta_1 \frown \beta_2 \cdots$. By Lemma 4.7, β is an execution fragment of $\mathcal{B}$.

Clearly, $\beta.fstate = \mathbf{x}_{\mathcal{B}}$. By Lemma 3.9, $trace(\beta) = trace(\alpha)$. Thus β has the required properties.

2. *α is a finite sequence ending with a closed trajectory.* Similar to the first case.
3. *α is a finite sequence ending with an open trajectory.* Similar to the first case, using Lemma 4.21. □

The next corollary states that forward simulations constitute a sound technique for proving trace inclusion between timed automata.

Corollary 4.23 *Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs and let R be a forward simulation from $\mathcal{A}$ to $\mathcal{B}$. Then $\mathcal{A} \leq \mathcal{B}$.*

Proof: Suppose $\beta \in traces_{\mathcal{A}}$. Then $\beta \in tracefrags_{\mathcal{A}}(\mathbf{x}_{\mathcal{A}})$ for some start state $\mathbf{x}_{\mathcal{A}}$ of $\mathcal{A}$. Property 1 of the definition of simulation implies the existence of a start state $\mathbf{x}_{\mathcal{B}}$ of $\mathcal{B}$ such that $\mathbf{x}_{\mathcal{A}}\ R\ \mathbf{x}_{\mathcal{B}}$. Then Theorem 4.22 implies that $\beta \in tracefrags_{\mathcal{B}}(\mathbf{x}_{\mathcal{B}})$. Since $\mathbf{x}_{\mathcal{B}}$ is a start state of $\mathcal{B}$, this implies that $\beta \in traces_{\mathcal{B}}$, as needed. □

Example 4.24 (Time-bounded channels). Consider two instances of the specification in Fig. 4.1, `TimedChannel(b1, M)` and `TimedChannel(b2, M)`, where `b1` $\leq$ `b2`. We define a forward simulation R from `TimedChannel(b1, M)` to `TimedChannel(b2, M)` below. If $\mathbf{x}$ is a state of `TimedChannel(b1, M)` and $\mathbf{y}$ is a state of `TimedChannel(b2, M)`, then $\mathbf{x}\ R\ \mathbf{y}$ provided that the following conditions are satisfied:

1. $\mathbf{x}$(`now`) $=$ $\mathbf{y}$(`now`).
2. $|\mathbf{x}(\text{queue})| = |\mathbf{y}(\text{queue})|$. We use $|q|$ to denote the length of an object q of type queue.
3. $\forall i.\ 1 \leq i \leq |\mathbf{x}(\text{queue})|$, if $\mathbf{x}(\text{queue})(i) =$ `[m,u1]` then $\mathbf{y}(\text{queue})(i) =$ `[m,u2]`, for some `u2` with `u1` $\leq$ `u2`.

We can prove that R is a forward simulation from the automaton `TimedChannel(b1, M)` to the automaton `TimedChannel(b2, M)` by showing that R satisfies each of the three properties in the definition of a forward simulation relation. In each automaton there is a unique initial state that maps the variable `now` to 0 and `queue` to the empty sequence. It is obvious that the initial states, which are identical, are related by R and so the first property is satisfied.

For the rest of the proof, we let $\mathbf{x}$ and $\mathbf{y}$ be, respectively, states of `TimedChannel(b1, M)` and `TimedChannel(b2, M)` such that $\mathbf{x}\ R\ \mathbf{y}$. In order to show that the second property is satisfied, we need to consider two cases, one for each discrete action that may be performed by `TimedChannel(b1, M)`.

If `TimedChannel(b1, M)` performs a `send(m)` action and the state changes from $\mathbf{x}$ to $\mathbf{x}'$, then we need to find an execution fragment β of `TimedChannel(b2,M)` from $\mathbf{y}$ ending in $\mathbf{y}'$, such that $\mathbf{x}'\ R\, \mathbf{y}'$ and $trace(\beta)$ is the same as the trace of $\wp(\mathbf{x})$ `send(m)` $\wp(\mathbf{x}')$. The execution fragment $\beta = \wp(\mathbf{y})$ `send(m)` $\wp(\mathbf{y}')$ satisfies the required conditions. This follows from the hypothesis that $\mathbf{x}\ R\, \mathbf{y}$ and the definition of R, using the fact that the effect of a `send(m)` action of `TimedChannel(b1, M)`, `TimedChannel(b2, M)` are, respectively, adding the entry `[m,now + b1]` to $\mathbf{x}$(`queue`), and `[m,now + b2]` to $\mathbf{y}$(`queue`) where `b1` $\leq$ `b2`.

If `TimedChannel(b1, M)` performs a `receive(m)` action and the state changes from $\mathbf{x}$ to $\mathbf{x}'$, then we need to show that `receive(m)` is also enabled in $\mathbf{y}$ and that there is an execution fragment with the required properties that ends in a state $\mathbf{y}'$ such that $\mathbf{x}'\ R\, \mathbf{y}'$. In order to show that `receive(m)` is enabled in $\mathbf{y}$, we use the hypothesis that $\mathbf{x}\ R\, \mathbf{y}$, which implies that the first element of $\mathbf{y}$(`queue`) is of the form `[m,u]` for some `u`. The execution fragment $\wp(\mathbf{y})$ `receive(m)` $\wp(\mathbf{y}')$ of `TimedChannel(b1, M)` can be shown to satisfy the required conditions.

For the third property, we consider a closed trajectory τ of `TimedChannel(b1, M)` with $\tau.fstate = \mathbf{x}$ and show that there exists a closed execution fragment β of the automaton `TimedChannel(b2, M)` with $\beta.fstate = \mathbf{y}$, $trace(\beta) = trace(\tau)$, and $\tau.lstate = \beta.lstate$. It is easy to check that the trajectory τ' of `TimedChannel(b2, M)` with $\tau'.fstate = \mathbf{y}$ and $\tau'.ltime = \tau.ltime$ satisfies the required conditions.

Example 4.25 (Time-bounded channel that keeps all messages). In this example we define a variant of `TimedChannel` from Example 4.1 called `TimedChannel2`. The main difference between `TimedChannel` and `TimedChannel2` is that the message queue in `TimedChannel2` is implemented using a finite sequence of (message, delivery deadline) pairs `queue` and a pointer `ptr` that points to the next element that is to be delivered. Hence, the internal variables of `TimedChannel2` consist of `queue`, `now`, and `ptr`. The variable `ptr` initially has value 1, which indicates that it is pointing to the first element in the sequence. A `send(m)` action causes messages and deadlines to be added to the sequence as in `TimedChannel`. A `receive(m)` causes `ptr` to be incremented to make it point to the next element in the sequence instead of removing the first element. This stops when predicate tests if there is a packet in the queue with index greater than or equal to `ptr` and deadline equal to `now`. The automaton `TimedChannel` can be viewed as an optimized implementation of `TimedChannel2`.

We define below a forward simulation R from `TimedChannel` to `TimedChannel2`. If $\mathbf{x}$ is a state of `TimedChannel` and $\mathbf{y}$ is a state of `TimedChannel2`, then $\mathbf{x}\ R\, \mathbf{y}$ provided that the following conditions are satisfied:

1. $\mathbf{x}(\texttt{now}) = \mathbf{y}(\texttt{now})$.
2. $\mathbf{x}(\texttt{queue}) = \mathbf{y}(\texttt{queue})(\mathbf{y}(\texttt{ptr}) \cdots |\mathbf{y}(\texttt{queue})|)$.

```
automaton SendVal(u,r: Real, i: Index) where u > 0 ∧ (0 ≤ r < 1)
  signature
     external send(m: Real),
              receive(m:Real, j: Index, const i: Index) where j ≠ i
  states
     counter: discrete Real := 0,
     now: Real := 0,
  transitions
     external send(m,i)
       pre
         m = counter * u ∧ counter * u / (1 + r) ≤ now
       eff
         counter := counter + 1
     external receive(m,j,i)
  trajectories
     stop when
       now = counter * u / (1 - r)
     evolve
       d(now) = 1
```

FIGURE 4.8: Clock synchronization

Here, we assume the sequence representation of queues and use the subsequence notation from Chapter 2 to denote the part of the queue that starts with the index `ptr` and ends with the index **y**(queue).

Example 4.26 (Clock synchronization). In this example, we define a forward simulation from `ClockSync`(u, r, i) of Fig. 4.7 to an automaton that sends multiples of `u`. The specification of this automaton, which is called `SendVal`(u, r, i) is given in Fig. 4.8. We assume that the `Index` types in both automata are identical. The variable `counter` keeps track of which multiple of `u` is to be sent next, and variable `now` contains the current time. The automaton parameter `r` is used in the precondition of the `send` and the stopping condition of the trajectory definition to enforce bounds on the times of occurrence of `send`.

The following predicate defines a forward simulation R from automaton `ClockSync` (u, r, i) to automaton `SendVal`(u, r, i):

$$\texttt{now} * (1 - \texttt{r}) \leq \texttt{physclock} \leq \texttt{now} * (1 + \texttt{r}) \;\wedge\; \texttt{counter} * \texttt{u} = \texttt{nextsend} \geq \texttt{physclock}.$$

While automaton `ClockSync` is more intuitive as a specification, automaton `SendVal` is easier for analysis purposes, since its continuous dynamics is simpler.

4.5.2 Refinements

A *refinement* is a simple, special case of a forward simulation, often used in practice (see for instance [42, 43]), in which the relation between states of $\mathcal{A}$ and $\mathcal{B}$ is a partial function.

Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs. A *refinement* from $\mathcal{A}$ to $\mathcal{B}$ is a partial function F from $Q_{\mathcal{A}}$ to $Q_{\mathcal{B}}$, satisfying the following conditions, for all states $\mathbf{x}_{\mathcal{A}}$ and $\mathbf{x}_{\mathcal{B}}$ of $\mathcal{A}$ and $\mathcal{B}$, respectively:

1. If $\mathbf{x}_{\mathcal{A}} \in \Theta_{\mathcal{A}}$, then $\mathbf{x}_{\mathcal{A}} \in dom(F)$ and $F(\mathbf{x}_{\mathcal{A}}) \in \Theta_{\mathcal{B}}$.
2. If α is an execution fragment of $\mathcal{A}$ consisting of one action surrounded by two point trajectories and $\alpha.fstate \in dom(F)$, then $\alpha.lstate \in dom(F)$ and $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = F(\alpha.fstate)$, $trace(\beta) = trace(\alpha)$, and $\beta.lstate = F(\alpha.lstate)$.
3. If α is an execution fragment of $\mathcal{A}$ consisting of a single closed trajectory and $\alpha.fstate \in dom(F)$, then $\alpha.lstate \in dom(F)$ and $\mathcal{B}$ has a closed execution fragment β with $\beta.fstate = F(\alpha.fstate)$, $trace(\beta) = trace(\alpha)$, and $\beta.lstate = F(\alpha.lstate)$.

Note that, by a trivial inductive argument, the set of states for which F is defined contains all the reachable states of $\mathcal{A}$ (and is thus an invariant of this automaton).

Theorem 4.27 *Let $\mathcal{A}$ and $\mathcal{B}$ be two TAs and suppose $R \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$. Then R is a refinement from $\mathcal{A}$ to $\mathcal{B}$ iff R is a forward simulation from $\mathcal{A}$ to $\mathcal{B}$ and R is a partial function.*

The following theorem states a basic sanity property of refinements, namely closure under composition.

Theorem 4.28 *Let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be comparable TAs. If R_1 is a refinement from $\mathcal{A}$ to $\mathcal{B}$ and R_2 is a refinement from $\mathcal{B}$ to $\mathcal{C}$, then $R_2 \circ R_1$ is a refinement from $\mathcal{A}$ to $\mathcal{C}$.*

A *weak isomorphism* from $\mathcal{A}$ to $\mathcal{B}$ is a refinement F from $\mathcal{A}$ to $\mathcal{B}$ such that F^{-1} is a refinement from $\mathcal{B}$ to $\mathcal{A}$. We say that two automata $\mathcal{A}$ and $\mathcal{B}$ are *weakly isomorphic*, if there exists an isomorphism from $\mathcal{A}$ to $\mathcal{B}$ (or, equivalently from $\mathcal{B}$ to $\mathcal{A}$).

Example 4.29 (Refinements). In Example 4.24 we established a forward simulation between two instances of the TA in Fig. 4.1, `TimedChannel(b1, M)` and `TimedChannel(b2, M)` with $\texttt{b1} \leq \texttt{b2}$. It is not hard see that there also exists a refinement from `TimedChannel(b1, M)` to `TimedChannel(b2, M)`: just add $\texttt{b2} - \texttt{b1}$ to the deadline of each packet in the queue.

In Example 4.26 we defined a forward simulation from automaton `ClockSync` to automaton `SendVal`. In this case, however, there does not exist a refinement from `ClockSync` to `SendVal` if $\texttt{r} > 0$. The proof is by contradiction. Suppose that F is a refinement from `ClockSync` to `SendVal`. Then F maps the initial state of `ClockSync` to the initial state of

SendVal. Since send actions can be simulated, the state s0 of ClockSync with nextsend $=$ u and physclock $= 0$ is mapped by F to the state of SendVal with counter $= 1$ and now $= 0$. Consider an outgoing trajectory of s0 with positive limit time to a state s1 in which the physical clock runs maximally fast, and a trajectory with the same limit time to a state s2 in which the physical clock runs maximally slow. Since $\mathtt{r} > 0$, s1 and s2 are distinct. By the transfer property for trajectories, both s1 and s2 are mapped onto the same state of SendVal. Now observe that there exists a trajectory with positive limite time from s2 to s1. This trajectory cannot be simulated in SendVal, since in this automaton there are no nontrivial trajectories from a state to itself. Contradiction.

4.5.3 Backward Simulations

Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs. A *backward simulation* from $\mathcal{A}$ to $\mathcal{B}$ is a total relation $R \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ satisfying the following conditions, for all states $\mathbf{x}_{\mathcal{A}}$ and $\mathbf{x}_{\mathcal{B}}$ of $\mathcal{A}$ and $\mathcal{B}$, respectively:

1. If $\mathbf{x}_{\mathcal{A}} \in \Theta_{\mathcal{A}}$ and $\mathbf{x}_{\mathcal{A}}\ R\, \mathbf{x}_{\mathcal{B}}$, then $\mathbf{x}_{\mathcal{B}} \in \Theta_{\mathcal{B}}$.
2. If $\mathbf{x}_{\mathcal{A}}\ R\, \mathbf{x}_{\mathcal{B}}$ and α is an execution fragment of $\mathcal{A}$ with $\alpha.lstate = \mathbf{x}_{\mathcal{A}}$, consisting of one discrete action surrounded by two point trajectories, then $\mathcal{B}$ has a closed execution fragment β with $\beta.lstate = \mathbf{x}_{\mathcal{B}}$, $trace(\beta) = trace(\alpha)$, and $\alpha.fstate\ \ R\ \beta.fstate$.
3. If $\mathbf{x}_{\mathcal{A}}\ R\, \mathbf{x}_{\mathcal{B}}$ and α is an execution fragment of $\mathcal{A}$ with $\alpha.lstate = \mathbf{x}_{\mathcal{A}}$, consisting of one trajectory, then $\mathcal{B}$ has a closed execution fragment β with $\beta.lstate = \mathbf{x}_{\mathcal{B}}$, $trace(\beta) = trace(\alpha)$, and $\alpha.fstate\ \ R\ \beta.fstate$.

Backward simulations are closed under relational composition, and hence induce a preorder between timed automata.

Theorem 4.30 *Let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be comparable TAs. If R_1 is a backward simulation from $\mathcal{A}$ to $\mathcal{B}$ and R_2 is a backward simulation $\mathcal{B}$ to $\mathcal{C}$, then $R_2 \circ R_1$ is a backward simulation from $\mathcal{A}$ to $\mathcal{C}$.*

Theorem 4.31 *Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs and let R be a backward simulation from $\mathcal{A}$ to $\mathcal{B}$. Let $\mathbf{x}_{\mathcal{A}}$ and $\mathbf{x}_{\mathcal{B}}$ be states of $\mathcal{A}$ and $\mathcal{B}$, respectively, such that $\mathbf{x}_{\mathcal{A}}\ R\, \mathbf{x}_{\mathcal{B}}$. Let β be the trace of a closed execution fragment of $\mathcal{A}$ from $\mathbf{y}_{\mathcal{A}}$ with last state $\mathbf{x}_{\mathcal{A}}$. Then there exists $\mathbf{y}_{\mathcal{B}}$ such that β is also the trace of a closed execution fragment of $\mathcal{B}$ from $\mathbf{y}_{\mathcal{B}}$ with last state $\mathbf{x}_{\mathcal{B}}$ and $\mathbf{y}_{\mathcal{A}}\ R\, \mathbf{y}_{\mathcal{B}}$.*

Proof: Fix some $R, \mathbf{x}_{\mathcal{A}}, \mathbf{x}_{\mathcal{B}}$, and β satisfying the conditions in the statement of the theorem. Let $\alpha \in frags_{\mathcal{A}}(\mathbf{y}_{\mathcal{A}})$ for some state $\mathbf{y}_{\mathcal{A}}$ of $\mathcal{A}$ with $trace(\alpha) = \beta$ and $\alpha.lstate = \mathbf{x}_{\mathcal{A}}$. By using Axioms **T1** and **T2**, we can write α as the concatenation of a sequence of closed execution fragments, $\alpha = \alpha_0 \frown \alpha_1 \frown \cdots \alpha_n$, where each α_i is either a closed trajectory or an action surrounded by two point trajectories, $\alpha_i.lstate = \alpha_{i+1}.fstate$, for $0 \leq i \leq n-1$, and $\alpha_n.lstate = \mathbf{x}_{\mathcal{A}}$.

By using the definition of a backward simulation, working backwards from α_n, we can construct an execution fragment $\alpha' = \alpha'_0 \frown \alpha'_1 \frown \cdots \alpha'_n$ from a state $\mathbf{y}_{\mathcal{B}}$ of $\mathcal{B}$ such that (a) $\alpha'.lstate = \mathbf{x}_{\mathcal{B}}$, (b) for all i, $0 \leq i \leq n$, $\alpha_i.fstate \; R \; \alpha'_i.fstate$ and $trace(\alpha'_i) = trace(\alpha_i)$, and (c) for all i, $0 \leq i \leq n-1$, $\alpha'_i.lstate = \alpha'_{i+1}.fstate$. Using Lemma 4.7, we can see that α' is an execution fragment of $\mathcal{B}$. By Lemma 3.9, $trace(\alpha) = trace(\alpha')$ as needed. □

The next corollary states that backward simulations constitute a sound technique for proving inclusion of closed traces between timed automata.

Corollary 4.32 *Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs and let R be a backward simulation from $\mathcal{A}$ to $\mathcal{B}$. Then every closed trace of $\mathcal{A}$ is a trace of $\mathcal{B}$.*

Proof: Suppose R is a backward simulation from $\mathcal{A}$ to $\mathcal{B}$ and β is a closed trace of $\mathcal{A}$. Then $\beta = trace(\alpha)$ for some closed execution α of $\mathcal{A}$. Let $\mathbf{x}_{\mathcal{A}}$ and $\mathbf{y}_{\mathcal{A}}$ be the first and last states of α, respectively. By the totality of relation R, there exists some state $\mathbf{y}_{\mathcal{B}}$ of $\mathcal{B}$ such that $\mathbf{y}_{\mathcal{A}} \; R \; \mathbf{y}_{\mathcal{B}}$. By Theorem 4.31, there exists $\mathbf{x}_{\mathcal{B}}$ of $\mathcal{B}$ such that β is the trace of a closed execution fragment of $\mathcal{B}$ from $\mathbf{x}_{\mathcal{B}}$ with last state $\mathbf{y}_{\mathcal{B}}$ and $\mathbf{x}_{\mathcal{A}} \; R \; \mathbf{x}_{\mathcal{B}}$. Property 1 of the definition of a backward simulation relation implies that $\mathbf{x}_{\mathcal{B}}$ is a start state of $\mathcal{B}$. It follows that $\beta \in traces_{\mathcal{B}}$, as needed. □

Image-finite backward simulations constitute a sound technique for proving inclusion of (all) traces between timed automata.

Theorem 4.33 *Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs and let R be an image-finite backward simulation from $\mathcal{A}$ to $\mathcal{B}$. Then $traces_{\mathcal{A}} \subseteq traces_{\mathcal{B}}$.*

Proof: Let $\beta \in traces_{\mathcal{A}}$. If β is closed, then Corollary 4.32 implies that β is a trace of $\mathcal{B}$. From now on we assume β is not closed.

Let $\alpha \in execs_{\mathcal{A}}$ with $trace(\alpha) = \beta$. Note that any such α is either an infinite sequence $\tau_0 \; a_1 \; \tau_1 \cdots$ or a finite sequence $\tau_0 \; a_1 \; \tau_1 \; \cdots \tau_n$, where the final trajectory τ_n is right open. In either case, using Axioms **T1** and **T2**, we can construct an infinite sequence $\alpha_0 \, \alpha_1 \; \cdots$ of closed execution fragments such that $\alpha = \alpha_0 \frown \alpha_1 \frown \cdots$, where α_0 is a point trajectory, each α_i is either a closed trajectory or an action surrounded by two point trajectories, and $\alpha_i.lstate = \alpha_{i+1}.fstate$ for each i, $0 \leq i$.

We construct a directed graph G whose nodes are pairs $(\mathbf{x}, i)$ consisting of a state of $\mathcal{B}$ and an index such that $(\alpha_i.lstate, \mathbf{x}) \in R$. In G, there is an edge from $(\mathbf{x}, i)$ to $(\mathbf{x}', j)$ exactly if $j = i + 1$ and there is an $\alpha' \in frags_{\mathcal{B}}(\mathbf{x})$ with $trace(\alpha') = trace(\alpha_{i+1})$ such that $\alpha'.lstate = \mathbf{x}'$. By image-finiteness of R and the definition of the edge set, each node has finite outdegree. By using the definition of a backward simulation and the edge set of G, we can show that each node

$(\mathbf{x}, i)$ is reachable from some root node $(\mathbf{z}, 0)$ for some start state $\mathbf{z}$ of $\mathcal{B}$. Since R is image-finite there are finitely many roots of G.

The directed graph G satisfies the hypotheses of Lemma 2.3, which implies that there is an infinite path in G starting from a root. An edge from a node $(\mathbf{x}, i)$ to $(\mathbf{x}', i+1)$ along this infinite path corresponds to a closed execution fragment γ_{i+1} of $\mathcal{B}$ for i, $0 \le i$, such that $\gamma_{i+1}.fstate = \mathbf{x}$, $\gamma_{i+1}.lstate = \mathbf{x}'$ and $trace(\gamma_{i+1}) = trace(\alpha_{i+1})$. By Lemma 4.7, $\gamma = \gamma_1 \frown \gamma_2 \frown \cdots$ is an execution of $\mathcal{B}$ and by Lemma 3.9, $trace(\gamma) = trace(\gamma_1) \frown trace(\gamma_2) \cdots$. Since $trace(\gamma_{i+1}) = trace(\alpha_{i+1})$ for all i, $0 \le i$, and α_0 is a point trajectory, by Lemma 3.9, we get $trace(\gamma) = trace(\alpha) = \beta$. □

Example 4.34 (A backward simulation relation). This example illustrates the difference between forward and backward simulations. We consider two automata $\mathcal{A}$ and $\mathcal{B}$ and show that a forward simulation from $\mathcal{A}$ to $\mathcal{B}$ does not exist while we exhibit a backward simulation from $\mathcal{A}$ to $\mathcal{B}$.

Let $\mathcal{A}$ and $\mathcal{B}$ be two comparable automata specified below. The trajectories consist of a set of point trajectories. This implies that the automaton does not allow time to pass—everything happens at time 0.

- $X_{\mathcal{A}} = \{stateA\}$ and $X_{\mathcal{B}} = \{stateB\}$, where $stateA$ is a discrete variable with $type(stateA) = \{x_{\mathcal{A}}, y_{\mathcal{A}}, q_{\mathcal{A}}, s_{\mathcal{A}}\}$ and $stateB$ is a discrete variable with $type(stateB) = \{x_{\mathcal{B}}, y_{\mathcal{B}}, y'_{\mathcal{B}}, q_{\mathcal{B}}, s_{\mathcal{B}}\}$.
- $Q_{\mathcal{A}} = val(X_{\mathcal{A}})$ and $Q_{\mathcal{B}} = val(X_{\mathcal{B}})$. We write $\mathbf{x}_{\mathcal{A}}$ for the valuation that maps $stateA$ to $x_{\mathcal{A}}$, $\mathbf{y}_{\mathcal{A}}$ for the valuation that maps $stateA$ to $y_{\mathcal{A}}$, etc. Similarly, we write $\mathbf{x}_{\mathcal{B}}$ for the valuation that maps $stateB$ to $x_{\mathcal{B}}$, $\mathbf{y}_{\mathcal{B}}$ for the valuation that maps $stateB$ to $y_{\mathcal{B}}$, etc.
- $\Theta_{\mathcal{A}} = \{\mathbf{x}_{\mathcal{A}}\}$ and $\Theta_{\mathcal{B}} = \{\mathbf{x}_{\mathcal{B}}\}$.
- $E_{\mathcal{A}} = E_{\mathcal{B}} = \{a, b, c\}$ and $H_{\mathcal{A}} = H_{\mathcal{B}} = \emptyset$.
- $\mathcal{D}_{\mathcal{A}} = \{(\mathbf{x}_{\mathcal{A}}, a, \mathbf{y}_{\mathcal{A}}), (\mathbf{y}_{\mathcal{A}}, b, \mathbf{q}_{\mathcal{A}}), (\mathbf{y}_{\mathcal{A}}, c, \mathbf{s}_{\mathcal{A}})\}$ and $\mathcal{D}_{\mathcal{B}} = \{(\mathbf{x}_{\mathcal{B}}, a, \mathbf{y}_{\mathcal{B}}), (\mathbf{x}_{\mathcal{B}}, a, \mathbf{y}'_{\mathcal{B}}), (\mathbf{y}_{\mathcal{B}}, b, \mathbf{q}_{\mathcal{B}}), (\mathbf{y}'_{\mathcal{B}}, c, \mathbf{s}_{\mathcal{B}})\}$.
- $\mathcal{T}_{\mathcal{A}} = \{\wp(\mathbf{v}) \mid \mathbf{v} \in Q_{\mathcal{A}}\}$ and $\mathcal{T}_{\mathcal{B}} = \{\wp(\mathbf{v}) \mid \mathbf{v} \in Q_{\mathcal{B}}\}$.

Fig. 4.9 displays automata $\mathcal{A}$ and $\mathcal{B}$ as directed multigraphs. The nodes in the graph represent states and the edges represent discrete transitions where a label on an edge stands for the action involved in the transition.

An obvious candidate for a forward simulation from $\mathcal{A}$ to $\mathcal{B}$ is the relation

$$R = \{(\mathbf{x}_{\mathcal{A}}, \mathbf{x}_{\mathcal{B}}), (\mathbf{y}_{\mathcal{A}}, \mathbf{y}_{\mathcal{B}}), (\mathbf{y}_{\mathcal{A}}, \mathbf{y}'_{\mathcal{B}}), (\mathbf{q}_{\mathcal{A}}, \mathbf{q}_{\mathcal{B}}), (\mathbf{s}_{\mathcal{A}}, \mathbf{s}_{\mathcal{B}})\}.$$

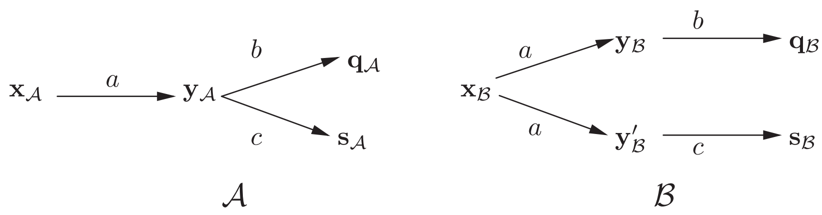

FIGURE 4.9: Difference between forward and backward simulations

However, observe that even though $\mathbf{y}_{\mathcal{A}}$ and $\mathbf{y}_{\mathcal{B}}$ are related by R, the execution fragment $\wp(\mathbf{y}_{\mathcal{A}})\ c\ \wp(\mathbf{s}_{\mathcal{A}})$ of $\mathcal{A}$ cannot be matched by any execution fragment of $\mathcal{B}$ starting with state $\mathbf{y}_{\mathcal{B}}$. Similarly, even though $\mathbf{y}_{\mathcal{A}}$ and $\mathbf{y}'_{\mathcal{B}}$ are related by R, the execution fragment $\wp(\mathbf{y}_{\mathcal{A}})\ b\ \wp(\mathbf{q}_{\mathcal{A}})$ of $\mathcal{A}$ cannot be matched by any execution fragment of $\mathcal{B}$ starting with $\mathbf{y}'_{\mathcal{B}}$. Therefore, R is not a forward simulation. In fact, there is no forward simulation relation from $\mathcal{A}$ to $\mathcal{B}$: there are finitely many possibilities for forward simulations from $\mathcal{A}$ to $\mathcal{B}$ and we see that none of them is a forward simulation by examining all the possibilities. The main reason for this is that while $\mathcal{A}$ makes the nondeterministic choice between performing b or c after performing a, $\mathcal{B}$ makes its choice earlier at the same time it performs a.

There is, however, a backward simulation from $\mathcal{A}$ to $\mathcal{B}$: the relation R defined above is a backward simulation.

4.5.4 History Relations

A relation $R \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ is a *history relation* from $\mathcal{A}$ to $\mathcal{B}$ if R is a forward simulation from $\mathcal{A}$ to $\mathcal{B}$ and R^{-1} is a refinement from $\mathcal{B}$ to $\mathcal{A}$. History relations induce a preorder between timed automata.

An automaton $\mathcal{B}$ is obtained from an automaton $\mathcal{A}$ by *adding history variables* if there exists a set of variables X such that

1. $X_{\mathcal{B}} = X_{\mathcal{A}} \cup X$ and $X_{\mathcal{A}} \cap X = \emptyset$,
2. $Q_{\mathcal{B}} \lceil X_{\mathcal{A}} \subseteq Q_{\mathcal{A}}$, and
3. relation $\{(\mathbf{x}, \mathbf{y}) \mid \mathbf{y} \in Q_{\mathcal{B}}$ and $\mathbf{y} \lceil X_{\mathcal{A}} = \mathbf{x}\}$ is a history relation from $\mathcal{A}$ to $\mathcal{B}$.

The method of adding history variables is typically used to make it possible to establish an implementation relationship using a refinement. If a refinement does not exist from a low-level automaton to a high-level one, it can often be made to exist by adding history variables to the low-level automaton.

Example 4.35 (Adding history variables to obtain a refinement). We cannot show that `TimedChannel` is an implementation of `TimedChannel2` from Example 4.25 by using a refinement. This is because we have no way of specifying what the subsequence before the pointer should be in `TimedChannel2` when relating the states of the two automata. This example shows how we can add history variables to `TimedChannel` (actually, we add just one variable) to obtain a new automaton that is related to `TimedChannel2` by a refinement.

Let `log` be a discrete variable whose static type is the same as the static type of `queue` in `TimedChannel` and let the initial value of `log` be the empty sequence. We define a new automaton `TimedChannelH` whose set of variables consists of the variables of `TimedChannel` and the variable `log`. The rest of the definition of `TimedChannelH` is the same as `TimedChannel` except for the transition definition for `receive(m)`. A `receive(m)` event in `TimedChannelH` not only removes the first message from the message queue but also appends this message to the sequence contained in `log`.

Let X_1, X_2 be the set of variables and Q_1, Q_2 be the set of states of `TimedChannel` and `TimedChannelH`, respectively. It is easy to verify that the relation $\{(\mathbf{x}, \mathbf{y}) \mid \mathbf{y} \in Q_2 \text{ and } \mathbf{y} \lceil X_1 = \mathbf{x}\}$ is a history relation from `TimedChannel` to `TimedChannelH`. This means that `TimedChannelH` is obtained from `TimedChannel` by adding a history variable.

We now define a refinement F from `TimedChannelH` to `TimedChannel2` as follows: In our definition we assume the following conventions. Concatenation on the left corresponds to putting an element on the front of a queue. Recall also that we use juxtaposition for concatenation of sequences. If $\mathbf{x}$ is a state of `TimedChannelH` and $\mathbf{y}$ is a state of `TimedChannel2`, then $F(\mathbf{x}) = \mathbf{y}$ where

1. $\mathbf{y}(\texttt{now}) = \mathbf{x}(\texttt{now})$.
2. $\mathbf{y}(\texttt{queue}) = \mathbf{x}(\texttt{log}) ^\frown \mathbf{x}(\texttt{queue})$.
3. $\mathbf{y}(\texttt{ptr}) = |\mathbf{x}(\texttt{log})| + 1$.

Whenever an automaton $\mathcal{B}$ is obtained from $\mathcal{A}$ by adding history variables, then there exists a history relation from $\mathcal{A}$ to $\mathcal{B}$ by definition. Theorem 4.36 states that the converse also holds, if weakly isomorphic automata are considered.

Theorem 4.36 *Let $\mathcal{A}$ and $\mathcal{B}$ be two comparable TAs. Suppose that there is a history relation from $\mathcal{A}$ to $\mathcal{B}$. Then, there exists a TA $\mathcal{C}$ that is weakly isomorphic to $\mathcal{B}$ and is obtained from $\mathcal{A}$ by adding history variables.*

Proof: Assume, without loss of generality, that $X_{\mathcal{A}}$ and $X_{\mathcal{B}}$ are disjoint. Let R be a history relation from $\mathcal{A}$ to $\mathcal{B}$. Define automaton $\mathcal{C}$ as follows:

- $X_{\mathcal{C}} = X_{\mathcal{A}} \cup X_{\mathcal{B}}$.
- $Q_{\mathcal{C}} = \{\mathbf{x} \in val(X_{\mathcal{C}}) \mid (\mathbf{x} \lceil X_{\mathcal{A}}, \mathbf{x} \lceil X_{\mathcal{B}}) \in R\}$.

- $\Theta_{\mathcal{C}} = \{\mathbf{x} \in Q_{\mathcal{C}} \mid \mathbf{x} \lceil X_{\mathcal{B}} \in \Theta_{\mathcal{B}}\}$.
- $E_{\mathcal{C}} = E_{\mathcal{B}}$ and $H_{\mathcal{C}} = H_{\mathcal{B}}$.
- $\mathbf{x} \xrightarrow{a}_{\mathcal{C}} \mathbf{y}$ iff $\mathbf{x} \lceil X_{\mathcal{B}} \xrightarrow{a}_{\mathcal{B}} \mathbf{y} \lceil X_{\mathcal{B}}$.
- $\mathcal{T}_{\mathcal{C}} = \{\tau \in trajs(Q_{\mathcal{C}}) \mid \tau \lceil X_{\mathcal{B}} \in \mathcal{T}_{\mathcal{B}}\}$.

Let $F : Q_{\mathcal{C}} \rightarrow Q_{\mathcal{B}}$ be the projection function such that $F(\mathbf{x}) = \mathbf{x} \lceil X_{\mathcal{B}}$ for all $\mathbf{x} \in Q_{\mathcal{C}}$. It is easy to check that F is a weak isomorphism from $\mathcal{C}$ to $\mathcal{B}$. We verify that $\mathcal{C}$ is obtained from $\mathcal{A}$ by adding history variables. Let $X_{\mathcal{B}}$ be the variable set X required in the definition of a history variable and let $R' = \{(\mathbf{x}, \mathbf{y}) \mid \mathbf{y} \in Q_{\mathcal{C}} \wedge \mathbf{y} \lceil X_{\mathcal{A}} = \mathbf{x}\}$. We need to show that R' is a history relation from $\mathcal{A}$ to $\mathcal{C}$.

1. R' is a forward simulation from $\mathcal{A}$ to $\mathcal{C}$. By definitions of the relations F, R', and the automaton $\mathcal{C}$, $R' = F^{-1} \circ R$. Since F^{-1} is a refinement from $\mathcal{B}$ to $\mathcal{C}$, by Theorem 4.27, we know that it is a forward simulation from $\mathcal{B}$ to $\mathcal{C}$. Since R is a forward simulation from $\mathcal{A}$ to $\mathcal{B}$, by Theorem 4.20 we have R' is a forward simulation from $\mathcal{A}$ to $\mathcal{C}$, as needed.
2. R'^{-1} is a refinement from $\mathcal{C}$ to $\mathcal{A}$. We use that $R'^{-1} = R^{-1} \circ F$. Since F is a refinement from $\mathcal{C}$ to $\mathcal{B}$ and R^{-1} is a refinement from $\mathcal{B}$ to $\mathcal{A}$, by Theorem 4.28, we have R'^{-1} is a refinement from $\mathcal{C}$ to $\mathcal{A}$, as needed. □

In the untimed case, forward simulations are essentially the same as history relations (or variables) combined with refinements [44, Th. 5.8]. Clearly, since history relations and refinements are both special cases of forward simulations, and forward simulations compose, forward simulations are at least as powerful as arbitrary combinations of history relations and refinements. Conversely, if there is a forward simulation from $\mathcal{A}$ to $\mathcal{B}$, then there exists an automaton $\mathcal{C}$ with a history relation from $\mathcal{A}$ to $\mathcal{C}$ and a refinement from $\mathcal{C}$ to $\mathcal{B}$. In [9] a corresponding result is claimed for timed automata (Theorem 7.8), but the proof turns out to be flawed. Example 7.13 of [9] constitutes a counterexample to Theorem 7.8 of [9]. Below, we have translated the example to conform to this chapter.

Example 4.37 (Forward simulations more powerful than combination history relations and refinements). Consider the automata `A` and `B` specified in Fig. 4.10. The two automaton definitions are very similar. While in `A` an `a`-action is enabled when `init` = `true` and the value of `now` is a rational number, in `B` an `a`-action is enabled when `init` = `true` and the value of `now` is an integer. While automaton `A` has a perfect clock with rate 1, automaton `B` measures time with a clock that may run either too slow or too fast, in an arbitrary fashion.

```
automaton A
  signature
    external a
  states
    init: Bool := true,
    now: Real := 0
  transitions
    external a
      pre
        init ∧ rational(now)
      eff
        init := false
  trajectories
    evolve
      d(now) = 1
```

```
automaton B
  signature
    external a
  states
    init: Bool := true,
    now: Real := 0
  transitions
    external a
      pre
        init ∧ integer(now)
      eff
        init := false
  trajectories
    evolve
      d(now) > 0
```

FIGURE 4.10: The power of forward simulations

It is easy to check that the predicate

$$\texttt{natural(B.now)} \wedge \texttt{A.init} = \texttt{B.init}$$

determines a forward simulation from A to B. However, there does not exists a timed automaton C with a history relation from A to C and a refinement from C to B. The proof is by contradiction: suppose C is such a timed automaton. Let $\mathbf{x}_0$ be a start state of C, let F be a history relation from A to C, and let R be a refinement from C to B. Then, by the start condition of a history relation, the start state (0, true) of A is related to $\mathbf{x}_0$ by F. By the start condition of a refinement, R maps $\mathbf{x}_0$ to the start state (0, true) of B. Since in A there is a trajectory with limit time 1 from (0, true) to (1, true), the transfer property for F gives that in C there is a trajectory τ with limit time 1 from $\mathbf{x}_0$ to some state $\mathbf{x}_1$ that is related by F to (1, true). Next, the transfer property for R gives that in B there is a trajectory with limit time 1 from (0, true) to state $R(\mathbf{x}_1) = (t, \texttt{true})$, for some $t > 0$. Since state (1, true) in A enables an a-action, $\mathbf{x}_1$ enables an execution fragment in which an a-action takes place within 0 time. Since $\mathbf{x}_1$ is mapped by R to $(t, \texttt{true})$, it follows by the transfer property for R that t in fact equals some natural number $n > 0$. By Axioms **T1** and **T2**, we can write τ as the concatenation $\tau_0\ \tau_1\ \cdots \tau_n$ of $n+1$ trajectories that all have limit time $1/(n+1)$. Using the fact that F is a history relation and the limit times of the trajectories τ_i are rational, we may infer that the last state of each trajectory τ_i enables an execution fragment in which an a-action takes place within 0 time. Using the fact that R is a refinement, we may infer that there is a trajectory in B from (0, true) to $(n, \texttt{true})$ on which there are at least $n+2$ states (including the first and last state) in which an a-action is enabled. This contradicts the fact that in B actions a are only enabled at integer times, which implies that there are only $n+1$ such states on any trajectory from $(0, \texttt{true})$ to $(n, \texttt{true})$.

4.5.5 Prophecy Relations

A relation $R \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{B}}$ is a *prophecy relation* from $\mathcal{A}$ to $\mathcal{B}$ if R is a backward simulation from $\mathcal{A}$ to $\mathcal{B}$ and R^{-1} is a refinement from $\mathcal{B}$ to $\mathcal{A}$. Prophecy relations induce a preorder between timed automata.

An automaton $\mathcal{B}$ is obtained from an automaton $\mathcal{A}$ by *adding prophecy variables* if there exists a set of variables X such that

1. $X_{\mathcal{B}} = X_{\mathcal{A}} \cup X$ and $X_{\mathcal{A}} \cap X = \emptyset$,
2. $Q_{\mathcal{B}} \lceil X_{\mathcal{A}} \subseteq Q_{\mathcal{A}}$, and
3. relation $\{(\mathbf{x}, \mathbf{y}) \mid \mathbf{y} \in Q_{\mathcal{B}} \text{ and } \mathbf{y} \lceil X_{\mathcal{A}} = \mathbf{x}\}$ is a prophecy relation from $\mathcal{A}$ to $\mathcal{B}$.

Example 4.38 (Adding prophecy variables to obtain a refinement). We consider adding a prophecy variable to the automaton $\mathcal{A}$ from Example 4.34. Let $\mathcal{C}$ be the automaton defined as follows:

- $X_{\mathcal{C}} = X_{\mathcal{A}} \cup \{v\}$, where v is a discrete variable with $type(v) = \{b, c\}$.
- $Q_{\mathcal{C}} = \{\mathbf{x}_{\mathcal{C}}, \mathbf{x}'_{\mathcal{C}}, \mathbf{y}_{\mathcal{C}}, \mathbf{y}'_{\mathcal{C}}, \mathbf{q}_{\mathcal{C}}, \mathbf{s}_{\mathcal{C}}\}$ such that

$$
\begin{array}{lcl}
\mathbf{x}_{\mathcal{C}} \lceil X_{\mathcal{A}} = \mathbf{x}_{\mathcal{A}} & \text{and} & \mathbf{x}_{\mathcal{C}}(v) = b \\
\mathbf{x}'_{\mathcal{C}} \lceil X_{\mathcal{A}} = \mathbf{x}_{\mathcal{A}} & \text{and} & \mathbf{x}'_{\mathcal{C}}(v) = c \\
\mathbf{y}_{\mathcal{C}} \lceil X_{\mathcal{A}} = \mathbf{y}_{\mathcal{A}} & \text{and} & \mathbf{y}_{\mathcal{C}}(v) = b \\
\mathbf{y}'_{\mathcal{C}} \lceil X_{\mathcal{A}} = \mathbf{y}_{\mathcal{A}} & \text{and} & \mathbf{y}'_{\mathcal{C}}(v) = c \\
\mathbf{q}_{\mathcal{C}} \lceil X_{\mathcal{A}} = \mathbf{q}_{\mathcal{A}} & \text{and} & \mathbf{q}_{\mathcal{C}}(v) = b \\
\mathbf{s}_{\mathcal{C}} \lceil X_{\mathcal{A}} = \mathbf{s}_{\mathcal{A}} & \text{and} & \mathbf{s}_{\mathcal{C}}(v) = c
\end{array}
$$

- $\Theta_{\mathcal{C}} = \{\mathbf{x}_{\mathcal{C}}, \mathbf{x}'_{\mathcal{C}}\}$.
- $E_{\mathcal{C}} = \{a, b, c\}$ and $H_{\mathcal{C}} = \emptyset$.
- $\mathcal{D}_{\mathcal{C}} = \{(\mathbf{x}_{\mathcal{C}}, a, \mathbf{y}_{\mathcal{C}}), (\mathbf{x}'_{\mathcal{C}}, a, \mathbf{y}'_{\mathcal{C}}), (\mathbf{y}_{\mathcal{C}}, b, \mathbf{q}_{\mathcal{C}}), (\mathbf{y}'_{\mathcal{C}}, c, \mathbf{s}_{\mathcal{C}})\}$.
- $\mathcal{T}_{\mathcal{C}} = \{\wp(\mathbf{v}) \mid \mathbf{v} \in Q_{\mathcal{C}}\}$.

Fig. 4.11 displays automata $\mathcal{A}$ and $\mathcal{C}$ as directed multipgraphs.

Relation $R = \{(\mathbf{x}_{\mathcal{A}}, \mathbf{x}_{\mathcal{C}}), (\mathbf{x}_{\mathcal{A}}, \mathbf{x}'_{\mathcal{C}}), (\mathbf{y}_{\mathcal{A}}, \mathbf{y}_{\mathcal{C}}), (\mathbf{y}_{\mathcal{A}}, \mathbf{y}'_{\mathcal{C}}), (\mathbf{q}_{\mathcal{A}}, \mathbf{q}_{\mathcal{C}}), (\mathbf{s}_{\mathcal{A}}, \mathbf{s}_{\mathcal{C}})\}$ is a backward simulation from $\mathcal{A}$ to $\mathcal{C}$ and R^{-1} is a refinement. Therefore, $\mathcal{C}$ is obtained by adding a prophecy variable to $\mathcal{A}$. Note that there is no refinement from $\mathcal{A}$ to $\mathcal{B}$ defined in Example 4.34. However, relation $F = \{(\mathbf{x}_{\mathcal{C}}, \mathbf{x}_{\mathcal{B}}), (\mathbf{x}'_{\mathcal{C}}, \mathbf{x}_{\mathcal{B}}), (\mathbf{y}_{\mathcal{C}}, \mathbf{y}_{B}), (\mathbf{y}'_{\mathcal{C}}, \mathbf{y}'_{\mathcal{B}}), (\mathbf{q}_{\mathcal{C}}, \mathbf{q}_{\mathcal{B}}), (\mathbf{s}_{\mathcal{C}}, \mathbf{s}_{B})\}$ is a refinement from $\mathcal{C}$ to $\mathcal{B}$.

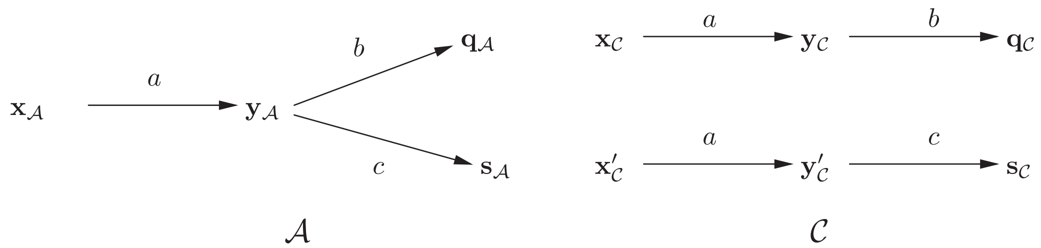

FIGURE 4.11: Prophecy variable

Theorem 4.39 *Let $\mathcal{A}$ and $\mathcal{B}$ be two comparable TAs such that $V_{\mathcal{A}}$ and $V_{\mathcal{B}}$ are disjoint. Suppose that there is a prophecy relation from $\mathcal{A}$ to $\mathcal{B}$. Then, there exists an automaton $\mathcal{C}$ that is isomorphic to $\mathcal{B}$ and is obtained from $\mathcal{A}$ by adding prophecy variables.*

Proof: The proof is analogous to the proof of Theorem 4.36. We assume a backward simulation relation R instead of a forward simulation relation. We construct the automaton $\mathcal{C}$ as in Theorem 4.36 and verify that it is obtained from $\mathcal{A}$ by adding a prophecy variable. □

CHAPTER 5

Operations on Timed Automata

In this chapter we introduce three kinds of operations on timed automata: parallel composition, hiding, and adding lower and upper bounds for tasks.

5.1 COMPOSITION

The composition operation for timed automata allows an automaton representing a complex system to be constructed by composing automata representing individual system components. Our composition operation identifies external actions with the same name in different component automata. When any component automaton performs a discrete step involving an action a, so do all component automata that have a as an external action. The composition operator for timed automata is simpler than it is for general hybrid automata since all the variables in a timed automaton are internal.[2] All the proofs of this section are as in [6], with simplifications due to the absence of external variables.

5.1.1 Definitions and Basic Results

Formally, we say that timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if $H_1 \cap A_2 = H_2 \cap A_1 = \emptyset$ and $X_1 \cap X_2 = \emptyset$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible, then their *composition* $\mathcal{A}_1 \| \mathcal{A}_2$ is defined to be the structure $\mathcal{A} = (X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$ where

- $X = X_1 \cup X_2$.
- $Q = \{\mathbf{x} \in val(X) \mid \mathbf{x} \lceil X_i \in Q_i, i \in \{1, 2\}\}$.
- $\Theta = \{\mathbf{x} \in Q \mid \mathbf{x} \lceil X_i \in \Theta_i, i \in \{1, 2\}\}$.
- $E = E_1 \cup E_2$ and $H = H_1 \cup H_2$.
- For each $\mathbf{x}, \mathbf{x}' \in Q$ and each $a \in A$, $\mathbf{x} \xrightarrow{a}_{\mathcal{A}} \mathbf{x}'$ iff for $i \in \{1, 2\}$, either $a \in A_i$ and $\mathbf{x} \lceil X_i \xrightarrow{a}_i \mathbf{x}' \lceil X_i$ or $a \notin A_i$ and $\mathbf{x} \lceil X_i = \mathbf{x}' \lceil X_i$.
- $\mathcal{T} \subseteq trajs(Q)$ is given by $\tau \in \mathcal{T} \Leftrightarrow \tau \downarrow X_i \in \mathcal{T}_i, i \in \{1, 2\}$.

[2]The composition operation for general hybrid automata requires external variables to be identified as well as external actions. When any component automaton follows a particular trajectory for an external variable v, then so do all component automata of which v is an external variable.

Theorem 5.1 *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are timed automata, then $\mathcal{A}_1 \| \mathcal{A}_2$ is a timed automaton.*

The following "projection lemma" says that execution fragments of a composition of timed automata project to give executions fragments of the component automata. Moreover, certain properties of the fragments of the composition imply, or are implied by, similar properties for the component fragments.

Lemma 5.2 *Let $\mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2$ and let α be an execution fragment of $\mathcal{A}$. Then $\alpha \lceil (A_1, X_1)$ and $\alpha \lceil (A_2, X_2)$ are execution fragments of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. Furthermore,*

1. *α is time bounded iff both $\alpha \lceil (A_1, X_1)$ and $\alpha \lceil (A_2, X_2)$ are time bounded.*
2. *α is admissible iff both $\alpha \lceil (A_1, X_1)$ and $\alpha \lceil (A_2, X_2)$ are admissible.*
3. *α is closed iff both $\alpha \lceil (A_1, X_1)$ and $\alpha \lceil (A_2, X_2)$ are closed.*
4. *α is non-Zeno iff both $\alpha \lceil (A_1, X_1)$ and $\alpha \lceil (A_2, X_2)$ are non-Zeno.*
5. *α is an execution iff both $\alpha \lceil (A_1, X_1)$ and $\alpha \lceil (A_2, X_2)$ are executions.*

The following lemma says that we obtain the same result for an execution fragment α of a composition if we first extract the trace and then restrict to one of the components, or if we first restrict to the component and then take the trace.

Lemma 5.3 *Let $\mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2$ and let α be an execution fragment of $\mathcal{A}$. Then, for $i = 1, 2$, $trace(\alpha) \lceil (E_i, \emptyset) = trace(\alpha \lceil (A_i, X_i))$.*

The following theorem is a fundamental result that relates the set of traces of a composed automaton to the sets of traces of its components. Set inclusion in one direction expresses the idea that a trace of a composition "projects" to yield traces of the components. Set inclusion in the other direction expresses the idea that traces of components can be "pasted" to yield a trace of the composition.

Theorem 5.4 *Let $\mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2$. Then $traces_{\mathcal{A}}$ is exactly the set of $(E, \emptyset)$-sequences whose restrictions to $\mathcal{A}_1$ and $\mathcal{A}_2$ are traces of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively.*
That is, $traces_{\mathcal{A}} = \{\beta \mid \beta$ is an $(E, \emptyset)$-sequence and $\beta \lceil (E_i, \emptyset) \in traces_{\mathcal{A}_i}, i \in \{1, 2\}\}$.

Notation: The compatibility conditions for composition require the set of internal variables of each automaton to be disjoint from the set of internal variables of all the other automata in the composition. We use a general scheme to disambiguate the internal variables of components in order to avoid possible name clashes that can violate the compatibility conditions. If $\mathcal{A}$ is the name of an automaton and v is an internal variable of $\mathcal{A}$, then we refer to this variable as $\mathcal{A}.v$ in the composite automaton. But if no confusion is possible, we write v rather than $\mathcal{A}.v$.

Example 5.5 (Periodic sending process with timeouts). Let $\mathcal{C}$ be the composition of three automata from Examples 4.1, 4.2, and 4.4:

$$\mathcal{C} = \texttt{PeriodicSend(u1, M)} \parallel \texttt{TimedChannel(b, M)} \parallel \texttt{Timeout(u2, M)}$$

where $\texttt{M} = \{\texttt{m1}, \ldots, \texttt{mn}\}$ and $\texttt{b} + \texttt{u1} < \texttt{u2}$. In a setting where $\texttt{b} < \texttt{u1}$, the following sequence is a trace of $\mathcal{C}$:

$$\alpha = \overline{\texttt{u1}}\ \texttt{send(m1)}\ \overline{\texttt{b}}\ \texttt{receive(m1)}\ \overline{\texttt{u1} - \texttt{b}}\ \texttt{send(m2)}\ \overline{\texttt{b}}\ \texttt{receive(m2)}\ \overline{\texttt{u1} - \texttt{b}}\ \cdots$$

where $\bar{t}$ denotes the trace with as domain $[0, t]$ and as range the set consisting of the function with the empty domain. The following invariant states that $\mathcal{C}$ never performs a `timeout` action.

Invariant 1: In any reachable state $\mathbf{x}$ of $\mathcal{C}$, $\mathbf{x}(\texttt{suspected}) = \texttt{false}$.

In order to prove this invariant we can use auxiliary invariants for the component automata, such as the one established in Example 4.11, and an auxiliary global invariant such as the one below, which establishes the fact that every message is delivered before the variable `Timeout.clock` reaches the point at which a `timeout` action occurs.

Invariant 2: In any reachable state $\mathbf{x}$ of $\mathcal{C}$,

1. if $\mathbf{x}(\texttt{queue})$ is not empty then there is a packet `p` such that $\texttt{p} \in \mathbf{x}(\texttt{queue})$ and $\texttt{p.deadline} - \mathbf{x}(\texttt{now}) < \texttt{u2} - \mathbf{x}(\texttt{Timeout.clock})$.
2. if $\mathbf{x}(\texttt{queue})$ is empty then $\texttt{u1} - \mathbf{x}(\texttt{PeriodicSend.clock}) + \texttt{b} < \texttt{u2} - \mathbf{x}(\texttt{Timeout.clock})$.

Example 5.6 (Periodic sending process with failures and timeouts). In this example, we consider a composite automaton defined exactly like the one in Example 5.5 except that the automaton `PeriodicSend(ul,M)` is replaced with `PeriodicSend(ul,M)`, the periodic sending process with failures. Let $\mathcal{C} = \texttt{PeriodicSend2(ul,M)} \parallel \texttt{TimedChannel(b,M)} \parallel \texttt{Timeout(u2,M)}$. The following sequence is a trace of $\mathcal{C}$:

$$\overline{\texttt{u1}}\ \texttt{send(m1)}\ \overline{\texttt{b}}\ \texttt{receive(m1)}\ \overline{\texttt{b}}\ \texttt{fail}\ \overline{\texttt{u2} - \texttt{b}}\ \texttt{timeout}\ \overline{\infty}.$$

According to this sample trace, the first message sent by the periodic sending process is received exactly `b` time units after it is sent. The periodic sending process fails $2 \times \texttt{b}$ time units after sending its first message. The timeout process performs a `timeout` since no second message arrives within the next `u2` time units after the receipt of the first message.

The following invariant states that a `timeout` performed by $\mathcal{C}$ can be used to conclude that the sender process has failed. We again assume that `b + u1 < u2`.

Invariant 1: In any reachable state **x** of C,

$$\mathbf{x}(\texttt{Timeout.suspected}) \Rightarrow \mathbf{x}(\texttt{PeriodicSend2.failed}).$$

The automaton $\mathcal{C}$ is guaranteed to perform a `timeout` to signal the failure of a process, within a specified amount of time after the occurrence of a fail event. The following is a formal statement of this property.

Let α be an admissible execution of $\mathcal{C}$ in which a `fail` event occurs. Let `t` be the point in time at which the first `fail` event occurs in α. Then a `timeout` event occurs in α in the interval [`t + u2 - u1`, `t + b + u2`].

Example 5.7 (Clock synchronization). In this example we consider the composition of three clock synchronization automata with six time-bounded channel automata. A graphical representation of the composite automaton is given in Fig. 5.1. The abbreviation CS_i represents the automaton `ClockSync` from Example 4.6. The abbreviation $TC_{i,j}$ represents the automaton `TimedChannel` from Example 4.1, the time-bounded channel with maximum delay `b`, but with the `send(m)` and `receive(m)` actions renamed to `send(m,i)` and `receive(m,i,j)`,

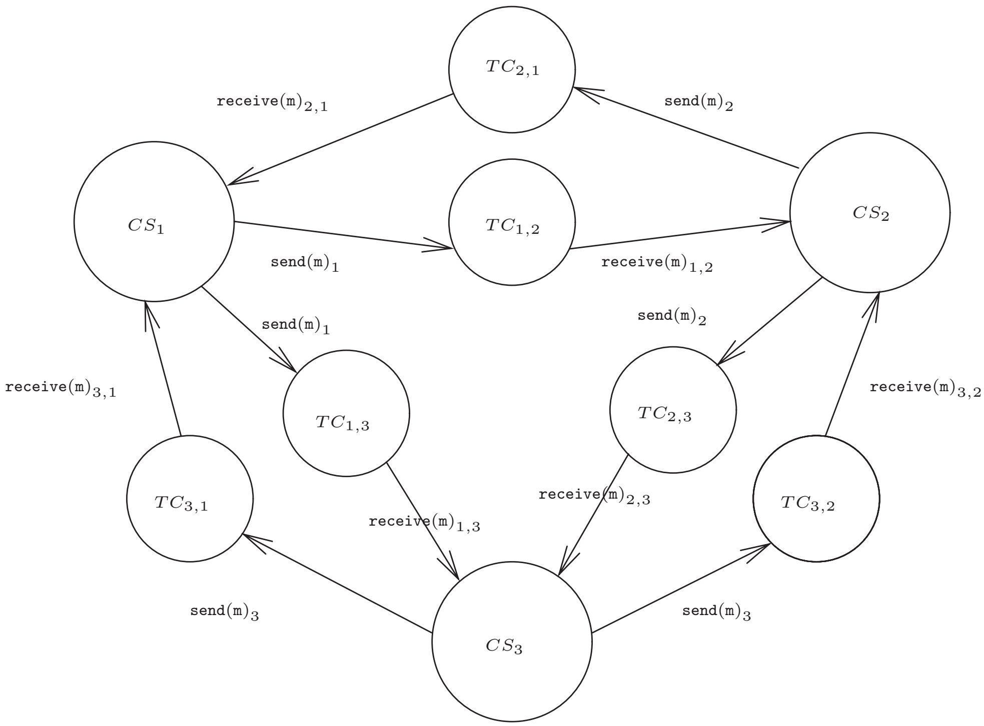

FIGURE 5.1: Clock synchronization network.

respectively, to enable communication of real-valued messages from `ClockSync` to `ClockSync`. Let

$$\mathcal{C} \;=\; CS_1 \parallel CS_2 \parallel CS_3 \parallel TC_{1,2} \parallel TC_{2,1} \parallel TC_{1,3} \parallel TC_{3,1} \parallel TC_{2,3} \parallel TC_{3,2}.$$

A physical clock diverges from real time at the largest rate when it evolves with rate (`1 + r`) or (`1 - r`). For example, if a physical clock evolves with rate `1 + r`, then at time t, its value is $t \times$ `(1 + r)`. Hence, the largest possible difference between a physical clock and the real time is ($t \times \mathtt{r}$). This property is stated by the following invariant.

Invariant 1: In any reachable state $\mathbf{x}$ of $\mathcal{C}$, at any time $t \in \mathsf{T}$, for any $i \in \{1, 2, 3\}$, $|\mathbf{x}(CS_i.\mathtt{physclock}) - t| \leq t \times \mathtt{r}$.

Two physical clocks in $\mathcal{C}$ diverge at the largest rate when one evolves with rate (`1 + r`) and the other with (`1 - r`). It follows from Invariant 1 that at any time t the largest possible difference between the physical clock values for two processes is $2 \times t \times \mathtt{r}$. This property is formalized by the following invariant.

Invariant 2: In any reachable state $\mathbf{x}$ of C, at any time $t \in \mathsf{T}$, for any $i, j \in \{1, 2, 3\}$, $|\mathbf{x}(CS_i.\mathtt{physclock}) - \mathbf{x}(CS_j.\mathtt{physclock})| \leq 2 \times t \times \mathtt{r}$.

The following invariant states that in any reachable state there exists a process j such that the logical clock of each process in the system is smaller than or equal to the physical clock of j. This follows from the definition of a logical clock and the fact that physical clocks always increase.

Invariant 3: In any reachable state $\mathbf{x}$ of $\mathcal{C}$, there exists $j \in \{1, 2, 3\}$ such that for all $i \in \{1, 2, 3\}$, $\mathbf{x}(CS_i.\mathtt{logclock}) \leq \mathbf{x}(CS_j.\mathtt{physclock})$.

The following invariant states that in any reachable state there exists a process j such that the logical clock of each process in the system is larger than or equal to the physical clock of j. This follows from the definition of a logical clock.

Invariant 4: In any reachable state $\mathbf{x}$ of $\mathcal{C}$, there exists $j \in \{1, 2, 3\}$ such that for all $i \in \{1, 2, 3\}$, $\mathbf{x}(CS_i.\mathtt{logclock}) \geq \mathbf{x}(CS_j.\mathtt{physclock})$.

Invariants 3 and 4 together are called *validity* properties. They express the condition that all the logical clocks remain in an envelope bounded by the maximum and minimum physical clock values in the system. The following invariant formalizes the property that all the logical clocks at a given time lie within the envelope formed by the largest and the smallest physical clock values in the system. It follows from Invariants 1, 3, and 4 that any point in this envelope can diverge from real time t by at most $t \times \mathtt{r}$ time units.

Invariant 5: In any reachable state $\mathbf{x}$ of $\mathcal{C}$, at any time $t \in \mathsf{T}$, for any $i \in \{1, 2, 3\}$, $|\mathbf{x}(CS_i.\mathtt{logclock}) - t| \leq t \times \mathtt{r}$.

Finally, we state a property about the *agreement* of logical clocks in $\mathcal{C}$. It says that the difference between two logical clocks is always bounded by a constant (which depends on the message-sending interval and the bounds on clock drift and message delay).

Invariant 6: In any reachable state $\mathbf{x}$ of $\mathcal{C}$, for all $i, j \in \{1, 2, 3\}$, $|\mathbf{x}(CS_i.\mathtt{logclock}) - \mathbf{x}(CS_j.\mathtt{logclock})| \leq \mathtt{u} + (\mathtt{b} \times (1 + \mathtt{r}))$.

To see why Invariant 6 holds, fix j to be a process with the largest physical clock in $\mathbf{x}$, and fix i to be any other process. Let v_j, v_i be the logical clock values of j and i respectively in state $\mathbf{x}$. Note that v_j is also the physical clock value of j in $\mathbf{x}$. By Invariant 3, we know that $v_i \leq v_j$. To show Invariant 6, it suffices to show that $v_j - v_i \leq \mathtt{u} + (\mathtt{b} \times (1 + \mathtt{r}))$.

Let α be a finite execution that leads to state $\mathbf{x}$. There are two cases to consider.

1. Some message sent by j arrives at i in α. Consider the last such message and let v_1 be the value that it contains. Let v_2 be the newly adjusted logical clock value of i immediately after the message arrives. We know that $v_i \geq v_2 \geq v_1$.

   If j sends a later message to i in α, then it sends the next later message when its physical clock has value $v_1 + \mathtt{u}$. By assumption, this message does not arrive at i. Therefore, the real time that elapses after sending it must be at most $\mathtt{b}$. It follows that the physical clock increase of j since sending this message is at most $\mathtt{b} \times (1 + \mathtt{r})$ and so $v_j \leq v_1 + \mathtt{u} + \mathtt{b} \times (1 + \mathtt{r})$. On the other hand, if j does not send a later message to i in α, then $v_j \leq v_1 + \mathtt{u}$. In either case, we have $v_j \leq v_1 + \mathtt{u} + \mathtt{b} \times (1 + \mathtt{r})$. Since $v_i \geq v_1$, we have $v_j - v_i \leq \mathtt{u} + \mathtt{b} \times (1 + \mathtt{r})$, as needed for Invariant 6.
2. No message sent by j arrives at i in α. Since the first send occurs at time 0 and $\mathtt{b}$ is the largest possible communication delay, the fact that i has not received the first message sent by j at time 0 implies that $t \leq \mathtt{b}$. Since both clocks start at 0, we have $v_j \leq \mathtt{b} \times (1 + \mathtt{r})$ and $v_i \geq 0$. Therefore, $v_j - v_i \leq \mathtt{u} + \mathtt{b} \times (1 + \mathtt{r})$, which suffices for Invariant 6.

5.1.2 Substitutivity Results

Theorem 5.4, which relates the set of traces of a composed automaton to the set of traces of component automata, is fundamental for compositional reasoning. We now introduce another important class of results, *substitutivity* results, that are useful for decomposing verification of composite automata. These results are best understood by viewing one of the components of a composition as the system and the other as the environment with which the system interacts.

The following result states that if a TA $\mathcal{A}_1$ can be shown to implement another one $\mathcal{A}_2$, with no assumptions about their environments, then $\mathcal{A}_1$ can be shown to implement $\mathcal{A}_2$ in a given environment $\mathcal{B}$.

Theorem 5.8 *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{B}$ are TAs, $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same external actions, and each of $\mathcal{A}_1$ and $\mathcal{A}_2$ is compatible with $\mathcal{B}$. If $\mathcal{A}_1 \leq \mathcal{A}_2$, then $\mathcal{A}_1 \| \mathcal{B} \leq \mathcal{A}_2 \| \mathcal{B}$.*

Commutativity of the composition operation together with repeated application of Theorem 5.8 gives the following corollary.

Corollary 5.9 *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{B}_1$, and $\mathcal{B}_2$ are TAs, $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same external actions, $\mathcal{B}_1$ and $\mathcal{B}_2$ have the same external actions, and each of $\mathcal{A}_1$ and $\mathcal{A}_2$ is compatible with each of $\mathcal{B}_1$ and $\mathcal{B}_2$. If $\mathcal{A}_1 \leq \mathcal{A}_2$ and $\mathcal{B}_1 \leq \mathcal{B}_2$, then $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$.*

We can strengthen Corollary 5.9 slightly by the following corollary: if $\mathcal{A}_1$ implements $\mathcal{A}_2$ in an environment $\mathcal{B}_2$, then $\mathcal{A}_1$ composed with an environment that is more restrictive than $\mathcal{B}_2$ (whose set of external behaviors is smaller than that of $\mathcal{B}_2$) implements $\mathcal{A}_2$ composed with $\mathcal{B}_2$.

Corollary 5.10 *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{B}_1$, and $\mathcal{B}_2$ are TAs, $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same external actions, $\mathcal{B}_1$ and $\mathcal{B}_2$ have the same external actions, and each of $\mathcal{A}_1$ and $\mathcal{A}_2$ is compatible with each of $\mathcal{B}_1$ and $\mathcal{B}_2$. If $\mathcal{A}_1 \| \mathcal{B}_2 \leq \mathcal{A}_2 \| \mathcal{B}_2$ and $\mathcal{B}_1 \leq \mathcal{B}_2$, then $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$.*

Proof: Let $\beta \in traces_{\mathcal{A}_1 \| \mathcal{B}_1}$. By Theorem 5.4, $\beta \lceil (E_{\mathcal{A}_1}, \emptyset) \in traces_{\mathcal{A}_1}$ and $\beta \lceil (E_{\mathcal{B}_1}, \emptyset) \in traces_{\mathcal{B}_1}$. Since $\mathcal{B}_1 \leq \mathcal{B}_2$, $\beta \lceil (E_{\mathcal{B}_1}, \emptyset) \in traces_{\mathcal{B}_2}$. Since $\mathcal{B}_1$ and $\mathcal{B}_2$ have the same external actions, it follows that $\beta \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. We have $\beta \lceil (E_{\mathcal{A}_1}, \emptyset) \in traces_{\mathcal{A}_1}$ and $\beta \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. By Theorem 5.4, $\beta \in traces_{\mathcal{A}_1 \| \mathcal{B}_2}$. Since $\mathcal{A}_1 \| \mathcal{B}_2 \leq \mathcal{A}_2 \| \mathcal{B}_2$ by assumption, $\beta \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$, as needed. $\square$

For other preorders, we also get substitutivity results, for example:

Theorem 5.11 *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{B}$ are TAs, $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same external actions, and each of $\mathcal{A}_1$ and $\mathcal{A}_2$ is compatible with $\mathcal{B}$.*

1. *If every closed trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$, then every closed trace of $\mathcal{A}_1 \| \mathcal{B}$ is a trace of $\mathcal{A}_2 \| \mathcal{B}$.*
2. *If every admissible trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$, then every admissible trace of $\mathcal{A}_1 \| \mathcal{B}$ is a trace of $\mathcal{A}_2 \| \mathcal{B}$.*
3. *If every non-Zeno trace of $\mathcal{A}_1$ is a trace of $\mathcal{A}_2$, then every non-Zeno trace of $\mathcal{A}_1 \| \mathcal{B}$ is a trace of $\mathcal{A}_2 \| \mathcal{B}$.*

Example 5.12 (A counterexample for a desirable substitutivity theorem). Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same external actions, $\mathcal{B}_1$ and $\mathcal{B}_2$ have the same external actions, and that each of $\mathcal{A}_1$ and $\mathcal{A}_2$ is compatible with each of $\mathcal{B}_1$ and $\mathcal{B}_2$. If we view $\mathcal{A}_2$ and $\mathcal{B}_2$ as specifications and want to prove that $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$, it would be useful to have a theorem that says if $\mathcal{A}_1 \| \mathcal{B}_2 \leq \mathcal{A}_2 \| \mathcal{B}_2$ and $\mathcal{A}_2 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$ then $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$. That is, if $\mathcal{A}_1$ implements $\mathcal{A}_2$ in the context of $\mathcal{B}_2$ and $\mathcal{B}_1$ implements $\mathcal{B}_2$ in the context of $\mathcal{A}_2$, we would like to conclude that $\mathcal{A}_1 \| \mathcal{B}_1$ implements $\mathcal{A}_2 \| \mathcal{B}_2$. We show by means of a counterexample that it is impossible to prove such a theorem. The problem arises with the infinite behaviors of $\mathcal{A}_1 \| \mathcal{B}_2$.

As examples for $\mathcal{A}_1$, $\mathcal{B}_1$, $\mathcal{A}_2$, and $\mathcal{B}_2$, consider, respectively, the automata `CatchUpA`, `CatchUpB`, `BoundedAlternateA`, and `BoundedAlternateB` in Figs. 5.2 and 5.3. All automata have the same set of actions, consisting of the external actions `a` and `b`. `CatchUpA` can perform an arbitrary number of `b` actions and can perform an `a` provided that `counta` $\leq$ `countb` and one time unit has elapsed since the occurrence of the last action. `CatchUpA` allows `counta` to increase to one more than `countb`. `CatchUpB` can perform an arbitrary number of `a` actions, and can perform a `b` provided that `counta` is at least one more than `countb`. `CatchUpB` allows `countb` to reach `counta`.

`BoundedAlternateA` has an infinite number of start states, each giving a different finite bound on the number of `a` actions it can perform. Similarly, `BoundedAlternateB` has an infinite number of start states, each giving a different finite bound on the number of `b` actions it can perform. Note that the absence of trajectory definitions in the specifications of these automata imply that they are timing-independent. That is, there is no constraint on the timing of actions.

The automata `CatchUpA` and `CatchUpB` strictly alternate `a`'s and `b`'s until a maximum count is reached, when put in the context of, respectively, `BoundedAlternateA` and `BoundedAlternateB`. Hence, on the one hand

$$(\texttt{CatchUpA} \| \texttt{BoundedAlternateB}) \leq (\texttt{BoundedAlternateA} \| \texttt{BoundedAlternateB})$$

and

$$(\texttt{BoundedAlternateA} \| \texttt{CatchUpB}) \leq (\texttt{BoundedAlternateA} \| \texttt{BoundedAlternateB}).$$

On the other hand, (`CatchUpA`$\|$`CatchUpB`) can perform an infinite sequence of alternating `a` and `b` actions, which is not allowed by (`BoundedAlternateA`$\|$`BoundedAlternateB`). Hence, (`CatchUpA`$\|$`CatchUpB`) does not implement (`BoundedAlternateA`$\|$`BoundedAlternateB`).

In Chapter 7, we revisit the substitutivity issue and prove Theorem 7.8, a variant of the desirable theorem considered in the above example, by assuming certain conditions on the environments $\mathcal{A}_2$ and $\mathcal{B}_2$.

```
automaton CatchUpA
  signature
    external a, b
  states
    counta: Nat := 0, countb: Nat := 0,
    now: Real := 0, next: discrete Real := 0
  transitions
    external a
      pre
        (counta ≤ countb)
          ∧ (now = next)
      eff
        counta := counta + 1;
        next := now + 1
    external b
      eff
      countb := countb + 1;
      next := now + 1
  trajectories
    stop when
      now = next
    evolve
      d(now) = 1
```

```
automaton CatchUpB
  signature
    external a, b
  states
    counta: Nat := 0, countb: Nat := 0,
    now: Real := 0, next: discrete Real := 0
  transitions
    external a
      eff
        counta := counta + 1
        next := now + 1
    external b
      pre
       (countb + 1) ≤ counta
         ∧ now = next
      eff
       countb := countb + 1;
       next := now + 1
  trajectories
    stop when
      now = next
    evolve
      d(now) = 1
```

FIGURE 5.2: CatchUpA and CatchUpB.

```
automaton BoundedAlternateA
  signature
    external a, b
  states
    myturn: Bool := true,
    maxout: Nat
  transitions
    external a                          external b
      pre                                 eff
        myturn ∧ (maxout > 0)          myturn := true
      eff
        myturn := false;
        maxout := maxout - 1
```

```
automaton BoundedAlternateB
  signature
    external a, b
  states
    myturn: Bool := false,
    maxout: Nat
  transitions
    external a                          external b
      eff                                 pre
        myturn := true                      myturn ∧ (maxout > 0)
                                          eff
                                            myturn := false;
                                            maxout := maxout - 1
```

FIGURE 5.3: BoundedAlternateA and BoundedAlternateB.

5.2 HIDING

We now define an operation that "hides" external actions of a timed automaton by reclassifying them as internal actions. This prevents them from being used for further communication and means that they are no longer included in traces. The operation is parametrized by a set of external actions: If $\mathcal{A}$ is a timed automaton $E \subseteq E_{\mathcal{A}}$, then $\mathsf{ActHide}(E, \mathcal{A})$ is the timed automaton $\mathcal{B}$ that is equal to $\mathcal{A}$ except that $E_{\mathcal{B}} = E_{\mathcal{A}} - E$ and $H_{\mathcal{B}} = H_{\mathcal{A}} \cup E$.

Lemma 5.13 *If* $E \subseteq E_{\mathcal{A}}$, *then* $\mathsf{ActHide}(E, \mathcal{A})$ *is a TA.*

The following lemma characterizes the traces of the automaton that results from applying a hiding operation.

Lemma 5.14 *If* $\mathcal{A}$ *is a TA and* $E \subseteq E_{\mathcal{A}}$*, then* $traces_{\mathsf{ActHide}(E,\mathcal{A})} = \{\beta \lceil (E_{\mathcal{A}} - E, \emptyset) \mid \beta \in traces_{\mathcal{A}}\}$.

Using Lemma 5.14, it is straightforward to establish that the hiding operation respects the implementation relation.

Theorem 5.15 *Suppose* $\mathcal{A}$ *and* $\mathcal{B}$ *are TAs with* $\mathcal{A} \leq \mathcal{B}$*, and suppose* $E \subseteq E_{\mathcal{A}}$*. Then* $\mathsf{ActHide}(E, \mathcal{A}) \leq \mathsf{ActHide}(E, \mathcal{B})$.

Example 5.16 (Clock and manager). Consider a simple system consisting of a "clock" and a "manager". The clock ticks once every $[\mathtt{c1}, \mathtt{c2}]$ time units and the manager issues a "grant" within $\mathtt{b}$ time units after counting $\mathtt{k} > 0$ ticks. We assume $0 \leq \mathtt{b} < \mathtt{c1} \leq \mathtt{c2}$. The problem is to prove upper and lower bounds on the time between successive grant actions.

Fig. 5.4 gives a formal specification of the clock in terms of the TA `Clock(c1, c2)` and the manager in terms of the TA `Manager(k, b)`. The full system with the `tick` actions hidden can be defined by

$$\mathtt{System} = \mathsf{ActHide}(\{\mathtt{tick}\}, \mathtt{Clock} \| \mathtt{Manager}).$$

Consider the automaton `Specification` displayed in Fig. 5.5. This automaton is equal to `Clock`, except for some renamings. We claim that the manager issues a grant once every $[\mathtt{c1} * \mathtt{k} - \mathtt{b}, \mathtt{c2} * \mathtt{k} + \mathtt{b}]$ time units. An equivalent formulation of this claim is

$$\mathtt{System} \leq \mathtt{Specification}(\mathtt{c1} * \mathtt{k} - \mathtt{b}, \mathtt{c2} * \mathtt{k} + \mathtt{b}).$$

In order to prove the claim, one may first establish that the predicate

$$\mathtt{Inv} \triangleq 0 \leq \mathtt{x} \leq \mathtt{c2} \wedge (\mathtt{count} = 0 \Rightarrow \mathtt{x} = \mathtt{y} \leq \mathtt{b}) \wedge 0 \leq \mathtt{count} \leq \mathtt{k}$$

defines an invariant of `System`, and use this to verify that the conjunction of `Inv` and

$$\mathtt{c1} * (\mathtt{k} - \mathtt{count}) - \mathtt{b} \leq \mathtt{z} - \mathtt{x} \leq \mathtt{c2} * (\mathtt{k} - \mathtt{count})$$

defines a forward simulation from `System` to $\mathtt{Specification}(\mathtt{c1} * \mathtt{k} - \mathtt{b}, \mathtt{c2} * \mathtt{k} + \mathtt{b})$.

5.3 EXTENDING TIMED AUTOMATA WITH BOUNDS

In this section, we define a new class of automata, "TA with bounds" where the basic definition of a timed automaton is extended with the notion of a task and a pair of bounds (a lower and an upper bound) for each task. We then define an operation that transforms a given TA with bounds to another TA. This operation supports specifying a system by thinking in terms of tasks and bounds as in the timed automata of Merritt *et al.* [7] and the phase transition systems of Maler *et al.* [12].

```
automaton Clock(c1,c2: Real) where 0 < c1 ∧ c1 ≤ c2
  signature
     external tick
  states
     x: Real := 0
  transitions
     external tick
       pre
         x ≥ c1
       eff
         x := 0
  trajectories
     stop when
       x = c2
     evolve
       d(x) = 1
```

```
automaton Manager(k: Int, b: Real) where b > 0 ∧ k > 0
  signature
     external tick, grant
  states
     y: Real := 0,
     count : Int := k
  transitions
     external tick
       eff
         count := count - 1;
         if count = 0 then y := 0
      external grant
        pre
          count = 0
        eff
          count := k
  trajectories
     stop when
       count = 0 ∧ y = b
     evolve
       d(y) = 1
```

FIGURE 5.4: Automata Clock and Manager.

```
automaton Specification(lb,ub: Real) where 0 < lb ∧ lb ≤ ub
  signature
    external grant
  states
    z: Real := 0
  transitions
    external grant
      pre
        z ≥ lb
      eff
        z := 0
  trajectories
    stop when
      z = ub
    evolve
      d(z) = 1
```

FIGURE 5.5: Automaton `Specification`.

In defining the operation for extending timed automata with bounds, we restrict attention to a class of automata where the enabling and disabling of actions during trajectories follow certain rules. Specifically, our operation is defined on automata in which each action is enabled or disabled throughout an entire trajectory, or becomes enabled once during a trajectory and remains so until the end of that trajectory. The given restrictions ensure that the result of applying the operation to a TA is another TA and that the resulting TA satisfies the restrictions.

Let $\mathcal{A}$ be a TA, C a set of actions of $\mathcal{A}$, and $\mathcal{T}$ the set of trajectories of $\mathcal{A}$. We say that $\mathcal{T}$ is *well formed* with respect to C if for each $\tau \in \mathcal{T}$ and for each $t \in dom(\tau)$ both of the following conditions hold:

1. (Stability) If C is enabled in $\tau(t)$, then for all $t' \in dom(\tau)$ with $t < t'$, C is enabled in $\tau(t')$.
2. (Left-closedness) If C is not enabled in $\tau(t)$, then there exists a $t' \in dom(\tau)$ with $t < t'$ such that C is not enabled in $\tau(t')$.

A *TA with bounds*, $\mathcal{A} = (\mathcal{B}, C, l, u)$ consists of

- a timed automaton $\mathcal{B} = (X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$.
- a set $C \subseteq E \cup H$ of actions called a *task*; we assume that $\mathcal{T}$ is well formed with respect to C.
- a lower time bound $l \in R^{\geq 0}$ and an upper time bound $u \in R^{\geq 0} \cup \{\infty\}$ with $l \leq u$.

Lower and upper bounds are used to specify how much time is allowed to pass between the enabling and the performance of an action. If l is the lower bound for a task C, then an action in C must remain enabled at least for l time units before being performed. If u is the upper bound for a task C, then an action in C can remain enabled at most u time units without being performed: it must either be performed or become disabled within u time units.

We now define an operation Extend, which transforms a TA $\mathcal{A}$ with bounds to another TA $\mathcal{A}'$ that incorporates the new bounds, in addition to the timing constraints already present in $\mathcal{A}$. Let $\mathcal{A} = (\mathcal{B}, C, l, u)$ be a TA with bounds, where $\mathcal{B} = (X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$. Then $\mathsf{Extend}(\mathcal{A})$ is the TA $\mathcal{A}' = (X', Q', \Theta', E', H', \mathcal{D}', \mathcal{T}')$ where

- $X' = X \cup \{now, \mathit{first}, \mathit{last}\}$, where
  1. now, first, and last are new variables that do not appear in X.
  2. now is an analog variable such that $type(now) = \mathsf{R}$.
  3. first and last are discrete variables where $type(\mathit{first}) = \mathsf{R}$ and $type(\mathit{last}) = \mathsf{R} \cup \{\infty\}$.
- $Q' = \{\mathbf{x} \in val(X') \mid \mathbf{x} \lceil X \in Q\}$.
- Θ' consists of all the states $\mathbf{x} \in Q'$ that satisfy the following conditions:
  1. $\mathbf{x} \lceil X \in \Theta$.
  2. $\mathbf{x}(now) = 0$.
  3. $\mathbf{x}(\mathit{first}) = \begin{cases} l & \text{if } C \text{ is enabled in } \mathbf{x} \lceil X, \\ 0 & \text{otherwise.} \end{cases}$

     $\mathbf{x}(\mathit{last}) = \begin{cases} u & \text{if } C \text{ is enabled in } \mathbf{x} \lceil X, \\ \infty & \text{otherwise.} \end{cases}$
- $E' = E$ and $H' = H$. We write $A' \triangleq E' \cup H'$.
- if $a \in A'$, then $(\mathbf{x}, a, \mathbf{x}') \in \mathcal{D}'$ exactly if all of the following conditions hold:
  1. $(\mathbf{x} \lceil X) \xrightarrow{a}_{\mathcal{A}} (\mathbf{x}' \lceil X)$.
  2. $\mathbf{x}'(now) = \mathbf{x}(now)$.
  3. (a) If $a \in C$, then $\mathbf{x}(\mathit{first}) \leq \mathbf{x}(now)$.
     (b) If C is enabled both in $\mathbf{x} \lceil X$ and $\mathbf{x}' \lceil X$ and $a \notin C$, then $\mathbf{x}(\mathit{first}) = \mathbf{x}'(\mathit{first})$ and $\mathbf{x}(\mathit{last}) = \mathbf{x}'(\mathit{last})$.
     (c) If C is enabled in $\mathbf{x}' \lceil X$ and either C is not enabled in $\mathbf{x} \lceil X$ or $a \in C$, then $\mathbf{x}'(\mathit{first}) = \mathbf{x}(now) + l$ and $\mathbf{x}'(\mathit{last}) = \mathbf{x}(now) + u$.
     (d) If C is not enabled in $\mathbf{x}' \lceil X$, then $\mathbf{x}'(\mathit{first}) = 0$ and $\mathbf{x}'(\mathit{last}) = \infty$.

- $\mathcal{T}'$ is a set that consists of all $\tau \in trajs(Q')$ that satisfy the following conditions:
  1. $(\tau \downarrow X) \in \mathcal{T}$.
  2. $d(now) = 1$.
  3. (a) If for all $t \in dom(\tau)$, C is enabled in $\tau \downarrow X(t)$ then *first* and *last* are constant throughout τ.

     (b) If for all $t \in dom(\tau)$, C is disabled in $\tau \downarrow X(t)$ then *first* and *last* are constant throughout τ.

     (c) If for all $t' \in [0, t)$, C is disabled in $\tau(t')$ and for all $t' \in dom(\tau) - [0, t)$, C is enabled in $\tau(t')$ then

        i. *first* and *last* are constant in $[0, t)$.
        ii. $\tau(t)(first) = \tau(t)(now) + l$ and $\tau(t)(last) = \tau(t)(now) + u$.
        iii. *first* and *last* are constant in $dom(\tau) - [0, t)$.

     (d) $now \leq last$.

The transformation is based on the idea of augmenting the state of the original automaton with a variable to represent current time (now) and the earliest time ($first$) and the latest time ($last$) a task can be performed. All these variables represent time in absolute terms. Item $3(a)$ in the definition of $\mathcal{D}'$ expresses the new lower bound constraint and Item $3(d)$ in the definition of $\mathcal{T}'$ the new upper bound constraint.

Let $\mathcal{A}$ be a TA with bounds $(\mathcal{B}, C, l, u)$. In a start state $\mathbf{x}$ of $\mathsf{Extend}(\mathcal{A})$, the variables *first* and *last* are initialized to l and u, respectively, if C is enabled in $\mathbf{x}$. If C is not enabled in $\mathbf{x}$, then *first* is set to 0 and *last* is set to ∞. Step $3(c)$ in the definition of $\mathcal{D}'$ and Step $3(c)$ in the definition of $\mathcal{T}'$ show how the variables *first* and *last* are updated. When C becomes newly enabled by a discrete transition or when a C action leads to a state in which C is enabled, *first* is set to $now + l$ and *last* is set to $now + u$. The variables *first* and *last* are updated similarly when C becomes newly enabled in the course of a trajectory.

Theorem 5.17 *Suppose that* $\mathcal{A} = (\mathcal{B}, C, l, u)$ *is a TA with bounds. Then* $\mathsf{Extend}(\mathcal{A})$ *is a TA with a set of trajectories that is well formed with respect to* C.

Proof: The proof follows from the definitions of TA and the operation Extend. Step 3(a) in the definition of $\mathcal{D}'$ adds a new lower bound constraint, which makes enabling start at some particular time. Step 3(b) in the definition of $\mathcal{T}'$, adds a new upper bound constraint, which stops trajectories at a particular time and does not add any enabling or disabling to trajectories. $\square$

In the rest of this section, we sometimes speak of variables, states, and traces of a TA with bounds. If $\mathcal{A} = (\mathcal{B}, C, l, u)$ is a TA with bounds, variables, states, and traces of $\mathcal{A}$ refer to, respectively, the states and the traces of the underlying automaton $\mathcal{B}$.

Theorem 5.18 *Suppose $\mathcal{A}$ is a TA with bounds. Then $traces_{\mathsf{Extend}(\mathcal{A})} \subseteq traces_{\mathcal{A}}$.*

Proof: Let $F : Q' \rightarrow Q$ be defined as follows: $F(\mathbf{x}) = \mathbf{x} \lceil X$, where X is the set of internal variables of $\mathcal{A}$. It is easy to check that F is a refinement from $\mathsf{Extend}(\mathcal{A})$ to $\mathcal{A}$. By Theorem 4.27 and Corollary 4.23, we conclude that $traces_{\mathsf{Extend}(\mathcal{A})} \subseteq traces_{\mathcal{A}}$. □

Lemma 5.19 *Suppose that $\mathcal{A} = (\mathcal{B}, C, l, u)$ is a TA with bounds. For any reachable state $\mathbf{x}$ of $\mathsf{Extend}(\mathcal{A})$, if C is enabled in $\mathbf{x} \lceil X$ in $\mathcal{A}$, then $\mathbf{x}(last) \leq \mathbf{x}(now) + u$.*

Proof: Consider a closed execution α of $\mathsf{Extend}(\mathcal{A})$. Using Axioms **T1** and **T2** for trajectories, we can write α as a concatenation of closed execution fragments $\alpha_0 \frown \alpha_1 \frown \cdots \frown \alpha_k$, where α_0 is a point trajectory and each α_i for $i \geq 1$ is either a trajectory or a discrete action surrounded by two point trajectories such that for all $0 \leq i \leq k-1$, $\alpha_i.lstate = \alpha_{i+1}.fstate$. We prove the invariant by induction on the length k of the sequence of execution fragments.

For the base case, suppose that C is enabled in $\alpha_0.fstate \lceil X$. Since α is an execution, we know that $\alpha_0.fstate$ is a start state of $\mathsf{Extend}(\mathcal{A})$. By definition of $\mathsf{Extend}(\mathcal{A})$, $\alpha_0.fstate(last) = u$. Since $\alpha_0.fstate(now) = 0$, $\alpha_0.fstate(last) \leq \alpha_0.fstate(now) + u$, as required.

For the inductive step, we assume that the property is true for the sequence $\alpha_0 \frown \alpha_1 \frown \cdots \frown \alpha_k$, and show that it is true in the sequence α_{k+1} in $\alpha_0 \frown \alpha_1 \frown \cdots \frown \alpha_k \frown \alpha_{k+1}$. There are two cases to consider depending on whether α_{k+1} is a discrete action surrounded by two point trajectories or a trajectory.

1. α_{k+1} is an action a surrounded by two point trajectories $\wp(\mathbf{y})$ and $\wp(\mathbf{y}')$. Suppose that C is enabled in $\mathbf{y}' \lceil X$ in $\mathcal{A}$. There are two subcases to consider:

   a) C is enabled in $\mathbf{y} \lceil X$ and $a \notin C$. Then, $\mathbf{y}'(last) = \mathbf{y}(last)$ and $\mathbf{y}'(now) = \mathbf{y}(now)$. By inductive hypothesis, $\mathbf{y}(last) \leq \mathbf{y}(now) + u$. Therefore, $\mathbf{y}'(last) \leq \mathbf{y}'(now) + u$, as needed.

   b) C is disabled in $\mathbf{y} \lceil X$ or $a \in C$. Then, by definition of $\mathsf{Extend}(\mathcal{A})$, $\mathbf{y}'(last) = \mathbf{y}'(now) + u$, which suffices.

2. α_{k+1} is a trajectory. Suppose that C is enabled in $\alpha_{k+1}.lstate \lceil X$ in $\mathcal{A}$. There are two subcases to consider:

   a) C is enabled in $\alpha_{k+1}.fstate \lceil X$ in $\mathcal{A}$. By inductive hypothesis $\alpha_{k+1}.fstate(last) \leq \alpha_{k+1}.fstate(now) + u$. By the well-formedness assumption, we know that C must be enabled throughout α_{k+1} and by definition of Extend($\mathcal{A}$) *last* is constant throughout α_{k+1}. Since the value of *now* increases, it is easy to see that $\alpha_{k+1}.lstate(last) \leq \alpha_{k+1}.lstate(now) + u$.

b) C is disabled in $\alpha_{k+1}.fstate \lceil X$ in $\mathcal{A}$. Then, since it is enabled in $\alpha_{k+1}.lstate \lceil X$ by the well-formedness assumption, it becomes enabled at some point t in the domain of α_{k+1} and remains enabled thereafter. Therefore, $\alpha_{k+1}(t)(last) = \alpha_{k+1}(t)(now) + u$, by definition of Extend($\mathcal{A}$). Since *last* remains constant after it is set and the value of *now* increases, $\alpha_{k+1}.lstate(last) \leq \alpha_{k+1}.lstate(now) + u$ holds. □

The following theorem shows that the executions of an automaton obtained by applying the transformation Extend to a TA with bounds respect the time bounds specified by the lower bound l and the upper bound u.

Theorem 5.20 *Let $\mathcal{A} = (\mathcal{B}, C, l, u)$ be a TA with bounds. Then,*

1. *there does not exist a closed execution fragment α of* Extend($\mathcal{A}$) *from a reachable state, where $\alpha.ltime > u$, C is enabled in $\mathcal{A}$ in all the states of $\alpha \lceil (A, X)$ and no action in C occurs in α.*
2. *there does not exist a closed execution fragment α of* Extend($\mathcal{A}$) *from a reachable state, where $\alpha.ltime < l$, such that C is not enabled in $\mathcal{A}$ in the first state of $\alpha \lceil (A, X)$ and an action in C occurs in α.*

Proof:

1. Suppose, for the sake of contradiction, that there exists a closed execution fragment $\alpha = \tau_0\ a_1 \tau_1\ a_2 \ldots\ \tau_n$ of Extend($\mathcal{A}$) from a reachable state, where $\alpha.ltime > u$, C is enabled in $\mathcal{A}$ in all the states of $\alpha \lceil (A, X)$, and none of the a_i in α is in C. By definition of trajectories for Extend($\mathcal{A}$) it must be the case that $\alpha.lstate(now) \leq \alpha.lstate(last)$.

   Since C is enabled in $\mathcal{A}$ in all states in α, by Lemma 5.19 we have $\alpha.fstate(last) \leq \alpha.fstate(now) + u$. By definition of Extend($\mathcal{A}$), *last* remains constant throughout α; therefore, $\alpha.lstate(last) = \alpha.fstate(last)$. Since $\alpha.fstate(last) \leq \alpha.fstate(now) + u$, it follows that $\alpha.lstate(last) \leq \alpha.fstate(now) + u$. By definition of α, we have $\alpha.lstate(now) = \alpha.fstate(now) + \alpha.ltime$. It follows that $\alpha.fstate(now) + \alpha.ltime \leq \alpha.fstate(now) + u$. This implies $\alpha.ltime \leq u$. But this gives us the needed contradiction since $\alpha.ltime > u$.
2. We assume that α is a closed execution fragment of Extend($\mathcal{A}$) from a reachable state where $\alpha.ltime < l$, such that C is not enabled in $\mathcal{A}$ in the first state of α and an action in C occurs in α. Let $(\mathbf{x}, a, \mathbf{x}')$ be the first discrete transition of Extend($\mathcal{A}$) in α such that $a \in C$. We show that the condition $\mathbf{x}(first) \leq \mathbf{x}(now)$, which has to hold for the discrete transition to occur, cannot be true, hence arrive at a contradiction.

By Theorem 5.17, the set of trajectories of Extend($\mathcal{A}$) is well formed with respect to C. Therefore, C can become enabled by either a discrete transition or during a trajectory, and remains enabled until the occurrence of $(\mathbf{x}, a, \mathbf{x}')$.

a) C becomes enabled by a discrete transition and remains enabled in $\mathcal{A}$ until the occurrence of $(\mathbf{x}, a, \mathbf{x}')$. Let $(\mathbf{y}, b, \mathbf{y}')$ be the discrete transition of $\mathcal{A}$ that enables C. From step 3(c) in the definition of $\mathcal{D}'$ we know that *first* is set to $\mathbf{y}(now) + l$ when C becomes enabled. From step 3(b) in the definition of $\mathcal{D}'$ and step 3(a) in the definition of $\mathcal{T}'$, we know that it remains constant so that $\mathbf{x}(first) = \mathbf{y}(now) + l$. Since $(\mathbf{x}, a, \mathbf{x}')$ is a discrete transition of Extend($\mathcal{A}$), it must be the case that $\mathbf{x}(first) \leq \mathbf{x}(now)$. Since $\mathbf{x}(now) \leq \mathbf{y}(now) + \alpha.ltime$ and $\mathbf{x}(first) = \mathbf{y}(now) + l$ it follows that $\mathbf{y}(now) + l \leq \mathbf{y}(now) + \alpha.ltime$. But we know by assumption that $\alpha.ltime < l$, which gives the needed contradiction.

b) C becomes enabled at some point in the course of a trajectory τ and remains enabled in $\mathcal{A}$ until the occurrence of $(\mathbf{x}, a, \mathbf{x}')$. Let $\mathbf{y}$ be a state in the range of τ where C becomes enabled. From step 3(c) in the definition of $\mathcal{T}'$ we know that *first* is set to $\mathbf{y}(now) + l$ when C becomes enabled and it remains constant in τ so that $\mathbf{x}(first) = \mathbf{y}(now) + l$. From step 3(b) in the definition of $\mathcal{D}'$ and step 3(a) in the definition of $\mathcal{T}'$, we know that *first* remains constant until the occurrence of $(\mathbf{x}, a, \mathbf{x}')$. Since $(\mathbf{x}, a, \mathbf{x}')$ is a discrete transition of Extend($\mathcal{A}$), it must be the case that $\mathbf{x}(first) \leq \mathbf{x}(now)$. Since $\mathbf{x}(now) \leq \mathbf{y}(now) + \alpha.ltime$ and $\mathbf{x}(first) = \mathbf{y}(now) + l$ it follows that $\mathbf{y}(now) + l \leq \mathbf{y}(now) + \alpha.ltime$. But we know by assumption that $\alpha.ltime < l$, which gives the needed contradiction.

Example 5.21 (Fischer's algorithm specified using tasks and bounds). In Example 4.5 we presented the specification of Fischer's mutual exclusion algorithm as a TA. This example illustrates an alternative way of specifying the same algorithm by using a TA with bounds.

Recall that, formally, we define a TA with bounds as a TA augmented with a single task along with lower and upper bounds for that task. The automaton in Fig. 5.6 is, however, augmented with a set of tasks and bounds (we omit from the figure those transition definitions that are the same as in Example 4.5). This is for notational convenience and the automaton in Fig. 5.6 should be viewed as the automaton representing the cumulative result of adding in successive steps two tasks for each index. We assume that Extend is applied once for each task. That is, we start with the timing-independent version of `FischerME`, apply Extend to the automaton augmented with the task {`set(i)`} to add the lower bound 0 and the upper bound `u_set`, then apply Extend to the resulting automaton augmented with {`check(i)`} to add the lower bound `l_check` and the upper bound ∞. Such two successive applications are allowed

```
type Index = enumeration of p1, p2, p3, p4

type PcValue =  enumeration of rem, test, set, check,
                               leavetry, crit, reset, leaveexit

automaton FischerME(u_set, l_check: Real)
 where u_set ≥ 0 ∧ l_check ≥0 ∧ u_set < l_check
 signature
  external try(i:Index), crit(i:Index), exit(i:Index), rem(i:Index)
  internal test(i:Index), set(i:Index),
           check(i:Index), reset(i:Index)
 states
    x: Null[Index] := nil,
    pc: Array[Index,PcValue] := constant(rem)
 transitions
    internal test(i)
       pre
         pc[i] = test
       eff
         if x = nil then
            pc[i] := set
    internal set(i)
       pre
         pc[i] = set
       eff
         x := embed(i);
         pc[i] := check
    internal check(i)
       pre
         pc[i] = check
       eff
         if x = embed(i) then pc[i] := leavetry
         else pc[i] := test
 tasks
     set = {set(i)} for i: Index; check = {check(i)} for i: Index

 bounds
     set = [0,u_set]; check = [l_check, infty]
```

FIGURE 5.6: Fischer's mutual exclusion algorithm with bounds.

since the result of the first application of Extend satisfies the the well-formedness conditions for the set of trajectories.

The result of these successive applications yields an automaton similar to the one in Example 4.5. The only difference is that the mechanical application of the transformation would reset the value of `firstcheck[i]` to 0 as an affect of `check(i)` while we do not reset `firstcheck[i]` explicitly in Example 4.5, when it becomes disabled. This is because we make use of the facts that the value of `firstcheck[i]` is used only in determining whether `check(i)` is enabled and that `check(i)` becomes enabled only in the poststate of `set(i)`, which also sets the value of `firstcheck[i]`. Note that this discrepancy does not give rise to any difference in the behaviors of the two automata. □

CHAPTER 6

Timed I/O Automata

In this chapter we refine the timed automaton model of Chapter 4 by distinguishing between input and output actions. Typically, an interaction between a system and its environment is modeled by using output and input actions to represent, respectively, the external events under the control of the system and the environment. We extend the results on simulation relations and composition from Chapters 4 and 5 to this new setting. We also introduce special kinds of timed I/O automata: I/O feasible, progressive, and receptive TIOAs.

6.1 DEFINITION OF TIMED I/O AUTOMATA

A *timed I/O automation (TIOA)* $\mathcal{A}$ is a tuple $(\mathcal{B}, I, O)$ where

- $\mathcal{B} = (X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$ is a timed automaton.
- I and O partition E into *input* and *output actions*, respectively. Actions in $L \triangleq H \cup O$ are called *locally controlled*; we again write $A \triangleq E \cup H$.
- the following additional axioms are satisfied:

  E1 (Input action enabling). For every $\mathbf{x} \in Q$ and every $a \in I$, there exists $\mathbf{x}' \in Q$ such that $\mathbf{x} \xrightarrow{a} \mathbf{x}'$.

  E2 (Time-passage enabling). For every $\mathbf{x} \in Q$, there exists $\tau \in \mathcal{T}$ such that $\tau.fstate = \mathbf{x}$ and either

  1. $\tau.ltime = \infty$ or
  2. τ is closed and some $l \in L$ is enabled in $\tau.lstate$.

Input action enabling is the input enabling condition of ordinary I/O automata [45]; it says that a TIOA is able to perform an input action at any time. The time-passage enabling condition says that a TIOA either allows time to advance forever, or it allows time to advance for a while, up to a point where it is prepared to react with some locally controlled action. The condition ensures what is called time reactivity in [46] and timelock freedom in [47], that is, whenever time progress stops there exists at least one enabled transition. Because TIOAs have no external variables, **E1** and **E2** are slightly simpler than the corresponding axioms for HIOAs.

Notation: As we did for TAs, we often denote the components of a TIOA $\mathcal{A}$ by $\mathcal{B}_{\mathcal{A}}$, $I_{\mathcal{A}}$, $O_{\mathcal{A}}$, $X_{\mathcal{A}}$, $Q_{\mathcal{A}}$, $\Theta_{\mathcal{A}}$, etc., and those of a TIOA $\mathcal{A}_i$ by H_i, I_i, O_i, X_i, Q_i, Θ_i, etc. We sometimes omit these subscripts, where no confusion is likely. We abuse notation slightly by referring to a TIOA $\mathcal{A}$ as a TA when we intend to refer to $\mathcal{B}_{\mathcal{A}}$.

Example 6.1 (TAs viewed as TIOAs). The automaton `TimedChannel` described in Example 4.1 can be turned into a TIOA by classifying the `send` actions as inputs and the `receive` actions as outputs. Since there is no precondition for `send` actions, they are enabled in each state, so clearly the input enabling condition **E1** holds. It is also easy to see that Axiom **E2** holds: in each state either `queue` is nonempty, in which case a `receive` output action is enabled after a point trajectory, or `queue` is empty, in which case time can advance forever.

The automaton `ClockSync` of Example 4.6 can be turned into a TIOA by classifying the `send` actions as outputs and the `receive` actions as inputs. Axiom **E1** then holds trivially. Axiom **E2** holds since from each state either time can advance forever or we have an outgoing trajectory (possibly of length 0) to a state in which `physclock` = `nextsend` and from there a `send` output action is enabled.

6.2 EXECUTIONS AND TRACES

An *execution fragment*, *execution*, *trace fragment*, or *trace* of a TIOA $\mathcal{A}$ is defined to be an execution fragment, execution, trace fragment, or trace of the underlying TA $\mathcal{B}_{\mathcal{A}}$, respectively.

We say that an execution fragment of a TIOA is *locally-Zeno* if it is Zeno and contains infinitely many locally controlled actions, or equivalently, if it has finite limit time and contains infinitely many locally controlled actions.

6.3 SPECIAL KINDS OF TIMED I/O AUTOMATA

6.3.1 Feasible and I/O Feasible TIOAs

A TIOA $\mathcal{A} = (\mathcal{B}, I, O)$ is defined to be *feasible* provided that its underlying TA $\mathcal{B}$ is feasible according to the definition given in Section 4.3. As noted in Section 4.3, feasibility is a basic requirement that any TA (or TIOA) should satisfy. I/O feasibility is a strengthened version of feasibility that takes inputs into account. It says that the automaton is capable of providing some response from any state, for any sequence of input actions and any amount of intervening time-passage. In particular, it should allow time to pass to infinity if the environment does not submit any input actions. Formally, we define a TIOA to be *I/O feasible* provided that, for each state $\mathbf{x}$ and each $(I, \emptyset)$-sequence β, there is some execution fragment α from $\mathbf{x}$ such that $\alpha \lceil (I, \emptyset) = \beta$. That is, an I/O feasible TIOA accommodates arbitrary input actions occurring at arbitrary times. The given $(I, \emptyset)$-sequence β describes the inputs and the amounts of intervening times.

6.3.2 Progressive TIOAs

A progressive TIOA never generates infinitely many locally controlled actions in finite time. Formally, a TIOA $\mathcal{A}$ is *progressive* if it has no locally-Zeno execution fragments.

The following lemma says that any progressive TIOA is capable of advancing time forever.

Lemma 6.2 *Every progressive TIOA is feasible.*

Proof: Let $\mathcal{A}$ be a progressive TIOA and let $\mathbf{x}$ be a state of $\mathcal{A}$. Since $\mathcal{A}$ is a TIOA it satisfies Axiom **E2**. We construct an admissible execution fragment $\alpha = \alpha_0 \frown \alpha_1 \frown \alpha_2 \cdots$ from $\mathbf{x}$ as follows:

1. $\alpha_0 = \wp(\mathbf{x})$.
2. For each $i > 0$,
   (a) if there exists a trajectory τ from $\alpha_{i-1}.lstate$ such that $\tau.ltime = \infty$, then α_i is the final execution fragment in the sequence and $\alpha_i = \tau$.
   (b) otherwise, let τ_i be a closed execution fragment from $\alpha_{i-1}.lstate$ such that $l \in L$ is enabled in $\tau_i.lstate$. Define $\alpha_i = \tau_i\ l\ \tau_{i+1}$ where $\tau_{i+1} = \wp(\mathbf{y})$ and $\tau_i.lstate \xrightarrow{l} \mathbf{y}$.

The above construction either ends after finitely many stages such that the last trajectory of α is admissible or goes through infinitely many stages such that α contains infinitely many local actions. In the former case, we know that α is admissible since it ends with an admissible tracjectory. In the latter case, since $\mathcal{A}$ is progressive, the fact that α has infinitely many local actions implies that α is admissible, as needed. □

The following lemma says that a progressive TIOA is capable of allowing any amount of time to pass from any state.

Lemma 6.3 *Let $\mathcal{A}$ be a progressive TIOA, let $\mathbf{x}$ be a state of $\mathcal{A}$, and let $\tau \in trajs(\emptyset)$. Then there exists an execution fragment α of $\mathcal{A}$ such that $\alpha.fstate = \mathbf{x}$ and $\alpha \lceil (I, \emptyset) = \tau$.*

Proof: The result follows from the construction used in the proof of Lemma 6.2. Let α be an admissible execution fragment from $\mathbf{x}$ constructed as in the proof of Lemma 6.2. Let α' be a prefix of α such that $\alpha' \lceil (\emptyset, \emptyset) = \tau$. Since our construction uses no actions from I, we have $\alpha' \lceil (I, \emptyset) = \alpha' \lceil (\emptyset, \emptyset) = \tau$, as needed. □

The following theorem says that a progressive TIOA is capable not just of allowing arbitrary amounts of time to pass, but also of allowing arbitrary input actions at arbitrary times.

Theorem 6.4 *Every progressive TIOA is I/O feasible.*

Proof: Let $\mathcal{A}$ be a progressive TIOA, let $\mathbf{x}$ be a state of $\mathcal{A}$, and let $\beta = \tau_0\, a_1\, \tau_1\, a_2\, \tau_2 \cdots$ be an $(I, \emptyset)$-sequence. We construct a finite or infinite sequence $\alpha_0\, \alpha_1\, \cdots$ of execution fragments such that

1. $\alpha_0.fstate = \mathbf{x}$.
2. for each nonfinal index i, $\alpha_i.lstate = \alpha_{i+1}.fstate$.
3. for each i, $(\alpha_0 \frown \alpha_1 \frown \cdots \frown \alpha_i) \lceil (I, \emptyset) = \tau_0\, a_1\, \tau_1 \cdots \tau_i$.

The construction is carried out recursively. To define α_0, we start with $\mathbf{x}$ and use Lemma 6.3 to span τ_0. For $i > 0$, we define α_i by starting with $\alpha_{i-1}.lstate$, using Axiom **E1** to perform the input action a_i and move to a new state and then using Lemma 6.3 to span τ_i.

Let $\alpha = \alpha_0 \frown \alpha_1 \frown \cdots$. By Lemma 3.8, α is an execution fragment of $\mathcal{A}$ from $\mathbf{x}$ such that $\alpha \lceil (I, \emptyset) = \beta$, as needed. □

6.3.3 Receptive Timed I/O Automata

In this section, we define the notion of *receptiveness* for TIOAs. A TIOA will be defined to be receptive provided that it admits a *strategy* for resolving its nondeterministic choices that never generate infinitely many locally controlled actions in finite time. This notion has an important consequence: A receptive TIOA provides some response from any state, for any sequence of discrete input actions at any times. This implies that the automaton has a nontrivial set of execution fragments, in fact, it has execution fragments that accommodate any inputs from the environment. The automaton cannot simply stop at some point and refuse to allow time to elapse; it must allow time to pass to infinity if the environment does so. Previous studies of receptiveness properties include [8, 41, 48, 49]. The notion of receptiveness for TIOAs as discussed here is a special case of the same notion for HIOAs [6].

We build our definition of receptiveness on our earlier definition of progressive TIOAs. Namely, we define a *strategy* for resolving nondeterministic choices and define receptiveness in terms of the existence of a progressive strategy.

We define a *strategy* for a TIOA $\mathcal{A}$ to be a TIOA $\mathcal{A}'$ that differs from $\mathcal{A}$ only in that $\mathcal{D}' \subseteq \mathcal{D}$ and $\mathcal{T}' \subseteq \mathcal{T}$. That is, we require

- $\mathcal{D}' \subseteq \mathcal{D}$,
- $\mathcal{T}' \subseteq \mathcal{T}$,
- $X = X'$, $Q = Q'$, $\Theta = \Theta'$, $H = H'$, $I = I'$, and $O = O'$.

Our strategies are nondeterministic and memoryless. They provide a way of choosing some of the evolutions that are possible from each state $\mathbf{x}$ of $\mathcal{A}$. The fact that the state set Q' of $\mathcal{A}'$ is the same as the state set Q of $\mathcal{A}$ implies that $\mathcal{A}'$ chooses evolutions from every state of $\mathcal{A}$.

Notions of strategy have been used also in previous studies of receptiveness [8, 41, 48, 49]. However, in these earlier works, strategies have been formalized using two-player games rather than automata. Defining strategies using automata allows us to avoid introducing extra mathematical machinery.

Lemma 6.5 *If $\mathcal{A}'$ is a strategy for $\mathcal{A}$, then every execution fragment of $\mathcal{A}'$ is also an execution fragment of $\mathcal{A}$.*

We define a TIOA to be *receptive* if it has a progressive strategy. The following theorem says that any receptive TIOA can respond to any inputs from the environment.

Theorem 6.6 *Every receptive TIOA is I/O feasible.*

Proof: The proof follows from the definitions, Theorem 6.4, and Lemma 6.5. □

Example 6.7 (Progressive and receptive TIOAs). The time-bounded channel automaton described in Example 4.1 is not progressive since it allows for an infinite execution in which `send` and `receive` actions alternate without any passage of time in between. The time-bounded channel automaton is receptive, however, as we may construct a progressive strategy for it by adding a condition `head(queue).deadline = now` to the precondition of the `receive` action. In this way we enforce that the channel operates maximally slow and messages are delivered only at their delivery deadline. The clock synchronization automaton of Example 4.6 is progressive (and therefore receptive) since it can generate only a locally controlled action each time its physical clock advances by u time units and the real time that elapses between two locally produced actions is at least $u \times (1-r)$ time units.

6.4 IMPLEMENTATION RELATIONSHIPS

Two TIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *comparable* if their inputs and outputs coincide, that is, if $I_1 = I_2$ and $O_1 = O_2$. If $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, then $\mathcal{A}_1 \leq \mathcal{A}_2$ is defined to mean that the traces of $\mathcal{A}_1$ are included among those of $\mathcal{A}_2$: $\mathcal{A}_1 \leq \mathcal{A}_2 \triangleq traces_{\mathcal{A}_1} \subseteq traces_{\mathcal{A}_2}$.

Lemma 6.8 *Let $\mathcal{A}_1$, $\mathcal{A}_2$ be two comparable TIOAs and let $\mathcal{B}_1$, $\mathcal{B}_2$ be, respectively, the underlying TAs for $\mathcal{A}_1$ and $\mathcal{A}_2$. Then $\mathcal{B}_1$ and $\mathcal{B}_2$ are comparable and $\mathcal{A}_1 \leq \mathcal{A}_2$ iff $\mathcal{B}_1 \leq \mathcal{B}_2$.*

Proof: The proof follows from the definitions. □

6.5 SIMULATION RELATIONS

The definition of forward simulation for TIOAs is the same as for TAs. Formally, if $\mathcal{A}_1 = (\mathcal{B}_1, I_1, O_1)$ and $\mathcal{A}_2 = (\mathcal{B}_2, I_2, O_2)$ are two comparable TIOAs, then a forward simulation from $\mathcal{A}_1$ to $\mathcal{A}_2$ is a forward simulation from $\mathcal{B}_1$ to $\mathcal{B}_2$.

Theorem 6.9 *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable TIOAs and there is a forward simulation from $\mathcal{A}_1$ to $\mathcal{A}_2$, then $\mathcal{A}_1 \leq \mathcal{A}_2$.*

The definitions and results about backward simulations, history, and prophecy relations for timed automata from Chapter 4 carry over to timed automata with input and output distinction in a similar fashion.

CHAPTER 7

Operations on Timed I/O Automata

7.1 COMPOSITION

In this chapter we define the operations of composition and hiding and present projection, pasting, and substitutivity results for TIOAs. We revisit the special kinds of TIOAs introduced in Chapter 6 and show that the classes of progressive and receptive TIOA are closed under composition, while this is not true for the class of I/O feasible automata.

7.1.1 Definitions and Basic Results

The definition of composition for TIOAs is based not only on the corresponding definition for TAs, but also takes the input/output structure into account. We require that precisely one component should control any given internal or output action. We say that TIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are *compatible* if, for $i \neq j$, $X_i \cap X_j = H_i \cap A_j = O_i \cap O_j = \emptyset$.

Lemma 7.1 *If $\mathcal{A}_1 = (\mathcal{B}_1, I_1, O_1)$ and $\mathcal{A}_2 = (\mathcal{B}_2, I_2, O_2)$ are compatible TIOAs, then $\mathcal{B}_1$ and $\mathcal{B}_2$ are compatible TAs.*

If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible TIOAs, then their *composition* $\mathcal{A}_1 \| \mathcal{A}_2$ is defined to be the tuple $\mathcal{A} = (\mathcal{B}, I, O)$ where

- $\mathcal{B} = \mathcal{B}_1 \| \mathcal{B}_2$,
- $I = (I_1 \cup I_2) - (O_1 \cup O_2)$, and
- $O = O_1 \cup O_2$.

Thus, an external action of the composition is classified as an output if it is an output of one of the component automata or otherwise it is classified as an input. The composition of two TIOAs is guaranteed to be a TIOA:

Theorem 7.2 *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are TIOAs, then $\mathcal{A}_1 \| \mathcal{A}_2$ is a TIOA.*

Proof: The proof is straightforward except for showing that Axiom **E2** is satisfied by the composition. Let $\mathbf{x}$ be a state of $\mathcal{A}_1 \| \mathcal{A}_2$. We need to show the existence of a trajectory from $\mathbf{x}$ that satisfies **E2**.

By definition of $\mathcal{A}_1 \| \mathcal{A}_2$, $\mathbf{x} \lceil X_1$ is a state of $\mathcal{A}_1$ and $\mathbf{x} \lceil X_2$ is a state of $\mathcal{A}_2$. We know that both $\mathcal{A}_1$ and $\mathcal{A}_2$ satisfy **E2**. Let τ_1 be a trajectory of $\mathcal{A}_1$ with $\tau_1.fstate = \mathbf{x} \lceil X_1$ that satisfies **E2**, let τ_2 be a trajectory of $\mathcal{A}_2$ with $\tau_2.fstate = \mathbf{x} \lceil X_2$ that satisfies **E2**, and consider the following cases:

1. $\tau_1.ltime = \infty$ and $\tau_2.ltime = \infty$. Then, define τ such that $\tau \downarrow X_1 = \tau_1$ and $\tau \downarrow X_2 = \tau_2$.
2. $\tau_1.ltime = \infty$ and τ_2 is closed where some $l \in L_2$ is enabled in $\tau_2.lstate$. Then, define τ such that $\tau \downarrow X_1 = \tau_1 \lceil dom(\tau_2)$ and $\tau \downarrow X_2 = \tau_2$.
3. τ_1 is closed where some $l \in L_1$ is enabled in $\tau_1.lstate$ and $\tau_2.ltime = \infty$. Then, define τ such that $\tau \downarrow X_1 = \tau_1$ and $\tau \downarrow X_2 = \tau_2 \lceil dom(\tau_1)$.
4. τ_1 is closed where some $l \in L_1$ is enabled in $\tau_1.lstate$ and τ_2 is closed where some $l \in L_2$ is enabled in $\tau_2.lstate$. If $dom(\tau_1) \subseteq dom(\tau_2)$, then define τ such that $\tau \downarrow X_1 = \tau_1$ and $\tau \downarrow X_2 = \tau_2 \lceil dom(\tau_1)$. Otherwise, define τ such that $\tau \downarrow X_1 = \tau_1 \lceil dom(\tau_2)$ and $\tau \downarrow X_2 = \tau_2$.

In all the cases, by definition of trajectories for a TIOA, τ is a trajectory of $\mathcal{A}_1 \| \mathcal{A}_2$ from $\mathbf{x}$, which satisfies **E2** by construction. □

Note that this theorem is stronger than the corresponding theorem [6, Th. 6.12] for general HIOAs. Two HIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ are required to be strongly compatible for their composition to be a HIOA. This extra condition is needed to rule out dependencies among external variables that may prevent the component automata from evolving together. The absence of external variables in TIOA eliminates this kind of problematic behavior. Thus, for the timed case, we do not require the notion of strong compatibility that was needed for the hybrid case.

Composition of TIOAs satisfies the following projection and pasting result, which follows from Theorem 5.4.

Theorem 7.3 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be comparable TIOAs, and let $\mathcal{A} = \mathcal{A}_1 \| \mathcal{A}_2$. Then $traces_{\mathcal{A}}$ is exactly the set of $(E, \emptyset)$-sequences whose restrictions to $\mathcal{A}_1$ and $\mathcal{A}_2$ are traces of $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively. That is, $traces_{\mathcal{A}} = \{\beta \mid \beta$ is an $(E, \emptyset)$-sequence and $\beta \lceil (E_i, \emptyset) \in traces_{\mathcal{A}_i}, i = \{1, 2\}\}$.*

7.1.2 Substitutivity Results

The following theorem is analogous to Theorem 5.8 for TAs without input/output distinction. It shows that the introduction of this distinction does not cause any changes to the substitutivity results we obtained for general TAs.

Theorem 7.4 *Suppose $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable TIOAs with $\mathcal{A}_1 \leq \mathcal{A}_2$. Suppose that $\mathcal{B}$ is a TIOA that is compatible with each of $\mathcal{A}_1$ and $\mathcal{A}_2$. Then $\mathcal{A}_1 \| \mathcal{B} \leq \mathcal{A}_2 \| \mathcal{B}$.*

The following corollaries are analogous to Corollaries 5.9 and 5.10.

Corollary 7.5 *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{B}_1$, and $\mathcal{B}_2$ are TIOAs, $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, $\mathcal{B}_1$ and $\mathcal{B}_2$ are comparable, and each of $\mathcal{A}_1$ and $\mathcal{A}_2$ is compatible with each of $\mathcal{B}_1$ and $\mathcal{B}_2$. If $\mathcal{A}_1 \leq \mathcal{A}_2$ and $\mathcal{B}_1 \leq \mathcal{B}_2$, then $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$.*

Corollary 7.6 *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{B}_1$, and $\mathcal{B}_2$ are TIOAs, $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, $\mathcal{B}_1$ and $\mathcal{B}_2$ are comparable, and each of $\mathcal{A}_1$ and $\mathcal{A}_2$ is compatible with each of $\mathcal{B}_1$ and $\mathcal{B}_2$. If $\mathcal{A}_1 \| \mathcal{B}_2 \leq \mathcal{A}_2 \| \mathcal{B}_2$ and $\mathcal{B}_1 \leq \mathcal{B}_2$, then $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$.*

The basic substitutivity theorem, Theorem 7.4, is desirable for any formalism for interacting processes. For design purposes, it enables one to refine individual components without violating the correctness of the system as a whole. For verification purposes, it enables one to prove that a composite system satisfies its specification by proving that each component satisfies its specification, thereby breaking down the verification task into more manageable pieces. However, it might not always be possible or easy to show that each component $\mathcal{A}_1$ (resp. $\mathcal{B}_1$) satisfies its specification $\mathcal{A}_2$ (resp. $\mathcal{B}_2$) without using any assumptions about the environment of the component. *Assume–guarantee* style results such as those presented in [49–56] are special kinds of substitutivity results that state what *guarantees* are expected from each component in an environment constrained by certain *assumptions*. Since the environment of each component consists of the other components in the system, assume–guarantee style results need to break the circular dependencies between the assumptions and guarantees for components. We present below two assume–guarantee style theorems Theorem 7.7 and Corollary 7.8, taken from [57], which can be used for proving that a system specified as a composite automaton $\mathcal{A}_1 \| \mathcal{B}_1$ implements a specification represented by a composite automaton $\mathcal{A}_2 \| \mathcal{B}_2$.

The main idea behind Theorem 7.7 is to assume that $\mathcal{A}_1$ implements $\mathcal{A}_2$ in a context represented by $\mathcal{B}_2$, and symmetrically that $\mathcal{B}_1$ implements $\mathcal{B}_2$ in a context represented by $\mathcal{A}_2$, where $\mathcal{A}_2$ and $\mathcal{B}_2$ are automata whose trace sets are closed under limits. The requirement about limit-closure implies that $\mathcal{A}_2$ and $\mathcal{B}_2$ specify trace safety properties. Moreover, we assume that the trace sets of $\mathcal{A}_2$ and $\mathcal{B}_2$ are closed under time-extension. That is, the automata allow arbitrary time-passage. This is the most general assumption one could make to ensure that $\mathcal{A}_2 \| \mathcal{B}_2$ does not impose stronger constraints on time-passage than $\mathcal{A}_1 \| \mathcal{B}_1$. Recall that the definition of time extension of a hybrid sequence can be found in Section 3.4.1.

Theorem 7.7 *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{B}_1$, and $\mathcal{B}_2$ are TIOAs such that $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable, $\mathcal{B}_1$ and $\mathcal{B}_2$ are comparable, and each of $\mathcal{A}_1$ and $\mathcal{A}_2$ is compatible with each of $\mathcal{B}_1$ and $\mathcal{B}_2$. Suppose further* *that*

1. *the sets* $traces_{\mathcal{A}_2}$ *and* $traces_{\mathcal{B}_2}$ *are closed under limits.*
2. *the sets* $traces_{\mathcal{A}_2}$ *and* $traces_{\mathcal{B}_2}$ *are closed under time-extension.*
3. $\mathcal{A}_1 \| \mathcal{B}_2 \leq \mathcal{A}_2 \| \mathcal{B}_2$ *and* $\mathcal{A}_2 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$.

Then $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$.

Proof: We first prove by induction on the length of traces of $\mathcal{A}_1 \| \mathcal{B}_1$ that every closed trace of $\mathcal{A}_1 \| \mathcal{B}_1$ is a trace of $\mathcal{A}_2 \| \mathcal{B}_2$.

For the base case, let β be a trace of $\mathcal{A}_1 \| \mathcal{B}_1$ such that $\beta \in trajs(\emptyset)$ (a single trajectory over the empty set of variables). By Axiom **T0** in the definition of a TA, we know that $\mathcal{A}_2$ and $\mathcal{B}_2$ have traces α_1 and α_2 such that $\alpha_1.ltime$=$\alpha_2.ltime$=0. By Assumption 2 we have $\alpha_1 \frown \beta \in traces_{\mathcal{A}_2}$ and $\alpha_2 \frown \beta \in traces_{\mathcal{B}_2}$. Since, $\alpha_1 \frown \beta = \beta$ and $\alpha_2 \frown \beta = \beta$, it follows that $\beta \in traces_{\mathcal{A}_2}$ and $\beta \in traces_{\mathcal{B}_2}$. By pasting using Theorem 7.3, $\beta \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$, as needed.

For the inductive step we consider the following cases:

1. $\beta = \beta' \, a \, \tau$, where a is an output action of $\mathcal{A}_1$ and τ is a point trajectory. Then $\beta \lceil (E_{\mathcal{A}_1}, \emptyset) \in traces_{\mathcal{A}_1}$ by projection using Theorem 7.3. By inductive hypothesis, $\beta' \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$. So $\beta' \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$, by projection using Theorem 7.3. Let α be an execution of $\mathcal{B}_2$ such that $trace(\alpha) = \beta' \lceil (E_{\mathcal{B}_2}, \emptyset)$. Since $\mathcal{A}_1$ and $\mathcal{B}_1$ are compatible TIOAs, $\mathcal{B}_1$ and $\mathcal{B}_2$ are comparable, and a is an output action of $\mathcal{A}_1$, we know that either a is an input action of $\mathcal{B}_2$ or the action set of $\mathcal{B}_2$ does not contain a. In the former case, by the input-enabling axiom (**E1**) we know that there exists $\mathbf{x}'$ such that $(\alpha.lstate, a, \mathbf{x}')$ is a discrete transition of $\mathcal{B}_2$. It follows that $\beta \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. In the latter case, since $\beta \lceil (E_{\mathcal{B}_2}, \emptyset) = \beta' \lceil (E_{\mathcal{B}_2}, \emptyset)$ and $\beta' \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$, we get $\beta \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. By pasting using Theorem 7.3, $\beta \in traces_{\mathcal{A}_1 \| \mathcal{B}_2}$. Then by Assumption 3, $\beta \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$.
2. $\beta = \beta' \, b \, \tau$, where b is an output action of $\mathcal{B}_1$ and τ is a point trajectory. This case is symmetric with the previous one.
3. $\beta = \beta' \, c \, \tau$, where c is an input action of both $\mathcal{A}_1$ and $\mathcal{B}_1$ and τ is a point trajectory. By inductive hypothesis, $\beta' \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$. By projection using Theorem 7.3 we get $\beta' \lceil (E_{\mathcal{A}_2}, \emptyset) \in traces_{\mathcal{A}_2}$ and $\beta' \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. Let α be an execution of $\mathcal{A}_2$ such that $trace(\alpha)$=$\beta' \lceil (E_{\mathcal{A}_2}, \emptyset)$. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable and a is an input action of $\mathcal{A}_1$ we know that a is an input action of $\mathcal{A}_2$. By the input-enabling axiom (**E1**) we know that there exists $\mathbf{x}'$ such that $(\alpha'.lstate, a, \mathbf{x}')$ is a discrete transition of $\mathcal{A}_2$. It follows that $\beta \lceil (E_{\mathcal{A}_2}, \emptyset) \in traces_{\mathcal{A}_2}$. Similarly, let α' be an execution of $\mathcal{B}_2$ such that $trace(\alpha') = \beta' \lceil (E_{\mathcal{B}_2}, \emptyset)$. Since $\mathcal{B}_1$ and $\mathcal{B}_2$ are comparable and a is an input action of

$\mathcal{B}_1$ we know that a is an input action of $\mathcal{B}_2$. By the input-enabling axiom (**E1**) we know that there exists $\mathbf{y}'$ such that $(\alpha'.lstate, a, \mathbf{y}')$ is a discrete transition of $\mathcal{B}_2$. It follows that $\beta \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. By pasting using Theorem 7.3, we get $\beta \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$.

4. $\beta = \beta' \, d \, \tau$, where d is an input action of $\mathcal{A}_1$ but not an action of $\mathcal{B}_1$ and τ is a point trajectory. By inductive hypothesis, $\beta' \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$. By projection using Theorem 7.3, we have $\beta' \lceil (E_{\mathcal{A}_2}, \emptyset) \in traces_{\mathcal{A}_2}$ and $\beta' \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. Let α be an execution of $\mathcal{A}_2$ such that $trace(\alpha) = \beta' \lceil (E_{\mathcal{A}_2}, \emptyset)$. Since $\mathcal{A}_1$ and $\mathcal{A}_2$ are comparable TIOAs and a is an input action of $\mathcal{A}_1$, a must be an input action of $\mathcal{A}_2$. By the input-enabling axiom (**E1**) we know that there exists $\mathbf{x}'$ such that $(\alpha.lstate, a, \mathbf{x}')$ is a discrete transition of $\mathcal{A}_2$. It follows that $\beta \lceil (E_{\mathcal{A}_2}, \emptyset) \in traces_{\mathcal{A}_2}$. Since $\mathcal{B}_1$ and $\mathcal{B}_2$ are comparable and a is not an action of $\mathcal{B}_1$, a cannot be an external action of $\mathcal{B}_2$. Therefore, $\beta \lceil (E_{\mathcal{B}_2}, \emptyset) = \beta' \lceil (E_{\mathcal{B}_2}, \emptyset)$. Since $\beta' \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$ we get $\beta \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. By pasting using Theorem 7.3, we get $\beta \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$.
5. $\beta = \beta' e \tau$, where e is an input action of $\mathcal{B}_1$ but not an action of $\mathcal{A}_1$ and τ is a point trajectory. This case is symmetric with the previous one.
6. $\beta = \beta' \frown \beta''$, where β'' is a hybrid sequence consisting of a single trajectory τ. By inductive hypothesis, $\beta' \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$. By projection using Theorem 7.3, we get $\beta' \lceil (E_{\mathcal{A}_2}, \emptyset) \in traces_{\mathcal{A}_2}$ and $\beta' \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. By Assumption 2, we have $\beta' \lceil (E_{\mathcal{A}_2}, \emptyset) \frown \beta'' \lceil (E_{\mathcal{A}_2}, \emptyset) \in traces_{\mathcal{A}_2}$ and $\beta' \lceil (E_{\mathcal{B}_2}, \emptyset) \frown \beta'' \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. Then by pasting using Theorem 7.3, $\beta \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$, as needed.

We have thus shown that every closed trace of $\mathcal{A}_1 \| \mathcal{B}_1$ is a trace of $\mathcal{A}_2 \| \mathcal{B}_2$. Now consider any non closed trace β of $\mathcal{A}_1 \| \mathcal{B}_1$. This β can be written as the limit of a sequence $\beta_1 \beta_2 \cdots$ of closed traces of $\mathcal{A}_1 \| \mathcal{B}_1$. By the first part of the proof we know that each $\beta_i \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$, and by projection using Theorem 7.3 each $\beta_i \lceil (E_{\mathcal{A}_2}, \emptyset)$ is a closed trace of $\mathcal{A}_2$, and $\beta_i \lceil (E_{\mathcal{B}_2}, \emptyset)$ is a closed trace of $\mathcal{B}_2$. Since restriction is a continuous operation (Lemma 3.8), we know that $\beta \lceil (E_{\mathcal{A}_2}, \emptyset)$ is the limit of the $\beta_i \lceil (E_{\mathcal{A}_2}, \emptyset)$ and similarly $\beta \lceil (E_{\mathcal{B}_2}, \emptyset)$ is the limit of the $\beta_i \lceil (E_{\mathcal{B}_2}, \emptyset)$. Since the sets $traces_{\mathcal{A}_2}$ and $traces_{\mathcal{B}_2}$ are limit-closed by Assumption 1, we get $\beta \lceil (E_{\mathcal{A}_2}, \emptyset) \in traces_{\mathcal{A}_2}$ and $\beta \lceil (E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. Finally, by pasting using Theorem 7.3, we get $\beta \in traces_{\mathcal{A}_2 \| \mathcal{B}_2}$. □

Note that automata with FIN and timing independence (see Section 4.3 for definitions) constitute examples for context automata $\mathcal{A}_2$ and $\mathcal{B}_2$ that satisfy Assumptions 1 and 2. The property FIN implies Assumption 1 (Lemma 4.18) and timing independence implies Assumption 2.

Theorem 7.7 has a corollary, Corollary 7.8, which can be used in the decomposition of proofs even when $\mathcal{A}_2$ and $\mathcal{B}_2$ neither admit arbitrary time-passage nor have limit-closed trace sets. The main idea behind this corollary is to assume that $\mathcal{A}_1$ implements $\mathcal{A}_2$ in a context $\mathcal{B}_3$ that is a variant of $\mathcal{B}_2$, and symmetrically that $\mathcal{B}_1$ implements $\mathcal{B}_2$ in a context $\mathcal{A}_3$ that is a variant of $\mathcal{A}_2$. That is, the correctness of implementation relationship between $\mathcal{A}_1$ and $\mathcal{A}_2$ does not depend on all the environment constraints, but just on those expressed by $\mathcal{B}_3$ (symmetrically for $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{A}_3$). In order to use this corollary to prove $\mathcal{A}_1\|\mathcal{B}_1 \leq \mathcal{A}_2\|\mathcal{B}_2$ one needs to be able to find appropriate variants of $\mathcal{A}_2$ and $\mathcal{B}_2$ that meet the required closure properties. This corollary prompts one to pin down what is essential about the behavior of the environment in proving the intended implementation relationship and also allows one to avoid the unnecessary details of the environment in proofs.

Corollary 7.8 *Suppose $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_3$, $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$ are TIOAs such that $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$ are comparable, $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$ are comparable, and $\mathcal{A}_i$ is compatible with $\mathcal{B}_j$ for $i, j \in \{1, 2, 3\}$. Suppose further that*

1. *the sets $traces_{\mathcal{A}_3}$ and $traces_{\mathcal{B}_3}$ are closed under limits.*
2. *the sets $traces_{\mathcal{A}_3}$ and $traces_{\mathcal{B}_3}$ are closed under time-extension.*
3. *$\mathcal{A}_2\|\mathcal{B}_3 \leq \mathcal{A}_3\|\mathcal{B}_3$ and $\mathcal{A}_3\|\mathcal{B}_2 \leq \mathcal{A}_3\|\mathcal{B}_3$.*
4. *$\mathcal{A}_1\|\mathcal{B}_3 \leq \mathcal{A}_2\|\mathcal{B}_3$ and $\mathcal{A}_3\|\mathcal{B}_1 \leq \mathcal{A}_3\|\mathcal{B}_2$.*

Then $\mathcal{A}_1\|\mathcal{B}_1 \leq \mathcal{A}_2\|\mathcal{B}_2$.

Proof: Since $\mathcal{A}_1\|\mathcal{B}_3 \leq \mathcal{A}_2\|\mathcal{B}_3$ by Assumption 4 and $\mathcal{A}_2\|\mathcal{B}_3 \leq \mathcal{A}_3\|\mathcal{B}_3$ by Assumption 3, we get $\mathcal{A}_1\|\mathcal{B}_3 \leq \mathcal{A}_3\|\mathcal{B}_3$. Similarly, we have $\mathcal{A}_3\|\mathcal{B}_1 \leq \mathcal{A}_3\|\mathcal{B}_2 \leq \mathcal{A}_3\|\mathcal{B}_3$. Since $\mathcal{A}_1\|\mathcal{B}_3 \leq \mathcal{A}_3\|\mathcal{B}_3$ and $\mathcal{A}_3\|\mathcal{B}_1 \leq \mathcal{A}_3\|\mathcal{B}_3$, by using Assumptions 1 and 2 and Theorem 7.7 we have $\mathcal{A}_1\|\mathcal{B}_1 \leq \mathcal{A}_3\|\mathcal{B}_3$.

Let β be a trace of $\mathcal{A}_1\|\mathcal{B}_1$. By projection using Theorem 7.3, $\beta\lceil(E_{\mathcal{A}_1}, \emptyset) \in traces_{\mathcal{A}_1}$ and $\beta\lceil(E_{\mathcal{B}_1}, \emptyset) \in traces_{\mathcal{B}_1}$. Since $\mathcal{A}_1\|\mathcal{B}_1 \leq \mathcal{A}_3\|\mathcal{B}_3$, we know that $\beta \in traces_{\mathcal{A}_3\|\mathcal{B}_3}$. By projection using Theorem 7.3, $\beta\lceil(E_{\mathcal{A}_3}, \emptyset) \in traces_{\mathcal{A}_3}$ and $\beta\lceil(E_{\mathcal{B}_3}, \emptyset) \in traces_{\mathcal{B}_3}$. By pasting using Theorem 7.3, we have $\beta \in traces_{\mathcal{A}_1\|\mathcal{B}_3}$ and $\beta \in traces_{\mathcal{A}_3\|\mathcal{B}_1}$. By Assumption 4, we get $\beta \in traces_{\mathcal{A}_2\|\mathcal{B}_3}$ and $\beta \in traces_{\mathcal{A}_3\|\mathcal{B}_2}$. Then, by projection using Theorem 7.3, $\beta\lceil(E_{\mathcal{A}_2}, \emptyset) \in traces_{\mathcal{A}_2}$ and $\beta\lceil(E_{\mathcal{B}_2}, \emptyset) \in traces_{\mathcal{B}_2}$. Finally, by pasting using Theorem 7.3, we have $\beta \in traces_{\mathcal{A}_2\|\mathcal{B}_2}$, as needed. ☐

Example 7.9 (Using environment assumptions to prove safety). This example illustrates that, in cases where specifications $\mathcal{A}_2$ and $\mathcal{B}_2$ satisfy certain closure properties, it is possible to decompose the proof of $\mathcal{A}_1\|\mathcal{B}_1 \leq \mathcal{A}_2\|\mathcal{B}_2$ by using Theorem 7.7, even if it is not the case that $\mathcal{A}_1 \leq \mathcal{A}_2$ or $\mathcal{B}_1 \leq \mathcal{B}_2$.

The automata `AlternateA` and `AlternateB` in Fig. 7.1 are timing-independent automata in which no consecutive outputs occur without inputs happening in between. `AlternateA` and `AlternateB` perform a handshake, outputting an alternating sequence of

```
automaton AlternateA
  signature
    output a, input b
  states
    myturn: Bool := true
  transitions
    output a                              input b
      pre                                   eff
        myturn                                myturn := true
      eff
        myturn := false
```

```
automaton AlternateB
  signature
    input a, output b
  states
    myturn: Bool := false
  transitions
    input a                               output b
      eff                                   pre
        myturn := true                        myturn
                                            eff
                                              myturn := false
```

FIGURE 7.1: `AlternateA` and `AlternateB`.

`a` and `b` actions when they are composed. The automata `CatchUpA` and `CatchUpB` in Fig. 5.2 are timing-dependent automata that do not necessarily alternate inputs and outputs as `AlternateA` and `AlternateB`. `CatchUpA` can perform an arbitrary number of `b` actions and can perform an `a` provided that `counta` $\leq$ `countb`. It allows `counta` to increase to one more than `countb`. `CatchUpB` can perform an arbitrary number of `a` actions and can perform a `b` provided that `counta` $\geq$ `countb` $+ 1$. It allows `countb` to reach `counta`. Timing constraints require each output to occur exactly one time unit after the last action. `CatchUpA` and `CatchUpB` perform an alternating sequence of `a` actions and `b` actions when they are composed.

Suppose that we want to prove that `CatchUpA` $\|$ `CatchUpB` $\leq$ `AlternateA` $\|$ `AlternateB`. We cannot apply the basic substituvity theorem Theorem 7.7, in particular Corollary 7.5, since the assertions `CatchUpA` $\leq$ `AlternateA` and `CatchUpB` $\leq$ `AlternateB` are not true. Consider the trace $\overline{1}$ `b` $\overline{1}$ `a` $\overline{1}$ `a` $\overline{1}$ of `CatchUpA`. After having performed one `b` and one `a`, `CatchUpA` can perform another `a`. But, this is impossible for `AlternateA`, which needs an input to enable the second `a`. `AlternateA` and `CatchUpA` behave similarly only when put in a context that imposes alternation.

It is easy to check that AlternateA and AlternateB satisfy the closure properties required by Assumptions 1 and 2 of Theorem 7.7 and, hence can be substituted for $\mathcal{A}_2$ and $\mathcal{B}_2$ respectively. Similarly, we can easily check that Assumption 3 is satisfied if we substitute CatchUpA for $\mathcal{A}_1$ and CatchUpB for $\mathcal{B}_1$.

Example 7.10 (Extracting essential environment assumptions with auxiliary automata). This example illustrates that it may be possible to decompose verification, using Corollary 7.8, in cases where Theorem 7.7 is not applicable. If the aim is to show $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$ where $\mathcal{A}_2$ and $\mathcal{B}_2$ do not satisfy the assumptions of Theorem 7.7, then we find appropriate context automata $\mathcal{A}_3$ and $\mathcal{B}_3$ that abstract from those details of $\mathcal{A}_2$ and $\mathcal{B}_2$ that are not essential in proving $\mathcal{A}_1 \| \mathcal{B}_1 \leq \mathcal{A}_2 \| \mathcal{B}_2$.

Consider the automata UseOldInputA and UseOldInputB in Fig. 7.2. UseOldInputA keeps track of the next time it is supposed to perform an output, which may be never (infty). The number of outputs that UseOldInputA can perform is bounded by a natural number. In the case of repeated b inputs, it is the oldest input that determines when the next output will occur. The automaton UseOldInputB is the same as UseOldInputA (inputs and outputs reversed) except that the next variable of UseOldInputB is set to infty initially. Note that UseOldInputA and UseOldInputB are not timing-independent and their trace sets are not limit-closed. For each automaton, there are infinitely many start states, one for each natural number. We can build an infinite chain of traces, where each element in the chain corresponds to an execution starting from a distinct start state. The limit of such a chain, which contains infinitely many outputs, cannot be a trace of UseOldInputA or UseOldInputB since the number of outputs they can perform is bounded by a natural number. The automaton UseNewInputA in Fig. 7.3 behaves in a manner similarly to UseOldInputA except for the handling of inputs. In the case of repeated b inputs, it is the most recent input that determines when the next output will occur. The automaton UseNewInputB in Fig. 7.3 is the same as UseNewInputA (inputs and outputs reversed) except that the next variable of UseNewInputB is set to infty initially. Suppose that we want to prove that

$$\texttt{UseNewInputA} \| \texttt{UseNewInputB} \leq \texttt{UseOldInputA} \| \texttt{UseOldInputB}.$$

Theorem 7.7 is not applicable here because the high-level automata UseOldInputA and UseOldInputB do not satisfy the required closure properties. However, we can use Corollary 7.8 to decompose verification. It requires us to find auxiliary automata that are less restrictive than UseOldInputA and UseOldInputB but that are restrictive enough to express the constraints that should be satisfied by the environment, for UseNewInputA to implement UseOldInputA and for UseNewInputB to implement UseOldInputB.

```
signature
   output a, input b
states
   maxout: Nat, now: Real := 0, next: AugmentedReal := 0
transitions
   output a                                  input b
     pre                                       eff
       (maxout > 0) ∧ (now = next)               if next = infty
     eff                                          then next := now + 1
       maxout := maxout - 1;
       next := infty
trajectories
   stop when
     now = next
   evolve
     d(now) = 1
```

```
 signature
   input a, output b
states
   maxout: Nat, now: Real := 0, next: AugmentedReal := infty
transitions
   input a                             output b
     eff                                 pre
       if next = infty                     (maxout > 0) ∧ (now = next)
        then next := now + 1             eff
                                           maxout := maxout - 1;
                                           next := infty
trajectories
   stop when
     now = next
   evolve
     d(now) = 1
```

FIGURE 7.2: `UseOldInputA` and `UseOldInputB`.

The automata `AlternateA` and `AlternateB` in Fig. 7.1 can be used as auxiliary automata in this example. They satisfy the closure properties required by Corollary 7.8 and impose alternation, which is the only additional condition to ensure the needed trace inclusion.

We can define a forward simulation relation from `UseNewInputA` ∥ `UseNewInputB` to `UseOldInputA` ∥ `UseOldInputB`, which is based on the equality of the `next` = `infty` predicate of the implementation and the specification automata. The fact that this simulation relation uses only the predicate `next` = `infty` reinforces the idea that the auxiliary contexts, which

```
signature
   output a, input b
states
   maxout: Nat, now: Real := 0, next: AugmentedReal := 0
transitions
   output a                              input b
     pre                                   eff
       (maxout > 0) ∧ (now = next)          next := now + 1
     eff
       maxout := maxout - 1;
       next := infty
trajectories
   stop when
     now = next
   evolve
     d(now) = 1
```

```
signature
   input a, output b
states
   maxout: Nat, now: Real := 0, next: AugmentedReal := infty
transitions
   input a                               output b
     eff                                   pre
       next := now + 1                       (maxout > 0) ∧ (now = next)
                                           eff
                                             maxout := maxout - 1;
                                             next := infty
trajectories
   stop when
     now = next
   evolve
     d(now) = 1
```

FIGURE 7.3: UseNewInputA and UseNewInputB.

only keep track of their turn, capture exactly what is needed for the proof of UseNewInputA ∥ UseNewInputB ≤ UseOldInputA ∥ UseOldInputB. We can observe that a direct proof of this assertion would require one to deal with state variables such as maxout and next of both UseOldInputA and UseOldInputB, which do not play any essential role in the proof. On the other hand, by decomposing the proof along the lines of Corollary 7.8 some of the unnecessary details can be avoided. Even though, this is a toy example with an easy proof, it should not be hard to observe how this simplification would scale to large proofs.

7.1.3 Composition of Special Kinds of TIOAs

The following example illustrates that the set of I/O feasible TIOAs is not closed under composition.

Example 7.11 (Two I/O feasible TIOAs whose composition is not I/O feasible). Consider two I/O feasible TIOAs $\mathcal{A}$ and $\mathcal{B}$, where $O_{\mathcal{A}} = I_{\mathcal{B}} = \{a\}$ and $O_{\mathcal{B}} = I_{\mathcal{A}} = \{b\}$. Suppose that $\mathcal{A}$ performs its output a at time 0 and then waits, allowing time to pass, until it receives input b. If and when it receives b, it responds with output a without allowing any time to pass (and ignoring any inputs that occur before it has a chance to perform its output). On the other hand, $\mathcal{B}$ starts out waiting, allowing time to pass, until it receives input a. If and when it receives a, it responds with output b without allowing time to pass.

It is not difficult to see that $\mathcal{A}$ and $\mathcal{B}$ are individually I/O feasible. We claim that the composition $\mathcal{A}\|\mathcal{B}$ is not I/O feasible. To see this, consider the start state of $\mathcal{A}\|\mathcal{B}$ and the unique input sequence β with $\beta.ltime = \infty$; β simply allows time to pass to infinity. The composition $\mathcal{A}\|\mathcal{B}$ has no way of accommodating this input, since it will never allow time to pass beyond 0.

On the other hand, the following theorems say that the classes of progressive and receptive TIOAs are closed under composition.

Theorem 7.12 *If $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible progressive TIOAs, then their composition is also progressive.*

Proof: The proof is similar to the proof of Theorem 7.4 in [6]. The main idea behind the proof is that a Zeno execution of $\mathcal{A}_1\|\mathcal{A}_2$ with infinitely many locally controlled actions contains infinitely many locally controlled actions of either $\mathcal{A}_1$ or $\mathcal{A}_2$. Suppose without loss of generality that the automaton that contributes infinitely many locally controlled actions is $\mathcal{A}_1$. Then the projection onto $\mathcal{A}_1$ violates progressiveness for $\mathcal{A}_1$. □

Theorem 7.13 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two compatible TIOAs with strategies $\mathcal{A}'_1$ and $\mathcal{A}'_2$, respectively. Then $\mathcal{A}'_1\|\mathcal{A}'_2$ is a strategy for $\mathcal{A}_1\|\mathcal{A}_2$.*

Proof: The proof is a straightforward one, similar to the proof of Theorem 7.7 in [6]. □

Now we can state the main result of this section, which follows easily from the previous two theorems. It shows that the class of receptive TIOAs is closed under composition.

Theorem 7.14 *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two compatible receptive TIOAs with progressive strategies $\mathcal{A}'_1$ and $\mathcal{A}'_2$, respectively. Then $\mathcal{A}_1\|\mathcal{A}_2$ is a receptive TIOA with progressive strategy $\mathcal{A}'_1\|\mathcal{A}'_2$.*

Example 7.15 (Composition of receptive TIOAs). Theorem 7.14 implies that the composition of clock synchronization automata with channel automata described in Example 5.7 (viewed as TIOAs as explained in Example 6.1) is receptive. By Theorem 6.6 we also have that it is I/O feasible.

Actually, the fact that the set of I/O feasible TIOAs is not closed under composition motivated the definition of the more restrictive class of receptive TIOAs. That is, receptiveness is a reasonable sufficient condition that implies I/O feasibility, and that also is preserved by composition.

The special case of the HIOA model, represented by the TIOA model, has simpler and stronger composition theorems than the general HIOA model. In particular, the main compositionality result for receptive HIOAs (Theorem 7.12 in [6]) has a more intricate proof than ours. It makes an assumption about the existence of strongly compatible strategies (discussed briefly at the end of Section 7.1.1) and needs an additional lemma that shows that if two HIOAs $\mathcal{A}_1$ and $\mathcal{A}_2$ have strongly compatible strategies $\mathcal{A}_1'$ and $\mathcal{A}_2'$, then $\mathcal{A}_1$ and $\mathcal{A}_2$ are also strongly compatible.

7.2 HIDING

We extend the definition of action hiding to any TIOA $\mathcal{A}$. For TIOAs, we consider hiding outputs only (but not inputs), by converting them to internal actions. Namely, if $O \subseteq O_{\mathcal{A}}$, then $\mathsf{ActHide}(O, \mathcal{A})$ is the TIOA $\mathcal{B}$ that is equal to $\mathcal{A}$ except that $O_{\mathcal{B}} = O_{\mathcal{A}} - O$ and $H_{\mathcal{B}} = H_{\mathcal{A}} \cup O$.

Lemma 7.16 *If $\mathcal{A}$ is a TIOA and $O \subseteq O_{\mathcal{A}}$, then $\mathsf{ActHide}(O, \mathcal{A})$ is a TIOA.*

Lemma 7.17 *If $\mathcal{A}$ is a TIOA and $O \subseteq O_{\mathcal{A}}$, then $traces_{\mathsf{ActHide}(O,\mathcal{A})} = \{\beta \lceil (O_{\mathcal{A}} - O, V_{\mathcal{A}}) \mid \beta \in traces_{\mathcal{A}}\}$.*

Theorem 7.18 *Suppose $\mathcal{A}$ and $\mathcal{B}$ are TIOAs with $\mathcal{A} \leq \mathcal{B}$, and suppose $O \subseteq O_{\mathcal{A}}$. Then $\mathsf{ActHide}(O, \mathcal{A}) \leq \mathsf{ActHide}(O, \mathcal{B})$.*

CHAPTER 8

Conclusions and Future Work

In this monograph, we have presented a new framework for describing and analyzing the behavior of timed systems. This framework is a mathematical framework that uses TIOA for the representation of systems. The TIOA framework is a special case of the HIOA modeling framework [6]. We used what we have learned in developing the HIOA framework to revise the earlier work on TIOA models. Our main motivation was to have a TIOA model that is compatible with the new HIOA model. We sought to benefit from the new style used in describing hybrid behavior in simplifying the prior definitions and results on TIOA.

Designers of real-time systems or timing-based algorithms can use the TIOA framework to describe complex systems and to decompose them into manageable pieces. In particular, they can use the TIOA framework to describe their systems at multiple levels of abstraction, to establish implementation relationships between these levels, and to decompose their systems into more primitive, interacting components. Although the framework as presented in this monograph provides only conceptual tools for modeling and manual proof methods, it also is a natural basis for building computerized modeling and analysis.

We are currently working on the development of a toolset based on this mathematical framework that will consist of (a) a formal modeling language called TIOA, (b) a front-end processor for TIOA, incorporating syntax and static semantic checking, and providing interfaces to computer-aided design tools, (c) a simulation tool allowing simulation of specifications and paired simulations of a specification and an abstract implementation, and (d) a theorem-proving link through an interface to the theorem-prover PVS [58]. We refer to [5, 36–38] for more information on the TIOA toolset. The described project builds upon our prior work on the IOA language [59].

On the theoretical side, we have done preliminary research toward extending the TIOA framework with support for reasoning about safety and liveness properties of timed systems. We have defined notions of fairness and proved results that state under which conditions the "fair" traces of a TIOA can be shown to be included in the fair traces of another. We have started investigating the consequences of composition on automata with liveness properties and the use of receptiveness and strategies in this context [60]. In [61], we study *urgency predicates* as an alternative to the **stop when** clauses that are used in this monograph for the specification of progress properties. The results of these lines of preliminary work are not included in this version of the monograph because the adequacy of our definitions and methods are yet to be assessed on a larger class of nontrivial examples.

We will also continue our work on establishing formal relationships with other models that are comparable to ours, showing that the TIOA framework is general enough to express previous results from other frameworks, such as [7–12].

References

[1] S. Garland and N. A. Lynch, Using I/O automata for developing distributed systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, New York, Ch. 13, pp. 285–312, 2000.

[2] J. Sifakis, Modeling real-time systems – challenges and work directions. In *Proc. Embedded Software, 1st Int. Workshop (EMSOFT '01),* Tahoe City, CA. Lecture Notes in Computer Science, Vol. 2211. pp. 373–389, 2001.

[3] J. Sifakis, Modeling real-time systems. In *Proc. 25th IEEE Real-Time Systems Symposium (RTSS '04)*. IEEE Comput. Soc. Press, Los Alamitos, CA, pp. 5–6, 2004. Invited Talk.

[4] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, San Fransisco, CA, 1996.

[5] D. Kaynar, N. A. Lynch, and S. Mitra, Specifying and proving timing properties with TIOA tools. In *Proc. 5th IEEE Int. Real-Time Systems Symposium, Work in Progress Session (RTSS WIP)*, Lisbon, Portugal, December 2004, pp. 96–99.

[6] N. A. Lynch, R. Segala, and F. W. Vaandrager, Hybrid I/O automata. *Information and Computation*, **185**(1):105–157, 2003. doi:10.1016/S0890-5401(03)00067-1

[7] M. Merritt, F. Modugno, and M. Tuttle, Time constrained automata. In J. C. M. Baeten and J. F. Groote, editors, *Proc. CONCUR 91,* Amsterdam. Lecture Notes in Computer Science, Vol. 527. Springer-Verlag, Berlin, pp. 408–423, 1991.

[8] R. Segala, R. Gawlick, J. F. Søgaard-Andersen, and N. A. Lynch, Liveness in timed and untimed systems. *Information and Computation*, **141**(2):119–171, 1998. doi:10.1006/inco.1997.2671

[9] N. A. Lynch and F. W. Vaandrager, Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, **128**(1):1–25, 1996. doi:10.1006/inco.1996.0060

[10] N. A. Lynch and F. W. Vaandrager, Action transducers and timed automata. *Formal Aspects of Computing*, **8**(5):499–538, 1996. doi:10.1007/BF01211907

[11] R. Alur and D. L. Dill, A theory of timed automata. *Theoretical Computer Science*, **126**:183–235, 1994. doi:10.1016/0304-3975(94)90010-8

[12] O. Maler, Z. Manna, and A. Pnueli, From timed to hybrid systems. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991. Lecture Notes in Computer Science, Vol. 600. Springer-Verlag, Berlin, pp. 447–484, 1992.

[13] P. Petterson, *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*, PhD thesis. Department of Computer Systems, Uppsala University, 1999. Technical Report DoCs 99/101.

[14] R. DePrisco, B. Lampson, and N. A. Lynch, Revisiting the Paxos algorithm. In M. Mavronicolas and P. Tsigas, editors, *Distributed Algorithms, Proc. 11th Int. Workshop, WDAG'97*, Saarbrücken, Germany, September 1997. Lecture Notes in Computer Science, Vol. 1320. Springer-Verlag, Berlin, pp. 111–125, 1997.

[15] R. Alur, Timed automata. In *Proc. 11th Int. Conf. Computer-Aided Verification (CAV)*. Lecture Notes in Computer Science, Vol. 1633. Springer-Verlag, Berlin, pp. 8–22, 1999. An earlier and longer version appears in NATO-ASI Summer School on Verification of Digital and Hybrid Systems, 1998.

[16] R. Alur, S. La Torre, and P. Madhusudan, Perturbed timed automata. In *Proc. 8th Int. Workshop Hybrid Systems: Computation and Control (HSCC),* Zurich, Zwitserland, Lecture Notes in Computer Science, Vol. 3414. Springer-Verlag, Berlin, pp. 70–85, 2005.

[17] R. Alur and P. Madhusudan, Decision problems for timed automata: A survey. In *Int. School Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT),* Bertinoro, Italy. Lecture Notes in Computer Science, Vol. 3185. Springer-Verlag, Berlin, pp. 1–24, 2004.

[18] K. G. Larsen, P. Pettersson, and W. Yi, Uppaal in a nutshell. *Journal of Software Tools for Technology Transfer*, **1/2**:134–152, 1997. doi:10.1007/s100090050010

[19] R. Milner, *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer-Verlag, Berlin, 1980.

[20] C. Robson, *TIOA and UPPAAL*, Master's thesis. MIT Department of Electrical Engineering and Computer Science, 2004.

[21] S. Yovine, Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, **1**(1/2):123–133, 1997. doi:10.1007/s100090050009

[22] C. Daws, A. Olivero, S. Tripakis, and S. Yovine, The tool Kronos. In *Proc. of Hybrid Systems III, Verification and Control*. Lecture Notes in Computer Science, Vol. 1066. Springer-Verlag, Berlin, pp. 208–219, 1996.

[23] R. Alur, *Techniques for Automatic Verification of Real-Time Systems*, PhD thesis. Stanford University, 1991.

[24] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, Symbolic model-checking for real-time systems. *Information and Computation*, **111**(2):193–244, 1994. doi:10.1006/inco.1994.1045

[25] M. Bozga, S. Graf, Il. Ober, Iul. Ober, and J. Sifakis, The IF toolset. In *Proc. Formal Methods for the Design of Real-Time Systems*. Lecture Notes in Computer Science, Vol. 3185. Springer-Verlag, Berlin, pp. 237–267, 2004.

[26] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P. H. Ho, X. Nicolin, A. Olivero, J. Sifakis, and S. Yovine, The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, **138**:3–34, 1995. doi:10.1016/0304-3975(94)00202-T

[27] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi, HyTech: A model checker for hybrid systems. In O. Grumberg, editor, *Proc. 9th Int. Conf. Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 1254. Springer-Verlag, Berlin, pp. 460–463, 1997.

[28] N. A. Lynch and A. Shvartsman, RAMBO: A reconfigurable atomic memory service for dynamic networks. In D. Malkhi, editor, *Distributed Computing, Proc. 16th Int. Symp. DIStributed Computing (DISC),* Toulouse, France, October 2002. Lecture Notes in Computer Science, Vol. 2508. Springer-Verlag, Berlin, pp. 173–190, 2002. Technical Report MIT-LCS-TR-856.

[29] C. A. Gunter, *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, MA, 1992.

[30] M. Hennessy, *Algebraic Theory of Processes*. MIT Press, Cambridge, MA, 1988.

[31] D. E. Knuth, *Fundamental Algorithms*, 2nd edition. The Art of Computer Programming, Vol. 1. Addision-Wesley, Reading, MA, 1973.

[32] E. D. Sontag, *Mathematical Control Theory – Deterministic Finite Dimensional Systems*, Texts in Applied Mathematics, Vol. 6. Springer-Verlag, Berlin, 1990.

[33] A. Pnueli, Development of hybrid systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Proc. 3rd Int. School Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94),* Lübeck, Germany, September 1994. Lecture Notes in Computer Science, Vol. 863. Springer-Verlag, Berlin, pp. 77–85, 1994.

[34] J. W. Polderman and J. C. Willems, *Introduction to Mathematical Systems Theory: A Behavioural Approach*. Texts in Applied Mathematics, Vol. 26. Springer-Verlag, Berlin, 1998.

[35] S. Mitra, Y. Wang, N. A. Lynch, and E. Feron, Safety verification of model helicopter controller using hybrid input/output automata. In O. Maler and A. Pnueli, editors, *Proc. Hybrid Systems: Computation and Control*, Prague, the Czech Republic April 3–5, 2003. Lecture Notes in Computer Science, Vol. 2623. Springer-Verlag, Berlin, pp. 343–358, 2003.

[36] D. Kaynar, N. A. Lynch, S. Mitra, and S. Garland, The TIOA language, May 2005. Available: `http://theory.csail.mit.edu/tds/reflist.html`.

[37] S. Garland, TIOA user guide and reference manual, September 2005. Available: `http://theory.csail.mit.edu/tds/reflist.html`.

[38] S. Garland, D. Kaynar, N. A. Lynch, J. Tauber, and M. Vaziri, TIOA tutorial, May 2005. Available: `http://theory.csail.mit.edu/tds/reflist.html`.

[39] R. Gawlick, R. Segala, J. F. Søgaard-Andersen, and N. A. Lynch, Liveness in timed and untimed systems. In S. Abiteboul and E. Shamir, editors, *Proc. 21st ICALP,* Jerusalem. Lecture Notes in Computer Science, Vol. 820. Springer-Verlag, Berlin, 1994. A full version appears as MIT Technical Report number MIT/LCS/TR-587.

[40] N. A. Lynch, R. Segala, F. W. Vaandrager, and H. B. Weinberg, Hybrid I/O automata. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*. Lecture Notes in Computer Science, Vol. 1066. Springer-Verlag, Berlin, pp. 496–510, 1996.

[41] N. A. Lynch, R. Segala, F. W. Vaandrager, and H. B. Weinberg, Hybrid I/O automata, Report CSI-R9907. Computing Science Institute, University of Nijmegen, April 1999.

[42] J. M. T Romijn, A timed verification of the IEEE 1394 leader election protocol. *Formal Methods in System Design*, **19**(2):165–194, 2001. Special issue on FMICS'99. doi:10.1023/A:1011284000753

[43] D. P. L. Simons and M. I. A. Stoelinga, Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k. *International Journal on Software Tools for Technology Transfer*, **3**(4):469–485, 2001.

[44] N. A. Lynch and F. W. Vaandrager, Forward and backward simulations, I: Untimed systems. *Information and Computation*, **121**(2):214–233, 1995. doi:/10.1006/inco.1995.1134

[45] N. A. Lynch and M. R. Tuttle, An introduction to input/output automata. *CWI Quarterly*, **2**(3):219–246, 1989.

[46] S. Bornot and J. Sifakis, An algebraic framework for urgency. *Information and Computation*, **163**:172–202, 2000. doi:10.1006/inco.2000.2999

[47] Howard Bowman, Modelling timeouts without timelocks. In J.-P. Katoen, editor, *ARTS'99, 5th Int. AMAST Workshop Real-time and Probabilistic Systems*. Lecture Notes in Computer Science, Vol. 1601. Springer, Berlin, pp. 334–353, May 1999.

[48] D. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, ACM Distinguished Dissertations. MIT Press, Cambridge, MA, 1988.

[49] M. Abadi and L. Lamport, Composing specifications. *ACM Transactions on Programming Languages and Systems*, **1**(15):73–132, 1993. doi:10.1145/151646.151649

[50] C. B. Jones, Specification and design of parallel programs. In R. E. A. Mason, editor, *Information Processing 83: Proc. of the IFIP 9th World Congress*, North-Holland, 1983, pp. 321–332.

[51] A. Pnueli, In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI. Springer-Verlag, Berlin, pp. 123–144, 1984.

[52] E. W. Stark, A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, Vol. 206. Springer-Verlag, Berlin, pp. 369–391, 1985.

[53] M. Abadi and L. Lamport, Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, **17**(3):507–534, 1995. doi:10.1145/203095.201069

[54] T. A. Henzinger, S. Qadeer, and S. K. Rajamani, Decomposing refinement proofs using assume-guarantee reasoning. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*. IEEE Comput. Soc. Press, Los Alamitos, CA, pp. 245–252, 2000.

[55] S. Tasiran, R. Alur, R. P. Kurshan, and R. K. Brayton, Verifying abstractions of timed systems. In *Proc. 7th Conf. Concurrency Theory (CONCUR)*. Lecture Notes in Computer Science, Vol. 1119. Springer-Verlag, Berlin, 1996.

[56] G. Frehse, *Compositional Verification of Hybrid Systems using Simulation Relations*, PhD thesis. Radboud University Nijmegen, October 2005.

[57] D. Kaynar and N. A. Lynch, Decomposing verification of timed I/O automata. In Y. Lakhnech and S. Yovine, editors, *Proc. Joint Int. Conf. Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004,* Grenoble, France, September 22–24, 2004. Lecture Notes in Computer Science, Vol. 3253. Springer-Verlag, Berlin, pp. 84–101, 2004.

[58] S. Owre, J. Rushby, N. Shankar, and F. von Henke, Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, **21**(2):107–125, 1995. doi:10.1109/32.345827

[59] S. Garland, N. A. Lynch, and M. Vaziri, *IOA: A Language for Specifying, Programming, and Validating Distributed Systems*. MIT Laboratory for Computer Science, Cambridge, MA, 2001. Available: http://theory.lcs.mit.edu/tds/ioa.html.

[60] D. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager, The theory of timed I/O automata, Technical Report MIT-LCS-TR-917a. MIT Laboratory for Computer Science, Cambridge, MA, 2004. Available: `http://theory.csail.mit.edu/tds/reflist.html`.

[61] B. Gebremichael and F.W. Vaandrager, Specifying urgency in timed I/O automata. In *Proc. 3rd IEEE Int. Conf. Software Engineering and Formal Methods (SEFM 2005),* Koblenz, Germany, September 5–9, 2005. IEEE Computer Soc. Press, Los Alamitos, CA, pp. 64–73, 2005.

Index