The Theory of Timed I/O Automata
Dilsun K. Kaynor, Nancy Lynch, Roberto Segala, and Frits Vaandrager
The Theory of Timed I/O Automata

Dilsun K. Kaynar and Nancy Lynch
MIT Computer Science and Artificial Intelligence Laboratory
{dilsun,lynch}@csail.mit.edu

Roberto Segala
Dipartimento di Informatica, Università di Verona
roberto.segala@univr.it

Frits Vaandrager
Institute for Computing and Information Sciences,
Radboud University Nijmegen
F.Vaandrager@cwi.nl

March 2, 2005

Abstract

This monograph presents the Timed Input/Output Automaton (TIOA) modeling framework, a basic mathematical framework to support description and analysis of timed systems. An important feature of this model is its support for decomposing timed system descriptions. In particular, the framework includes a notion of external behavior for a timed I/O automaton, 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.
Contents

1 Introduction 4
   1.1 Overview ................................... 4
   1.2 Evolution of the TIOA framework .......... 6
   1.3 Related work ................................ 7
   1.4 Paper Organization .......................... 9

2 Mathematical Preliminaries 9
   2.1 Functions and Relations .................... 9
   2.2 Sequences .................................. 10
   2.3 Partial Orders ................................ 11
   2.4 A Basic Graph Lemma ........................ 12

3 Describing Timed System Behavior 12
   3.1 Time ....................................... 12
   3.2 Static and Dynamic Types .................. 13
   3.3 Trajectories ................................ 14
      3.3.1 Basic Definitions ....................... 14
      3.3.2 Prefix Ordering ......................... 14
      3.3.3 Concatenation ........................... 15
   3.4 Hybrid Sequences ............................ 16
      3.4.1 Basic Definitions ....................... 16
      3.4.2 Prefix Ordering ......................... 17
      3.4.3 Concatenation ........................... 17
      3.4.4 Restriction .............................. 18

4 Timed Automata 19
   4.1 Definition of Timed Automata ............... 19
   4.2 Executions and Traces ...................... 30
   4.3 Special Kinds of Timed Automata ............ 34
   4.4 Implementation Relationships ............... 37
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 [11, 41, 42].

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 work, 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 infinitestate, 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, \( A \), to *implement* another TIOA, \( B \), namely, any trace that can be exhibited by \( A \) is also allowed by \( B \). In this case, \( A \) might be more deterministic than \( B \), in terms of either discrete transitions or trajectories. For instance, \( B \) might be allowed to perform an output action at an arbitrary time before noon, whereas \( A \) produces the same output sometime between 10 and 11AM. The notion of a *simulation relation* from \( A \) to \( B \) provides a sufficient condition for demonstrating that \( A \) implements \( B \). A simulation relation is defined to satisfy three conditions, one relating start states, one relating discrete transitions, and one relating trajectories of \( A \) and \( B \).

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 \( A_1 \) implements \( A_2 \) then the composition of \( A_1 \) and \( B \) implements the composition of \( A_2 \) and \( B \). 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 "pasting" 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 [24]. 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, $A$, in the TIOA framework is defined in terms of the existence of a strategy, which is defined as a subautomaton of $A$ that chooses some of the evolutions from each state of $A$.

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 [19].

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 [25] by Lynch, Segala and Vaandrager. 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 [32, 40, 30, 28], because we have observed that the new HIOA modeling framework of Lynch, Segala and Vaandrager offered a way of improving and simplifying previous work on timed I/O automaton models [40, 30, 28]. 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 timed I/O automata [30, 32, 40, 28] and other related models [5, 31, 35, 9] can be re-cast 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 timed I/O automaton 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 [5, 35]. 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 paper is compatible with our longer-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 [5, 3]. 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 [7]. The clocks in perturbed timed automata can change at a rate within the interval $[1 - \epsilon, 1 + \epsilon]$, where $\epsilon$ 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 [6]. 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 [35, 23] 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 [33] along with other notions such as committed and urgent locations. Uppaal also supports 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 [39].

It would be interesting to work on formal semantics for Uppaal based on some variation of our restricted hybrid I/O automaton 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 non-empty 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 [46] 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. Kronos, as Uppaal, can perform model-checking using a symbolic representation of the infinite state space by sets of linear constraints. 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 [8], 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 [4]. 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 [16], 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 timed I/O automaton modeling framework presented in this paper can be used to express models that use lower and upper time bounds on tasks or actions [32, 31].
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 [9]. 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 [9, 29]. We are interested in finding a systematic way of describing such behavior with our new timed I/O automaton model.

1.4 Paper Organization

The rest of this paper is organized as follows. Section 2 contains mathematical preliminaries. Section 3 defines notions that are useful for describing the behavior of timed systems, most importantly, trajectories and timed sequences. Section 4 defines \textit{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. Section 5 presents composition and hiding operations for TAs, along with operations for adding bounds that relate TIOAs to other timed automaton models. Section 6 defines \textit{timed I/O automata (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. Section 7 presents compositionality results for TIOAs in general, and for the special classes of progressive and receptive TIOAs. Examples are included throughout.

2 Mathematical Preliminaries

In this section, we give basic mathematical definitions and notation that will be used as a foundation for our definitions of timed automata and timed I/O automata. 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 \( \text{dom}(f) \) and \( \text{range}(f) \), respectively. If also \( S \) is a set, then we write \( f \upharpoonright S \) for the restriction of \( f \) to \( S \), that is, the function \( g \) with \( \text{dom}(g) = \text{dom}(f) \cap S \) such that \( g(c) = f(c) \) for each \( c \in \text{dom}(g) \).

We say that two functions \( f \) and \( g \) are \textit{compatible} if \( f \upharpoonright \text{dom}(g) = g \upharpoonright \text{dom}(f) \). If \( f \) and \( g \) are compatible functions then we write \( f \cup g \) for the unique function \( h \) with
dom(h) = dom(f) ∪ dom(g) satisfying the condition: for each c ∈ dom(h), if c ∈ dom(f) then h(c) = f(c) and if c ∈ 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) = \( \bigcup \{ \text{dom}(f) \mid f \in F \} \) satisfying the condition: for each f ∈ F and c ∈ 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) | S for each c ∈ 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 ∈ dom(g).

We say that two functions f and g whose ranges are sets of functions are pointwise compatible if for each c ∈ dom(f) ∩ 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 \( \cup \) 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. 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 over S is a function from a downward-closed subset of \( \mathbb{Z}^{\geq 0} \) to S. Thus, the domain of a sequence is either the set of all positive integers, or is of the form \( \{1, \ldots, k\} \) for some k. In the first case we say that the sequence is infinite, and in the second case 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 \( S^\omega \), respectively. Concatenation of a finite sequence \( \rho \) with a finite or infinite sequence \( \sigma \) is denoted by \( \rho \sqcup \sigma \). The empty sequence, that is, the sequence with the empty domain is denoted by \( \lambda \). The sequence containing one element \( c \in S \) is abbreviated as c. We say that a sequence \( \sigma \) is a prefix of a sequence \( \rho \), denoted by \( \sigma \preceq \rho \), if \( \sigma = \rho | \text{dom}(\sigma) \). Thus, \( \sigma \preceq \rho \) if either \( \sigma = \rho \) or \( \sigma \) is finite and \( \rho = \sigma \sqcup \sigma' \) for some sequence \( \sigma' \). If \( \sigma \) is a nonempty sequence then head(\sigma) denotes the first element of \( \sigma \) and tail(\sigma) denotes \( \sigma \) with its first element removed. Moreover, if \( \sigma \) is finite, then last(\sigma) denotes the last element of \( \sigma \) and init(\sigma) denotes \( \sigma \) with its last element removed. Let \( \sigma \) and \( \sigma' \) be sequences over S. Then \( \sigma' \) is a subsequence of \( \sigma \) provided that there exists a monotone increasing function \( f : \text{dom}(\sigma') \to \text{dom}(\sigma) \) such that \( \sigma'(i) = \sigma(f(i)) \) for all \( i \in \text{dom}(\sigma') \). If \( 1 \leq j_1 \leq j_2 \leq |\sigma| \), then we define \( \sigma(j_1 \ldots j_2) \) to be the subsequence of \( \sigma \) obtained by extracting the elements in positions \( j_1, \ldots, j_2 \); that is, \( \sigma' \) is the subsequence obtained from function \( f \) of length \( j_2 - j_1 + 1 \), where \( f(i) = i + j_1 - 1 \) for all \( i \in \text{dom}(\sigma') \).
2.3 Partial Orders

We recall some basic definitions and results regarding partial orders, and in particular, complete partial orders (cpos) from [14, 15]. 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 [15] [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 $\bigsqcup P = \bigsqcup \{c_i \mid i \in I\}$.

A finite or infinite sequence of elements, $c_0 c_1 c_2 \ldots$, of a partially ordered set $(S, \sqsubseteq)$ is called a chain if $c_i \sqsubseteq c_{i+1}$ for each non-final 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 $\text{K}(S)$ 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 $\{d \in \text{K}(S) \mid d \sqsubseteq c\}$. 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

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: The proof is an extension of König's Lemma [22].

3 Describing Timed System Behavior

In this section, we give 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 section comes from the paper on the HIOA modeling framework [25]. The reader is referred to [25] for the proofs that are not included here.

3.1 Time

Throughout this paper, we fix a time axis $T$, which is a subgroup of $(\mathbb{R}, +)$, 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 $\mathbb{R}$ of real numbers, but the set $\mathbb{Z}$ of integers and the singleton set $\{0\}$ are also examples of allowed time axes. We define $T^{\geq 0} \triangleq \{ t \in T \mid t \geq 0 \}$.

An interval $J$ is a nonempty, convex subset of $T$. We denote intervals as usual: $[t_1, t_2] = \{ t \in T \mid t_1 \leq t \leq t_2 \}$, $(t_1, t_2) = \{ t \in T \mid t_1 \leq t < t_2 \}$ etc. An interval $J$ is left-closed (right-closed) if it has a minimum (resp., maximum) element, and 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 $\mathbb{R} \cup \{-\infty, \infty\}$. For $K \subseteq T$ and $t \in T$, 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 the paper where we use $\mathbb{R}$ as the time domain we assume that the relation $\leq$ on $\mathbb{R}$ extends to a relation on $\mathbb{R} \cup \{\infty\}$ such that $\infty \leq \infty$ and for all $t \in \mathbb{R}$, $t < \infty$.
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:

- \( \text{type}(v) \), the (static) type of \( v \). This is a nonempty set of values.
- \( \text{dtype}(v) \), the dynamic type of \( v \). This is a set of functions from left-closed intervals of \( \mathbb{T} \) to \( \text{type}(v) \) that satisfies the following properties:
  
  1. (Closure under time shift)
     For each \( f \in \text{dtype}(v) \) and \( t \in \mathbb{T} \), \( f + t \in \text{dtype}(v) \).
  
  2. (Closure under subinterval)
     For each \( f \in \text{dtype}(v) \) and each left-closed interval \( J \subseteq \text{dom}(f) \), \( f \restriction J \in \text{dtype}(v) \).
  
  3. (Closure under pasting)
     Let \( f_0, f_1, f_2, \ldots \) be a sequence of functions in \( \text{dtype}(v) \) such that, for each index \( i \) such that \( f_i \) is not the final function in the sequence, \( \text{dom}(f_i) \) is right-closed and \( \max(\text{dom}(f_i)) = \min(\text{dom}(f_{i+1})) \). Then the function \( f \) defined by \( f(t) = f_i(t) \), where \( i \) is the smallest index such that \( t \in \text{dom}(f_i) \), is in \( \text{dtype}(v) \).

Example 3.1 (Discrete variables) Let \( v \) be any variable and let \( \text{Constant} \) be the set of constant functions from a left-closed interval of \( \mathbb{T} \) to \( \text{type}(v) \). Then \( \text{Constant} \) is closed under time shift and subinterval. If the dynamic type of \( v \) is obtained by closing \( \text{Constant} \) under the pasting operation, then \( v \) is called a discrete variable. This is essentially the same as the definition of a discrete variable in [31].

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

Example 3.3 (Standard real-valued function classes) If we take \( \mathbb{T} = \mathbb{R} \) and \( \text{type}(v) = \mathbb{R} \), 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 [43], 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 subsection, 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 $v$ for $V$ is a function that associates with each variable $v \in V$ a value in $\text{type}(v)$. 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 $v \in V$, $\tau \downarrow v \in d\text{type}(v)$.

A trajectory for $V$ is a $J$-trajectory for $V$, for any $J$. We write $\text{trajs}(V)$ for the set of all trajectories for $V$.

A trajectory for $V$ with domain $[0,0]$ is called a point trajectory for $V$. If $v$ is a valuation for $V$ then $\varphi(v)$ denotes the point trajectory for $V$ that maps 0 to $v$. 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 $J = T^{\geq 0}$. 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 $\tau$ is a trajectory then $\tau\.ltime$, the limit time of $\tau$, is the supremum of $\text{dom}(\tau)$. We define $\tau\.first$, the first valuation of $\tau$, to be $\tau(0)$, and if $\tau$ is closed, we define $\tau\.last$, the last valuation of $\tau$, to be $\tau(\tau\.ltime)$. For $\tau$ a trajectory and $t \in T^{\geq 0}$, we define

\[
\begin{align*}
\tau \preceq t & \triangleq \tau \upharpoonright [0,t], \\
\tau \prec t & \triangleq \tau \upharpoonright (0,t), \\
\tau \succeq t & \triangleq (\tau \upharpoonright [t,\infty)) - t.
\end{align*}
\]

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 $\tau \preceq \infty \triangleq \tau$ and $\tau \prec \infty \triangleq \tau$.

3.3.2 Prefix Ordering

Trajectory $\tau$ is a prefix of trajectory $u$, denoted by $\tau \preceq u$, if $\tau$ can be obtained by restricting $u$ to a subset of its domain. Formally, if $\tau$ and $u$ are trajectories for $V$, then $\tau \preceq u$ iff
\( \tau = v \upharpoonright \text{dom}(\tau) \). Alternatively, \( \tau \leq v \) iff there exists a \( t \in \mathbb{T}^{\geq 0} \cup \{\infty\} \) such that \( \tau = v \leq_t \) or \( \tau = v <_t \). If \( \tau \leq v \) then clearly dom(\( \tau \)) \( \subseteq \) dom(\( v \)). If \( T \) is a set of trajectories for \( V \), then \( \text{pref}(T) \) denotes the prefix closure of \( T \), defined by:

\[
\text{pref}(T) \triangleq \{ \tau \in \text{trajs}(V) \mid \exists v \in T : \tau \leq v \}.
\]

We say that \( T \) is prefix closed if \( T = \text{pref}(T) \).

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 \( \text{trajs}(V) \) 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 \( \tau \) and \( \tau' \) are trajectories for \( V \), with \( \tau \) closed. Then the concatenation \( \tau \odot \tau' \) is the function given by

\[
\tau \odot \tau' \triangleq \tau \cup (\tau' \upharpoonright [(0, \infty) + \tau, \text{time}]).
\]

Because dynamic types are closed under time shift and pasting, it follows that \( \tau \odot \tau' \) is a trajectory for \( V \). Observe that \( \tau \odot \tau' \) is finite (resp., closed, full) if and only if \( \tau' \) 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 \( \tau \) and \( v \) be trajectories for \( V \) with \( \tau \) closed. Then

\[
\tau \leq v \iff \exists \tau' : v = \tau \odot \tau'.
\]

Note that if \( \tau \leq v \), then the trajectory \( \tau' \) such that \( v = \tau \odot \tau' \) is unique except that it has an arbitrary value for \( \tau', \text{final} \). Note also that the "\( \leq \)" 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 \( \tau_0 \tau_1 \tau_2 \ldots \) be a (finite or infinite) sequence of trajectories such that \( \tau_i \) is closed for each nonfinal index \( i \). Define trajectories \( \tau'_0, \tau'_1, \tau'_2, \ldots \) inductively by

\[
\begin{align*}
\tau'_0 & \triangleq \tau_0, \\
\tau'_{i+1} & \triangleq \tau'_i \bigcap \tau_{i+1} \text{ for nonfinal } i.
\end{align*}
\]

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

### 3.4 Hybrid Sequences

In this subsection, 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 \( \alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots \), where

1. each \( \tau_i \) is a trajectory in \( \text{trajs}(V) \),
2. each \( a_i \) is an action in \( A \),
3. if \( \alpha \) is a finite sequence then it ends with a trajectory, and
4. if \( \tau_i \) is not the last trajectory in \( \alpha \) then \( \text{dom}(\tau_i) \) 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 [37], where state changes at a single real time are modeled using a notion of “superdense time”. Specifically, hybrid behavior is modeled in [37] using functions from an extended time domain, which includes countably many elements for each real time, to states.

If \( \alpha \) is a hybrid sequence, with notation as above, then we define the limit time of \( \alpha \), \( \alpha.ltime \), to be \( \sum_i \tau_i.ltime \). A hybrid sequence \( \alpha \) is defined to be:
• time-bounded if $\alpha$.ltime is finite.

• admissible if $\alpha$.ltime = $\infty$.

• closed if $\alpha$ is a finite sequence and the domain of its final trajectory is a closed interval.

• Zeno if $\alpha$ is neither closed nor admissible, that is, if $\alpha$ 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 $\alpha$ is not Zeno.

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

If $\alpha$ is a closed $(A, V)$-sequence, where $V = \emptyset$ and $\beta \in \text{trajs}(\emptyset)$, we call $\alpha \sqsubseteq \beta$ a time-extension of $\alpha$.

### 3.4.2 Prefix Ordering

We say that $(A, V)$-sequence $\alpha = \tau_0 a_1 \tau_1 \ldots$ is a prefix of $(A, V)$-sequence $\beta = v_0 b_1 v_1 \ldots$, denoted by $\alpha \leq \beta$, provided that (at least) one of the following holds:

1. $\alpha = \beta$.

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

Like the set of trajectories over $V$, the set of $(A, V)$-sequences is 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 $\alpha$ and $\alpha'$ are $(A, V)$-sequences with $\alpha$ closed. Then the concatenation $\alpha \sqcup \alpha'$ is the $(A, V)$-sequence given by

$$\alpha \sqcup \alpha' \triangleq \text{init}(\alpha) \ (\text{last}(\alpha) \sqcup \text{head}(\alpha')) \ \text{tail}(\alpha').$$

(Here, init, last, head and tail are ordinary sequence operations.)
Lemma 3.7 Let $\alpha$ and $\beta$ be $(A,V)$-sequences with $\alpha$ closed. Then

$$\alpha \leq \beta \iff \exists \alpha' : \beta = \alpha \cdot \alpha'.$$

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

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 $\alpha_i$ is closed for each nonfinal index $i$. Define $(A,V)$-sequences $\alpha'_0, \alpha'_1, \ldots$ inductively by

$$\alpha'_0 \triangleq \alpha_0,$$
$$\alpha'_{i+1} \triangleq \alpha'_i \cdot \alpha_{i+1} \text{ for nonfinal } i.$$

Lemma 3.7 implies that for each nonfinal $i$, $\alpha'_i \leq \alpha'_{i+1}$. We define the concatenation $\alpha_0 \cdot \alpha_1 \ldots$ to be the limit of the chain $\alpha'_0 \alpha'_1 \ldots$; 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 $\alpha$, denoted by $\alpha \restriction (A',V')$, is obtained by first projecting all trajectories of $\alpha$ 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:

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

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

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:

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

Lemma 3.8 $(A',V')$-restriction is a continuous operation.
Lemma 3.9 \((\alpha_0 \land \alpha_1 \land \cdots) \models (A, V) = \alpha_0 \models (A, V) \land \alpha_1 \models (A, V) \land \cdots\).

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

Lemma 3.11 Let \(\alpha\) be a hybrid sequence \(A\) a set of actions and \(V\) a set of variables.

1. \(\alpha\) is time-bounded if and only if \(\alpha \models (A, V)\) is time-bounded.
2. \(\alpha\) is admissible if and only if \(\alpha \models (A, V)\) is admissible.
3. If \(\alpha\) is closed then \(\alpha \models (A, V)\) is closed.
4. If \(\alpha\) is non-Zeno then \(\alpha \models (A, V)\) is non-Zeno.

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

4 Timed Automata

In this section, as a preliminary step toward defining timed I/O automata, 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 added in Section 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) \(A = (X, Q, \Theta, E, H, D, T)\) consists of:
A set $X$ of internal variables.

A set $Q \subseteq \text{val}(X)$ 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 $D \subseteq Q \times A \times Q$ of discrete transitions. We use $x \xrightarrow{a,A} x'$ as shorthand for $(x, a, x') \in D$. Here and elsewhere, we sometimes drop the subscript and write $x \xrightarrow{a} x'$, when we think $A$ should be clear from the context. We say that $a$ is enabled in $x$ if $x \xrightarrow{a} x'$ for some $x'$. We say that a set $C$ of actions is enabled in a state $x$ if some action in $C$ is enabled in $x$.

A set $T$ of trajectories for $X$ such that $\tau(t) \in Q$ for every $\tau \in T$ and $t \in \text{dom}(\tau)$. Given a trajectory $\tau \in T$ we denote $\tau.fval$ by $\tau.fstate$ and, if $\tau$ is closed, we denote $\tau.fval$ by $\tau.lstate$. When $\tau.fstate = x$ and $\tau.lstate = x'$, we sometimes write $x \xrightarrow{\tau,A} x'$.

We require that the following axioms hold:

T0 (Existence of point trajectories)
If $x \in Q$ then $\phi(x) \in T$.

T1 (Prefix closure)
For every $\tau \in T$ and every $\tau' \leq \tau$, $\tau' \in T$.

T2 (Suffix closure)
For every $\tau \in T$ and every $t \in \text{dom}(\tau)$, $\tau \triangleright t \in T$.

T3 (Concatenation closure)
Let $\tau_0 \tau_1 \tau_2 \ldots$ be a sequence of trajectories in $T$ such that, for each nonfinal index $i$, $\tau_i$ is closed and $\tau_i.lstate = \tau_{i+1}.fstate$. Then $\tau_0 \circ \tau_1 \circ \tau_2 \cdots \in T$.

Thus, a timed automaton is essentially a hybrid automaton in the sense of [25] in which $W$, the set of external variables, is empty. (The only difference is the addition of the axiom T0, which does not affect any of the results of [25].) This definition differs from previous definitions of timed automata [28, 40] 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 $A$ by $X_A$, $Q_A$, $\Theta_A$, $E_A$, etc., and the components of a TA $A_i$ by $X_i$, $Q_i$, $\Theta_i$, $E_i$, 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 $\mathbb{R}$, $\tau$ 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 $\text{dom}(\tau) \rightarrow \text{type}(v)$, which gives the value of $v$ at all times during trajectory $\tau$. Similarly, we view any expression $e$ containing variables from $V$ as a function with domain $\text{dom}(\tau)$. 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 $\tau$ satisfies the algebraic equation

$$v = e$$

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

$$d(v) = e$$

if, for every $t \in \text{dom}(\tau)$, $v(t) = v(0) + \int_0^t e(t')dt'$. Equivalently, for every $t_1, t_2 \in \text{dom}(\tau)$ 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 [38], who call functions defined in this way “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 $t \in \text{dom}(\tau)$, $e(t) \leq v(t)$. That is, the constraint expressed by the inequality $e \leq v$ holds for each state of trajectory $\tau$. Similarly, the inequality

$$v \leq e$$

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

$$e \leq d(v)$$

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

$$d(v) \leq e$$

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

21
Conventions for automata specifications: In all the examples of this monograph we assume the time axis $T$ to be $\mathbb{R}$ and specify timed automata by using a variant of the language presented in [34, 20].

An automaton specification consists of four main parts: a signature, which lists the actions along with their kinds 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.

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 $\mathbb{R} \cup \{\infty\}$.

The dynamic types of variables are specified implicitly. 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 3.1 and 3.2. Note that according to these definitions, a variable of type Real is both analog and discrete if its dynamic type consists of piecewise constant functions only. The keyword discrete is used to qualify the type Real in the declaration of such variables. 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 invariants, 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 predicate in the invariant clause, 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 $d(v) = e$ for $d(v) = e$, $d(v) \leq e$ for $d(v) \leq e$ and $e \leq d(v)$ for $e \leq d(v)$. We assume that the value of a discrete variable is constant throughout a trajectory. That is, time-passage
does not change the value of discrete variables. If the evolution of a variable follows a continuous function throughout a trajectory then we write \textit{continuous}(v). Note that our interpretation of differential inequalities allows “jumps” in trajectories. Therefore, extra caution is needed in the specification of real variables that may evolve at a varying rate. If a variable is assumed to be continuous throughout each trajectory this must be explicitly stated in the trajectory definition.

\textbf{Example 4.1 (Time-bounded channel)} The automaton TimedChannel(b:Real, M:Type)

\begin{verbatim}
automaton TimedChannel(b: Real, M: Type)
    signature
        external send(m: M), receive(m: M)
    states
        queue: Queue[M] := {},
        now: Real := 0,
        initially b ≥ 0
    transitions
        external send(m)
            eff
                queue := append((m,now+b),queue)
        external receive(m)
            pre
                ∃u: Real (m,u) = head(queue)
            eff
                queue := tail(queue)
    trajectories
        stop when
            ∃(m,u): (M,Real) (m,u) ∈ queue ∧ (now = u)
        evolve
            d(now) = 1
\end{verbatim}

Figure 1: Time-bounded channel

in Figure 1 is the specification of a reliable FIFO channel that delivers its messages within a certain time 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 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 whose 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 queue is kept constant by trajectories and 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: Real, M: Type)

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

Figure 2: Periodic sending process

in Figure 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.

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(u: Real, M: Type) process from Example 4.2 does not model failures. We now consider a variant of PeriodicSend(u: Real, M: Type) where the process may fail and stop doing any discrete actions. The specification of this new automaton is given in Figure 3.
automaton PeriodicSend2(u: Real, M: Type)
  signature
    external send(m: M), fail
  states
    failed: Bool := false,
    clock: Real := 0
  initially u ≥ 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 3: Periodic sending process with failures

The discrete variable failed in automaton PeriodicSend2(u:Real, M:Type) is a boolean flag that records whether the process is failed. It is initialized to false and is set to true when a fail action occurs. The trajectory specification of PeriodicSend2(u:Real, M:Type) shows that time can advance without any bound when the process is failed.

Example 4.4 (Timeout process) The automaton Timeout(u:Real, M:Type) in Figure 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(u:Real, M:Type) performs a timeout action, thereby “suspecting” the other process. When a message arrives it “unsuspects” the other process. Timeout(u:Real, M:Type) may suspend and unsuspend repeatedly.

The discrete variable suspected is a flag that shows whether Timeout(u:Real, M:Type) suspects that the other process is failed. The variable clock is a timer that records the amount of time that has elapsed 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
automaton Timeout(u: Real, M: Type)
signature
  external receive(m: M), timeout
states
  suspected: Bool := false,
  clock Real := 0
initially u > 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: Timeout

a result of a timeout.

The discrete variable suspected remains constant throughout each trajectory. The trajectory specification also 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 mutual exclusion algorithm) The automaton FischerME(uset, lcheck: Real) presented in Figures 5 and 6 is the specification of a shared memory mutual exclusion algorithm which 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 five elements, one of which represents a special value. Note, however, that this specification can be generalized to any finite number of participants. The automaton parameters uset and lcheck represent upper and lower time bounds for the set(i) and check(i) actions respectively. We assume that uset < lcheck.

The shared variable x can be assigned any value of type Index where nil is regarded as a special value. If a process is in the critical region, then the variable x contains the
type Index = enumeration of nil, p1, p2, p3, p4

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

automaton FischerME(u_set, l_check: Real)
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: 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
initially u_set ≥ 0 ∧ l_check ≥ 0 ∧ u_set < l_check

Figure 5: Fischer’s mutual exclusion algorithm: Signature and states

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 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), test(i), crit(i), and exit(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 trajectory specification says that the values of discrete variables are kept constant
by trajectories. 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.
transitions

external try(i)
  pre
  pc[i] = rem
  eff
  pc[i] := test

internal test(i)
  pre
  pc[i] = test
  eff
  if x = p0 then
    pc[i] := set;
    lastset[i] := now + u_set

internal set(i)
  pre
  pc[i] = set
  eff
  x := 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 = i then pc[i] := leavetry
  else pc[i] := test

trajectories

stop when
  ∃ i: Index now = lastset[i]

 evolve
d(now) = 1

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 := i;
  pc[i] := leaveexit

external rem(i)
  pre
  pc[i] = leaveexit
  eff
  pc[i] := rem

Figure 6: Fischer's mutual exclusion algorithm: Transitions and trajectory definitions
automaton ClockSync(u,r: Real, i: Index)

signature
    external send(m: Real, const i: Index),
    receive(m: Real, j: Index, const i: Index) where j ≠ i

states
    nextsend: Real := 0,
    maxother: Real := 0,
    physclock: Real := 0

initially u > 0 ∧ (0 ≤ r < 1)

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
    invariant
        continuous(physclock)
    stop when
        physclock = nextsend
    evolve
        (1 - r) ≤ d(physclock) ≤ (1 + r)

Figure 7: Clock synchronization

Example 4.6 (Clock synchronization)

The automaton ClockSync(u,r:Real, i:Index) in Figure 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.

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(u,r:Real, i:Index) has a physical clock physclock, which
may drift from the real time with a drift rate bounded by \( r \). It uses the variable \( \text{maxother} \) to keep track of the largest physical clock value of the other processes in the system. The variable \( \text{nextsend} \) records when it is supposed to send its physical clock to the other processes. The logical clock, \( \text{logclock} \), is defined to be the maximum of \( \text{maxother} \) and \( \text{physclock} \). Formally \( \text{logclock} \) is a derived variable, which is a function whose value is defined in terms of the state variables.

A \( \text{send}(m,i) \) transition is enabled when \( m = \text{physclock} \) and \( \text{nextsend} = \text{physclock} \). It causes the value of \( \text{nextsend} \) to be updated so that the next send can occur when \( \text{physclock} \) has advanced by \( u \) time units. The transition definition for \( \text{receive}(m,j,i) \) specifies the effect of receiving a message from another process \( j \) in the system. Upon the receipt of a message \( m \) from \( j \), \( i \) sets \( \text{maxother} \) to the maximum of \( m \) and the current value of \( \text{maxother} \), 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 \( \text{physclock} \) 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 \( \text{physclock} = \text{nextsend} \).

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 \( A \) is an \((A, V)\)-sequence \( \alpha = \tau_0 \ a_1 \ \tau_1 \ a_2 \ \tau_2 \ ... \), where (1) each \( \tau_i \) is a trajectory in \( T \), and (2) if \( \tau_i \) is not the last trajectory in \( \alpha \) then \( \tau_i.\text{Istate} \overset{a_{i+1}}{\rightarrow} \tau_{i+1}.\text{fstate} \). 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 \( \text{frugs}_A \) for the set of all execution fragments of \( A \).

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

A state of \( A \) is reachable if it is the last state of some closed execution of \( A \). A property that is true for all reachable states of an automaton is called an invariant assertion, or invariant, for short.
Lemma 4.7 Let $\alpha_0 \alpha_1 \ldots$ be a finite or infinite sequence of execution fragments of $\mathcal{A}$ such that, for each nonfinal index $i$, $\alpha_i$ is closed and $\alpha_i.fstate = \alpha_{i+1}.fstate$. Then $\alpha_0 \sqsubseteq \alpha_1 \sqsubseteq \ldots$ is an execution fragment of $\mathcal{A}$.

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

Lemma 4.8 Let $\alpha$ and $\beta$ be execution fragments of $\mathcal{A}$ with $\alpha$ closed. Then

$$\alpha \leq \beta \iff \exists \alpha' \in \mathfrak{frgs}_{\mathcal{A}} : \beta = \alpha \sqcup \alpha'.$$

Proof: Implication "$\Rightarrow$" follows directly from the corresponding implication in Lemma 3.7. Implication "$\Leftarrow$" follows from the definitions and 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 $\alpha$ is an execution fragment, then the trace of $\alpha$, denoted by $\text{trace}(\alpha)$, is the $(E,\emptyset)$-restriction of $\alpha$, $\alpha \upharpoonleft (E,\emptyset)$. A trace fragment of a timed automaton $\mathcal{A}$ from a state $x$ of $\mathcal{A}$ is the trace of an execution fragment of $\mathcal{A}$ whose first state is $x$. We write $\text{trace}_{\mathfrak{frgs}_{\mathcal{A}}}(x)$ for the set of trace fragments of $\mathcal{A}$ from $x$. 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 $\text{traces}_{\mathcal{A}}$ for the set of traces of $\mathcal{A}$.

In the earlier timed automaton models [28, 40], execution fragments were defined in a similar style 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 section and give sample executions and traces for these automata.

Example 4.9 (Periodic sending process) Consider the automaton $\text{PeriodicSend}(u:\text{Real}, M:\text{Type})$ from Example 4.2 where $u$ is instantiated to the real number 3 and the message type parameter $M$ is instantiated to the set $\{m_1, m_2, \ldots \}$. The following sequence is an execution of the automaton:

$$\alpha = \tau_0 \text{send}(m_1) \tau_1 \text{send}(m_2) \tau_2 \text{send}(m_3) \tau_3 \ldots$$
where \( \tau_i: [0, 3] \rightarrow val(\{\text{clock}\}) \) are defined such that \( \tau_i(t)(\text{clock}) = t \) for all \( t \in [0, 3] \).

The functions \( \tau_i \) are defined for closed intervals of length 3, starting at time 0. They describe the evolution of the variable \( \text{clock} \), which is 0 at the start of each \( \tau_i \) and increases with rate 1 for 3 time units. The discrete \text{send} events occur periodically, every 3 time units and reset the \text{clock} variable to 0.

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

\[
\alpha = \tau'_0 \text{ send}(m1) \tau'_1 \text{ send}(m2) \tau'_2 \text{ send}(m3) \tau'_3 \ldots
\]

where \( \tau'_i: [0, 3] \rightarrow val(\emptyset) \).

Since the range of each function \( \tau'_i \) contains only the function with the empty domain, \( trace(\alpha) \) does not contain any information about what happens to the value of \text{clock} as time progresses. Since the domains of each \( \tau_i \) and \( \tau'_i \) are identical, \( \alpha \) and \( trace(\alpha') \) express the same information about the amount of time that elapses between discrete steps.

\[\square\]

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

\[
\alpha = \tau_0 \text{ receive}(m1) \tau_1 \text{ timeout} \tau_2 \text{ receive}(m2) \tau_3 \text{ timeout} \tau_4
\]

where \( Val = val(\{\text{suspected, clock}\}) \) and the functions \( \tau_0, \tau_1, \tau_2, \tau_3, \tau_4 \) are defined as follows:

\[
\begin{align*}
\tau_0 &: [0, 2] \rightarrow Val \text{ where } \tau_0(t)(\text{suspected}) = \text{false} \text{ and } \tau_0(t)(\text{clock}) = t \text{ for all } t \in [0, 2], \\
\tau_1 &: [0, 5] \rightarrow Val \text{ where } \tau_1(t)(\text{suspected}) = \text{false} \text{ and } \tau_1(t)(\text{clock}) = t \text{ for all } t \in [0, 5], \\
\tau_2 &: [0, 1] \rightarrow Val \text{ where } \tau_2(t)(\text{suspected}) = \text{true} \text{ and } \tau_2(t)(\text{clock}) = 5 + t \text{ for all } t \in [0, 1], \\
\tau_3 &: [0, 5] \rightarrow Val \text{ where } \tau_3(t)(\text{suspected}) = \text{false} \text{ and } \tau_3(t)(\text{clock}) = t \text{ for all } t \in [0, 5], \\
\tau_4 &: [0, \infty) \rightarrow Val \text{ where } \tau_4(t)(\text{suspected}) = \text{true} \text{ and } \tau_4(t)(\text{clock}) = 5 + t \text{ for all } t \in [0, \infty).
\end{align*}
\]

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 \text{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 \text{Timeout}(u:\text{Real}, M:\text{Type})\) 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 $\tau_4$ with a function on a closed interval would yield a non-Zeno execution that is not admissible.

The trace of the execution $\alpha$ can be obtained by letting the range of $\tau_i$ 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 1:** In any reachable state $x$ of automaton \texttt{TimedChannel(b:Real, M:Type)}, for all $(m, u)$ in $x(queue)$, $x(now) \leq u \leq x(now) + b$.

Such an invariant can be proved by induction. Recall that reachable states are the final states of closed executions. Axioms \textbf{T1} and \textbf{T2} allow us to view any closed execution as a concatenation of closed execution fragments, $a_0 = a_1 \ldots = a_k$, where every $a_i$ is either a closed trajectory or a discrete action surrounded by point trajectories, and where $a_i.state = a_{i+1}.state$ 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 $a_i$.

---

**Example 4.12 (Fischer’s mutual exclusion)** The main safety property that needs to be satisfied by the automaton \texttt{FischerME(u.set, l.check:Real)} from Example 4.5 is mutual exclusion. This safety property can be expressed as an invariant assertion:

**Invariant 1:** In any reachable state $x$ of \texttt{FischerME(u.set, l.check:Real)}, there do not exist $i:Index$ and $j:Index$ such that $x(pc)[i] = crit$ and $x(pc)[j] = crit$.

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 4.12:

**Invariant 2:** In any reachable state $x$ of \texttt{FischerME(u.set, l.check:Real)}, if $x(pc)[i] = check$, $x(x) = i$, and $x(pc)[j] = set$, then $x(firstcheck)[i] > x(lastset)[j]$.

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 $i$, then $i$ will allow enough time for $j$ to set $x$ to $j$, before performing the check. If this timing constraint were not satisfied, it would be possible for $i$ to check that $x = i$ before $j$ sets $x$ to $j$. Both of the processes would then observe $x$ to contain their own index and enter the critical region.
Lemma 4.13 If $\alpha$ is an execution of $A$ then

1. $\alpha$ is time-bounded if and only if $\text{trace}(\alpha)$ is time-bounded.
2. $\alpha$ is admissible if and only if $\text{trace}(\alpha)$ is admissible.
3. If $\alpha$ is closed then $\text{trace}(\alpha)$ is closed.
4. If $\alpha$ is non-Zeno then $\text{trace}(\alpha)$ is non-Zeno.

Proof: It follows directly from the restriction of $(A,V)$-sequences. ■

Lemma 4.14 If $\beta$ is a trace of $A$ then

1. If $\beta$ is closed then there exists an execution $\alpha$ of $A$ such that $\text{trace}(\alpha) = \beta$ and $\alpha$ is closed.
2. If $\beta$ is non-Zeno then there exists an execution $\alpha$ of $A$ such that $\text{trace}(\alpha) = \beta$ and $\alpha$ is non-Zeno.

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

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

Example 4.15 (Constructing a closed execution from a closed trace) Consider the Zeno hybrid sequence $\alpha = \phi(v) a \phi(v) a \phi(v) \ldots$ given in Example 3.12. Suppose that $\alpha$ is an execution of $A$ and that $a$ is an internal action of $A$. Then, $\text{trace}(\alpha) = \phi(v')$ where $\phi(v')$ is a trajectory over the empty set of variables. However, the fact that $\text{trace}(\alpha)$ is closed does not imply that $\alpha$ 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 $A$ with trace $\phi(v')$ as explained in the proof of Lemma 4.14. The execution consisting of the point trajectory $\phi(v')$ is a closed execution of $A$ with trace $\phi(v')$. ■

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 in later sections.
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 \( A \) has finite internal nondeterminism (FIN) provided that:

1. The set \( \Theta \) of start states is finite, and

2. For every state \( x \) of \( A \) and every trace fragment \( \beta \) of \( A \) from \( x \), the set \( \{ \alpha.\text{Istate} \mid \alpha \in \text{frags}_A(x) \land \text{trace}(\alpha) = \beta \} \) is finite.

Example 4.16 (Automata with FIN) The automata \( \text{TimedChannel}(b:\text{Real}, M:\text{Type}) \), \( \text{PeriodicSend}(u:\text{Real}, M:\text{Type}) \), \( \text{PeriodicSend2}(u:\text{Real}, M:\text{Type}) \), and \( \text{Timeout}(u:\text{Real}, M:\text{Type}) \) 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 \( x \) and every trace fragment \( \beta \) from \( x \), there is a unique execution fragment \( \alpha \) such that \( \text{trace}(\alpha) = \beta \).

Example 4.17 (Automata without FIN) We show that \( \text{FischerME}(u:\text{set}, l:\text{check}:\text{Real}) \) and \( \text{ClockSync}(a,r:\text{Real}, i:\text{Index}) \) 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 \( x \) be the start state of \( \text{FischerME}(u:\text{set}, l:\text{check}:\text{Real}) \) and \( \beta = \tau_0 \text{try}(i) \tau_1 \) be a trace of the same automaton where the domains of the functions \( \tau_0 \) and \( \tau_1 \) 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 \( \alpha \), \( \text{trace}(\alpha) = \beta \), if and only if \( \alpha.\text{time} = u \), \( \text{try}(i) \) occurs at time 0, and all the actions in \( \alpha \) that occur after \( \text{try}(i) \) are internal actions. There are infinitely many different times that the internal actions may occur, and infinitely many values \( \text{lastcheck} \) and \( \text{firstcheck} \) could have, by the time \( u \). Therefore, the set \( \{ \alpha.\text{Istate} \mid \alpha \in \text{frags}_A(x) \land \text{trace}(\alpha) = \tau_0 \text{try}(i) \tau_1 \} \) is not finite and \( \text{FischerME}(u:\text{set}, l:\text{check}:\text{Real}) \) does not have FIN.

Now, let \( x \) be the start state of \( \text{ClockSync}(a,r:\text{Real}, i:\text{Index}) \) where \( x(\text{physclock}) = x(\text{nextsend}) = x(\text{maxother}) = 0 \) and \( \beta = \tau_0 \text{send}(0) \tau_1 \) be a trace of \( \text{ClockSync}(a,r:\text{Real}, i:\text{Index}) \) where the domains of functions \( \tau_0 \) and \( \tau_1 \) 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 \( \alpha \) in which \( \text{send}(0) \) occurs at time 0 and is followed by a trajectory \( \tau \) such that \( \tau.\text{time} = u \), we have \( \text{trace}(\alpha) = \beta \). For any such \( \alpha \), \( \alpha.\text{Istate}(\text{physclock}) \) can be any value in the interval \([u(1-r), u(1+r)]\). Therefore, the set \( \{ \alpha.\text{Istate} \mid \alpha \in \text{frags}_A(x) \land \text{trace}(\alpha) = \tau_0 \text{send}(0) \tau_1 \} \) is not finite and \( \text{ClockSync}(a,r:\text{Real}, i:\text{Index}) \) does not have FIN.

35
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 $A$ has FIN and $x \in Q$. Suppose that $\beta_1\beta_2 \ldots$ is a chain of trace fragments of $A$ from $x$. Then the hybrid sequence $\lim \beta_i$ is a trace fragment of $A$ from $x$.

**Proof:** This is analogous to the proof of Lemma 4.3 of [28]. Suppose that $A$ is a timed automaton that has FIN, $x$ is a state of $A$, and $\beta_1 \beta_2 \ldots$ is a chain of trace fragments of $A$ from $x$. We define a relation $\text{after}$ between trace fragments from $x$ and states of $A$: $\text{after} = \{(\beta, y) \mid \exists \alpha \in \text{frgs}_{A}(x). \text{trace}(\alpha) = \beta \land \alpha.lstate = y\}$.

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

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

Hence, $G$ satisfies the hypotheses of Lemma 2.3, which implies that there is an infinite execution fragment starting from $x$ whose trace is $\lim \beta_i$. Lemma 2.3 is an extension of Konig's lemma.

There are two references to automata with FIN later in the paper. 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.

**Feasible Timed Automata:** A timed automaton $A$ is feasible provided that, for every state $x$ of $A$, there exists an admissible execution fragment of $A$ from $x$.

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

**Timing-Independent Timed Automata:** A timed automaton $A$ 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 Corollary 7.8.

36
4.4 Implementation Relationships

Timed automata $A_1$ and $A_2$ are comparable if they have the same external interface, that is, if $E_1 = E_2$. If $A_1$ and $A_2$ are comparable then we say that $A_1$ implements $A_2$, denoted by $A_1 \preceq A_2$, if the traces of $A_1$ are included among those of $A_2$, that is, if $\text{traces}_{A_1} \subseteq \text{traces}_{A_2}$.\footnote{In [28, 13, 26, 27], 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 paper using the newer, more inclusive definition imply corresponding results for the earlier definition. For example, we have the following property: If $A_1 \preceq A_2$ then the set of traces that arise from closed or admissible executions of $A_1$ is a subset of the set of traces that arise from closed or admissible executions of $A_2$. This follows from Lemmas 4.13 and 4.14.}

Other preorders between timed automata could also be used as implementation relationships, for example, if $A_1$ and $A_2$ are comparable timed automata, we could consider:

- Every closed trace of $A_1$ is a trace of $A_2$.
- Every admissible trace of $A_1$ is a trace of $A_2$.
- Every non-Zeno trace of $A_1$ is a trace of $A_2$.

**Theorem 4.19** Let $A_1$ and $A_2$ be comparable TAs.

1. If every closed trace of $A_1$ is a trace of $A_2$ and $A_2$ has FIN, then $A_1 \preceq A_2$.
2. If every admissible trace of $A_1$ is a trace of $A_2$ and $A_1$ is feasible, then every closed trace of $A_1$ is a trace of $A_2$.
3. If every admissible trace of $A_1$ is a trace of $A_2$, $A_1$ is feasible, and $A_2$ has FIN, then $A_1 \preceq A_2$.

**Proof:** Part 1 follows from Lemma 4.18.

For Part 2, consider a closed trace $\beta$ of $A_1$. By feasibility of $A_1$, we may extend $\beta$ to an admissible trace $\beta'$ of $A_1$. Then by assumption, $\beta'$ is also a trace of $A_2$. By prefix closure of the set of traces, $\beta$ is a trace of $A_2$.

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 types of simulation relations: forward and backward simulations.
Forward simulations are more commonly used than 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 non-deterministic 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 \( A \) and \( B \) be comparable TAs. A forward simulation from \( A \) to \( B \) is a relation \( R \subseteq Q_A \times Q_B \) satisfying the following conditions, for all states \( x_A \) and \( x_B \) of \( A \) and \( B \), respectively:

1. If \( x_A \in \Theta_A \) then there exists a state \( x_B \in \Theta_B \) such that \( x_A R x_B \).
2. If \( x_A R x_B \) and \( \alpha \) is an execution fragment of \( A \) consisting of one action surrounded by two point trajectories, with \( \alpha.fstate = x_A \), then \( B \) has a closed execution fragment \( \beta \) with \( \beta.fstate = x_B \), \( trace(\beta) = trace(\alpha) \), and \( \alpha.Istate \in R \beta.Istate \).
3. If \( x_A R x_B \) and \( \alpha \) is an execution fragment of \( A \) consisting of a single closed trajectory, with \( \alpha.fstate = x_A \), then \( B \) has a closed execution fragment \( \beta \) with \( \beta.fstate = x_B \), \( trace(\beta) = trace(\alpha) \), and \( \alpha.Istate \in R \beta.Istate \).

Forward simulation relations induce a preorder between timed automata.

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

The definition of a forward simulation from \( A \) to \( B \) yields a correspondence for open trajectories of \( A \):

**Lemma 4.21** Let \( A \) and \( B \) be comparable TAs and let \( R \) be a forward simulation from \( A \) to \( B \). Let \( x_A \) and \( x_B \) be states of \( A \) and \( B \), respectively, such that \( x_A R x_B \). Let \( \alpha \) be an execution fragment of \( A \) from state \( x_A \) consisting of a single open trajectory. Then \( B \) has an execution fragment \( \beta \) with \( \beta.fstate = x_B \) and \( trace(\beta) = trace(\alpha) \).

**Proof:** Let \( \tau \) be the single open trajectory in \( \alpha \). Using axioms \( T1 \) and \( T2 \), we construct an infinite sequence \( \tau_0 \tau_1 \ldots \) of closed trajectories of \( A \) such that \( \tau = \tau_0 \tau_1 \ldots \). Then, working recursively, we construct a sequence \( \beta_0 \beta_1 \ldots \) of closed execution fragments of \( B \) such that \( \beta_0.Istate = x_B \) and, for each \( i \), \( \tau_i.Istate \in R \beta_i.Istate \), \( \beta_i.Istate = \beta_{i+1}.fstate \), 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 \( \beta = \beta_0 \cap \beta_1 \cap \cdots \). By Lemma 4.7, \( \beta \) is an execution fragment of \( B \). Clearly, \( \beta.fstate = x_B \). By Lemma 3.9 applied to both \( \alpha \) and \( \beta \), \( \text{trace}(\beta) = \text{trace}(\alpha) \). Thus \( \beta \) has the required properties.

**Theorem 4.22** Let \( A \) and \( B \) be comparable TAs and let \( R \) be a forward simulation from \( A \) to \( B \). Let \( x_A \) and \( x_B \) be states of \( A \) and \( B \), respectively, such that \( x_A R x_B \). Then \( \text{tracefrags}_A(x_A) \subseteq \text{tracefrags}_B(x_B) \).

**Proof:** Suppose that \( \delta \) is the trace of an execution fragment of \( A \) that starts from \( x_A \); we prove that \( \delta \) is also a trace of an execution fragment of \( B \) that starts from \( x_B \). Let \( \alpha = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots \) be an execution fragment of \( A \) such that \( \alpha.fstate = x_A \) and \( \delta = \text{trace}(\alpha) \). We consider cases:

1. \( \alpha \) is an infinite sequence.
   
   Using axioms \( T1 \) and \( T2 \), we can write \( \alpha \) as an infinite concatenation \( a_0 \sim a_1 \sim a_2 \cdots \), in which the execution fragments \( a_i \) with \( i \) even consist of a trajectory only, and the execution fragments \( a_i \) with \( i \) odd consist of a single discrete step surrounded by two point trajectories.
   
   We define inductively a sequence \( \beta_0 \beta_1 \ldots \) of closed execution fragments of \( B \), such that \( \beta_0.fstate = x_B \) and, for all \( i \), \( \beta_i.fstate = \beta_{i+1}.fstate \), \( \alpha_i.lstate R \beta_i.lstate \), and \( \text{trace}(\beta_i) = \text{trace}(\alpha_i) \). We use Property 3 of the definition of a simulation for the construction of the \( \beta_i \)'s with \( i \) even, and Property 2 for the construction of the \( \beta_i \)'s with \( i \) odd. Let \( \beta = \beta_0 \cap \beta_1 \cap \beta_2 \cdots \). By Lemma 4.7, \( \beta \) is an execution fragment of \( B \). Clearly, \( \beta.fstate = x_B \). By Lemma 3.9, \( \text{trace}(\beta) = \text{trace}(\alpha) \). Thus \( \beta \) has the required properties.

2. \( \alpha \) is a finite sequence ending with a closed trajectory.
   
   Similar to the first case.

3. \( \alpha \) is a finite sequence ending with an open trajectory.
   
   Similar to the first case, using Lemma 4.21.

**Corollary 4.23** Let \( A \) and \( B \) be comparable TAs and let \( R \) be a forward simulation from \( A \) to \( B \). Then \( \text{traces}_A \subseteq \text{traces}_B \).

**Proof:** Suppose \( \beta \in \text{traces}_A \). Then \( \beta \in \text{tracefrags}_A(x_A) \) for some start state \( x_A \) of \( A \). Property 1 of the definition of simulation implies the existence of a start state \( x_B \) of \( B \) such that \( x_A R x_B \). Then Theorem 4.22 implies that \( \beta \in \text{tracefrags}_B(x_B) \). Since \( x_B \) is a start state of \( B \), this implies that \( \beta \in \text{traces}_B \), as needed.
Example 4.24 (Time-bounded channels) Consider two instances of the specification in Figure 1, TimedChannel(b1, M) and TimedChannel(b2, M) where b1 ≤ b2. We define a forward simulation $R$ from TimedChannel(b1, M) to TimedChannel(b2, M) below. If $x$ is a state of TimedChannel(b1, M) and $y$ is a state of TimedChannel(b2, M), then $x R y$ provided that the following conditions are satisfied:

1. $x(\text{now}) = y(\text{now})$.
2. $|x(\text{queue})| = |y(\text{queue})|$.
3. $\forall i, 1 \leq i \leq |x(\text{queue})|$, if $x(\text{queue})(i) = (m, u1)$ then $y(\text{queue})(i) = (m, u2)$ and $u1 \leq u2$.

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 $x$ and $y$ be, respectively, states of TimedChannel(b1, M) and TimedChannel(b2, M) such that $x R y$. 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 $x$ to $x'$ then we need to find an execution fragment $\beta$ of TimedChannel(b2, M) from $y$ ending in $y'$, such that $x' R y'$ and $\text{trc}(\beta)$ is the same as the trace of $\phi(x) \text{send}(m) \phi(y)$. The execution fragment $\beta = \phi(y) \text{send}(m) \phi(y')$ satisfies the required conditions. This follows from the hypothesis that $x R y$ and the definition of $R$, using the fact that the effect of a send($m$) action of TimedChannel(b1, M) on TimedChannel(b2, M) are, respectively, adding the entry $(m, \text{now} + b1)$ to $x(\text{queue})$, and $(m, \text{now} + b2)$ to $y(\text{queue})$ where $b1 \leq b2$.

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

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

$\blacksquare$
Example 4.25 (Time-bounded channel that keeps all messages) In this example we define a variant of TimedChannel(b, M) from Example 4.1 called TimedChannel2(b, M). The main difference between TimedChannel(b, M) and TimedChannel2(b, M) is that the message queue in TimedChannel2(b, M) 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(b, M) 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(b, M). 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. The automaton TimedChannel(b, M) can be viewed as an optimized implementation of TimedChannel2(b, M).

We define below a forward simulation $R$ from TimedChannel(b, M) to TimedChannel2(b, M). If $x$ is a state of TimedChannel(b, M) and $y$ is a state of TimedChannel2(b, M), then $x \ R \ y$ provided that the following conditions are satisfied:

1. $x(\text{now}) = y(\text{now})$.
2. $x(\text{queue}) = y(\text{queue}) (y(\text{ptr}) \ldots [y(\text{queue})])$.

Example 4.26 (Clock synchronization)

In this example, we define a forward simulation from ClockSync(u,r:Real, i:Index) of Figure 7 to an automaton that sends multiples of $u$. The specification of this automaton, which is called SendVal(u,r:Real, i:Index) is given in Figure 8. We assume that the definitions of type Index 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.

We now define a forward simulation $R$ from the automaton ClockSync(u, r, i) to the automaton SendVal(u, r, i) where u, r and and i r are actual parameters. If $x$ is a state of the automaton ClockSync(u, r, i) and $y$ is a state of SendVal(u, r, i), then $x \ R \ y$ provided that the following conditions are satisfied:

1. $y(\text{now})(1-r) \leq x(\text{physclock}) \leq y(\text{now})(1+r)$.
2. $y(\text{counter}) = x(\text{nextsend}) / u$.  

41
Figure 8: Clock synchronization

4.5.2 Refinements

Let \( A \) and \( B \) be comparable TAs. A refinement from \( A \) to \( B \) is a function \( F \subseteq Q_A \times Q_B \), satisfying the following conditions, for all states \( x_A \) and \( x_B \) of \( A \) and \( B \), respectively:

1. If \( x_A \in Q_A \) then \( F(x_A) \in Q_B \).

2. If \( \alpha \) is an execution fragment of \( A \) consisting of one action surrounded by two point trajectories, with \( \alpha.fstate = x_A \), then \( B \) has a closed execution fragment \( \beta \) with \( \beta.fstate = F(x_A) \), \( trace(\beta) = trace(\alpha) \), and \( \beta.Istate = F(\alpha.Istate) \).

3. If \( \alpha \) is an execution fragment of \( A \) consisting of a single closed trajectory, with \( \alpha.fstate = x_A \), then \( B \) has a closed execution fragment \( \beta \) with \( \beta.fstate = F(x_A) \), \( trace(\beta) = trace(\alpha) \), and \( \beta.Istate = F(\alpha.Istate) \).

Theorem 4.27 Let \( A \) and \( B \) be two TAs and suppose \( R \subseteq Q_A \times Q_B \). Then \( R \) is a refinement from \( A \) to \( B \) if and only if \( R \) is a forward simulation from \( A \) to \( B \) and \( R \) is a function.

Theorem 4.28 Let \( A, B \) and \( C \) be comparable TAs. If \( R_1 \) is a refinement from \( A \) to \( B \) and \( R_2 \) is a refinement from \( B \) to \( C \), then \( R_2 \circ R_1 \) is a refinement from \( A \) to \( C \).
An 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 isomorphic, if there exists an isomorphism from $\mathcal{A}$ to $\mathcal{B}$ (or, equivalently from $\mathcal{B}$ to $\mathcal{A}$).

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 $R \subseteq Q_A \times Q_B$ satisfying the following conditions, for all states $x_A$ and $x_B$ of $\mathcal{A}$ and $\mathcal{B}$, respectively:

1. If $x_A \in \Theta_A$ and $x_A R x_B$ then $x_B \in \Theta_B$.

2. If $x_A R x_B$ and $\alpha$ is an execution fragment of $\mathcal{A}$ with $\alpha.\text{Istate} = x_A$, consisting of one discrete action surrounded by two point trajectories, then $\mathcal{B}$ has a closed execution fragment $\beta$ with $\beta.\text{Istate} = x_B$, $\text{trace}(\beta) = \text{trace}(\alpha)$, and $\alpha.\text{Fstate} R \beta.\text{Fstate}$.

3. If $x_A R x_B$ and $\alpha$ is an execution fragment of $\mathcal{A}$ with $\alpha.\text{Istate} = x_A$, consisting of one trajectory, then $\mathcal{B}$ has a closed execution fragment $\beta$ with $\beta.\text{Istate} = x_B$, $\text{trace}(\beta) = \text{trace}(\alpha)$, and $\alpha.\text{Fstate} R \beta.\text{Fstate}$.

Backward simulations induce a preorder between timed automata.

**Theorem 4.29** 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 4.30** Let $\mathcal{A}$ and $\mathcal{B}$ be comparable TAs and let $R$ be a backward simulation from $\mathcal{A}$ to $\mathcal{B}$. Let $x_A$ and $x_B$ be states of $\mathcal{A}$ and $\mathcal{B}$, respectively, such that $x_A R x_B$. Let $\beta$ be the trace of a closed execution fragment of $\mathcal{A}$ from $y_A$ with last state $x_A$. Then there exists $y_B$ such that $\beta$ is also the trace of a closed execution fragment of $B$ from $y_B$ with last state $x_B$ and $y_A R y_B$.

**Proof:** Fix some $R$, $x_A$, $x_B$ and $\beta$ satisfying the conditions in the statement of the theorem. Let $\alpha \in \text{frags}_A(y_A)$ for some state $y_A$ of $\mathcal{A}$ with $\text{trace}(\alpha) = \beta$. By using the axioms $\text{T1}$ and $\text{T2}$, we can write $\alpha$ as the concatenation of a sequence of closed execution fragments, $\alpha = \alpha_0 \circ \alpha_1 \circ \ldots \alpha_n$, where each $\alpha_i$ is either a closed trajectory or an action surrounded by two point trajectories, and $\alpha_i.\text{Istate} = \alpha_{i+1}.\text{Fstate}$ for $0 \leq i \leq n$.

By using the definition of a backward simulation, working backwards from $\alpha_n$, we can construct an execution fragment $\alpha' = \alpha_0' \circ \alpha_1' \circ \ldots \alpha_n'$ from a state $y_B$ of $\mathcal{B}$ such that (a) $\alpha'.\text{Istate} = x_B$, (b) for all $i$, $0 \leq i \leq n$, $\alpha_i.\text{Fstate} R \alpha_i'.\text{Fstate}$ and $\text{trace}(\alpha_i') = \text{trace}(\alpha_i)$, (c) for all $i$, $0 \leq i \leq n - 1$, $\alpha_i'.\text{Istate} = \alpha_{i+1}'.\text{Fstate}$. Using Lemma 4.7, we can see that $\alpha'$ is an execution fragment of $\mathcal{B}$. By Lemma 3.9, $\text{trace}(\alpha) = \text{trace}(\alpha')$ as needed.

\[\Box\]
Corollary 4.31 Let $A$ and $B$ be comparable TAs and let $R$ be a backward simulation from $A$ to $B$. Then every closed trace of $A$ is a trace of $B$.

Proof: Suppose $R$ is a backward simulation from $A$ to $B$ and $\beta$ is a closed trace of $A$. Then $\beta = \text{trace}(\alpha)$ for some closed execution $\alpha$ of $A$. Let $x_A$ and $y_A$ be the first and last states of $\alpha$ respectively. By the totality of relation $R$, there exists some state $y_B$ of $B$ such that $y_A R y_B$. By Theorem 4.30, there exists $x_B$ of $B$ such that $\beta$ is the trace of a closed execution fragment of $B$ from $x_B$ with last state $y_B$ and $x_A R x_B$. Property 1 of the definition of a backward simulation relation implies that $x_B$ is a start state of $B$. It follows that $\beta \in \text{traces}_B$, as needed.

Theorem 4.32 Let $A$ and $B$ be comparable TAs and let $R$ be an image-finite backward simulation from $A$ to $B$. Then $\text{traces}_A \subseteq \text{traces}_B$.

Proof: Let $\beta \in \text{traces}_A$. If $\beta$ is closed then Corollary 4.31 implies that $\beta$ is a trace of $B$. From now on we assume $\beta$ is not closed.

Let $\alpha \in \text{execs}_A$ with $\text{trace}(\alpha) = \beta$. Note that any such $\alpha$ is either an infinite sequence $\tau_0 \alpha_1 \tau_1 \ldots$ or a finite sequence $\tau_0 \alpha_1 \tau_1 \ldots \tau_n$ where the final trajectory $\tau_n$ is right open. In either case, using the axioms T1 and T2, we can construct an infinite sequence $\alpha_0 \alpha_1 \ldots$ of closed execution fragments such that $\alpha = \alpha_0 \rhd \alpha_1 \rhd \ldots$ where $\alpha_0$ is a point trajectory, each $\alpha_i$ is either a closed trajectory or an action surrounded by two point trajectories, and $\alpha_i.Istate = \alpha_{i+1}.Istate$ for each $i$, $0 \leq i$.

We construct a directed graph $G$ whose nodes are pairs ($x, i$) consisting of a state of $B$ and an index such that $(\alpha_i.Istate, x) \in R$. In $G$, there is an edge from ($x, i$) to ($x', j$) exactly if $j = i + 1$ and there is an $\alpha' \in \text{frags}_B(x)$ with $\text{trace}(\alpha') = \text{trace}(\alpha_{i+1})$ such that $\alpha'.Istate = x'$. Since $R$ is image-finite there are finitely many roots of $G$. 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 ($x, i$) is reachable from some root node ($z, 0$) for some start state $z$ of $B$.

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 ($x, i$) to ($x', i + 1$) along this infinite path corresponds to a closed execution fragment $\gamma_i+1$ of $B$ for $i$, $0 \leq i$ such that $\gamma_{i+1}.Istate = x$, $\gamma_{i+1}.Istate = x'$ and $\text{trace}(\gamma_{i+1}) = \text{trace}(\alpha_{i+1})$. By Lemma 4.7, $\gamma = \gamma_1 \rhd \gamma_2 \rhd \ldots$ is an execution of $B$ and by Lemma 3.9, $\text{trace}(\gamma) = \text{trace}(\gamma_1) \rhd \text{trace}(\gamma_2) \rhd \ldots$. Since $\text{trace}(\gamma_i+1) = \text{trace}(\alpha_{i+1})$ for all $i$, $0 \leq i$, and $\alpha_0$ is a point trajectory, by Lemma 3.9, we get $\text{trace}(\gamma) = \text{trace}(\alpha) = \beta$. 

44
Example 4.33 (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.

- \( V_A = \{ \text{state}_A \} \) and \( V_B = \{ \text{state}_B \} \) where:
  - \( \text{state}_A \) is a discrete variable with \( \text{type}(\text{state}_A) = \{ x_A, y_A, q_A, s_A \} \), and
  - \( \text{state}_B \) is a discrete variable with \( \text{type}(\text{state}_B) = \{ x_B, y_B, q_B, s_B \} \).

- \( Q_A = \text{val}(V_A) \) and \( Q_B = \text{val}(V_B) \). We write \( x_A \) for the valuation that maps \( \text{state}_A \) to \( x_A \), \( y_A \) for the valuation that maps \( \text{state}_A \) to \( y_A \), etc. Similarly, we write \( x_B \) for the valuation that maps \( \text{state}_B \) to \( x_B \), \( y_B \) for the valuation that maps \( \text{state}_B \) to \( y_B \), etc.

- \( \Theta_A = \{ x_A \} \) and \( \Theta_B = \{ x_B \} \).

- \( E_A = E_B = \{ a, b, c \} \) and \( H_A = H_B = \emptyset \).

- \( D_A = \{ (x_A, a, y_A), (y_A, b, q_A), (y_A, c, s_A) \} \), and
  - \( D_B = \{ (x_B, a, y_B), (x_B, a, y_B'), (y_B, b, q_B), (y_B, c, s_B) \} \).

- \( T_A = \{ \varphi(v) \mid v \in Q_A \} \), and \( T_B = \{ \psi(v) \mid v \in Q_B \} \)

The following are representations of 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.

\[
\begin{align*}
\text{A} & \quad \text{B} \\
\begin{array}{c}
\xrightarrow{a} y_A \\
\xrightarrow{b} q_A \\
\xrightarrow{c} s_A
\end{array} & \quad \begin{array}{c}
\xrightarrow{a} y_B \\
\xrightarrow{b} q_B \\
\xrightarrow{c} s_B
\end{array}
\end{align*}
\]

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

\[
R = \{ (x_A, x_B), (y_A, y_B), (y_A, y_B'), (q_A, q_B), (s_A, s_B) \}.
\]

However, observe that even though \( y_A \) and \( y_B \) are related by \( R \), the execution fragment \( \varphi(y_A) \ c \ \varphi(s_A) \) of \( \mathcal{A} \) cannot be matched by any execution fragment of \( \mathcal{B} \) starting with \( y_B \). Similarly, even though \( y_A \) and \( y_B' \) are related by \( R \), the execution fragment \( \varphi(y_A) \ b \ \varphi(q_A) \) of \( \mathcal{A} \) cannot be matched by any execution fragment of \( \mathcal{B} \) starting with \( y_B' \). 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 $A$ to $B$ and we see that none of them is a forward simulation by examining all the possibilities. The main reason for this is that while $A$ makes the nondeterministic choice between performing $b$ or $c$ after performing $a$, $B$ makes its choice earlier at the same time it performs $a$.

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

\[ \Box \]

### 4.5.4 History Relations

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

An automaton $B$ is obtained from an automaton $A$ by adding history variables if there exists a set of variables $V$ such that

1. $V_B = V_A \cup V$ and $V_A \cap V = \emptyset$,

2. $Q_B \subseteq \text{val}(V_B)$ such that $Q_B \upharpoonright V_A \subseteq Q_A$, and

3. The relation $\{(x, y) \mid y \in Q_B \text{ and } y \upharpoonright V_A = x\}$ is a history relation from $A$ to $B$.

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 higher-level one, it can often be made to exist by adding history variables to the low-level automaton.

**Example 4.34 (Adding history variables to obtain a refinement)** We cannot show that TimedChannel$(b, M)$ is an implementation of TimedChannel$12(b, M)$ 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 TimedChannel$12(b, M)$ when relating the states of the two automata. This example shows how we can add history variables to TimedChannel$(b, M)$ (actually, we add just one variable) to obtain a new automaton that is related to TimedChannel$12(b, M)$ by a refinement.

Let $\log$ be a discrete variable whose static type is the same as the static type of queue in TimedChannel$(Ob, M)$ and let the initial value of $\log$ be the empty sequence. We define a new automaton TimedChannel$1H(b, M)$ whose set of variables consists of the variables of TimedChannel$(b, M)$ and the variable $\log$. The rest of the definition of TimedChannel$1H(b: \text{Real}, M: \text{Type})$ is the same as TimedChannel$(b, M)$ except for the transition definition for receive$(m)$. A receive$(m)$ event in TimedChannel$1H(b, M)$ not only removes the first message from the message queue but also appends this message to the sequence contained in $\log$. 

46
Let $V_1, V_2$ be the set of variables and $Q_1, Q_2$ be the set of states of $\text{TimedChannel}(b, M)$ and $\text{TimedChannelH}(b, M)$ respectively. It is easy to verify that the relation $\{(x, y) \mid y \in Q_2 \text{ and } y \upharpoonright V_1 = x\}$ is a history relation from $\text{TimedChannel}(b, M)$ to $\text{TimedChannelH}(b, M)$. This means that $\text{TimedChannelH}(b, M)$ is obtained from $\text{TimedChannel}(b, M)$ by adding a history variable.

We now define a refinement $F$ from $\text{TimedChannelH}(b, M)$ to $\text{TimedChannel2}(b, M)$ 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 $x$ is a state of $\text{TimedChannelH}(b, M)$ and $y$ is a state of $\text{TimedChannel2}(b, M)$, then $F(x) = y$ where:

1. $y(\text{now}) = x(\text{now})$.
2. $y(\text{queue}) = x(\text{log}) \circ x(\text{queue})$ such that $|x(\text{log})| = y(\text{ptr}) - 1$.

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 4.35 states that the converse also holds, if isomorphic automata are considered.

**Theorem 4.35** 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 history 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 history variables.

**Proof:** Let $R$ be a history relation from $\mathcal{A}$ to $\mathcal{B}$. Define automaton $\mathcal{C}$ as follows:

- $V_\mathcal{C} = V_\mathcal{A} \cup V_\mathcal{B}$.
- $Q_\mathcal{C} = \{ x \in \text{val}(V_\mathcal{C}) \mid (x \upharpoonright V_\mathcal{A}, x \upharpoonright V_\mathcal{B}) \in R \}$.
- $\Theta_\mathcal{C} = \{ x \in Q_\mathcal{C} \mid x \upharpoonright V_\mathcal{B} \in \Theta_\mathcal{B} \}$.
- $E_\mathcal{C} = E_\mathcal{B}$ and $H_\mathcal{C} = H_\mathcal{B}$.
- $x \xrightarrow{a}_\mathcal{C} y$ if and only if $x \upharpoonright V_\mathcal{B} \xrightarrow{a}_\mathcal{B} y \upharpoonright V_\mathcal{B}$.
- $x \xrightarrow{\tau}_\mathcal{C} y$ if and only if $x \upharpoonright V_\mathcal{B} \xrightarrow{\tau}_\mathcal{B} y \upharpoonright V_\mathcal{B}$ where $\tau = \tau \downarrow V_\mathcal{B}$.

Let $F : Q_\mathcal{C} \to Q_\mathcal{B}$ be defined such that $F(x) = x \upharpoonright V_\mathcal{B}$ for all $x \in Q_\mathcal{C}$. The function $F$ is an isomorphism from $\mathcal{C}$ to $\mathcal{B}$: It is easy to check that $F$ is a refinement from $\mathcal{C}$ to $\mathcal{B}$. We can also easily verify that $F^{-1}$ is a refinement from $\mathcal{B}$ to $\mathcal{C}$, by definition of $\mathcal{C}$ and the fact that $R^{-1}$ is a function from the states of $\mathcal{B}$ to the states of $\mathcal{A}$.

Now, we verify that $\mathcal{C}$ is obtained from $\mathcal{A}$ by adding history variables. Let $V_\mathcal{B}$ be the variable set $V$ required in the definition of a history variable and let $R' = \{(x, y) \mid y \in Q_\mathcal{C} \land y \upharpoonright V_\mathcal{A} = x\}$. We need to show that $R'$ is a history relation from $\mathcal{A}$ to $\mathcal{C}$.
1. \( R' \) is a forward simulation from \( A \) to \( C \).
   By definitions of the relations \( F, R' \) and the automaton \( C, R' = F^{-1} \circ R \). Since \( F^{-1} \)
   is a refinement from \( B \) to \( C \), by Theorem 4.27, we know that it is a forward simulation
   from \( B \) to \( C \). Since \( R \) is a forward simulation from \( A \) to \( B \), by Theorem 4.20 we have
   \( R' \) is a forward simulation from \( A \) to \( C \), as needed.

2. \( R'^{-1} \) is a refinement from \( C \) to \( A \).
   By definitions of the relations \( F, R' \) and the automaton \( C, R'^{-1} = R^{-1} \circ F \). Since \( F \)
   is a refinement from \( C \) to \( B \) and \( R^{-1} \) is a refinement from \( B \) to \( A \), by Theorem 4.28,
   we have \( R'^{-1} \) is a refinement from \( C \) to \( A \), as needed.

The following theorem shows that forward simulations are essentially the same as
history relations combined with refinements.

**Theorem 4.36** Let \( A \) and \( B \) be two comparable TAs such that \( V_A \) and \( V_B \) are disjoint.
There is a forward simulation from \( A \) to \( B \) if and only if there exists a TA \( C \) such that
there is a history relation from \( A \) to \( C \) and a refinement from \( C \) to \( B \).

**Proof:** To prove the implication \( \Rightarrow \), suppose \( R \) is a forward simulation from \( A \) to \( B \). Let
\( C \) be an automaton defined as follows:

- \( V_C = V_A \cup V_B \).
- \( Q_C = \{ x \in val(V_C) \mid (x \upharpoonright V_A, x \upharpoonright V_B) \in R \} \).
- \( \Theta_C = \{ x \in Q_C \mid x \upharpoonright V_A \in \Theta_A \land x \upharpoonright V_B \in \Theta_B \} \).
- \( E_C = E_A \) and \( H_C = H_A \).
- \( x \overset{a}{\Rightarrow}_C y \) if and only if both of the following conditions hold:
  1. \( x \upharpoonright V_A \overset{a}{\Rightarrow}_A y \upharpoonright V_A \).
  2. There exists an execution fragment \( \beta \) of \( B \) such that \( \beta.fstate = x \upharpoonright V_B, \beta.Istate = y \upharpoonright V_B \), and \( trace(\beta) = trace(\varphi(x \alpha \varphi(y)) \).
- \( x \overset{\tau}{\Rightarrow}_C y \) if and only if both of the following conditions hold:
  1. \( \tau_1 = \tau \upharpoonright V_A \in T_A \) and \( x \upharpoonright V_A \overset{\tau}{\Rightarrow}_A y \upharpoonright V_A \).
  2. \( \tau_2 = \tau \upharpoonright V_B \in T_B \) and \( x \upharpoonright V_B \overset{\tau}{\Rightarrow}_B y \upharpoonright V_B \).
Let \( \pi_A \) and \( \pi_B \) be the functions that restrict states of \( C \) to, respectively, \( V_A \) and \( V_B \). It follows from the definitions that \( \pi_A^{-1} \) is a history relation from \( A \) to \( C \) and \( \pi_B \) is a refinement from \( C \) to \( B \).

For the implication \( \Leftarrow \), suppose that there is a history relation from \( A \) to \( C \) and that there is a refinement from \( C \) to \( B \). Then, by definition of a history relation, we know that there is a forward simulation from \( A \) to \( C \). We also know that there is a forward simulation from \( C \) to \( B \) by Theorem 4.27. It follows that there is a forward simulation from \( A \) to \( B \), as needed.

Example 4.37 (Theorem 4.36 applied to time-bounded channels) In Example 4.25, we demonstrated a forward simulation from the automaton \( \text{TimedChannel}(b, M) \) to the automaton \( \text{TimedChannel12}(\text{Ob}, M) \). Theorem 4.36 implies that there exists an automaton \( A \) such that there is a history relation from \( \text{TimedChannel}(b, M) \) to \( A \) and a refinement from \( A \) to \( \text{TimedChannel12}(\text{Ob}, M) \). The automaton \( \text{TimedChannel1H}(b, M) \) from Example 4.34 is a witness for \( A \).

4.5.5 Prophecy Relations

A relation \( R \subseteq Q_A \times Q_B \) is a prophecy relation from \( A \) to \( B \) if \( R \) is a backward simulation from \( A \) to \( B \) and \( R^{-1} \) is a refinement from \( B \) to \( A \). Prophecy relations induce a preorder between timed automata.

An automaton \( B \) is obtained from an automaton \( A \) by adding prophecy variables if there exists a set of variables \( V \) such that

1. \( V_B = V_A \cup V \) and \( V_A \cap V = \emptyset \),
2. \( Q_B \subseteq \text{val}(V_B) \) such that \( Q_B | V_A \subseteq Q_A \), and
3. The relation \( \{(x, y) \mid y \in Q_B \text{ and } y \upharpoonright V_A = x\} \) is a prophecy relation from \( A \) to \( B \).

Example 4.38 (Adding prophecy variables to obtain a refinement) In this example we consider adding a prophecy variable to the automaton \( A \) from Example 4.33. Let \( C \) be an automaton defined as follows:

- \( V_C = V_A \cup \{v\} \) where \( v \) is a discrete variable with \( \text{type}(v) = \{b, c\} \).
• $Q_C = \{x_C, x'_C, y_C, y'_C, q_C, s_C\}$ such that
  \[
  x_C \vdash V_A = x_A \text{ and } x_C \vdash \{v\} = b \\
  x'_C \vdash V_A = x_A \text{ and } x'_C \vdash \{v\} = c \\
  y_C \vdash V_A = y_A \text{ and } y_C \vdash \{v\} = b \\
  y'_C \vdash V_A = y_A \text{ and } y'_C \vdash \{v\} = c \\
  q_C \vdash V_A = q_A \text{ and } q_C \vdash \{v\} = b \\
  s_C \vdash V_A = s_A \text{ and } s_C \vdash \{v\} = c 
  \]

• $\Theta_C = \{x_C, x'_C\}$.

• $E_C = \{a, b, c\}$.

• $D_C = \{(x_C, a, y_C), (x'_C, a, y'_C), (y_C, b, q_C), (y'_C, c, s_C)\}$.

• $T_C = \{y_C(v) \mid v \in Q_C\}$.

The relation $R = \{(x_A, x_C), (x_A, x'_C), (y_A, y_C), (y_A, y'_C), (q_A, q_C), (s_A, s_C)\}$ is a backward simulation from $A$ to $C$ and $R^{-1}$ is a refinement. Therefore, $C$ is obtained by adding a prophecy variable to $A$. Note that there is no refinement from $A$ to $B$ defined in Example 4.33. However, the relation $F = \{(x_C, x_B), (x'_C, x_B), (y_C, y_B), (y'_C, y_B), (q_C, q_B), (s_C, s_B)\}$ is a refinement from $C$ to $B$.

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

**Proof:** The proof is analogous to the proof of Theorem 4.35. We assume a backward simulation relation $R$ instead of a forward simulation relation. We construct the automaton $C$ as in Theorem 4.35 and verify that it is obtained from $A$ by adding a prophecy variable.
Theorem 4.40 Let $A$ and $B$ be two comparable TAs such that $V_A$ and $V_B$ are disjoint. There is a backward simulation from $A$ to $B$ if and only if there exists a TA $C$ such that there is a prophecy relation from $A$ to $C$ and a refinement from $C$ to $B$.

Proof: The proof is analogous to the proof of Theorem 4.36. We assume a backward simulation relation $R$ instead of a forward simulation. The construction of the automaton $C$ and the reasoning that follows are similar.

Example 4.41 (Theorem 4.40 applied to Examples 4.33 and 4.38) In Example 4.33, we demonstrated a backward simulation from $A$ to $B$. Theorem 4.40 implies that there exists an automaton $C$ such that there is a prophecy relation from $A$ to $C$ and a refinement from $C$ to $B$. The automaton $C$ defined in Example 4.38 constitutes a witness for $C$.

5 Operations on Timed Automata

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

5.1 Composition

5.1.1 Definitions and Basic Results

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.\footnote{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.}

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

- $X = X_1 \cup X_2$.
- $Q = \{x \in val(X) \mid x \mid X_i \in Q_i, i \in \{1, 2\}\}$. 

51
• \( \Theta = \{ x \in Q \mid x \upharpoonright X_i \in \Theta_i, \ i \in \{1,2\} \} \).

• \( E = E_1 \cup E_2 \) and \( H = H_1 \cup H_2 \).

• For each \( x, x' \in Q \) and each \( a \in A \), \( x \xrightarrow{\sigma} A \ x' \) iff for \( i \in \{1,2\} \), either (1) \( a \in A_i \) and \( x \upharpoonright X_i \xrightarrow{\sigma} i \) \( x' \upharpoonright X_i \), or (2) \( a \not\in A_i \) and \( x \upharpoonright X_i = x' \upharpoonright X_i \).

• \( T \subseteq \text{traj}(X) \) is given by \( \tau \in T \iff \tau \downarrow X_i \in T_i, \ i \in \{1,2\} \).

**Theorem 5.1** If \( A_1 \) and \( A_2 \) are timed automata then \( A_1 \| A_2 \) is a timed automaton.

**Lemma 5.2** Let \( A = A_1 \| A_2 \) and let \( \alpha \) be an execution fragment of \( A \). Then \( \alpha \upharpoonright (A_1, X_1) \) and \( \alpha \upharpoonright (A_2, X_2) \) are execution fragments of \( A_1 \) and \( A_2 \), respectively. Furthermore,

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

**Lemma 5.3** Let \( A = A_1 \| A_2 \), and let \( \alpha \) be an execution fragment of \( A \). Then, for \( i \in \{1,2\} \), \( \text{traj}(\alpha) \upharpoonright (E_i, \emptyset) = \text{traj}(\alpha \upharpoonright (A_i, X_i)) \).

The following theorem is a fundamental theorem that relates the set of traces of a composed automaton to the set 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 \( A = A_1 \| A_2 \). Then \( \text{traces}_A \) is exactly the set of \((E, \emptyset)\)-sequences whose restrictions to \( A_1 \) and \( A_2 \) are traces of \( A_1 \) and \( A_2 \), respectively. That is, \( \text{traces}_A = \{ \beta \mid \beta \text{ is an } (E, \emptyset) \text{-sequence and } \beta \upharpoonright (E_i, \emptyset) \in \text{traces}_{A_i}, i \in \{1,2\} \} \).

**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 \( A \) is the name of an automaton and \( v \) is an internal variable of \( A \), then we refer to this variable as \( A.v \) in the composite automaton.

**Example 5.5** (Periodic sending process with timeouts) Let \( C \) be the composition of three automata from Examples 4.1, 4.2 and 4.4:
$C = \text{PeriodicSend}(u_1, M) \parallel \text{TimedChannel}(b, M) \parallel \text{Timeout}(u_2, M)$

where $M = \{m_1, \ldots, m_n\}$ and $b + u_1 < u_2$. The following sequence is a trace of $C$.

$\alpha = \tau_0 \text{send}(m_1) \tau_1 \text{receive}(m_1) \tau_2 \text{send}(m_2) \tau_3 \text{receive}(m_2) \tau_4 \ldots$

where $e$ is the set consisting of the function with the empty domain and

$\tau_0 : [0, u_1] \rightarrow e \quad \tau_1 : [0, b] \rightarrow e \quad \tau_2 : [0, u_1 - b] \rightarrow e \quad \tau_3 : [0, b] \rightarrow e \quad \tau_4 : [0, u_1 - b] \rightarrow e$

The following invariant states that $C$ never performs a timeout action.

**Invariant 1:** In any reachable state $x$ of $C$, $x(\text{Timeout}.\text{suspected}) = false$.

In order to prove this invariant, we can use an auxiliary invariant such as the one below, which establishes the fact that every message is delivered before the variable now, which keeps track of real-time, reaches the point at which a timeout action occurs.

**Invariant 2:**

1. if $x(\text{TimedChannel}.\text{queue})$ is not empty then
   
   $x(\text{TimedChannel}.\text{queue})[1] < x(\text{TimedChannel}.\text{now} + u_2) - x(\text{Timeout}.\text{clock})$.

2. if $x(\text{TimedChannel}.\text{queue})$ is empty then
   
   $u_1 - x(\text{PeriodicSend}.\text{clock}) + b < u_2 - x(\text{Timeout}.\text{clock})$.

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 $\text{PeriodicSend}(u_1, M)$ is replaced with $\text{PeriodicSend}_2(u_1, M)$. Let $C = \text{PeriodicSend}_2(u_1, M) \parallel \text{TimedChannel}(b, M) \parallel \text{Timeout}(u_2, M)$. The following sequence is a trace of $C$.

$\tau_0 \text{send}(m_1) \tau_1 \text{receive}(m_1) \tau_2 \text{fail} \tau_3 \text{timeout} \tau_4$

where $e$ is the set consisting of the function with the empty domain and

$\tau_0 : [0, u_1] \rightarrow e \quad \tau_1 : [0, b] \rightarrow e \quad \tau_2 : [0, b] \rightarrow e \quad \tau_3 : [0, u_2 - b] \rightarrow e \quad \tau_4 : [0, \infty) \rightarrow e$
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 \( 2 \times b \) time units after sending its first message. The timeout process performs a timeout since no second message arrives within the next \( u_2 \) time units after the receipt of the first message.

The following invariant states that a timeout performed by \( C \) can be used to conclude that the sender process has failed.

**Invariant 1:** Let \( C = \text{PeriodicSend2}(u_1, M) \Vert \text{TimedChannel}(b, M) \Vert \text{Timeout}(u_2, M) \) and assume that \( b + u_1 < u_2 \). In any reachable state \( x \) of \( C \), if \( x(\text{Timeout}.\text{suspected}) = \text{true} \) then \( x(\text{PeriodicSend2}.\text{failed}) = \text{true} \).

The automaton \( C \) 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 \( \alpha \) be an execution of \( C \) and let \( t \) be the point in time at which a fail event occurs in \( \alpha \). Then \( \alpha \) includes a timeout event that occurs in the interval \( (t + b, t + b + u_2) \).

**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 below. The abbreviation \( CS_i \) represents the automaton \( \text{ClockSync}(u, r, i) \). The abbreviation \( TC_{i,j} \) represents the timed channel with maximum delay \( b \) that communicates real-valued messages from \( \text{ClockSync}(u, r, i) \) to \( \text{ClockSync}(u, r, j) \). We assume that the time-bounded channel automata used in this composition are defined as in Example 4.1 where receive and send actions in each instance are renamed such that they can be shared with clock synchronization automata. Let \( C = CS_1 \Vert CS_2 \Vert CS_3 \Vert TC_{1,2} \Vert TC_{2,1} \Vert TC_{1,3} \Vert TC_{3,1} \Vert TC_{2,3} \Vert TC_{3,2} \).
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 \((t \times r)\). This property is stated by the invariant below.

**Invariant 1:** In any reachable state \(x\) of \(C\), at any time \(t \in T\), for any \(i \in \{1, 2, 3\}\),
\[
|\text{x(ClockSync(u, r, i).physClock)} - t| \leq (t \times r).
\]

Two physical clocks in \(C\) 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 \((2 \times t \times r)\). This property is formalized by the following invariant.

**Invariant 2:** In any reachable state \(x\) of \(C\), at any time \(st \in T\), for any \(i, j \in \{1, 2, 3\}\),
\[
|\text{x(ClockSync(u, r, i).physClock)} - \text{x(ClockSync(u, r, j).physClock)}| \leq (2 \times t \times r)\] where \(i, j \in \{1, 2, 3\}\).

The following invariant states that in any reachable state there exists a process \(j\) such that the logical clock of each other 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 \(x\) of \(C\), there exists \(j \in \{1, 2, 3\}\) such that for all \(i \in \{1, 2, 3\}\), \(\text{x(ClockSync(u, r, i).logClock)} \leq \text{x(ClockSync(u, r, j).physClock)}\).

The following invariant states that in any reachable state there exists a process \(j\) such that the logical clock of each other 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 \(x\) of \(C\), there exists \(j \in \{1, 2, 3\}\) such that for all \(i \in \{1, 2, 3\}\), \(\text{x(ClockSync(u, r, i).logClock)} \geq \text{x(ClockSync(u, r, j).physClock)}\).

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 \((t \times r)\) time units.

**Invariant 5:** In any reachable state \(x\) of \(C\), at any time \(t \in T\), for any \(i \in \{1, 2, 3\}\)
\[
|\text{x(ClockSync(u, r, i).logClock)} - t| \leq (t \times r).
\]

Finally, we state a property about the agreement of logical clocks in \(C\).
**Invariant 6:** In any reachable state \( x \) of \( C \), for \( i, j \in \{1, 2, 3\} \),
\[
|\mathbf{x}(\text{ClockSync}(u, r, i).\text{logclock}) - \mathbf{x}(\text{ClockSync}(u, r, j).\text{logclock})| \leq u + (b \times (1 + r)).
\]

To see why Invariant 6 holds, fix \( j \) to be a process with the largest physical clock in \( x \), 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 \( x \). Note that \( v_j \) is also the physical clock value of \( j \) in \( x \). By Invariant 3, we know that \( v_i \leq v_j \). To show Invariant 6, it suffices to show that \( v_j - v_i \leq u + (b \times (1 + r)) \).

Let \( \alpha \) be a finite execution that leads to state \( x \). There are two cases to consider.

1. Some message sent by \( j \) arrives at \( i \) in \( \alpha \).
   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 \( \alpha \), then it sends the next later message when its physical clock has value \( v_1 + u \). By assumption, this message does not arrive at \( i \). Therefore, the real time that elapses after sending it must be at most \( b \). It follows that the physical clock increase of \( j \) since sending this message is at most \( b \times (1 + r) \) and so \( v_j \leq v_1 + u + b \times (1 + r) \). On the other hand, if \( j \) does not send a later message to \( i \) in \( \alpha \), then \( v_j \leq v_1 + u \). In either case, we have \( v_j \leq v_1 + u + b \times (1 + r) \). Since \( v_i \geq v_1 \), we have \( v_j - v_i \leq u + b \times (1 + r) \), as needed for Invariant 6.

2. No message sent by \( j \) arrives at \( i \) in \( \alpha \).
   Since the first send occurs at time 0 and \( sb \) is the largest possible communication delay, the fact that \( i \) has not received the first message sent by \( j \) at time 0 implies that \( t \leq b \). Since both clocks start at 0, we have \( v_j \leq b \times (1 + r) \) and \( v_i \geq 0 \). Therefore, \( v_j - v_i \leq u + b \times (1 + r) \), which suffices for Invariant 6.

\[\blacksquare\]

### 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 \( A_1 \) can be shown to implement another one \( A_2 \), with no assumptions about their environments, then \( A_1 \) can be shown to implement \( A_2 \) in a given environment \( B \).
Theorem 5.8 Suppose $A_1, A_2$ and $B$ are TAs, $A_1$ and $A_2$ have the same external actions, and each of $A_1$ and $A_2$ is compatible with $B$. If $A_1 \leq A_2$ then $A_1 || B \leq A_2 || B$.

Corollary 5.9 Suppose $A_1, A_2, B_1,$ and $B_2$ are TAs, $A_1$ and $A_2$ have the same external actions, $B_1$ and $B_2$ have the same external actions, and each of $A_1$ and $A_2$ is compatible with each of $B_1$ and $B_2$. If $A_1 \leq A_2$ and $B_1 \leq B_2$ then $A_1 || B_1 \leq A_2 || B_2$.

We can strengthen Corollary 5.9 slightly by the following corollary: if $A_1$ implements $A_2$ in an environment $B_2$, then $A_1$ composed with an environment that is more restrictive than $B_2$ (whose set of external behaviors is smaller than that of $B_2$), implements $A_2$ composed with $B_2$.

Corollary 5.10 Suppose $A_1, A_2, B_1,$ and $B_2$ are TAs, $A_1$ and $A_2$ have the same external actions, $B_1$ and $B_2$ have the same external actions, and each of $A_1$ and $A_2$ is compatible with each of $B_1$ and $B_2$. If $A_1 || B_2 \leq A_2 || B_2$ and $B_1 \leq B_2$ then $A_1 || B_1 \leq A_2 || B_2$.

Proof: Let $\beta \in traces_{A_1||B_1}$. By Theorem 5.4, $\beta \models (E_{A_1}, \emptyset) \in traces_{A_1}$ and $\beta \models (E_{B_1}, \emptyset) \in traces_{B_1}$. Since $B_1 \leq B_2$, $\beta \models (E_{B_2}, \emptyset) \in traces_{B_2}$. Since $B_1$ and $B_2$ have the same external actions, it follows that $\beta \models (E_{B_2}, \emptyset) \in traces_{B_2}$. We have $\beta \models (E_{A_1}, \emptyset) \in traces_{A_1}$ and $\beta \models (E_{B_2}, \emptyset) \in traces_{B_2}$. By Theorem 5.4, $\beta \in traces_{A_1||B_2}$. Since $A_1 || B_2 \leq A_2 || B_2$ by assumption, $\beta \in traces_{A_2||B_2}$, as needed.

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

Theorem 5.11 Suppose $A_1$, $A_2$ and $B$ are TAs, $A_1$ and $A_2$ have the same external actions, and each of $A_1$ and $A_2$ is compatible with $B$.

1. If every closed trace of $A_1$ is a trace of $A_2$ then every closed trace of $A_1 || B$ is a trace of $A_2 || B$.

2. If every admissible trace of $A_1$ is a trace of $A_2$ then every admissible trace of $A_1 || B$ is a trace of $A_2 || B$.

3. If every non-Zeno trace of $A_1$ is a trace of $A_2$ then every non-Zeno trace of $A_1 || B$ is a trace of $A_2 || B$.

Example 5.12 (A counterexample for a desirable substitutivity theorem)

Suppose $A_1$ and $A_2$ have the same external actions, $B_1$ and $B_2$ have the same external actions, and that each of $A_1$ and $A_2$ is compatible with each of $B_1$ and $B_2$. If we view
automaton CatchUpA
signature
  external a, b
states
  counta: Nat := 0, countb: Nat := 0,
  now: Real := 0, next: Real := 0
transitions
  external a
    pre
      (counta ≤ countb)
      ∧ (now = next)
    eff
      counta := counta + 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: Real := 0
transitions
  external a
    eff
      counta := counta + 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 9: CatchUpA and CatchUpB

58
\textbf{Figure 10: BoundedAlternateA and BoundedAlternateB}

$A_2$ and $B_2$ as specifications and want to prove that $A_1||B_1 \leq A_2||B_2$, it would be useful to have a theorem that says if $A_1||B_2 \leq A_2||B_2$ and $A_2||B_1 \leq A_2||B_2$ then $A_1||B_1 \leq A_2||B_2$. That is, if $A_1$ implements $A_2$ in the context of $B_2$ and $B_1$ implements $B_2$ in the context of $A_2$, we would like to conclude that $A_1||B_1$ implements $A_2||B_2$. We show by means of a counterexample that it is impossible to prove such a theorem. The problem arises with the infinite behaviors of $A_1||B_2$.

As examples for $A_1, B_1, A_2$, and $B_2$, consider, respectively, the automata \texttt{CatchUpA}, \texttt{CatchUpB}, \texttt{BoundedAlternateA}, \texttt{BoundedAlternateB} in Figures 9 and 10. All automata have the same set of actions, consisting of the external actions a and b. \texttt{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. \texttt{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 their inputs and outputs until a maximum count is reached for outputs, when put in the context of, respectively, BoundedAlternateA and BoundedAlternateB. Hence, \((\text{CatchUpA} \parallel \text{BoundedAlternateB}) \leq (\text{BoundedAlternateA} \parallel \text{BoundedAlternateB})\), and \((\text{BoundedAlternateA} \parallel \text{CatchUpB}) \leq (\text{BoundedAlternateA} \parallel \text{BoundedAlternateB})\). On the other hand, \((\text{CatchUpA} \parallel \text{CatchUpB})\) can perform an infinite sequence of alternating a and b actions, which is not allowed allowed by \((\text{BoundedAlternateA} \parallel \text{BoundedAlternateB})\). Hence, \((\text{CatchUpA} \parallel \text{CatchUpB})\) does not implement \((\text{BoundedAlternateA} \parallel \text{BoundedAlternateB})\).

In Section 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 \(A_2\) and \(B_2\).

### 5.2 Hiding

We define one hiding operation for timed automata, which hides external actions: if \(E \subseteq E_A\), then ActHide\((E, A)\) is the TA \(B\) that is equal to \(A\) except that \(E_B = E_A - E\) and \(H_B = H_A \cup E\).

**Lemma 5.13** If \(E \subseteq E_A\) then ActHide\((E, A)\) is a TA.

**Lemma 5.14** If \(A\) is a TA and \(E \subseteq E_A\) then \(\text{traces}_{\text{ActHide}(E, A)} = \{ \beta \parallel (E_A - E, \emptyset) \mid \beta \in \text{traces}_A \}\).

The following theorem states that the hiding operation respects the implementation relation.

**Theorem 5.15** Suppose \(A\) and \(B\) are TAs with \(A \leq B\), and suppose \(E \subseteq E_A\). Then ActHide\((E, A)\) \leq ActHide\((E, B)\).
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, Modugno, and Tuttle [32] and the phase transition systems of Maller, Manna and Pnueli [31].

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 \( T \) the set of trajectories of \( \mathcal{A} \). We say that \( T \) is well-formed with respect to \( C \) if each \( \tau \in T \) satisfies one of the following conditions:

1. For all \( t \in \text{dom}(\tau) \), \( C \) is enabled in \( \tau(t) \).
2. For all \( t \in \text{dom}(\tau) \), \( C \) is disabled in \( \tau(t) \).
3. There exists \( t \in \text{dom}(\tau) \) such that for all \( t' \in [0, t) \), \( C \) is disabled in \( \tau(t') \) and for all \( t' \in \text{dom}(\tau) - [0, t) \), \( C \) is enabled in \( \tau(t') \).

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

- A timed automaton \( \mathcal{B} = (X, Q, \Theta, E, H, D, T) \).
- A set \( C \subseteq E \cup H \) of actions called a task; we assume that \( T \) is well-formed with respect to \( C \).
- A lower time bound \( l \) and an upper time bound \( u \) for \( C \). We require that the following axioms are satisfied for \( l \) and \( u \):

  \[
  \begin{align*}
  \text{B1} & \quad l \in \mathbb{R}^{\geq 0} \text{ and } u \in \mathbb{R}^{\geq 0} \cup \{\infty\}. \\
  \text{B2} & \quad l \leq u.
  \end{align*}
  \]

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 \textbf{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 $\mathcal{B} = (X, Q, \Theta, E, H, D, T)$. Then \textbf{Extend}(\mathcal{A}) is the TA $\mathcal{A'} = (X', Q', \Theta', E', H', D', T')$ such that the components of $\mathcal{A'}$ consist of:

- $X' = X \cup \{\text{now}, \text{first}, \text{last}\}$ where:
  
  1. \text{now}, \text{first}, and \text{last} are new variables that do not appear in $X$.
  2. \text{now} is an analog variable such that $\text{type(\text{now})} = \mathbb{R}$.
  3. \text{first} and \text{last} are discrete variables where $\text{type(\text{first})} = \mathbb{R}$ and $\text{type(\text{last})} = \mathbb{R} \cup \{\infty\}$.

- $Q' = \{x \in \text{val}(X') \mid x \mid X \in Q\}$.

- $\Theta'$ consists of all the states $x \in Q'$ that satisfy the following conditions:
  
  1. $x \mid X \in \Theta$.
  2. $x(\text{now}) = 0$.
  3. $x(\text{first}) = \begin{cases} \text{l} & \text{if } C \text{ is enabled in } x \mid X, \\ 0 & \text{otherwise}. \end{cases}$
  
  $x(\text{last}) = \begin{cases} \text{u} & \text{if } C \text{ is enabled in } x \mid X, \\ \infty & \text{otherwise}. \end{cases}$

- $E' = E$ and $H' = H$. We write $\mathcal{A'} \triangleq E' \cup H'$.

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

- $T'$ is a set that consists of all $\tau \in \text{trajs}(X')$ that satisfy the following conditions:
  
  1. $(\tau \downarrow X) \in T$. 

62
2. \(d(now) = 1\).

3. (a) If for all \(t \in \text{dom}(\tau)\), \(C\) is enabled in \(\tau \downarrow X(t)\) then \(\text{first}\) and \(\text{last}\) are constant throughout \(\tau\).

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

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

i. \(\text{first}\) and \(\text{last}\) are constant in \([0,t)\).

ii. \(\tau(t)(\text{first}) = \tau(t)(now) + l\) and \(\tau(t)(\text{last}) = \tau(t)(now) + u\).

iii. \(\text{first}\) and \(\text{last}\) are constant in \(\text{dom}(\tau) - [0,t)\).

(d) \(now \leq \text{last}\).

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

Let \(A\) be a TA with bounds \((B, C, l, u)\). In a start state \(x\) of \(\text{Extend}(A)\), the variables \(\text{first}\) and \(\text{last}\) are initialized to \(l\) and \(u\) respectively, if \(C\) is enabled in \(x\). If \(C\) is not enabled in \(x\), then \(\text{first}\) is set to 0 and \(\text{last}\) is set to \(\infty\). Items 3(c) in the definition of \(D'\) and 3(c) in the definition of \(T'\) show how the variables \(\text{first}\) and \(\text{last}\) 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, \(\text{first}\) is set to \(now + l\) and \(\text{last}\) is set to \(now + u\). The variables \(\text{first}\) and \(\text{last}\) are updated similarly when \(C\) becomes newly enabled in the course of a trajectory.

**Theorem 5.16** Suppose that \(A = (B, C, l, u)\) is a TA with bounds. Then \(\text{Extend}(A)\) 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 \(\text{Extend}\). Step 3(a) in the definition of \(D'\) adds a new lower bound constraint, which makes enabling start at some particular time. Step 3(b) in the definition of \(T'\), adds a new upper bound constraint, which stops trajectories at a particular time and which does not add any enabling or disabling to trajectories.

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

**Theorem 5.17** Suppose \(A = (B, C, l, u)\) is a TA with bounds. Then \(\text{traces}_{\text{Extend}(A)} \subseteq \text{traces}_{A}\).
Proof: Let $F : Q' \to Q$ be defined as follows: $F(x) = x \upharpoonright X$ where $X$ is the set of internal variables of $A$. It is easy to check that $F$ is a refinement from $\text{Extend}(A)$ to $A$. By Theorem 4.27 and Corollary 4.23, we conclude that $\text{traces}_{\text{Extend}(A)} \subseteq \text{traces}_A$. \hfill \blacksquare

**Lemma 5.18** Suppose that $A$ is a TA with bounds. For any reachable state $x$ of $\text{Extend}(A)$, if $C$ is enabled in $x \upharpoonright X$ in $A$, then $x(\text{last}) \leq x(\text{now}) + u$.

Proof: Consider a closed execution $\alpha$ of $\text{Extend}(A)$. Using the axioms T1 and T2 for trajectories, we can write $\alpha$ as a concatenation of closed execution fragments $\alpha_0 \ldots \alpha_1 \ldots \alpha_k$ where $\alpha_0$ is a point trajectory, and each $\alpha_i$ for $i \geq 1$ is either a trajectory or a discrete action surrounded by two point trajectories such that for all $0 \leq i \leq k+1$, $\alpha_i.\text{Istate} = \alpha_{i+1}.\text{fstate}$. 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 $\alpha_0.\text{fstate} \upharpoonright X$. Since $\alpha$ is an execution, we know that $\alpha_0.\text{fstate}$ is a start state of $\text{Extend}(A)$. By definition of $\text{Extend}(A)$, $\alpha_0.\text{fstate}(\text{last}) = u$. Since $\alpha_0.\text{fstate}(\text{now}) = 0$, $\alpha_0.\text{fstate}(\text{last}) \leq \alpha_0.\text{fstate}(\text{now}) + u$, as required.

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

1. $\alpha_{k+1}$ is an action $a$ surrounded by two point trajectories. Suppose that $C$ is enabled in $\alpha_{k+1}.\text{Istate}$. There are two subcases to consider:
   
   (a) $C$ is enabled in $\alpha_k.\text{Istate} \upharpoonright X$ and $a \notin C$.
   Then, $\alpha_{k+1}.\text{fstate}(\text{last}) = \alpha_k.\text{fstate}(\text{last})$ and $\alpha_{k+1}.\text{fstate}(\text{now}) = \alpha_k.\text{fstate}(\text{now})$. By inductive hypothesis, $\alpha_k.\text{Istate}(\text{last}) \leq \alpha_k.\text{Istate}(\text{now}) + u$. Therefore, $\alpha_{k+1}.\text{Istate}(\text{last}) \leq \alpha_{k+1}.\text{Istate}(\text{now}) + u$, as needed.
   
   (b) $C$ is disabled in $\alpha_k.\text{Istate} \upharpoonright X$ or $a \in C$.
   Then, by definition of $\text{Extend}(A)$, $\alpha_{k+1}.\text{Istate}(\text{last}) = \alpha_{k+1}.\text{Istate}(\text{now}) + u$, which suffices.

2. $\alpha_{k+1}$ is a trajectory.
   Suppose that $C$ is enabled in $\alpha_{k+1}.\text{Istate} \upharpoonright X$ in $A$. There are two subcases to consider:
   
   (a) $C$ is enabled in $\alpha_{k+1}.\text{fstate} \upharpoonright X$ in $A$.
   By inductive hypothesis $\alpha_{k+1}.\text{fstate}(\text{last}) \leq \alpha_{k+1}.\text{fstate}(\text{now}) + u$. By the well-formedness assumption, we know that $C$ must be enabled throughout $\alpha_{k+1}$ and by definition of $\text{Extend}(A)$ last is constant throughout $\alpha_{k+1}$. Since the value of now increases, it is easy to see that $\alpha_{k+1}.\text{Istate}(\text{last}) \leq \alpha_{k+1}.\text{Istate}(\text{now}) + u$. 

64
(b) $C$ is disabled in $\alpha_{k+1}.fstate \mid X$ in $\mathcal{A}$.

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

\end{itemize}

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

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

1. There does not exist a closed execution fragment $\alpha$ of $\text{Extend}(\mathcal{A})$ from a reachable state, where $\alpha.ltime > u$, $C$ is enabled in $\mathcal{A}$ in all the states of $\alpha \mid (A, X)$ and no action in $C$ occurs in $\alpha$.

2. There does not exist a closed execution fragment $\alpha$ of $\text{Extend}(\mathcal{A})$ from a reachable state, where $\alpha.ltime < l$, such that $C$ is not enabled in $\mathcal{A}$ in the first state of $\alpha \mid (A, X)$ and an action in $C$ occurs in $\alpha$.

\end{theorem}

\begin{proof}

1. Suppose, for the sake of contradiction, that there exists a closed execution fragment $\alpha = \tau_0 a_0 \tau_1 a_2 \ldots \tau_n$ of $\text{Extend}(\mathcal{A})$ from a reachable state, where $\alpha.ltime > u$, $C$ is enabled in $\mathcal{A}$ in all the states of $\alpha \mid (A, X)$ and none of the $a_i$ in $\alpha$ is in $C$. By definition of trajectories for $\text{Extend}(\mathcal{A})$ it must be the case that $\alpha.lstate(\text{now}) \leq \alpha.lstate(\text{last})$.

Since $C$ is enabled in $\mathcal{A}$ in all states in $\alpha$, by Lemma 5.18 we have $\alpha.fstate(\text{last}) \leq \alpha.fstate(\text{now}) + u$. By definition of $\text{Extend}(\mathcal{A})$, $\text{last}$ remains constant throughout $\alpha$; therefore, $\alpha.lstate(\text{last}) = \alpha.fstate(\text{last})$. Since $\alpha.fstate(\text{last}) \leq \alpha.fstate(\text{now}) + u$, it follows that $\alpha.lstate(\text{last}) \leq \alpha.fstate(\text{now}) + u$. By definition of $\alpha$, we have $\alpha.lstate(\text{now}) = \alpha.fstate(\text{now}) + \alpha.ltime$. It follows that $\alpha.fstate(\text{now}) + \alpha.ltime \leq \alpha.fstate(\text{now}) + u$. This implies $\alpha.ltime \leq u$. But this gives us the needed contradiction since $\alpha.ltime > u$.

2. We assume that $\alpha$ is a closed execution fragment of $\text{Extend}(\mathcal{A})$ from a reachable state where $\alpha.ltime < l$, such that $C$ is not enabled in $\mathcal{A}$ in the first state of $\alpha$ and an action in $C$ occurs in $\alpha$. Let $(x, a, x')$ be the first discrete transition of $\text{Extend}(\mathcal{A})$ in $\alpha$ such that $a \in C$. We show that the condition $x(\text{first}) \leq x(\text{now})$, which has to hold for the discrete transition to occur, cannot be true, hence arrive at a contradiction.
By Theorem 5.16, the set of trajectories of $\text{Extend}(A)$ 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 $(x, a, x')$.

(a) $C$ becomes enabled by a discrete transition and remains enabled in $A$ until the occurrence of $(x, a, x')$.

Let $(y, b, y')$ be the discrete transition of $A$ that enables $C$. By item 3(c) in the definition of $D'$ we know that first is set to $y(now) + l$ when $C$ becomes enabled. By item 3(b) in the definition of $D'$ and 3(a) in the definition of $T'$, we know that it remains constant so that $x(first) = y(now) + l$. Since $(x, a, x')$ is a discrete transition of $\text{Extend}(A)$, it must be the case that $x(first) \leq x(now)$. Since $x(now) \leq y(now) + \alpha.ltime$ and $x(first) = y(now) + l$ it follows that $y(now) + l \leq y(now) + \alpha.ltime$. 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 $\tau$ and remains enabled in $A$ until the occurrence of $(x, a, x')$.

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

Example 5.20 (Fischer’s mutual exclusion 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 Figure 11 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 Figure 11 should be viewed as the automaton representing the cumulative result of adding in successive steps two tasks for each index. We assume that $\text{Extend}$ is applied once for each task. That is, we start with the timing-independent version of $\text{FischerME}(\text{uset}, \text{1check}: \text{Real})$, apply $\text{Extend}$ to the automaton augmented with the task $\{\text{set}(i)\}$ to add the lower bound 0 and the upper bound $\text{uset}$, then apply $\text{Extend}$
type PcValue = enumeration of rem, test, set, check,
             leavetry, crit, leaveexit

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

automaton FischerME(u_set, l_check: Real)
  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: Index := nil,
    pc: Array[Index,PcValue] := constant(rem),
    initially u_set >= 0 ∧ l_check >= 0 ∧ u_set < l_check
  transitions
  internal test(i)
    prec
    pc[i] = test
    eff
    if x = p0 then
      pc[i] := set;
      lastset[i]:=now+u_set
  internal set(i)
    prec
    pc[i] = set
    eff
    x := i;
    pc[i] := check;
    lastset[i]:= infty;
    firstcheck[i]:=now + l_check
  internal check(i)
    prec
    pc[i] = check ∧
    now >= firstcheck[i]
    eff
    if x = 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 11: Fischer’s mutual exclusion algorithm with bounds

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 since the result of
the first application of \texttt{Extend} satisfies 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[1] to 0 as an effect of check(i) while we do not reset firstcheck[i] explicitly in 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. ■

6 Timed I/O Automata

In this section we refine the timed automaton model of Section 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 Sections 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 automaton (TIOA) \( \mathcal{A} \) is a tuple \((\mathcal{B}, I, O)\) where

- \( \mathcal{B} = (X, Q, \Theta, E, H, D, T) \) 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; as before we write \( A \triangleq E \cup H \).
- The following additional axioms are satisfied:

\begin{enumerate}
  \item \textbf{E1} (Input action enabling)
    For every \( x \in Q \) and every \( a \in I \), there exists \( x' \in Q \) such that \( x \xrightarrow{a} x' \).
  \item \textbf{E2} (Time-passage enabling)
    For every \( x \in Q \), there exists \( \tau \in T \) such that \( \tau.fstate = x \) and either
    \begin{enumerate}
    \item \( \tau.\text{time} = \infty \), or
    \item \( \tau \) is closed and some \( l \in L \) is enabled in \( \tau.\text{state} \).
    \end{enumerate}
\end{enumerate}

Input action enabling is the input enabling condition of ordinary I/O automata; 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. 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 $A$ by $B_A, I_A, O_A, X_A, Q_A, \Theta_A$, etc., and those of a TIOA $A_i$ by $H_i, I_i, O_i, \ldots, X_i, Q_i, \Theta_i$, etc. We sometimes omit these subscripts, where no confusion is likely. We abuse notation slightly by referring to a TIOA $A$ as a TA when we intend to refer to $B_A$.

**Example 6.1 (TAs viewed as TIOAs)** The automaton TimedChannel(b,M) 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(u,r,1) 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 $\text{physClock} = \text{nextSend}$, 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 $A$ is defined to be an execution fragment, execution, trace fragment, or trace of the underlying TA $B_A$, 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 $A = (B, I, O)$ is defined to be feasible provided that its underlying TA $B$ 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 take 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 \( x \) and each \((I, \emptyset)\)-sequence \( \beta \), there is some execution fragment \( \alpha \) from \( x \) such that \( \alpha \vdash (I, \emptyset) = \beta \). That is, an I/O feasible TIOA accommodates arbitrary input actions occurring at arbitrary times. The given \((I, \emptyset)\)-sequence \( \beta \) 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 \( A \) 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 \( A \) be a progressive TIOA and let \( x \) be a state of \( A \). Since \( A \) is a TIOA it satisfies axiom E2. We construct an admissible execution fragment \( \alpha = \alpha_0 \cap \alpha_1 \cap \alpha_2 \cdots \) from \( x \) as follows.

1. \( \alpha_0 = \phi(x) \).
2. For each \( i > 0 \),
   
   (a) If there exists a trajectory \( \tau \) from \( \alpha_{i-1} \cdot \text{Istate} \) such that \( \tau \cdot \text{Itime} = \infty \) then \( \alpha_i \) is the final execution fragment in the sequence and \( \alpha_i = \tau \).
   
   (b) Otherwise, let \( \tau_i \) be a closed execution fragment from \( \alpha_{i-1} \cdot \text{Istate} \) such that \( l \in L \) is enabled in \( \tau_i \cdot \text{Istate} \). Define \( \alpha_i = \tau_i l \tau_{i+1} \) where \( \tau_{i+1} = \phi(y) \) and \( \tau_i \cdot \text{Istate} \xrightarrow{l} y \).

   The above construction either ends after finitely many stages such that the last trajectory of \( \alpha \) is admissible, or goes through infinitely many stages such that \( \alpha \) contains infinitely many local actions. In the former case, we know that \( \alpha \) is admissible since it ends with an admissible trajectory. In the latter case, since \( A \) is progressive, the fact that \( \alpha \) has infinitely many local actions implies that \( \alpha \) 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 \( A \) be a progressive TIOA, let \( x \) be a state of \( A \), and let \( \tau \in \text{traj}(\emptyset) \). Then there exists an execution fragment \( \alpha \) of \( A \) such that \( \alpha \cdot \text{Istate} = x \) and \( \alpha \vdash (I, \emptyset) = \tau \).
Proof: The result follows from the construction used in the proof of Lemma 6.2. Let \( \alpha \) be an admissible execution fragment from \( x \) constructed as in the proof of Lemma 6.2. Let \( \alpha' \) be a prefix of \( \alpha \) such that \( \alpha' \upharpoonright (\emptyset, \emptyset) = \tau \). Since our construction uses no actions from \( I \), we have \( \alpha' \upharpoonright (I, \emptyset) = \alpha' \upharpoonright (\emptyset, \emptyset) = \tau \), as needed.

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

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

Proof: Let \( A \) be a progressive TIOA, let \( x \) be a state of \( A \), and let \( \beta = \tau_0 a_1 \tau_1 a_2 \tau_2 \ldots \) be an \((I, \emptyset)\)-sequence. We construct a finite or infinite sequence \( a_0 a_1 \ldots \) of execution fragments such that:

1. \( a_0.f\text{state} = x \).
2. For each nonfinal index \( i \), \( a_i.l\text{state} = a_{i+1}.f\text{state} \).
3. For each \( i \), \( (a_0 \cap a_1 \cap \cdots \cap a_i) \upharpoonright (I, \emptyset) = \tau_0 a_1 \tau_1 \ldots \tau_i \).

The construction is carried out recursively. To define \( a_0 \), we start with \( x \) and use Lemma 6.3 to span \( \tau_0 \). For \( i > 0 \), we define \( a_i \) by starting with \( a_{i-1}.l\text{state} \), using axiom E1 to perform the input action \( a_i \) and move to a new state and then using Lemma 6.3 to span \( \tau_i \).

Let \( \alpha = a_0 \cap a_1 \cap \cdots \). By Lemma 3.8, \( \alpha \) is an execution fragment of \( A \) from \( x \) such that \( \alpha \upharpoonright (I, \emptyset) = \beta \), 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 generates 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 [10, 1, 40, 27].
The notion of receptiveness for TIOAs as discussed here is a special case of the same notion for HIOAs [25].

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 $A$ to be a TIOA $A'$ that differs from $A$ only in that $D' \subseteq D$ and $T' \subseteq T$. That is, we require:

- $D' \subseteq D$.
- $T' \subseteq T$.
- $X = X'$, $Q = Q'$, $\Theta = \Theta'$, $E = E'$, $H = H'$, $I = I'$, 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 $x$ of $A$. The fact that the state set $Q'$ of $A'$ is the same as the state set $Q$ of $A$ implies that $A'$ chooses evolutions from every state of $A$.

Notions of strategy have been used also in previous studies of receptiveness [10, 1, 40, 27]. 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 $A'$ is a strategy for $A$, then every execution fragment of $A'$ is also an execution fragment of $A$.

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 is similar to that of the corresponding theorem for HIOAs [25].

**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 $u = now$ to the precondition of the receive action. In this way we enforce that the channel operates maximally slow and messages are only delivered at their delivery deadline. The clock synchronization automaton of Example 4.6 is progressive (and therefore receptive) since it can only generate 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 $A_1$ and $A_2$ are comparable if their inputs and outputs coincide, that is, if $I_1 = I_2$ and $O_1 = O_2$. If $A_1$ and $A_2$ are comparable, then $A_1 \leq A_2$ is defined to mean that the traces of $A_1$ are included among those of $A_2$: $A_1 \leq A_2 \iff \text{traces}_{A_1} \subseteq \text{traces}_{A_2}$.

**Lemma 6.8** Let $A_1$, $A_2$ be two comparable TIOAs and let $B_1$, $B_2$ be, respectively, the underlying TAs for $A_1$ and $A_2$. Then $B_1$ and $B_2$ are comparable and $A_1 \leq A_2$ iff $B_1 \leq B_2$.

**Proof:** Immediate from the definitions.

6.5 Simulation Relations

The definition of forward simulation for TIOAs is the same as for TAs. Formally, if $A_1 = (B_1, I_1, O_1)$ and $A_2 = (B_2, I_2, O_2)$ are two comparable TIOAs, then a forward simulation from $A_1$ to $A_2$ is a forward simulation from $B_1$ to $B_2$.

**Theorem 6.9** If $A_1$ and $A_2$ are comparable TIOAs and there is a forward simulation from $A_1$ to $A_2$, then $A_1 \leq A_2$.

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

7 Operations on Timed I/O Automata

7.1 Composition

In this section, 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 Section 6 and show that the classes of progressive and receptive timed I/O automata 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 on the corresponding definition for TAs, but also takes the input/output structure into account. We say that TIOAs $A_1$ and $A_2$ are compatible if, for $i \neq j$, $X_i \cap X_j = H_i \cap A_j = O_i \cap O_j = \emptyset$. 

73
Lemma 7.1 If $A_1 = (B_1, I_1, O_1)$ and $A_2 = (B_2, I_2, O_2)$ are compatible TIOAs, then $B_1$ and $B_2$ are compatible TAs.

If $A_1$ and $A_2$ are compatible TIOAs then their composition $A_1||A_2$ is defined to be the tuple $A = (B, I, O)$ where

- $B = B_1||B_2,$
- $I = (I_1 \cup I_2) - (O_1 \cup O_2)$
- $O = O_1 \cup O_2.$

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

Theorem 7.2 If $A_1$ and $A_2$ are TIOAs then $A_1||A_2$ is a TIOA.

Proof: The proof is straightforward except for showing that Axiom E2 is satisfied by the composition. Let $x$ be a state of $A_1||A_2$. We need to show the existence of a trajectory from $x$ that satisfies E2.

By definition of $A_1||A_2$, $x \restriction X_1$ is a state of $A_1$ and $x \restriction X_2$ is a state of $A_2$. We know that both $A_1$ and $A_2$ satisfy E2. Let $\tau_1$ be a trajectory of $A_1$ with $\tau_1, fstate = x \restriction X_1$ that satisfies E2, let $\tau_2$ be a trajectory of $A_2$ with $\tau_2, fstate = x \restriction X_2$ that satisfies E2, and consider the following cases:

1. $\tau_1, ltime = \infty$ and $\tau_2, ltime = \infty.$
   Then, define $\tau$ such that $\tau \downarrow X_1 = \tau_1$ and $\tau \downarrow X_2 = \tau_2.$

2. $\tau_1, ltime = \infty$ and $\tau_2$ is closed where some $l \in L_2$ is enabled in $\tau_2, lstate.$
   Then, define $\tau$ such that $\tau \downarrow X_1 = \tau_1 \restriction dom(\tau_2)$ and $\tau \downarrow X_2 = \tau_2.$

3. $\tau_1$ is closed where some $l \in L_1$ is enabled in $\tau_1, lstate$ and $\tau_2, ltime = \infty.$
   Then, define $\tau$ such that $\tau \downarrow X_1 = \tau_1$ and $\tau \downarrow X_2 = \tau_2 \restriction dom(\tau_1).$

4. $\tau_1$ is closed where some $l \in L_1$ is enabled in $\tau_1, lstate$ and $\tau_2$ is closed where some $l \in L_2$ is enabled in $\tau_2, lstate.$
   If $dom(\tau_1) \subseteq dom(\tau_2),$ then define $\tau$ such that $\tau \downarrow X_1 = \tau_1$ and $\tau \downarrow X_2 = \tau_2 \restriction dom(\tau_1).$ Otherwise, define $\tau$ such that $\tau \downarrow X_1 = \tau_1 \restriction dom(\tau_2)$ and $\tau \downarrow X_2 = \tau_2.$

In all the cases, by definition of trajectories for a TIOA, $\tau$ is a trajectory of $A_1||A_2$ from $x$, which satisfies E2 by construction.
Note that this theorem is stronger than the corresponding theorem (Theorem 6.12 in [25]) for general HIOAs. Two HIOAs \( A_1 \) and \( A_2 \) are required to be “strongly compatible” for their composition to be a hybrid I/O automaton. 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 \( A_1 \) and \( A_2 \) be comparable TIOAs, and let \( A = A_1 || A_2 \). Then \( \text{traces}_{A} \) is exactly the set of \((E, \emptyset)\)-sequences whose restrictions to \( A_1 \) and \( A_2 \) are traces of \( A_1 \) and \( A_2 \), respectively. That is, \( \text{traces}_{A} = \{ \beta \mid \beta \text{ is an } (E, \emptyset)\text{-sequence and } \beta \upharpoonright (E_i, \emptyset) \in \text{traces}_{A_i}, i = \{1, 2\} \}. \)

### 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 the input/output distinction does not cause any changes to the substitutivity results we obtained for general TAs.

**Theorem 7.4** Suppose \( A_1 \) and \( A_2 \) are comparable TIOAs with \( A_1 \leq A_2 \). Suppose that \( B \) is a TIOA that is compatible with each of \( A_1 \) and \( A_2 \). Then \( A_1 || B \leq A_2 || B \).

The corollaries below follow from the Corollaries 5.9 and 5.10 of Theorem 5.8.

**Corollary 7.5** Suppose \( A_1, A_2, B_1, \) and \( B_2 \) are TIOAs, \( A_1 \) and \( A_2 \) are comparable, \( B_1 \) and \( B_2 \) are comparable, and each of \( A_1 \) and \( A_2 \) is compatible with each of \( B_1 \) and \( B_2 \). If \( A_1 \leq A_2 \) and \( B_1 \leq B_2 \) then \( A_1 || B_1 \leq A_2 || B_2 \).

**Corollary 7.6** Suppose \( A_1, A_2, B_1, \) and \( B_2 \) are TAs, \( A_1 \) and \( A_2 \) are comparable, \( B_1 \) and \( B_2 \) are comparable, and each of \( A_1 \) and \( A_2 \) is compatible with each of \( B_1 \) and \( B_2 \). If \( A_1 || B_2 \leq A_2 || B_2 \) and \( B_1 \leq B_2 \) then \( A_1 || B_1 \leq A_2 || B_2 \).

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 $A_1$ (resp. $B_1$) satisfies its specification $A_2$ (resp. $B_2$) without using any assumptions about the environment of the component. Assume-guarantee style results such as those presented in [18, 36, 44, 1, 2, 17, 45] 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, which can be used for proving that a system specified as a composite automaton $A_1||B_1$ implements a specification represented by a composite automaton $A_2||B_2$.

The main idea behind Theorem 7.7 is to assume that $A_1$ implements $A_2$ in a context represented by $B_2$, and symmetrically that $B_1$ implements $B_2$ in a context represented by $A_2$ where $A_2$ and $B_2$ are automata whose trace sets are closed under limits. The requirement about limit-closure implies that $A_2$ and $B_2$ specify trace safety properties. Moreover, we assume that the trace sets of $A_2$ and $B_2$ 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 $A_2||B_2$ does not impose stronger constraints on time-passage than $A_1||B_1$. Note that the definitions of limit and time extension of a hybrid sequence can be found in Section 3.4.1.

**Theorem 7.7** Suppose $A_1$, $A_2$, $B_1$, $B_2$ are TIOAs such that $A_1$ and $A_2$ are comparable, $B_1$ and $B_2$ are comparable, and $A_i$ is compatible with $B_i$ for $i \in \{1,2\}$. Suppose further that:

1. The sets $\text{traces}_{A_2}$ and $\text{traces}_{B_2}$ are closed under limits.
2. The sets $\text{traces}_{A_2}$ and $\text{traces}_{B_2}$ are closed under time-extension.
3. $A_1||B_2 \leq A_2||B_2$ and $A_2||B_1 \leq A_2||B_2$.

Then $A_1||B_1 \leq A_2||B_2$.

**Proof:** We first prove by induction on the length of traces of $A_1||B_1$ that every closed trace of $A_1||B_1$ is a trace of $A_2||B_2$.

For the base case, let $\beta$ be a trace of $A_1||B_1$ such that $\beta \in \text{truj}(0)$ (a single trajectory over the empty set of variables). By Axiom TO in the definition of a TA, we know that $A_2$ and $B_2$ have traces $\alpha_1$ and $\alpha_2$ such that $\alpha_1.\text{time} = \alpha_2.\text{time} = 0$. By Assumption 2 we have $\alpha_1 \land \beta \in \text{traces}_{A_2}$ and $\alpha_2 \land \beta \in \text{traces}_{B_2}$. Since, $\alpha_1 \land \beta = \beta$ and $\alpha_2 \land \beta = \beta$, it follows that $\beta \in \text{traces}_{A_2}$ and $\beta \in \text{traces}_{B_2}$. By pasting using Theorem 7.3, $\beta \in \text{traces}_{A_2||B_2}$, as needed.

For the inductive step we consider the following cases:
1. $\beta = \beta' \, a \, \tau$, where $a$ is an output action of $A_1$ and $\tau$ is a point trajectory. 

Then $\beta \, [(E_{A_1}, \emptyset)] \in \text{traces}_{A_1}$ by projection using Theorem 7.3. By inductive hypothesis, $\beta' \, [(E_{B_2}, \emptyset)] \in \text{traces}_{A_2||B_2}$. So $\beta' \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. By projection using Theorem 7.3.

Let $\alpha$ be an execution of $B_2$ such that $\text{trace}(\alpha) = \beta' \, [(E_{B_2}, \emptyset)]$. Since $A_1$ and $B_1$ are compatible TIOAs, $B_1$ and $B_2$ are comparable, and $a$ is an output action of $A_1$, we know that either $a$ is an input action of $B_2$ or the action set of $B_2$ does not contain $a$. In the former case, by the input-enabling axiom (E1) we know that there exists $x'$ such that $(a, lstate, a, x')$ is a discrete transition of $B_2$. It follows that $\beta \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. In the latter case, since $\beta \, [(E_{B_2}, \emptyset)] = \beta' \, [(E_{B_2}, \emptyset)]$ and $\beta' \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$ we get $\beta \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. By pasting using Theorem 7.3, $\beta \in \text{traces}_{A_1 || B_2}$. Then by Assumption 3, $\beta \in \text{traces}_{A_2 || B_2}$. 

2. $\beta = \beta' \, b \, \tau$, where $b$ is an output action of $B_1$ and $\tau$ 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 $A_1$ and $B_1$ and $\tau$ is a point trajectory.

By inductive hypothesis, $\beta' \in \text{traces}_{A_2 || B_2}$. By projection using Theorem 7.3 we get $\beta' \, [(E_{A_2}, \emptyset)] \in \text{traces}_{A_2}$ and $\beta' \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. Let $\alpha$ be an execution of $A_2$ such that $\text{trace}(\alpha) = \beta' \, [(E_{A_2}, \emptyset)]$. Since $A_1$ and $A_2$ are comparable and $a$ is an input action of $A_1$ we know that $a$ is an input action of $A_2$. By the input-enabling axiom (E1) we know that there exists $x'$ such that $(a', lstate, a, x')$ is a discrete transition of $A_2$. It follows that $\beta \, [(E_{A_2}, \emptyset)] \in \text{traces}_{A_2}$. Similarly, let $\alpha'$ be an execution of $B_2$ such that $\text{trace}(\alpha') = \beta' \, [(E_{B_2}, \emptyset)]$. Since $B_1$ and $B_2$ are comparable and $a$ is an input action of $B_1$ we know that $a$ is an input action of $B_2$. By the input-enabling axiom (E1) we know that there exists $y'$ such that $(a', lstate, a, y')$ is a discrete transition of $B_2$. It follows that $\beta \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. By pasting using Theorem 7.3, we get $\beta \in \text{traces}_{A_2 || B_2}$. 

4. $\beta = \beta' \, d \, \tau$, where $d$ is an input action of $A_1$ but not an action of $B_1$ and $\tau$ is a point trajectory.

By inductive hypothesis, $\beta' \in \text{traces}_{A_2 || B_2}$. By projection using Theorem 7.3, we have $\beta' \, [(E_{A_2}, \emptyset)] \in \text{traces}_{A_2}$ and $\beta' \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. Let $\alpha$ be an execution of $A_2$ such that $\text{trace}(\alpha) = \beta' \, [(E_{A_2}, \emptyset)]$. Since $A_1$ and $A_2$ are comparable TIOAs and $a$ is an input action of $A_1$, $a$ must be an input action of $A_2$. By the input-enabling axiom (E1) we know that there exists $x'$ such that $(a, lstate, a, x')$ is a discrete transition of $A_2$. It follows that $\beta \, [(E_{A_2}, \emptyset)] \in \text{traces}_{A_2}$. Since $B_1$ and $B_2$ are comparable and $a$ is not an action of $B_1$, $a$ cannot be an external action of $B_2$. Therefore, $\beta \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. Since $\beta \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$ we get $\beta \, [(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. By pasting using Theorem 7.3, we get $\beta \in \text{traces}_{A_2 || B_2}$. 

5. $\beta = \beta' \, d \tau$, where $d$ is an input action of $B_1$ but not an action of $A_1$ and $\tau$ is a point trajectory.

This case is symmetric with the previous one.
6. $\beta = \beta' \cap \beta''$, where $\beta'$ is a hybrid sequence consisting of a single trajectory $\tau$.

By inductive hypothesis, $\beta' \in \text{traces}_{A_2||B_2}$. By projection using Theorem 7.3, we get $\beta'[(E_{A_2}, \emptyset)] \in \text{traces}_{A_2}$ and $\beta'[(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. By Assumption 2, we have $\beta'[(E_{A_2}, \emptyset)] \cap \beta''[(E_{A_2}, \emptyset)] \in \text{traces}_{A_2}$ and $\beta'[(E_{B_2}, \emptyset)] \cap \beta''[(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$.

Then by pasting using Theorem 7.3, $\beta \in \text{traces}_{A_2||B_2}$, as needed.

We have thus shown that every closed trace of $A_1||B_1$ is a trace of $A_2||B_2$. Now consider any non-closed trace $\beta$ of $A_1||B_1$. This $\beta$ can be written as the limit of a sequence $\beta_1, \beta_2, \cdots$ of closed traces of $A_1||B_1$. By the first part of the proof we know that each $\beta_1 \in \text{traces}_{A_2||B_2}$, and by projection using Theorem 7.3 each $\beta_1[(E_{A_2}, \emptyset)]$ is a closed trace of $A_2$, and $\beta_1[(E_{B_2}, \emptyset)]$ is a closed trace of $B_2$. We know that $\beta'[(E_{A_2}, \emptyset)]$ is the limit of the $\beta_k[(E_{A_2}, \emptyset)]$, and similarly $\beta'[(E_{B_2}, \emptyset)]$ is the limit of the $\beta_k[(E_{B_2}, \emptyset)]$. Since the sets $\text{traces}_{A_2}$ and $\text{traces}_{B_2}$ are limit-closed by Assumption 1, we get $\beta'[(E_{A_2}, \emptyset)] \in \text{traces}_{A_2}$ and $\beta'[(E_{B_2}, \emptyset)] \in \text{traces}_{B_2}$. Finally, by pasting using Theorem 7.3, we get $\beta \in \text{traces}_{A_2||B_2}$.}

Note that automata with FIN and timing-independence (see Section 4.3 for definitions) constitute examples for context automata $A_2$ and $B_2$ 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 below, which can be used in the decomposition of proofs even when $A_2$ and $B_2$ neither admit arbitrary time-passage nor have limit-closed trace sets. The main idea behind this corollary is to assume that $A_1$ implements $A_2$ in a context $B_3$ that is a variant of $B_2$, and symmetrically that $B_1$ implements $B_3$ in a context that is a variant of $A_2$. That is, the correctness of implementation relationship between $A_1$ and $A_2$ does not depend on all the environment constraints, just on those expressed by $B_3$ (symmetrically for $B_1,B_2$, and $A_3$). In order to use this corollary to prove $A_1||B_1 \leq A_2||B_2$ one needs to be able to find appropriate variants of $A_2$ and $B_2$ 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 $A_1$, $A_2$, $A_3$, $B_1$, $B_2$, $B_3$ are TIOAs such that $A_1$, $A_2$, and $A_3$ are comparable, $B_1$, $B_2$, and $B_3$ are comparable, and $A_i$ is compatible with $B_i$ for $i \in \{1,2,3\}$. Suppose further that:

1. The sets $\text{traces}_{A_3}$ and $\text{traces}_{B_3}$ are closed under limits.
2. The sets $\text{traces}_{A_3}$ and $\text{traces}_{B_3}$ are closed under time-extension.
3. $A_2||B_3 \leq A_3||B_3$ and $A_3||B_2 \leq A_3||B_3$.
4. $A_1||B_3 \leq A_2||B_3$ and $A_3||B_1 \leq A_3||B_2$.

78
Then $A_1 || B_1 \leq A_2 || B_2$.

**Proof:** Since $A_2 \leq A_3$ by Assumption 3 and $A_1 || B_3 \leq A_2 || B_3$ by Assumption 4, we get $A_1 || B_3 \leq A_2 || B_3 \leq A_3 || B_3$, by Theorem 7.4. Similarly, we have $A_3 || B_1 \leq A_3 || B_2 \leq A_3 || B_3$. Since $A_1 || B_3 \leq A_3 || B_3$ and $A_3 || B_1 \leq A_3 || B_3$, by using Assumptions 1 and 2, and Theorem 7.7 we have $A_1 || B_1 \leq A_3 || B_3$.

Let $\beta$ be a trace of $A_1 || B_1$. By projection using Theorem 7.3, $\beta[(E_{A_1}, \emptyset) \in \text{traces}_{A_1}$ and $\beta[(E_{B_1}, \emptyset) \in \text{traces}_{B_1}$. Since $A_1 || B_1 \leq A_3 || B_3$, we know that $\beta \in \text{traces}_{A_3 || B_3}$. By projection using Theorem 7.3, $\beta[(E_{A_3}, \emptyset) \in \text{traces}_{A_3}$ and $\beta[(E_{B_3}, \emptyset) \in \text{traces}_{B_3}$. By pasting using Theorem 7.3, we have $\beta \in \text{traces}_{A_3 || B_3}$ and $\beta \in \text{traces}_{A_3 || B_1}$. By Assumption 4, we get $\beta \in \text{traces}_{A_3 || B_3}$ and $\beta \in \text{traces}_{A_3 || B_2}$. Then, by projection using Theorem 7.3, $\beta[(E_{A_2}, \emptyset) \in \text{traces}_{A_2}$ and $\beta[(E_{B_2}, \emptyset) \in \text{traces}_{B_2}$. Finally, by pasting using Theorem 7.3 we have $\beta \in \text{traces}_{A_2 || B_2}$, as needed.

**Example 7.9 (Using environment assumptions to prove safety)** This example illustrates that, in cases where specifications $A_2$ and $B_2$ satisfy certain closure properties, it is possible to decompose the proof of $A_1 || B_1 \leq A_2 || B_2$ by using Theorem 7.7, even if it is not the case that $A_1 \leq A_2$ or $B_1 \leq B_2$.

The automata $\text{AlternateA}$ and $\text{AlternateB}$ in Figure 12 are timing-independent automata in which no consecutive outputs occur without inputs happening in between. $\text{AlternateA}$ and $\text{AlternateB}$ perform a handshake, outputting an alternating sequence of $a$ and $b$ actions when they are composed. The automata $\text{CatchUpA}$ and $\text{CatchUpB}$ in Figure 9 are timing-dependent automata that do not necessarily alternate inputs and outputs as $\text{AlternateA}$ and $\text{AlternateB}$. $\text{CatchUpA}$ can perform an arbitrary number of $b$ actions, and can perform an $a$ provided that $\text{counta} \leq \text{countb}$. It allows $\text{counta}$ to increase to one more than $\text{countb}$. $\text{CatchUpB}$ can perform an arbitrary number of $a$ actions, and can perform a $b$ provided that $\text{counta} \geq \text{countb} + 1$. It allows $\text{countb}$ to reach $\text{counta}$. Timing constraints require each output to occur exactly one time unit after the last action. $\text{CatchUpA}$ and $\text{CatchUpB}$ perform an alternating sequence of $a$ actions and $b$ actions when they are composed.

Suppose that we want to prove that $\text{CatchUpA} || \text{CatchUpB} \leq \text{AlternateA} || \text{AlternateB}$. We cannot apply the basic substitutivity theorem Theorem 7.7, in particular Corollary 7.5, since the assertions $\text{CatchUpA} \leq \text{AlternateA}$ and $\text{CatchUpB} \leq \text{AlternateB}$ are not true. Consider the trace $\tau_0 \cdot b \cdot \tau_1 \cdot a \cdot \tau_2 \cdot a \cdot \tau_3$ of $\text{CatchUpA}$ where $\tau_0$, $\tau_1$, $\tau_2$ and $\tau_3$ are trajectories with limit time 1. After having performed one $b$ and one $a$, $\text{CatchUpA}$ can perform another $a$. But, this is impossible for $\text{AlternateA}$ which needs an input to enable the second $a$. $\text{AlternateA}$ and $\text{CatchUpA}$ behave similarly only when put in a context that imposes alternation.

It is easy to check that $\text{AlternateA}$ and $\text{AlternateB}$ satisfy the closure properties required by Assumptions 1 and 2 of Theorem 7.7 and, hence can be substituted for $A_2$.
automaton AlternateA
  signature
    output a, input b
  states
    myturn: Bool := true
  transitions
    output a
    pre
    myturn
    eff
    myturn := false

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

Figure 12: AlternateA and AlternateB

and $B_2$ respectively. Similarly, we can easily check that Assumption 3 is satisfied if we substitute CatchUpA for $A_1$ and CatchUpB for $B_1$.

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 $A_1 || B_1 \leq A_2 || B_2$, where $A_2$ and $B_2$ do not satisfy the assumptions of Theorem 7.7, then we find appropriate context automata $A_3$ and $B_3$ that abstract from those details of $A_2$ and $B_2$ that are not essential in proving $A_1 || B_1 \leq A_2 || B_2$.

Consider the automata UseOldInputA and UseOldInputB in Figure 13. UseOldInputA keeps track of whether or not it is UseOldInputA's turn, and when it is UseOldInputA's turn, it keeps track of the next time it is supposed to perform an output. The number of outputs that UseOldInputA can perform is bounded by a natural number. In the case
signature
output a, input b

states
myturn: Bool := true, maxout: Nat,
now: Real := 0, next: AugmentedReal := 0

transitions
output a
pre
myturn \land (maxout > 0) \land (now = next)

input b
pre
myturn := true;
if next = infty
then next := now + 1

eff
myturn := false;
maxout := maxout - 1;
next := infty

trajectories
stop when
now = next
evolve
d(now) = 1

signature
input a, output b

states
myturn: Bool := true, maxout: Nat,
now: Real := 0, next: AugmentedReal := 0

transitions
input a
pre
myturn := true;
if next = infty
then next := now + 1

eff
myturn := false;
maxout := maxout - 1;
next := infty

output b
pre
myturn \land (maxout > 0) \land (now = next)

eff
myturn := true;
if next = infty
then next := now + 1

trajectories
stop when
now = next
evolve
d(now) = 1

Figure 13: UseOldInputA and UseOldInputB

81
signature
  output a, input b
states
  myturn: Bool := true, maxout: Nat,
  now: Real := 0, next: AugmentedReal := 0
transitions
  output a
  pre
    myturn \land (maxout > 0)
    \land (now = next)
  eff
    myturn := true;
    next := now + 1
input b
  pre
    myturn := true;
    next := now + 1
  eff
    maxout := maxout - 1;
    next := infty
trajectories
  stop when
    now = next
  evolve
    d(now) = 1

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

Figure 14: UseNewInputA and UseNewInputB
of repeated \( b \) inputs, it is the oldest input that determines when the next output will occur. The automaton \( \text{UseOldInputB} \) is the same as \( \text{UseOldInputA} \) (inputs and outputs reversed) except that the turn variable of \( \text{UseOldInputB} \) is set to false initially. Note that \( \text{UseOldInputA} \) and \( \text{UseOldInputB} \) 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 \( \text{UseOldInputA} \) or \( \text{UseOldInputB} \) since the number of outputs they can perform is bounded by a natural number. The automaton \( \text{UseNewInputA} \) in Figure 14 behaves similarly to \( \text{UseOldInputA} \) 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 \( \text{UseNewInputB} \) in Figure 14 is the same as \( \text{UseNewInputA} \) (inputs and outputs reversed) except that the turn variable of \( \text{UseNewInputB} \) is set to false initially.

Suppose that we want to prove that: \( \text{UseNewInputA} || \text{UseNewInputB} \leq \text{UseOldInputA} || \text{UseOldInputB} \).

Theorem 7.7 is not applicable here because the high-level automata \( \text{UseOldInputA} \) and \( \text{UseOldInputB} \) 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 \( \text{UseOldInputA} \) and \( \text{UseOldInputB} \) but that are restrictive enough to express the constraints that should be satisfied by the environment, for \( \text{UseOldInputA} \) to implement \( \text{UseOldInputA} \) and for \( \text{UseNewInputB} \) to implement \( \text{UseOldInputB} \).

The automata \( \text{AlternateA} \) and \( \text{AlternateB} \) in Figure 12 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 \( \text{UseNewInputA} || \text{UseNewInputB} \) to \( \text{UseOldInputA} || \text{UseOldInputB} \), which is based on the equality of the turn variables of the implementation and the specification automata. The fact that this simulation relation only uses the equality of turn variables reinforces the idea that the auxiliary contexts, which only keep track of their turn, capture exactly what is needed for the proof of \( \text{UseNewInputA} || \text{UseNewInputB} \leq \text{UseOldInputA} || \text{UseOldInputB} \). 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 \( \text{UseOldInputA} \) and \( \text{UseOldInputB} \) 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 $A$ and $B$, where $O_A = I_B = \{a\}$ and $O_B = I_A = \{b\}$. Suppose that $A$ 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, $B$ 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 $A$ and $B$ are individually I/O feasible. We claim that the composition $A||B$ is not I/O feasible. To see this, consider the start state of $A||B$ and the unique input sequence $\beta$ with $\beta.\text{time} = \infty$; $\beta$ simply allows time to pass to infinity. The composition $A||B$ 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 $A_1$ and $A_2$ are compatible progressive TIOAs, then their composition is also progressive.

**Proof:** The proof is similar to the proof of Theorem 7.4 in [25]. The main idea behind the proof is that a Zeno execution of $A_1||A_2$ with infinitely many locally controlled contains infinitely many locally controlled actions of either $A_1$ or $A_2$. Suppose without loss of generality that the automaton that contributes infinitely many locally controlled actions is $A_1$. Then the projection onto $A_1$ violates progressiveness for $A_1$. □

**Theorem 7.13** Let $A_1$ and $A_2$ be two compatible TIOAs with strategies $A'_1$ and $A'_2$, respectively. Then $A'_1 || A'_2$ is a strategy for $A_1 || A_2$.

**Proof:** The proof is similar to the proof of Theorem 7.7 in [25]. □

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 $A_1$ and $A_2$ be two compatible receptive TIOAs with progressive strategies $A'_1$ and $A'_2$, respectively. Then $A_1 || A_2$ is a receptive TIOA with progressive strategy $A'_1 || A'_2$. 
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.

In fact, 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 [25]) 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 $A_1$ and $A_2$ which may not be strongly compatible have strongly compatible strategies $A'_1$ and $A'_2$, then $A_1$ and $A_2$ are also strongly compatible.

7.2 Hiding

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

Lemma 7.16 If $A$ is a TIOA and $O \subseteq O_A$ then $\text{ActHide}(O, A)$ is a TIOA.

Lemma 7.17 If $A$ is a TIOA and $O \subseteq O_A$ then $\text{traces}_{\text{ActHide}(O, A)} = \{ \beta \mid (O_A - O, V_A) \mid \beta \in \text{traces}_A \}$. 

Theorem 7.18 Suppose $A$ and $B$ are TIOAs with $A \leq B$, and suppose $O \subseteq O_A$. Then $\text{ActHide}(O, A) \leq \text{ActHide}(O, B)$.

8 Conclusions and Future Work

In this monograph, we have presented a new framework for describing and analyzing the behavior of timel systems. This framework is a mathematical framework that uses timed I/O automata for the representation of systems. The TIOA framework is a special case of the hybrid I/O automaton modeling framework [25]. We used what we have learned in developing the HIOA framework to revise the earlier work on timed I/O automaton models. Our main motivation was to have a timed I/O automaton 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 timed I/O automata.

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) the 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 [1]. The described project builds upon our prior work on the IOA language [12].

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 [21]. 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 non-trivial 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 [32, 31, 5, 30, 28, 40].

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 FA9550-00-1-0097. Frits Vaandrager was supported by EU IST project IST-2001-3530 Advanced Methods for Timed Systems (AMETIST) and PROGRESS project TES4999: Verification of Hard and Softly Timed Systems (HaST).

3Parts of the TIOA software are available upon request for experimental use. The project web site (http://tioa.csail.mit.edu) contains all the material that are available for general public use.
References


A Notational Conventions

$a, b$ 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
$x$ state
$v$ valuation
$A, B, C$ timed (I/O) automaton
$D$ set of discrete transitions
$T$ set of trajectories
$N$ the natural numbers
$R$ the real numbers
$T$ the time axis
$Z$ the integers
$V$ the universe of variables
$\alpha, \beta, \delta$ $(A, V)$-sequence
$\gamma$ sequence
$\lambda$ the empty sequence
$\pi$ projection function
$\sigma, \rho$ sequence
$\tau, \upsilon$ trajectory
$\Theta$ set of start states
Index

(A, V)-restriction, 18
(A, V)-sequence, 16

abstraction, 5
admissible, 17, 19
algebraic cpo, 11
AlternateA, 79
AlternateB, 79
Alur-Dill timed automaton, 7
analog variable, 13, 22
assume-guarantee, 76

backward simulation, see simulation relation, 45
BoundedAlternateA, 57
BoundedAlternateB, 57

CatchUpA, 57, 79
CatchUpB, 57, 79
chain, 11
clock synchronization, 29, 41
ClockSync(u, r: Real, i: Index), 29, 54, 69
compact element of a cpo, 11
comparable, 73
TA, 37
compatible, 73
TA, 51
complete partial order (cpo), 11
algebraic cpo, 11
compact element, 11
composition, 5, 51, 74
continuous, 11
cpo, see complete partial order
discrete action, 19
discrete transition, 19
discrete variable, 13, 22
dynamic type, 13
effect, 22
enabled, 20

execution, 30, 69
PeriodicSend(u: Real, M: Type), 31
Timeout(b: Real, M: Type), 32
execution fragment, 30, 31, 69
feasible, 36, 69
FIN, see finite internal nondeterminism, 78
finite internal nondeterminism (FIN), 35
Fischer’s mutual exclusion, 26, 33, 66
FischerME(u set, l check: Real), 26
FischerME(u set, l check: Real), 66
forward simulation, see simulation relation
clock synchronization, 41
time-bounded channels, 40

hiding, 60
HIOA, 6, 75
history relation, 46, 47, 73
time-bounded channels, 49
history variable, 46, 47
time-bounded channels, 46
hybrid automaton, 20, 51
Hybrid I/O Automaton modeling framework, 6, 85
hybrid sequence, 16, 16
admissible, 17
closed, 17
concatenation, 17
limit time, 16
prefix, 17
time-bounded, 17
Zeno, 17
HyTech, 8

I/O feasibility, 85
I/O feasible, 70, 84
implementation, 5, 37
invariant, 30
clock agreement, 56
clock validity, 55
ClockSync(u, r: Real, i: Index), 55, 56
failure and timeout, 54
FischerME(uset, Lcheck: Real), 33
TimedChannel(b: Real, M: Type), 33
timeout, 53
isomorphism, 43

Kronos, 8

limit of a chain, 11
linear hybrid automaton, 8
locally Zeno, 69

monotone, 11
non-Zeno, 17, 19

parallel composition, see composition
partial order, 11
complete partial order, 11
periodic sending process, 24, 31
periodic sending process with failures, 24
PeriodicSend(u: Real, M: Type), 24, 52
PeriodicSend2(u: Real, M: Type), 24, 53
point trajectory, see trajectory
precondition, 22
prefix, 10
progressive, 70, 72
prophecy relation, 49, 73
prophecy variable, 49

reachable, 30
receptive, 72, 85
receptiveness, 5, 71, 85
refinement, 42

sequence, 10
simulation relation, 5, 37
backward simulation, 38, 43, 73
forward simulation, 38, 73
refinement, 42
static type, 13
strategy, 71, 72
substitutivity, 56, 57, 75

TA, see timed automaton

TA with bounds, 61
task, 61
lower bound, 61
upper bound, 61
time axis, 12
time interval, 12
closed, 12
left-closed, 12
right-closed, 12
time-bounded channel, 23, 33, 40, 46, 49
timed automaton (TA), 19
timed automaton model, 19
Timed I/O automaton (TIOA), 5, 68
Timed Input/Output Automaton modeling framework, 4
TimedChannel(b: Real, M: Type), 23, 52, 53, 69
Timeout(u: Real, M: Type), 53
Timeout(u: Real, M: Type), 25, 52
timeout process, 25, 32
timing-independent, 36, 78
TIOA, see Timed I/O automaton
trace, 5, 31, 69
PeriodicSend(u: Real, M: Type), 32
Timeout(u: Real, M: Type), 33
trace fragment, 31, 69
trajectory, 14, 19
closed, 14
concatenation, 15, 16
full, 14
limit time, 14
open, 14
point trajectory, 14, 16
prefix, 14

Uppaal, 8
UseNewInputA, 80
UseNewInputB, 80
UseOldInputA, 80
UseOldInputB, 80

variables, 13, 14, 19
analog, 13
discrete, 13
dynamic types, 13
static type, 13

Zeno, 5, 17, 34