The Tempo Language User Guide and Reference Manual

Nancy A. Lynch, Stephen J. Garland, Dilsun Kaynar, Laurent Michel, Alex Shvartsman
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

February 3, 2008

Abstract

Tempo is a simple formal language for modeling distributed systems with (or without) timing constraints, as collections of interacting state machines called Timed Input/Output Automata. Tempo provides natural mathematical notations for describing systems, their properties, and relationships between their descriptions at different levels of abstraction. An associated Tempo Toolkit supports several validation methods for systems described using Tempo, including static analysis, simulation, interactive proof using the PVS theorem-prover, and model-checking using the Uppaal model-checker.

This three-part document consists of: (I) an informal tutorial that describes the underlying mathematical Timed Input/Output Automata framework and demonstrates how to use the Tempo language to model typical timed systems; (II) a systematic description of the Tempo language constructs; and (III) a reference manual containing a complete definition of the Tempo language.

Contents

1 Introduction

Tempo is a simple formal language for modeling distributed systems as collections of interacting state machines called *Timed Input/Output Automata* [2]. Timed Input/Output Automata are often referred to as *Timed I/O Automata*, or just *TIOAs*. The distributed systems in question may have timing constraints, for example, bounds on the time when certain events may occur, or bounds on the rates of change of component clocks. They may use time in significant ways, for example, for timeouts, or for scheduling events to occur periodically. Timed I/O Automata, and the Tempo language, provide good support for describing these constraints and capabilities.

1.1 Timed I/O Automata

The Timed I/O Automata mathematical framework is an extension of the classical I/O Automata framework [6, 7, 4], which was developed many years ago in the theoretical distributed algorithms research community. I/O Automata are very simple interacting asynchronous state machines, without any support for describing timing features. Although they are simple, I/O Automata provide a rich set of capabilities for modeling and analyzing distributed algorithms. I/O Automata support description of many properties that distributed algorithms are required to satisfy, and mathematical proofs that the algorithms in fact satisfy their required properties. These proofs are based on methods such as invariant assertions and compositional reasoning. I/O Automata also support representation of algorithms at different levels of abstraction, and proofs of consistency relationships between algorithm representations at different levels. Because of these capabilities, I/O Automata have been used fairly extensively for modeling and analyzing asynchronous distributed algorithms, and even for proving impossibility results about computability in asynchronous distributed settings.

However, ordinary I/O Automata cannot be used to describe distributed algorithms that use time explicitly, for example, those that use timeouts or schedule events periodically. And they do not provide explicit support for describing timing constraints such as bounds on message delay or clock rates. Moreover, without support for timing, I/O Automata could not be used for other applications such as practical communication protocols. These limitations led to the development of Timed I/O Automata, which include new features—most notably, *trajectories*— specifically designed for describing timing aspects of systems.

Like ordinary I/O Automata, Timed I/O Automata are simple interacting state machines. They have a well-developed, elegant theory, which is presented in a separate monograph [2]. Like I/O Automata, Timed I/O Automata provide a rich set of capabilities for system modeling and analysis. Methods used for analyzing TIOAs are essentially the same as those used for ordinary I/O automata: invariant assertions, compositional reasoning, and correspondences between levels of abstraction. However, all of these methods needed to be modified somewhat from their counterparts for I/O Automata, to take into account the timing of events.

1.2 Intended applications

Distributed algorithms are not the only application domain for which Timed I/O Automata are suited. In fact, Timed I/O Automata can be used to model practically any type of distributed system, including (wired and wireless) communication systems, real-time operating systems, embedded systems, automated process control systems, and even biological systems. The behavior of these systems generally includes both discrete state changes and continuous state evolution; TIOA

is designed to express both kinds of changes.

Many distributed systems involve a combination of computer components and real-world, physical entities such as vehicles, robots, or medical devices. Systems involving interaction between computer and real-world components usually have strong safety, reliability, and predictability requirements, stemming from the requirements of real-world applications. This makes it especially important to have good methods for modeling the systems precisely and analyzing their behavior rigorously. TIOA can be used to model both computer and real-world system components, as well as their interactions. It provides a simple, elegant, and powerful mathematical foundation for analyzing these systems.

1.3 The Tempo language and tools

I/O Automata and Timed I/O Automata are fine mathematical modeling frameworks for distributed systems. They have proved to be tractable for researchers to use, by hand, in describing and analyzing distributed algorithms, communication protocols, and embedded systems. However, computer support could make this type of work quite a bit easier, which is why we have been working on developing the Tempo Language and Toolkit.

The Tempo language provides simple formal notation for describing Timed I/O Automata precisely, based on the pseudocode notation that has been used in many research papers. It also allows specification of properties such as invariant assertions and relationships between automata at different levels of abstraction. The Tempo toolkit contains tools to support analysis of systems described using Tempo. These include lightweight tools, which check syntax and perform static semantic analysis; medium-weight tools, which simulate the action of an automaton and support model-checking using the Uppaal model-checker [3]; and heavyweight tools, which provide support for proving properties of automata using the PVS interactive theorem-prover [9]. The overall architecture of the Tempo toolkit has been designed to facilitate incorporation of other validation tools in the future.

The Tempo language has a rather minimal syntax, which corresponds closely to the simple semantics of the TIOA mathematical framework. In fact, the mapping between a Tempo automaton description and the TIOA that it denotes is pretty transparent. For example, an automaton's discrete transitions and continuous evolutions are described directly in Tempo, by "transitions" and "trajectories", respectively. The minimality of Tempo syntax and the close correspondence between Tempo syntax and TIOA semantics make it easy to analyze systems of TIOAs based directly on Tempo code.

The minimality of the Tempo language does not limit its expressive power: Tempo is capable of describing very general systems of TIOAs. Of course, many analysis tools—especially automated ones like model-checkers—are not capable of handling fully general Tempo programs. Our approach here is to define *sublanguages* of the general Tempo language that are suitable for use with particular tools. This approach contrasts with the usual approach taken by developers of automated tools, which limits the expressive power of the language at the outset. Writing system models in a general language such as Tempo makes it possible to use a variety of tools, both automated and interactive, to assist in validating the models.

The Tempo language is a variant of the earlier IOA language [1], which was designed for use with basic (untimed) I/O Automata. Over many years, MIT students and other researchers produced various tools for IOA, including a translator to the Larch theorem-prover, a simulator, and an automatic generator of distributed code from IOA models. However, these tools were never

engineered professionally for wide use. We hope and intend that the new Tempo tools will be usable by many people, including researchers, teachers, students, and system developers working on many types of distributed systems. Our target application areas include distributed algorithms, communication systems, embedded systems, and process control systems.

1.4 Organization

This document is organized in three parts:

1. An informal tutorial designed to get you familiar with the Tempo language. This part describes the underlying Timed I/O Automata mathematical framework, explains the "philosophy" of Tempo programming, and demonstrates how to use the Tempo language to model typical timed systems. This tutorial features six examples, which illustrate typical applications, including distributed algorithms, communication protocols, and vehicle control. Reading the tutorial should be sufficient for you to begin writing complete Tempo descriptions.

2. A more systematic, detailed description of the Tempo language constructs, including all of the control structures and the current data types.

3. A reference manual containing a complete definition of the syntax and semantics of the Tempo language.

This document draws from various descriptions of IOA [1] and TIOA [2]. It documents the Tempo language itself, but not the tools used to process Tempo programs. For documentation on the Tempo Simulator, see []. For documentation on theorem-proving using Tempo and PVS, see []. Finally, for documentation on model-checking using Tempo and Uppaal, see [].

Part I
Tempo Language Tutorial

2 Tutorial Introduction

Part I of the User Guide and Reference Manual is a tutorial on Tempo programming. The main content of this part is a collection of six examples, which together illustrate most interesting aspects of Tempo programming. We have chosen the examples from the application areas of *distributed algorithms*, *communication protocols*, and *hybrid systems*; hybrid systems are systems that exhibit both interesting discrete behavior and interesting continuous behavior, for example, process-control or vehicle-control systems. We hope this selection of examples will make it easy for people interested in any of these areas to get started writing Tempo programs.

We accompany the examples with discussion about their interesting properties and about their significance to their respective fields. We also use the examples as the basis for a running discussion about Tempo language design choices and usage patterns.

We begin, in Section 3, with a brief review of the underlying Timed I/O Automata model, basically summarizing technical material from [2]. The first three examples are basically toys. First, in Section 4, we present a simple timing-based shared-memory mutual exclusion algorithm designed by Mike Fischer. This example has become a standard initial case study for papers on formal methods for timed systems. It illustrates the use of invariants to prove correctness, in this case, invariants involving time. Next, in Section 5, we consider another toy example, this one involving a race between two tasks; the interesting issue here is the time taken for the tasks to complete their work. This example illustrates the use of abstraction relationships to prove system properties—in this case, time bounds. Then, in Section 6, we describe a simple timeout-based failure-detection system; this example illustrates the use of composition of TIOAs.

The last three examples are a little more complicated, and are designed as introductions to Tempo programming for particular application areas. In Section 7, we present a prototypical distributed algorithm, namely, a leader-election algorithm that uses a separate failure-detection service. Section 8 contains a prototypical protocol for wired communication networks, namely, a Bellman-Ford-style shortest-path-determination algorithm. Finally, Section 9 contains a hybrid system example: a simple vehicle controller.

3 The Timed I/O Automata Mathematical Framework

Tempo is based on the *Timed Input/Output Automata* mathematical framework, as described in the monograph [2]. Timed Input/Output Automata are often referred to as *Timed I/O Automata*, or just *TIOAs*. Here we give a brief introduction; you should refer to the monograph for all the details.

3.1 Timed I/O Automata

A Timed I/O Automaton is a kind of nondeterministic, possibly infinite-state, state machine. Formally, it is a tuple $(X, Q, \Theta, E, H, \mathcal{D}, \mathcal{T})$, where E is further partitioned into $I \cup O$. Here, X is the set of *state variables*, which are regarded as internal to the automaton. The state of a TIOA

is described by a valuation of the state variables, which is, formally, a mapping from X to values for the variables. Q is the set of states (a subset of the set of valuations of X), and Θ is the set of start states.

A TIOA also has two sets of actions: E representing the set of *external actions* and H representing the set of *hidden actions*. The external actions are subdivided into input actions I and output actions O. The external actions are used to describe the TIOA's externally-visible behavior, and in particular, its communications with other TIOAs.

The state of a Timed I/O Automaton can change in two ways: instantaneously by the occurrence of a *discrete transition*, or over time, according to a *trajectory*. D represents the set of discrete transitions; formally, each discrete transition is a $(state, action, state)$ triple. Thus, each discrete transition is labelled by some (internal or external) action. T represents the set of trajectories; formally, each trajectory is a function from a left-closed time interval to valuations of X, which describes the state evolution over the time interval. Trajectories may be continuous or discontinuous functions.

There are a few points worth noting here. First, every variable comes equipped with two *types*, a *static type* and a *dynamic type*. The static type simply describes the set of values that the variable may take on. The dynamic type, on the other hand, describes the allowable ways in which a variable may evolve. For instance, a variable may have static type Real and dynamic type equal to the set of piecewise continuous functions from time intervals to Reals. Dynamic types are used to constrain how the variables may evolve during trajectories.

Second, a TIOA must satisfy a few simple axioms. Most of these are closure properties for the set T of trajectories. Two other axioms describe "enabling" properties for input actions and for time-passage: Basically, a TIOA is not supposed to prevent the occurrence of an input action. And it is not supposed to prevent the passage of time unless it has some action that it wants to take before time is allowed to pass.

This last comment suggests that TIOAs may sometimes prevent the passage of time. If we think of TIOAs as standard *imperative programs*, this may seem somewhat odd—after all, how can a program prevent time from passing? However, if we instead think of TIOAs as *descriptive models* for parts of systems, it is not odd at all. Preventing time-passage is simply a formal device for expressing time bounds. It is useful, for example, for saying that certain events, like the delivery of a message, must happen by a certain time. We will soon see many examples of TIOAs that prevent time-passage as a way of enforcing time bounds.

Third, a TIOA *executes* by performing a sequence of alternating trajectories and discrete transitions, in which the states match up properly.

Finally, sometimes we may want to consider basic untimed automata, for example, to model asynchronous distributed algorithms. We may embed such automata in the TIOA framework by using trivial trajectories, which allow arbitrary amounts of time to pass, without any changes to the variables.[1]

Figure 3.1 illustrates a simple communication channel that can be modeled as a Timed I/O Automaton. Arrows in the figure represent external discrete actions, through which the channel automaton can interact with its environment. The incoming arrow represents the input action,

[1] As a caution to distributed algorithms readers, we note that TIOAs, as presented in [2], do not have facilities for describing liveness properties of the sort that say that some event must "eventually" occur. Such properties are currently outside the scope of the TIOA model and Tempo tools. Earlier drafts of [2], for example, [], do include a preliminary treatment of liveness properties, but more work is needed to fully incorporate this material into the model and language.

send(m), by means of which the environment can inject a message m into the channel. The outgoing arrow represents the output action *receive*(m), by means of which the channel can deliver a message m to its environment.

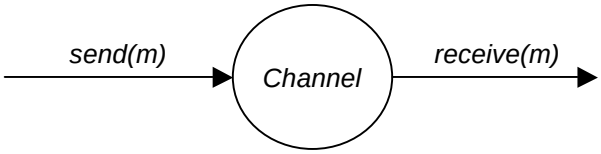

Figure 3.1: A communication channel modeled as a Timed I/O Automaton

Figure 3.1 by itself provides no information about how (or even whether) the channel's output actions are related to its input actions. To supply this missing information, we must specify the state variables, states, start states, hidden actions, discrete transitions, and trajectories. For example, suppose we want to represent a reliable FIFO channel in which every message that is sent is received within time b of when it is sent. Then we might represent the state of the channel in terms of a variable whose static type is a FIFO queue (finite sequence) of pairs of the form $(message, deliverydeadline)$. Initially, the queue is empty. Another useful state variable is a Real-valued variable *now*, which keeps track of the current real time, initially 0. An input action adds a message to the tail of the queue, together with its delivery deadline (as an absolute time, calculated as the current time plus b). An output action removes a message from the head of the queue. No hidden actions are needed.

To express the upper bound on message delivery time, we use the trajectories. The trajectories allow *now* to increase with rate 1, which keeps *now* always equal to the real time, and they require the queue to remain unchanged. Also, trajectories may not continue past the deadline of any message in the queue, that is, time is not allowed to advance beyond any current deadline. This is just another way of saying that, if time does indeed continue to advance, then the messages must be delivered by their deadlines.

If we wanted to describe a FIFO channel without any guarantees of timely message delivery, we could omit the *now* state component and use trivial trajectories that can advance time by any amount, while leaving the queue unchanged. (This description does not guarantee that messages ever get delivered.) Other types of channels may also be represented in this style, for example, channels that may lose, duplicate, or reorder messages.

3.2 Invariants

One of the most important concepts used in stating and proving properties of Timed I/O Automata in that of an *invariant*. An invariant of a TIOA $\mathcal{A}$ is simply a property that is true in every reachable state of $\mathcal{A}$, that is, in every state that can be reached from an initial state of $\mathcal{A}$ by means of an execution of $\mathcal{A}$. For example, one useful invariant for the time-bounded channel described above is the property that, for every message deadline d on the queue, $now \leq d \leq now + b$.

Invariants are typically proved using induction on the number of steps (discrete steps and trajectories) in an execution. This allows us to decompose the task of proving an invariant for a timed system into three separate subtasks: checking that the property is true in the initial state,

checking that it is preserved by every discrete transition, and checking that it is preserved by every trajectory. Checking the initial state is generally straightforward. The interesting work occurs in showing the two "preservation" properties. Here, showing that the invariant is preserved by discrete steps typically uses discrete proof methods like algebraic substitution. On the other hand, showing that the invariant is preserved by trajectories usually involves continuous reasoning, about continuous functions, differential equations, and integrals. The inductive proof structure separates these two kinds of reasoning, allowing them to be carried out separately, perhaps by different people.

3.3 Abstraction

A crucial feature of a system modeling framework is the ability to support *abstraction*. Abstraction refers to encapsulating complex behavior of a system component inside a simpler interface, so that the component can be understood, and used, without knowing the details of what is inside.

For TIOAs, the main abstraction mechanism is the projection of complete executions to external *traces*, which omit all information about state and about hidden actions. What is left is the information about what external (input and output) actions occur during the execution, together with the times at which these actions occur. A trace can be thought of as a sequence of external actions paired with their times of occurrence. Formally, they are presented slightly differently, as *hybrid sequences*, which are alternating trajectories and external actions. Here, the trajectories are trivial in the sense that they map time intervals to valuations of the empty set of variables, that is, to a special distinguished "null valuation". So, the only real information that is captured by such a trivial trajectory is the amount of time that passes. Although expressing traces in this way may seem a little unnatural, it fits more neatly into the general theory of TIOAs. Anyway, if this bothers you, you won't go very wrong by thinking of traces as sequences of external actions paired with their times of occurrence.

The *external behavior* of a TIOA is just the set of traces of all of its executions. For example, in the time-bounded channel described above, the external behavior consists of all sequences of *send* and *receive* actions that represent correct, exactly-once message delivery, each message delivered within time b. The timing is expressed by interspersing the actions with trivial trajectories that indicate how much time passes between the actions.

The TIOA framework includes notions of *implementation* and *simulation*, which can be used to view timed systems at multiple levels of abstraction. In particular, the TIOA framework defines what it means for one TIOA, $\mathcal{A}$, to *implement* another TIOA, $\mathcal{B}$, namely, that the external behavior of $\mathcal{A}$ is a subset of the external behavior of $\mathcal{B}$. That is, every trace exhibited by $\mathcal{A}$ is also allowed by $\mathcal{B}$. For example, our time-bounded channel implements the less constrained untimed channel described above.

The notion of a *simulation relation* from $\mathcal{A}$ to $\mathcal{B}$ provides a sufficient condition for demonstrating that $\mathcal{A}$ implements $\mathcal{B}$. A simulation relation is defined to satisfy three conditions, one relating start states of $\mathcal{A}$ and $\mathcal{B}$, one relating discrete transitions, and one relating trajectories. Simulation relations come in two flavors: *forward simulations* and *backward simulations*. Of the two, forward simulations are far more commonly used, and far easier to use. This is because, in proving a forward simulation, the reasoning about discrete transitions and trajectories follows the normal direction of program execution. Proving a backward simulation requires reasoning about discrete transitions and trajectories in the *reverse* direction from normal program execution. As for invariants, proofs for simulation relations decompose nicely into pieces that require two distinct types of reasoning (discrete vs. continuous).

3.4 Operations on Timed I/O Automata

The most important operation provided for TIOAs is *parallel composition*, by which individual TIOAs can be combined to produce a model for a larger timed system. The model for the composed system describes interactions among the components, which involve joint participation in discrete transitions. All TIOAs in a composition participate in trajectories concurrently, allowing the same amount of time to pass. Composition requires certain "compatibility" conditions, namely, that no output action is an output of more than one automaton, and that no hidden action of any automaton is shared with any other automaton. On the other hand, an output of one automaton is allowed to be an input of any number of other automata, and an input may be shared by any number of automata.

Notice that all communication between TIOAs is by means of discrete actions. TIOA does not provide directly for other forms of communication, such as *shared variable communication*. If you want to use TIOAs to model a system of processes communicating by means of shared variable, you have two options: (1) You can model the entire system of processes plus variables as a single automaton. This is what we do in Section 4, for the Fischer mutual exclusion algorithm. (2) You can model the shared variables as automata, with inputs representing invocations of operations and outputs representing responses. Then the entire system of processes and objects can be modeled as a composition of automata.

TIOA also does not support *continuous communication*, for example, transmission of a continuous signal between two components. The more general *Hybrid I/O Automata* modeling framework [5] allows both discrete and continuous communication among components.

The composition operation respects traces; for example, if $\mathcal{A}_1$ implements $\mathcal{A}_2$ then the composition of $\mathcal{A}_1$ and $\mathcal{B}$ implements the composition of $\mathcal{A}_2$ and $\mathcal{B}$. Composition also satisfies *projection* and *pasting* results, which are fundamental for compositional design and verification of systems: a trace of a composition of TIOAs "projects" to give traces of the individual TIOAs, and traces of components are "pastable" to give traces of the composition.

Finally, the TIOA framework provides a *hiding* operation for TIOAs, by which some output actions become reclassified as hidden. This implies that, when the new automaton is composed with other automata, these newly-hidden actions are no longer available for communication with the other TIOAs.

3.5 Summary

Thus, Timed I/O Automata provide precise representations for timed and untimed systems and their components. They allow descriptions of both discrete and continuous state changes. They enable us to view systems and to reason about them at different levels of abstraction. TIOAs may be composed, interacting through discrete actions only.

So far, we have talked only about the underlying mathematical framework. In the remaining sections of Part I of the tutorial we will introduce the actual Tempo language, via a series of examples.

4 Example 1: Fischer's Timed Mutual Exclusion Algorithm

We are now ready to present our first Tempo code example, the Fischer Timed Mutual Exclusion Algorithm. This simple algorithm has become famous as a standard test example for formal

methods for modeling and analyzing timed systems. It was invented by Mike Fischer, but it does not appear in any publication, unless you count an email to Leslie Lamport as a publication. An informal description of the example appears in [4], Chapter 24.

This example illustrates most of the basic constructs needed for writing a Tempo program for a single Timed I/O Automaton, in this case one modeling a shared-memory system. The example also demonstrates how to express invariants using Tempo, including invariants that involve time.

4.1 Overview of the algorithm

In the Fischer algorithm, a collection of "processes" attempt to arbitrate their entrance to the critical region by accessing a single read/write shared variable called *turn*. The *turn* variable is supposed to indicate whose turn it is to enter the critical region. It can take on a value that is a process name, or else the value *nil* to indicate that no process owns the variable.

Each process i that tries to enter the critical region first reads, or "tests", the *turn* variable, to determine if anyone currently owns it. If so, process i keeps retesting. When it finally sees that the variable is currently un-owned (value = *nil*), process i moves on to the next stage of its program, where it writes its own name i to the *turn* variable. Note that this write step is done separately from the read step that found the variable equal to *nil*; this means that there is some possibility that another process could modify *turn* in between the read and the write step.

After setting *turn* to its own name i, process i rechecks *turn* one more time to see if it is still equal to i. If so, process i proceeds to the critical region; if not, it goes back to the beginning of its program. When process i leaves the critical region, it resets *turn* to *nil*, in another write step.

As described so far, this algorithm admits the possibility that two processes could find themselves in their critical regions simultaneously. An example of an execution that makes this happen is as follows: processes i and j both enter the system, both test *turn*, both find *turn* =*nil*, and both proceed to their next stage. Next, process i sets *turn* to i, then checks and sees that *turn* is still equal to i, and proceeds to the criticial region. Then, process j sets *turn* to j, then checks and sees that *turn* is still equal to j, and proceeds to the critical region. At this point, both processes are in the critical region simultaneously.

The problem is that process i has time to both set and check the *turn* variable during the interval between when process j tests the variable and sets it. This problem can be solved by the simple expedient of imposing an upper bound *last_set* on the time between testing and setting, and a lower bound *first_check* on the time between setting and checking, with *last_set* strictly less than *first_check*. However, it is not completely obvious that this strategy solves the problem, guaranteeing mutual exclusion in all situations. This is why this example is interesting enough to serve as a test case for formal verification methods.

4.2 Tempo description

Figures 1 and 2 contain our Tempo code for the Fischer mutual exclusion algorithm.

The Tempo model describes the entire system as a single Timed I/O Automaton. As we noted in Section 3, TIOA (and Tempo) have no explicit facilities for modeling shared-variable communication. The two options are to treat the shared variables as separate object automata, or to use one big automaton to represent the entire shared memory system. Here we choose the latter option.

```
vocabulary fischer_types
  types process,
  PcValue : Enumeration [pc_rem, pc_test, pc_set, pc_check, pc_leavetry, pc_crit, pc_reset, pc_leaveexit]
end

automaton fischer(l_check, u_set: Real) where u_set < l_check ∧u_set ≥0 ∧l_check ≥0
  imports fischer_types

  signature
    output try(i: process)
    output crit(i: process)
    output exit(i: process)
    output rem(i: process)
    internal test(i: process)
    internal set(i: process)
    internal check(i: process)
    internal reset(i: process)

  states
    turn: Null[process] := nil;
    pc: Array[process, PcValue] := constant(pc_rem);
    now: Real := 0;
    last_set: Array[process, AugmentedReal] := constant(∞);
    first_check: Array[process, DiscreteReal] := constant(0);

  transitions
    output try(i)
      pre pc[i] =pc_rem;
      eff pc[i] := pc_test;

    internal test(i)
      pre pc[i] =pc_test;
      eff if turn =nil then
            pc[i] := pc_set;
            last_set[i] := (now + u_set);
          fi;

    internal set(i)
      pre pc[i] =pc_set;
      eff turn := embed(i);
          pc[i] := pc_check;
          last_set[i] := ∞;
          first_check[i] := now + l_check;
```

Code Sample 1: Tempo description of the Fischer Timed Mutual Exclusion algorithm

```
  internal check(i)
    pre pc[i] =pc_check ∧first_check[i] ≤now;
    eff if turn =embed(i) then
            pc[i] := pc_leavetry;
        else
            pc[i] := pc_test;
        fi;
        first_check[i] := 0;

  output crit(i)
    pre pc[i] =pc_leavetry;
    eff pc[i] := pc_crit;

  output exit(i)
    pre pc[i] =pc_crit;
    eff pc[i] := pc_reset;

  internal reset(i)
    pre pc[i] =pc_reset;
    eff pc[i] := pc_leaveexit;
        turn := nil;

  output rem(i)
    pre pc[i] =pc_leaveexit;
    eff pc[i] := pc_rem;

trajectories
  trajdef traj
    stop when
      ∃i: process (now =last_set[i]);
    evolve
      d(now) =1;
```

Code Sample 2: Tempo description of the Fischer Timed Mutual Exclusion algorithm, continued

The code begins with a declaration of data types used in the algorithm, under the heading **vocabulary**. Here, we define type *process* as an abstract data type. Also, we find it convenient to keep track of where each process is in its program, using explicit program counters; this device is common in modeling shared-memory programs. For this purpose, we define an **Enumeration** data type, *PcValue*, which simply lists the possible valued a process' program counter can take on. There are quite a lot of such values: the process could be in its *remainder region* (program counter = *pc_rem*), where it is not engaged in trying to enter the critical region. Or, it could be about to test, set, or check the *turn* variable. Or, it could be in various stages of entering or leaving the critical region—in the model presented here, we have separate program counter values to represent situations where the process has successfully completed the trying protocol, where it is actually in the critical region, where it is about to reset the *turn* variable upon leaving, and where it has successfully completed the exit protocol. Of course, you could represent less (or more) granularity if you like.

The actual automaton description begins with the name of the automaton, with formal parameters *l_check* and *u_set*. These are real numbers representing, respectively, a lower bound on the time between setting and checking, and an upper bound on the time between checking and setting. To reflect the needed conditions for these parameters, we include a **where** clause, saying (most importantly) that *u_set* must be strictly less than *l_check*. The automaton **imports** the declared types, so that we can use them within the body of the automaton definition.

Next, we have the automaton's **signature**, which describes its actions. Actions are classified as **input**, **output**, or **internal**, although here, we happen not to have any input actions. That is, the system we are considering is "closed". Since the entire system is being modelled by a single automaton, each type of action is parameterized by the name of the process that performs it. Here, the internal actions are associated with shared-variable accesses—the steps that test, set, check, and reset the *turn* variable. The output actions are those that mark processes' progress through the various high-level regions of their code: The *try*(i) action describes process i moving from its remainder region to its *trying region*, in which it executes a protocol to try to reach the critical region. The *crit*(i) action describes passage from the trying region to the critical region, and the *exit*(i) action describes passage from the critical region to the *exit region*, where process i performs its exit protocol. Finally, the *rem*(i) action describes passage from the exit region back to the remainder region.

Next, we have the list of variables that constitute the automaton's state. First we have the *turn* shared variable. Its type is **Null**[*process*], which is a new type that includes the type *process* plus the special value *nil*. In general, **Null** is a type constructor that, given any type not containing the special value *nil*, produces a new type that is the same as the original, with the addition of *nil*. The variable *turn* is set initially (that is, in the initial state of the underlying TIOA) to *nil*.

The next state variable, *pc*, represents the program counters for all of the processes; for this, we use an array of *PcValue* indexed by processes. Initially, all of the program counter values are set to *pc_rem*, which means that all of the processes start out in the remainder region.

The remaining three variables are introduced solely to express the needed timing constraints. First, the variable *now* is used to represent the real time. It is initialized at 0. The use of such a *now* variable is quite common, and convenient, in models for timed systems.

Second, the variable *last_set* is an array containing absolute real time upper bounds (*deadlines*) for the processes to perform **set** actions. Such a deadline will be in force for a process i only when its program counter is equal to *pc_set*, that is, when it is in fact ready to set the *turn* variable. In

this case, the value of *last_set*[*i*] will be a nonnegative real number; otherwise, that is, if the program counter is anything other than *pc_set*, the value will be ∞, representing the absence of any such deadline. The elements of the *last_set* array are defined to be of type*AugmentedReal*, which is a type that includes all (positive and negative) real numbers, plus two values corresponding to positive and negative infinity. Initially, since none of the program counters is *pc_set*, the values in the array are all ∞.

Third and finally, the variable *first_check* is an array containing absolute real time lower bounds (*earliest times*) for the processes to perform *check* actions, when their program counters are equal to *pc_check*. The elements of *first_check* are of type *DiscreteReal*, which means that they always have *Real* values, and moreover, they do not change between discrete actions. We do not need to use the type *AugmentedReal* here, because we will never need to set any of these lower bounds to be positive or negative infinity—the default, when no lower bound is in force, will be zero.

Next, we have the detailed description of the transitions of the automaton. Recall that transitions are (state, action, state) triples. The transitions are described in *guarded command* style, using small pieces of code that we call *transition definitions*. Each transition definition denotes a collection of transitions, all of which share a common action name.

Each transition definition begins with the action name and possible parameters. Next, it has a *precondition*, which is a predicate saying when the action is enabled to occur. And finally, it has an *effects* clause, which describes the discrete changes to the state that accompany the action. Input actions of TIOAs have no preconditions, in general, which reflects the assumption that TIOAs are *input-enabled*. However, this example contains no input actions to illustrate this.

To make the transitions easy to read, we have arranged them according to the typical order in which they should occur during execution. But note that this order is merely suggestive, and has no formal significance: the transitions are allowed to occur in *any* order, as long as their preconditions are satisfied.

We explain the transition definitions briefly, one at a time. First, a *try*(*i*) transition represents an entrance by process *i* into its trying region. This transition is allowed to occur (according to its precondition) whenever *pc*[*i*] =*pc_rem*, that is, whenever process *i* is in its remainder region. The result of this transition is simply to advance the program counter to *pc_test*, indicating that process *i* is ready to test the *turn* variable.

A *test*(*i*) transition represents process *i* testing the *turn* variable. This is allowed to occur whenever *pc*[*i*] =*test*. The *effects* show that two cases may arise: If process *i* finds the *turn* variable equal to *nil*, then it moves to the next stage of its program, which involves setting *turn* to its own index *i*. In this case, to record the needed upper bound on the time until it sets *turn*, the deadline variable *last_set*[*i*] is set to the real time deadline for the **set** action to occur. That deadline is calculated as the current time *now* plus the upper bound *u_set* given as a parameter of the automaton. On the other hand, if process *i* does not find the *turn* variable equal to *nil*, then it remains at the same point of its execution, ready to retest the *turn* variable.

A **set**(*i*) transition represents process *i* setting the *turn* variable to its own index. This is allowed to occur whenever *pc*[*i*] =**set**. In this case, the *effects* are just straight-line code, without any branching. Process *i* simply sets *turn* to its own index; however, because the *turn* variable is of type **Null**[*process*] rather than just *process*, we need to use the *embed* operator to produce a version of index *i* that is of the right type. Then process *i* moves to the next stage of its program, which involves rechecking the *turn* variable. Now that the **set**(*i*) action has occurred, we no longer need the *last_set*[*i*] deadline variable, so that is reset to its default value, ∞. However, we now need

to record the earliest time when process i could recheck the *turn* variable; thus, the earliest-time variable *first_check*[i] is set to the current time *now* plus the lower bound *l_check* given as a parameter of the automaton.

A *check*(i) transition represents process i checking the *turn* variable to verify that it is still equal to its own index i. The precondition here has two parts: first, it says that process i's program counter is set to *check*, as it should be. Second, it says that the current time, *now*, is at least as large as the earliest time at which this action is allowed to occur, as specified in *first_check*[i]. As for the *test* transitions, two interesting cases may arise: If process i finds that *turn* is still equal to i, then it moves to the next stage of its program, which involves leaving the trying region and entering the critical region. On the other hand, if it finds the *turn* variable equal to anything else, then it gives up the current attempt and goes back to the testing step. In either case, it resets the *first_check* earliest-time variable to its default value, 0.

The subsequent transitions are quite straightforward. A *crit*(i) transition represents process i moving into the critical region, and an *exit*(i) transition represents process i leaving the critical region. A *reset*(i) transition represents process i resetting the *turn* variable to its default value *nil*, and a *rem*(i) transition represents process i returning to its remainder region.

The final part of the automaton description is the set of **trajectories**, that is, the functions from time to states that describe how the state is permitted to evolve between discrete steps. Here, we have one kind of trajectory definition, named *traj*. This trajectory definition describes the evolution of the state in a way that allowed the current time *now* to increase at rate 1. All of the other state variables are of types that are defined to be discrete; these, by default, are not allowed to change during trajectories. The other part of the trajectory definition is a *stops* **when** condition, which says that a trajectory must stop if the state ever reaches a point where the current time *now* is equal to a specified deadline *last_set*[i], for any i. That is, time is not "allowed to pass" beyond any deadline currently in force.

This *stops* **when** condition is an example of a phenomenon we discussed in Section 3.1, whereby a TIOA can prevent the passage of time. This may look strange (at first) to some programmers, since programs of course cannot prevent time from passing. However, although the Fischer automaton may look similar to a program, it is not exactly that: it is a *descriptive model* that expresses both the usual sort of behavior expressed by a program, plus additional timing assumptions that might be expressed in other ways.

4.3 Properties of the algorithm

Tempo can be used to describe not just algorithms, but also properties that we would like the algorithms to satisfy. For example, the Fischer algorithm is supposed to satisfy the *mutual exclusion* property, saying that no two processes can simultaneously reside in their critical regions. This is a claim that the mutual exclusion is an *invariant* of the Fischer algorithm, that is, that it is true in all reachable states of the *fischer* TIOA. This claim can be expressed in Tempo as indicated in Figure 3.

```
invariant of fischer:
  ∀i: process ∀j: process
     (i ≠j ⇒(pc[i] ≠pc_crit ∨pc[j] ≠pc_crit));
```

Code Sample 3: Tempo description of the mutual exclusion property

By writing this invariant definition, we are claiming that the mutual exclusion predicate is in fact true in all reachable states. However, just writing such a definition in Tempo (and passing it through the Tempo front end) doesn't imply that the predicate is in fact always true. In order to verify that the predicate is indeed an invariant, we would have to carry out a formal proof, either manually or with the aid of an interactive theorem-prover, such as PVS. We could also check that the invariant holds during selected runs by simulating the protocol and checking the invariant after each simulated step. We could also use a model-checker, such as Uppaal, to check that the invariant holds for all reachable states, at least for special cases of the algorithm having small numbers of processes. All of these tasks are supported by existing Tempo tools.

For example, suppose we want to carry out an interactive proof of the invariant in Figure 3 using PVS. To do this, we will need to define and prove several other *auxiliary invariants*. Specifically, it is useful to know that, when process i is in (or immediately before or immediately after) its critical region, the *turn* variable must be set to i; moreover, no other process j can be about to set the *turn* variable. This property is stated in Figure 4.

```
invariant of fischer:
  ∀i: process ∀j: process
    (pc[i] =pc_leavetry ∨pc[i] =pc_crit ∨pc[i] =pc_reset
        ⇒(turn =embed(i) ∧pc[j] ≠pc_set));
```

Code Sample 4: Properties that hold when a process is in the critical region

Other useful auxiliary invariants involve variables that describe timing aspects of the protocol. Figure 5 contains some particularly simple properties involving time variables. These simply say that the value of the deadline *last_set*[i] is always in the future (no smaller than *now*); this is so whether this variable has a real value or the special value ∞. Moroever, when a process i is about to set the *turn* variable, the value of *last_set*[i] is in fact a real number, not ∞. And in this case it is never very large—it is at most *u_set* in the future, where *u_set* is the upper bound provided for the **set** action.

```
invariant of fischer:
  ∀i: process (now ≤last_set[i]);

invariant of fischer:
  ∀i: process
    (pc[i] =pc_set ⇒last_set[i] ≠∞);

invariant of fischer:
  ∀i: process
    (pc[i] =pc_set ⇒(last_set[i] ≤now + u_set));
```

Code Sample 5: Simple properties involving time

Finally, we have, in Figure 6, the key invariant for understanding why the algorithm works. It says that, if one process i is about to check the *turn* variable in a situation where the check might succeed, and if, at the same time, another process j is about to set the *turn* variable, then the **set** step must happen before the *check* step. This is exactly the condition that is needed to rule out the bad interleaving of steps discussed at the beginning of this section.

invariant of *fischer*:
$\forall i$: *process* $\forall j$: *process*
$(pc[i] = pc\_check \wedge turn = embed(i) \wedge pc[j] = pc\_set$
$\Rightarrow (last\_set[j] < first\_check[i]))$;

Code Sample 6: The key invariant involving time

A formal proof using PVS, using invariants like the ones above, is documented in the separate Tempo Theorem Prover User Guide and Reference Manual []. An informal proof sketch appears in [4], Chapter 24.

4.4 Discussion

The Fischer mutual exclusion example demonstrates how to write a Tempo program for a shared-memory system with timing constraints. Other shared-memory algorithms can be written in a similar way.

As in many shared-memory algorithms, the Fischer algorithm's processes have a rather sequential style; to model them using an essentially concurrent language like Tempo, we needed to define explicit structure (program counters) to keep track of the implicit sequential flow of control. Algorithms with more concurrency, such as typical communication protocols, have less need for such control structure.

We have modeled the entire shared-memory system as a single Timed I/O Automaton. A nice alternative approach, as noted earlier, is to organize the system as a collection of process automata and shared object automata. In this case, the processes and objects interact via *invocation actions* and *response actions*. An invocation action is an output of a process and an input to an object, and represents the invocation of some operation on that object. A response action is an output of an object and an input to a process, and represents the corresponding response. The objects' responses should be consistent with those of actual shared variables. The main advantage of this form of modeling is that it allows us to decompose the system into clearly separated components, using the formal Tempo composition facilities. The main penalty is the need for separate invocation and response steps, that is, the finer granularity of the model. Examples of this type of modeling of shared memory systems appear in [4], Chapter 13.

The Fischer example also shows how timing constraints can be expressed using special time-valued state variables, including a current time variable *now*, deadline variables, and earliest-time variables. These variables can be used just like other state variables, in the statements and proofs of invariants and simulation relations.

5 Example 2: Two-Task Race System

The second example consists of three things: a simple Two Task Race algorithm, a formal specification of the algorithm's desired behavior, and a simulation relation that relates the algorithm to the specification. The Two Task Race algorithm is quite trivial. It involves two tasks, a *main* task and a **set** task. The **set** task simply sets a Boolean flag (once). The *main* task increments a counter until the flag is set, then decrements it, and when the counter reaches zero, reports that it is done. The interesting issues here involve the timing of events: each task comes equipped with upper and

lower bounds on its step time, and the question we ask is *when* the final *report* might happen.

The behavior specification given for this algorithm expresses nothing more than the timing constraints for the *report* event. The simulation relation involves relationships between time-valued variables in the algorithm automaton and the specification automaton. An informal description of the example appears in [4], Chapter 23.

This example illustrates the use of Tempo to describe systems at two levels of abstraction, and to relate two such descriptions using a simulation relation. Furthermore, it shows how timing issues can be incorporated into multi-level system descriptions and simulation relations.

5.1 The algorithm

Tempo code for the Two Task Race algorithm appears in Figure 7. The automaton is named *TTR*, and has four real-valued parameters representing upper and lower bounds for the two tasks. In particular, *a1* and *a2* are (positive) lower and upper bounds for the *main* task, and *b1* and *b2* are (nonnegative) upper and lower bounds for the **set** task. The actions that we consider to be part of the *main* task are the *increment* and *decrement* internal actions, which increment and decrement the counter, respectively, plus the *report* output action, which reports that everything is done. The only action in the **set** task is the **set** internal action.

The state contains three "normal", non-timing-related variables. The variable *count* represents the counter that is manipulated by the *main* task. It is initialized at 0. The variable *flag* is the Boolean flag that gets set by the **set** task. The variable *reported* is another flag indicating whether the final *report* has happened.

In addition to these variables, the state contains five timing-related variables. The first is *now*, which represents the current time as before. The other four, *first_main*, *last_main*, *first_set*, and *last_set*, represent earliest times and deadlines for the two tasks. They are initialized to the lower and upper bounds given as parameters of the automaton. Their use is similar to the earliest-time and deadline variable in the Fischer mutual exclusion algorithm, in Section 4.

The automaton has only four transition definitions. An *increment* transition represents the *main* task incrementing the counter. It is allowed to occur if the *flag* has not been set, and also, if the current time is greater than or equal to the earliest time allowed for the *main* task to take its next step. This earliest time is recorded in the *first_main* variable, so the relevant test here is $now \geq first\_main$. The effect is to increment the *count* variable, and to reset the earliest time and deadline for the *main* task's next step. These are set to the current time plus the given lower and upper bounds for the *main* task.

A **set** transition represents the setting of the *flag* variable by the **set** task. This is allowed to happen if the flag is not yet set, and if the current time is greater than or equal to the earliest time allowed, which is recorded in *first_set*. Its effect is to set the flag, and then reset the earliest-time and deadline variables for the **set** task to their default values. They will not be needed again, and so will retain these default values forever.

The *decrement* transitions are analogous to the *increment* transitions. A *decrement* is allowed to occur if the flag has been set, if the counter is positive, and if the current time is greater than or equal to the earliest time allowed for the *main* task to take its next step. The effect is to decrement the *count* variable, and to reset the earliest time and deadline for the *main* task.

Finally, a *report* transition is allowed to happen if the counter has reached 0 and if the current time is greater than or equal to the earliest time for the *main* task. The two flags are also checked: The *flag* must be equal to *true*, to distinguish the case where the counter has returned to zero from

```
automaton TTR(a1, a2, b1, b2: Real) where
  a1 > 0 ∧a2 > 0 ∧b1 ≥0 ∧b2 ≥0 ∧a2 ≥a1 ∧b2 ≥b1

  signature
    internal increment
    internal decrement
    output report
    internal set

  states
    count: Int := 0;
    flag: Bool := false;
    reported: Bool := false;
    now: Real := 0;
    first_main: DiscreteReal := a1;
    last_main: AugmentedReal := a2;
    first_set: DiscreteReal := b1;
    last_set: AugmentedReal := b2;

  transitions

    internal increment
      pre ¬flag ∧now ≥first_main;
      eff count := count + 1;
          first_main := now + a1;
          last_main := now + a2;

    internal set
      pre ¬flag ∧now ≥first_set;
      eff flag := true;
          first_set := 0;
          last_set := ∞;

    internal decrement
      pre flag ∧count > 0 ∧now ≥first_main;
      eff count := count − 1;
          first_main := now + a1;
          last_main := now + a2;

    output report
      pre flag ∧count =0 ∧¬reported ∧now ≥first_main;
      eff reported := true;
          first_main := 0;
          last_main := ∞;

  trajectories
    trajdef traj
      stop when now =last_main ∨now =last_set;
      evolve
        d(now) =1;
```

Code Sample 7: Tempo description of the Two-Task-Race algorithm

the initial state, where nothing has yet happened. And the *reported* flag must be equal to *false*, to ensure that no *report* has already occurred.

The trajectories here simply allow time to pass at rate 1, but not past the point where either the *last_main* or the *last_set* deadline is reached.

It should be pretty clear that the Two Task Race algorithm results in a single *report* event occurring at some point in time. The interesting question here is *what* point in time. A little thought shows that the *report* occurs latest in situations where the *increment* events happen as quickly as possible, the **set** event happens as late as possible, and the *decrement* events happen as slowly as possible. On the other hand,the *report* occurs earliest in situations where the *increment* events happen as slowly as possible, the **set** happens as early as possible, and the *decrement* events happen as quickly as possible.

Exact calculations of the bounds that arise in these two cases require consideration of messy roundoffs. However, if we allow a little slack in the bounds, we can conclude that a good upper bound on the *report* time is $b2 + (b2 * a2 \,/\, a1) + a2$. Here, the first term, *b2*, describes the latest time when the **set** might occur, and the third term, *a2*, captures the time needed at the end for the *report*. The middle term essentially determines the largest number of increments that might occur (approximately $b2 \,/\, a1$) and then multiplies this number by *a2*, which is the longest time for a decrement.

Similar calculations yield a lower bound of $b1 + (b1 - a2) * a1 \,/\, a2$. Here, the first term describes the earliest time when the **set** might occur. The second term determines the smallest number of increments that might occur (approximately $(b1 - a2)/a2$), and then multiplies this number by *a1*, which is the shortest time for a decrement or report. (We don't have a third term of *a1* because the first decrement could conceivably occur immediately after the **set**.)

5.2 The behavior specification and simulation relation

In many cases, interesting properties of a system can be expressed in terms of its externally-visible behavior. This behavior may include not just *what* happens, but also *when* it happens. For TIOAs, external behavior is captured by external (input and output) actions, together with the times at which they occur. Formally, this external behavior is described by TIOA *traces*.

A useful technique for specifying a set of TIOA traces is by using another TIOA. For the Two Task Race example, the traces that should be specified are exactly those containing a single *report* event, occurring no later than time $b2 + (b2 * a2 \,/\, a1) + a2$, and no earlier than time $b1 + (b1 - a2) * a1 \,/\, a2$.

Figure 8 contains a Tempo description of a simple TIOA whose traces are exactly those that perform a single *report* output, and do so at some time in the interval [*c1*,*c2*]. The specification of interest for the Two Task Race algorithm can then be obtained by instantiating the parameters *c1* and *c2* with $b1 + (b1 - a2) * a1 \,/\, a2$ and $b2 + (b2 * a2 \,/\, a1) + a2$, respectively.

As usual, the bounds are captured by means of earliest-time and deadline variables, *first_report* and *last_report*, respectively. Another point of interest in this code is that *TTRSpec* has two separate trajectory definitions, named *pre_report* and *post_report* for obvious reasons. The **pre**−*report* trajectories are forced to stop if time reaches the *last_report* deadline. The *post_report* trajectories are allowed to continue indefinitely. Alternatively, the two trajectory definitions could be combined into one, with no invariant, a stopping condition that is the same as that for trajectory definition *pre_report*, and the evolves clause $d(now) = 1$. This would work because after the *report* occurs, *last_report* keeps its value at ∞, which means that the stopping condition would never be true.

```
automaton TTRSpec(c1, c2: Real) where c2 ≥0 ∧c2 ≥c1

  signature
    output report

  states
    reported: Bool := false;
    now: Real := 0;
    first_report: DiscreteReal := c1;
    last_report: AugmentedReal := c2;

  transitions

    output report
      pre ¬reported ∧now ≥first_report;
      eff reported := true;
          first_report := 0;
          last_report := ∞;

  trajectories

    trajdef pre_report
      invariant ¬reported;
      stop when now =last_report;
      evolve d(now) =1;

    trajdef post_report
      invariant reported;
      evolve d(now) =1;
```

Code Sample 8: Tempo description of the Two-Task-Race behavior specification

We would like to show that *TTR*(*a1*,*a2*,*b1*,*b2*) implements *TTRSpec*(*c1*,*c2*), where *c1* =*b1*+(*b1*−*a2*)∗*a1*/*a2* and *c2* =*b2* + (*b2* ∗ *a2* /*a1*) + *a2*. We can do this by defining an explicit relation between the states of *TTR* and *TTRSpec*, and proving that it is a *forward simulation* relation, as described in Section 3.

We can define a candidate forward simulation relation in Tempo using the code in Figure 9. This code both defines a mapping between the states of the two automata and asserts that the mapping is in fact a forward simulation. However, as for invariants, more work needs to be done to prove that the mapping is in fact a forward simulation. The code begins with the keyword **forward simulation**, followed by a name for the mapping and a set of parameters. Next, the code specifies the two automata involved in the mapping—here, *ttr* as an instance of the *TTR* automaton and *ttrspec* as an instance of the *TTRSpec* automaton. Notice that the mapping is given a direction, "from" the algorithm automaton and "to" the specification automaton.

Actual parameters for these two automata are specified in terms of the formal parameters of the forward simulation; here, the four parameters of the *TTR* automaton are simply the first four parameters of *F*, whereas the two parameters of *TTRSpec* are calculated using the formulas that we described above. Next, we have a **where** clause, which specifies constraints on the parameters of *F*; these constraints should imply any constraints used in **where** clauses in the definitions of the two component automata (and you can check that they do, in this example). They should also include any new constraints needed for the mapping itself, like the two calculations given here.

Next, we have the mapping itself, described as a predicate involving the state variables of the two automata. To refer to state variables of the two automata, we simply use the declared name, i.e., *ttr* or *ttrspec*. For instance, *ttr.now* refers to the state variable *now* of the *ttr* automaton.

In this example, the mapping begins with two simple equations saying that the *reported* and *now* variables have identical values in the two automata. The remaining four conjuncts of the predicate express relationships between values of the earliest-time and deadline variables of the two automata; the first two of these involve *ttrspec.last_report* and the last two involve *ttrspec.first_report*.

The first of the four conjuncts is an inequality that relates the deadline variable *ttrspec.last_report*—which represents the upper bound we are trying to prove—to deadline variables and earliest-time variables in *ttr*. This conjunct addresses the situation where the *flag* has not yet been set, and moreover, it might not be set until after another *increment* has occurred; this possibility is captured by the non-strict inequality *ttr.first_main* ≤*ttr.last_set*. In this case, we calculate a bound for *report* by considering the latest time when the flag may be set (*ttr.last_set*), calculating the largest possible count at that point, and then adding the longest possible times to decrement the count and perform the final *report*. The largest possible count here is obtained by adding the current count, *ttr.count*, to the largest number of additional increments that can occur. That number is calculated as (*ttr.first_set* − *ttr.last_main*) /*a1*) + 1. The times between successive decrements and between the last decrement and the report are taken to be *a2*.

The second of the four conjuncts again relates *ttrspec.last_report* to deadlines and earliest-time variables in *ttr*. However, this case addresses the simpler situation where the *flag* has either already been set, or else must be set before another *increment* occurs. In this case, we calculate a bound for *report* simply by considering the latest time when all the needed *decrement* actions and the final *report* can occur. This is calculated by considering the latest time when the first *decrement* may occur (*ttr.last_main*), and then considering the additional needed decrements and reports with intervening times of *a2*.

The third conjunct relates the earliest-time variable *ttrspec.first_report* to deadlines and earliest-time variables in *ttr*. It addresses the situation where the *flag* has not yet been set, and moreover,

forward simulation $F(a1,a2,b1,b2,c1,c2\text{: } Real)$
where $a1 > 0 \wedge a2 > 0 \wedge b1 \geq 0 \wedge b2 \geq 0 \wedge c2 \geq 0 \wedge a2 \geq a1$
$\wedge b2 \geq b1 \wedge c2 \geq c1$
$\wedge c1 = b1 + (b1 - a2) * a1/a2$
$\wedge c2 = b2 + (b2 * a2/a1) + a2$
from $ttr : TTR(a1,a2,b1,b2)$
to $ttrspec : TTRSpec(c1,c2)$

mapping

$ttr.reported = ttrspec.reported$
$\wedge ttr.now = ttrspec.now$

$\wedge((\neg ttr.flag \wedge ttr.first\_main \leq ttr.last\_set) \Rightarrow$
$ttrspec.last\_report \geq$
$(Real)(ttr.last\_set) +$
$(ttr.count + 2 + ((Real)(ttr.last\_set) - (Real)(ttr.first\_main)) / a1) * a2)$

$\wedge((\neg ttr.reported \wedge (ttr.flag \vee ttr.first\_main > ttr.last\_set)) \Rightarrow$
$ttrspec.last\_report \geq (Real)(ttr.last\_main) + ttr.count * a2)$

$\wedge((\neg ttr.flag \wedge ttr.last\_main < ttr.first\_set) \Rightarrow$
$ttrspec.first\_report \leq$
$(Real)(ttr.first\_set) +$
$(ttr.count + ((Real)(ttr.first\_set) - (Real)(ttr.last\_main)) / a2) * a1)$

$\wedge((ttr.flag \vee ttr.last\_main \geq ttr.first\_set) \Rightarrow$
$ttrspec.first\_report \leq$
$max(max((Real)(ttr.first\_main), (Real)(ttr.first\_set)), ttr.now) + ttr.count * a1);$

end

Code Sample 9: Forward simulation from Two Task Race algorithm to its specification

it is guaranteed not to be set until after another *increment* has occurred; this guarantee is captured by the strict inequality *ttr.last_main* < *ttr.first_set*. In this case, we calculate a bound for *report* by considering the earliest time when the flag may be set (*ttr.first_set*), calculating the smallest possible count at that point, and then adding the shortest possible times to decrement the count and perform the final *report*. The smallest possible count is obtained by adding the current count, *ttr.count*, to the smallest number of additional increments that can occur. That number is calculated as (*ttr.first_set* − *ttr.last_main*) / *a2*). The times between successive decrements and between the last decrement and the report are taken to be *a1*.

Finally, the fourth conjunct relates *ttrspec.first_report* to deadlines and earliest-time variables in *ttr*, in the situation where the *flag* has either already been set, or may be set before another *increment* occurs. In this case, we calculate a bound for *report* simply by considering the earliest time when all the needed *decrement* actions and the final *report* can occur. This is calculated by considering the earliest time when the first *decrement* may occur (*max*(*ttr.first_main*, *ttr.first_set*, *ttr.now*)), and then considering the additional needed decrements and reports with intervening times of *a1*.

To prove that a given relation is a forward simulation relation, one must show that it relates initial states of the two automata, and is preserved by every discrete transition and every trajectory of the lower-level automaton. Such a proof may be done by hand, or with an interactive theorem-prover. A formal proof for this example, using PVS, is documented in the separate Tempo Theorem Prover User Guide and Reference Manual []. An informal proof sketch appears in [4], Chapter 23.

5.3 Discussion

The Two Task Race example demonstrates how to write Tempo models for an algorithm and its behavior specification. Tempo behavior specifications are used to define external behavior of TIOAs, in terms of their traces. Traces capture both what happens and when it happens. For this example, the main focus is on when the final *report* event occurs.

The Two Task Race example also shows how to describe the connection between an algorithm TIOA and a behavior specification TIOA using a forward simulation relation. Forward simulations relate the states of the two automata, and are used in proving that the algorithm meets its specification. For this example, the simulation relation relates not only logical variables like Boolean flags, but also time-valued variables. This is a common pattern when the algorithm automaton and the specification automaton are time-sensitive.

6 Example 3: Timeout-Based Failure Detector

Our third example is a simple failure-detection system consisting of three automata: a "sender" process that sends periodic messages to announce that it is alive, a "receiver", or "detector" process that receives these messages and notes when the sender appears to have failed, and a timed channel from the sender to the receiver. The system is supposed to guarantee two properties: an *accuracy* property saying that the receiver times out the sender only if the sender has in fact failed, and a *completeness* property saying that, if the sender fails, then the receiver soon times it out.

This example illustrates the use of composition of TIOAs. The timed channel used here is a formal version of one described informally earlier, in Section 3.1.

6.1 The timed channel

The timed channel we consider here is a simple reliable, FIFO channel. It is described as the TIOA *TimedChannel*, in Figure 10.

```
vocabulary Message(M: Type)
        types
                Packet : Tuple[message: M, deadline: Real]
end

automaton TimedChannel(b: Real, M: Type) where b ≥0
        imports Message(M)

        signature
                input send(m:M)
                output receive(m:M)

        states
                queue: Seq[Packet] : = ∅;
                now: Real : = 0;

        transitions

                input send(m)
                        eff queue : = queue ⊢[m,now+b];

                output receive(m)
                        pre queue ≠∅∧head(queue).message =m;
                        eff queue : = tail(queue);

        trajectories
                trajdef traj
                        stop when ∃p: Packet (p ∈ queue ∧now =p.deadline);
                        evolve d(now) =1;
```

Code Sample 10: Tempo description of a timed channel

The vocabulary *Message*(*M*) is parameterized by a message type *M* and defines a *Packet* to be a pair consisting of a message and a deadline. The *TimedChannel* automaton itself takes two parameters, *b*, representing a nonnegative upper bound on the message delay, and the message type *M*. (We do allow the delay bound to be zero; in that case, we are saying that messages must be delivered without any time-passage.) As described informally in Section 3.1, the channel has only one type of input action, *send*, and one type of output action, *receive*. To represent FIFO behavior, we represent the state of the channel using a state variable *queue*, which contains a sequence of packets. Note that each of the packets in the *queue* contains not just the message being sent, but also a deadline giving an upper bound on its delivery time.

A *send*(*m*) transition simply adds the message *m* to the end of *queue*, along with its delivery deadline, calculated as *now* + *b*, where *b* is the delivery bound associated with the channel. A *receive*(*m*) transition is enabled at any point when *m* is the first message in the *queue*—the value of the deadline variable is not relevant here. The effect of the transition is to remove the message

from the head of the queue.

Time passes with rate 1 as usual, and may continue to pass as long as it does not exceed the value of the deadline for any message in the *queue*. Since the queue is FIFO, the *stops* **when** predicate could be stated equivalently as: $queue \neq \emptyset \wedge now = head(queue).deadline$.

6.2 The sender

The sender is described as the TIOA *PeriodicSend*, in Figure 11. This automaton is parameterized by a nonnegative real number u, representing the sending period, as well as the message type M.

```
automaton PeriodicSend(u: Real, M: Type) where u ≥0
    imports Message(M)
    signature
        output send(m:M)
        input fail
    states
        failed: Bool := false;
        clock: Real := 0;

    transitions

        output send(m)
            pre ¬failed ∧clock =u;
            eff clock := 0;

        input fail
            eff failed := true;

    trajectories

        trajdef traj
            stop when ¬failed ∧clock =u;
            evolve d(clock) =1;
```

Code Sample 11: Tempo description of a sending process

The signature allows the automaton to send any message from the alphabet M. The automaton has only one other action, a *fail* input, which represents the failure of the sender, by simply stopping. This action is an input, modeling the notion that a failure may occur at any time, and for any reason.

The state includes only two variables: a Boolean flag *failed*, which records whether the process has failed, and a real-valued *clock*, which measures the elapsed time since the previous *send* transition. The use of this *clock* variable is similar to our use of *now* in the *TimedChannel* and other automata, in that it measures time and increases at rate 1; it is slightly different in that this *clock* variable gets reset to 0 after each send, rather than increasing forever.

A *send*(m) transition is enabled when the clock reaches the value u, provided that the sender has not failed. The effect of the transition is simply to reset the clock to 0. Obviously, we intend that the *send*(m) transition should accomplish something more: it should actually place the message in the channel directed toward the receiver process. However, we do not describe this effect within the

code for the sender. Rather, when the sender is composed with the timed channel, a corresponding *send*(m) transition in the channel will place the message in the channel's message queue. A *fail* transition may occur at any time, and simply sets the *failed* flag to *true*.

The trajectories allow the clock to increase at rate 1. Time is required to stop (in order to permit a message to be sent) when the clock reaches the bound u; however, this stopping requirement holds only if the sender process has not failed. If the process fails, this model says that the clock simply increases forever, but the process does not use it any longer to schedule *send* transitions. Note that, if we had omitted the $\neg$*failed* conjunct, the automaton would force time to stop, but would be unable to perform its *send* transition; thus, it would stop time forever!

6.3 The receiver process

The final piece of the timeout system is the receiver process, which we describe as the TIOA *Timeout* in Figure 12. This automaton is also parameterized by a nonnegative real number called u, as well as the message type M. Here, u represents the length of time the receiver waits before declaring that the sender has failed.

```
automaton Timeout(u: Real, M: Type) where u ≥0
    imports Message(M)

    signature
        input receive(m:M)
        output timeout

    states
        suspected: Bool := false;
        clock: Real := 0;

    transitions

        input receive(m)
            eff clock := 0;
                suspected := false;

        output timeout
            pre clock =u ∧¬suspected;
            eff suspected := true;

    trajectories
        trajdef traj
            stop when clock =u ∧¬suspected;
            evolve d(clock) =1;
```

Code Sample 12: Tempo description of a receiver process for the timeout system

The signature allows the automaton to receive any message in alphabet M, and to output *timeout*. The state includes a Boolean variable *suspected*, which records the receiver's opinion about whether the sender has failed or not. The state also includes a real-valued *clock*, like the one used by the sender.

When a *receive*(m) transition occurs, the automaton ignores the actual message contents, and simply resets the clock. It also sets *suspected* to *false*, indicating that it does not currently suspect that the sender has failed. (In the simple case we are considering, where the sender can fail only once and can never recover, and where all messages are delivered reliably within a predictable time bound, this assignment will turn out to be unnecessary. However, it would make sense to consider the same receiver process in conjunction with more elaborate failure behavior for the sender, or a worse-behaved channel; in such cases, this assignment would play a useful role.)

The receiver process is permitted to output *timeout* if its clock ever reaches u, which means that time u has elapsed since the previous receipt of a message from the sender (or since the beginning of the execution). In that case, the process sets the *suspected* flag to *true*. The precondition of the *timeout* action includes a check that *suspected* is false, as a way of ensuring that the receiver outputs *timeout* only once.

The trajectories allow the clock to increase with rate 1. Time is required to stop (so that the process can output *timeout*) when the clock reaches u; however, this requirement holds only if the receiver has not already output *timeout*, that is, only if *suspected* is currently *false*. Note that, if we had omitted the $\neg$*suspected* conjunct, the automaton could keep outputting *timeout*, but would cause time to stop.

6.4 The complete timeout system

The three components described in the last three subsections can be combined into a single system: formally, the three TIOAs denoted by the three component Tempo programs are *composed* to yield a new TIOA. Figure 13 shows how this combination is specified using Tempo.

automaton *TimeoutSystem*(*u1*,*u2*,*b*: *Real*, *M*: **Type**) **where** $u1 \geq 0 \wedge u2 \geq 0 \wedge b \geq 0 \wedge u2 > (u1 + b)$
    components
        Sender: *PeriodicSend*(*u1*,*M*);
        Detector: *Timeout*(*u2*,*M*);
        Channel: *TimedChannel*(*b*,*M*);

Code Sample 13: Tempo description of a complete timeout system

The description begins by naming the composed system, in this case, *TimeoutSystem*. The system has three nonnegative real-valued parameters, *u1*, *u2*, and *b*, and another parameter *M* representing the message type. A **where** clause describes constraints on the parameter values; here, the reals are all nonnegative, and *u2* is strictly greater than *u1*+*b*.

The description continues with a list of the system components. Each component is given a short name by which it can be referred to within the context of the composition. The short name is followed by the definition of the component, in terms of automata previously defined, with their formal parameters instantiated in terms of the parameters of the composed system. Here, the *TimeoutSystem* consists of the three components *PeriodicSend*(*u1*,*M*), *Timeout*(*u2*,*M*), and *TimedChannel*(*b*,*M*), where $u2 > u1 + b$. The short names are *Sender*, *Detector*, and *Channel*.

We have already mentioned that this composed system is supposed to satisfy two key properties, an *accuracy* property and a *completeness* property. The accuracy property should say that the receiver times out the sender only if the sender has in fact failed. In terms of the given Tempo specifications, this property can be stated as: "For any execution α of *TimeoutSystem*(*u1*,*u2*,*b*),

where $u1 \geq 0 \wedge u2 \geq 0 \wedge b \geq 0 \wedge u2 > (u1 + b)$, if a *timeout* event occurs in α, then it is preceded by a *fail* event."

A good way to prove this accuracy property is to define an auxiliary invariant, saying that, if the detector suspects that the sender has failed, then in fact it has failed. Formally, we can write this as $Detector.suspected \Rightarrow Sender.failed$. Notice that we use the short names for the components here to indicate whose state variables we are talking about. It should not be hard to see that this invariant implies the accuracy property we really want.

To prove this invariant, we can use other auxiliary invariants, for instance, the one in Figure 14. This says that, if the sender has not failed, then a message is on the way to the detector and will get there in time to prevent the detector from timing out the sender. The statement uses two cases: If the queue is nonempty, then the (absolute) time by which some message in the queue will be received by the detector is strictly before the time at which the detector would perform a *timeout*. The latter time is calculated as the current time plus the remaining time left until the timeout, which is just the difference $u2 - Detector.clock$. On the other hand, if the queue is empty, then the absolute time by which some message will be sent by the sender and subsequently received by the detector is strictly before the time at which the detector would perform a *timeout*. The time when the sender will send the next message is the current time plus the remaining time left until the sender's clock reaches $u1$, which is the difference $u1 - Sender.clock$. Then the time for that message to be delivered is at most an additional b.

invariant of *TimeoutSystem*:

$$\neg Sender.failed \Rightarrow$$
$$((Channel.queue \neq \emptyset \Rightarrow (head(Channel.queue).deadline < Channel.now + u2 - Detector.clock))$$
$$\wedge$$
$$(Channel.queue = \emptyset \Rightarrow Channel.now + u1 - Sender.clock + b < Channel.now + u2 - Detector.clock));$$

Code Sample 14: Auxiliary invariant for proving accuracy of the timeout system

The completeness property should say that, if the sender fails, then before too long, the detector will perform a *timeout*. Specifically, we can say: "For any execution α of $TimeoutSystem(u1,u2,b,M)$, where $u1 \geq 0 \wedge u2 \geq 0 \wedge b \geq 0 \wedge u2 > (u1 + b)$, if a *fail* event occurs in α at real time t, then it is followed by a *timeout* event, within time at most $b + u2$." This bound can be argued informally, in terms of events: After the sender fails, within time at most b, any message it has already sent is delivered. Then within at most an additional time $u2$, the detector performs a *timeout*. The bound could also be proved more formally, using a specification automaton and a forward simulation, as in the Two Task Race example.

6.5 Discussion

The timeout example illustrates a typical, simple clock-based timeout mechanism for detecting process failures. This and similar mechanisms are used in many distributed algorithms and communication protocols. The example demonstrates the use of composition in Tempo, and indicates how one might prove global properties of a composed automaton.

In our formulation of the timeout failure detector, we have assumed that the *clock* variables of the sender and receiver automata progress at rate 1. Often in such examples, people make a somewhat weaker assumption: that the clock rate is always between $1 - \rho$ and $1 + \rho$, for some

known, small nonnegative real constant ρ. This weaker assumption could also be expressed using Tempo notation, as: $d(clock) \geq 1 - rho \wedge d(clock) \leq 1 + rho$. The same kinds of properties hold for the modified system as for the original system. However, we must increase the timeout bound to take the clock discrepancy into account, so that the receiver does not time out the sender incorrectly. Moreover, the increase in the timeout bound, and the clock uncertainty, lead also to an increase in the worst-case time to produce the timeout announcement. We leave the detailed calculations of these increases to the reader.

7 Example 4: Leader-Election Algorithm

This example illustrates a simple distributed algorithm in which several processes attempt to coordinate so that one of their number is distinguished as the "leader", at any point in time. We assume that processes may fail and recover. Since we would like the leader to be a non-failed process, the identity of the leader may have to change from time to time, during the operation of the system.

The Distributed Algorithms research literature contains many examples of leader election algorithms. The particular algorithm we consider here is based on one used in a proposed fault-tolerant extension of the DHCP IP-address-assignment protocol []. In that setting, leader election is used to choose the server that is currently responsible for managing a particular IP address. This application requires that the leader-election algorithm satisfy a kind of "mutual exclusion" property: it should never allow two processes to believe, at the same time, that they are the leader.

The algorithm we describe uses a separate failure-detection service, which provides information to the processes about which processes are currently alive. In practice, the failure-detection service would itself be implemented using another distributed algorithm, using a strategy based on timeouts as in Section 6. However, we present it abstractly here, as a single global service automaton.

The processes are assumed to have local clocks, which are reliable and increase at rate 1. Less synchronized clocks could also be used, with slight adjustments in bounds. We assume that, when a process recovers, it resumes its computation with all of its variable restored to their initial values. The only exception to this rule is the local clock, which we assume remains reliable. The processes use time for other purposes besides failure detection; in particular, a process must wait a specified amount of time after recovery before it can decide that it is the leader.

This leader-election algorithm is a typical example of a distributed algorithm in which a collection of processes coordinate by using a shared service. It is atypical in that the processes do not communicate in any other way except through the shared service; usually, the processes would also communicate via point-to-point or broadcast channels. We could also model such channels as TIOAs. An example of such a channel—a reliable FIFO point-to-point channel—is given in Section 6. Many other kinds of channels could also be defined; see Section **??** in Part II for some others.

7.1 The election processes

We assume that the processes of the election algorithm are named by elements of a finite set J, which is declared in the vocabulary *Processes* in Figure 15. We assume that the set of process names is totally ordered, and we assume an explicit operator to return the minimum index in a given set.

In our algorithm, the processes try to elect the process with the minimum name, among those that are currently alive. To determine who is alive, the processes use information that arrives from

```
vocabulary Processes
        types J
        operators min: Set[J] →J
end

imports Processes

automaton Elect(j:J, delta, e, u: Real) where delta > 0 ∧e > 0 ∧u > 0

 signature
        input status(L: Set[J], const j), fail(const j), recover(const j)
        internal elect(const j)
        output leader(const j)

 states
        live:Set[J] := {j};
        elected: Bool := false;
        clock: Real := 0;
        last_rectime: Real := 0;
        next_announce: AugmentedReal := ∞;

 transitions
        input status(L,j)
                eff if j ∈ live then
                        live := L;
                        if (j ≠min(live)) then
                          elected := false;
                        fi;
                    fi;

        input fail(j)
                eff live := ∅;
                    elected := false;
                    last_rectime := 0;
                    next_announce := ∞;

        input recover(j)
                eff live := {j};
                    last_rectime := clock;

        internal elect(j)
                 pre j ∈ live ∧j =min(live) ∧clock > last_rectime + delta ∧¬elected;
                 eff elected := true;
                      next_announce := clock;

        output leader(j)
                pre elected =true;
                eff next_announce := clock + u;


trajectories
        trajdef normal
          stop when j ∈ live ∧
                    ((j =min(live) ∧clock ≥last_rectime + delta + e ∧¬elected) ∨
                     clock =next_announce);
          evolve d(clock) =1;
```

Code Sample 15: Tempo description of a leader-election process

the failure-detection service. When a process recovers, and also at the beginning of execution, it waits a certain amount of time—strictly more than *delta*—before electing itself as the leader. This strategy prevents more than one process from deciding that they are the leader at the same time.

Automaton *Elect* has four parameters: its name *j*, which is of type *J*, and three reals, *delta*, *e*, and *u*, representing, respectively, the amount of time the process waits after recovery before it *may* elect itself the leader, a small positive constant used to bound when it *must* elect itself the leader, and an upper bound for announcements when it is the leader.

The automaton has three kinds of input actions. The first, of the form *status*(*L*, *j*), are intended to be produced by the failure-detection service. The first parameter, *L*, indicates a set of processes, representing those that the failure detector claims are currently alive. The second parameter, *j*, is simply the name of the process itself. Since that name is already fixed for the particular process instance (as a formal parameter of the automaton), we write this here with the keyword **const**, as **const** *j*. The other two inputs are failure and recovery actions, also parameterized by **const** *j*. The automaton also has an internal action, *elect*(**const** *j*)', by which it elects itself the leader, and an output action, *announce*(**const** *j*), by which it announces that it is the leader.

The state of the *Elect* process automaton includes the variable *live*, which keeps track of the latest information the process has about who is alive. Initially, it sets *live* to record only that process *j* itself is alive. The next variable, *elected*, is a Boolean flag saying whether process *j* is currently elected as the leader. The variable *clock* is a real-time clock. The variable *last_rectime* keeps track of the last time at which process *j* recovered from a failure (or the initial time, if this has never happened). Finally, *next_announce* is a deadline variable that keeps track of when process *j* should next announce that it is the leader.

A *status*(*L*,*j*) transition does nothing if process *j* is currently failed. Otherwise, if it is alive, it records the arriving set *L* in *live*. (We will assume a reliable, up-to-date, failure detection system, which implies that *j* itself will always be in the set *L* in this case.) Finally, if *j* is now not the minimum live process, process *j* sets *elected* to *false*.

The *fail* transitions cause the non-clock variables to be reset to default values. The *recover* transitions cause process *j* to record that it is alive, and also to record the current time as the last recovery time.

Process *j* may elect itself as leader at any point when it sees that its name is the minimum among the names of processes it believes to be alive, and when its *clock* is sufficiently greater than the last recovery time. The effect is simply to set the *elected* flag to *true* and to schedule an immediate leader announcement. Process *j* may announce that it is the leader at any point when it is the leader. The only effect is to reset the deadline for the next announcement to *clock* + *u*. This guarantees that, while it is the leader, it will announce this fact every so often.

The trajectories allow the clock to advance at rate 1. Time is required to stop when *j* should elect itself the leader, or when it is time to announce that it is the leader. Note the use of the constant *e* used here in defining the deadline for election. This constant does not appear in the corresponding precondition for the *elect* action. However, that precondition uses a strict inequality, which would not make sense in a stopping condition. This is because there is no "first" time when the clock strictly exceeds *last_rectime* + *delta*!

7.2 The failure-detection service

Tempo code for the failure-detection service appears in Figure 16. Many different kinds of failure detectors have appeared in Distributed Algorithms research papers, varying according to the

strength of their accuracy and completeness guarantees. (See the discussion of accuracy and completeness for the timeout system, in Section 6.) Here, we describe a failure detector with strong guarantees. Namely, it learns the true failure status of processes immediately, and correctly and frequently informs each process of the set of live processes. The parameter *delta* represents an upper bound on the time between status reports to each process.

automaton *FailureDetector*(*delta*: *Real*) **where** *delta* > 0
imports *Processes*

signature
input *fail*(*j*: *J*), *recover*(*j*: *J*)
output *status*(*L*: **Set**[*J*], *j*: *J*)

states
live: **Set**[*J*] : = {*j*:*J* **where** *true*};
clock: *Real* : = 0;
next_status: **Array**[*J*,*Real*] : = *constant*(0);

transitions

input *fail*(*j*)
eff *live* : = *live* − {*j*};

input *recover*(*j*)
eff *live* : = *live* ∪{*j*};

output *status*(*L*, *j*)
pre *L* =*live*;
eff *next_status*[*j*] : = *clock* + *delta*;

trajectories
trajdef *normal*
stop when ∃*j*:*J* (*next_status*[*j*] =*clock*);
evolve *d*(*clock*) =1;

Code Sample 16: Tempo description of the failure-detection service

The signature includes input actions *fail*(*j*) and *recover*(*j*) for every *j* in *J*. Note that each such action is also an input action of the appropriate automaton *Elect*(*j*). When we compose the pieces, each of these input actions will be shared by two automata. The only other actions are the *status*(*L*,*j*) outputs, by which the failure detector informs process *j* that *L* is the current set of live processes.

The automaton has a state variable *live*, which contains a set of processes, intended to be the set of all live processes. The presumption at the beginning is that all processes are alive, so *live* is initialized to the full set of processes. The automaton also has a real-time clock. Finally, it has an array *next_status* of deadline variables. For each *j*, *next_status*(*j*) keeps track of a deadline for the next time the failure detector should inform process *j* about the current failure status.

The *fail* and *recover* transitions simply update the *live* variable appropriately. The *status* transitions allow the failure detector to inform any process *j* about the (correct and current) failure

status, at any time. Their effect is simply to establish a new deadline for the next status update to process j.

Finally, the trajectories allow time to pass normally, forcing time to stop when a deadline is reached. Thus, the status updates are allowed to happen at any time, with no lower bound on the time between successive updates. However, there is an upper bound on this time, expressed by the stopping condition.

7.3 The complete leader-election system

The complete leader-election system combines the *Elect*(j) automata for all j with the failure detector, as in Figure 17. The system takes three parameters, *delta*, e, and u, where the first has the same meaning as in the failure detector and the last two have the same meanings they do in *Elect*. The components are the *Elect*(j, *delta*, e, u) automata for every j in J, plus *FailureDetector*(*delta*). We assign these short names $E[j]$ and *FD* by which we can refer to them within the composition.

automaton *LeaderSystem*(*delta*,e,u: *Real*) **where** $delta > 0 \wedge e > 0 \wedge u > 0$
  components
    $E[j:J]$: *Elect*(j, *delta*, e, u);
    FD: *FailureDetector*(*delta*);
  hidden
    status(L,j);

Code Sample 17: Tempo description of the leader-election system

We also specify that the *status* actions are hidden; formally, we define a new TIOA that is the same as the specified composition except that the *status* output actions are reclassified as internal rather than output actions. This is useful if we want to regard the *LeaderSystem* as a black box, to be used as a piece of a larger system. In that case, the *status* actions should be regarded as internal, since they are probably not relevant to the correct behavior of the leader election subsystem, as seen by the rest of the system.

What are the interesting properties of *LeaderSystem*? First, we would like to know that we don't have two leaders elected at the same time. And second, we would like to know that, soon after any point when the underlying system stabilizes, in the sense that failures and recoveries stop, a leader is actually elected. The first of these two properties could be stated as an invariant, as in Figure 18.

invariant of *LeaderSystem*:
  $\forall i$: J $\forall j$: J
    $(i \neq j \Rightarrow \neg(E[i].elected \wedge E[j].elected))$;

Code Sample 18: Unique leader invariant

Actually, this is probably not quite what we would like to say. We would like to know not just that two processes cannot be leaders *at exactly the same time*, but also that they cannot be leaders *at times that are too close together*. To see why, suppose we consider this leader-election algorithm as part of a larger distributed algorithm, in which a process j assumes that it can act as a leader if it has received a *leader* announcement "recently" from its local piece of the leader

sub-algorithm. Suppose that, in the larger algorithm, two processes simultaneously assume they may act as leaders—a situation that we would like to avoid. Then it must be that, in the leader subalgorithm, the two processes were both recently considered to be leaders. That is, they were leaders at times that are close together, though not necessarily at exactly the same time.

At any rate, the key to proving either version of this property is to suppose that two processes, $j1$ and $j2$, have *elected* $=$*true* simultaneously (or nearly simultaneously), and that $j1 < j2$. Then both processes must have been alive for a long time, which implies that $j2$ knows that $j1$ is alive. In that case, $j2$ would not classify itself as a leader, a contradiction. This argument can be turned into a formal proof using invariants and deadline variables, though a fair amount of work would be needed for that.

The second property is easy to see informally, though again, a formal proof would require some work.

7.4 Discussion

The leader-election example illustrates how to model a simple, typical distributed algorithm in which a collection of processes interact using a shared service. In other distributed algorithms the processes might also interact using shared channels as described in Section 6.

This example again illustrates the use of composition, and suggests how one might prove global properties of a composed automaton.

8 Example 5: Dynamic Bellman-Ford Shortest-Paths Protocol

The fifth example is intended to illustrate how to write a simple, fairly typical timing-based communication protocol using Tempo. The protocol we present is a dynamic version of the well-known Bellman-Ford shortest paths algorithm.

In the dynamic Bellman-Ford protocol, processes are located at the vertices of an edge-weighted directed graph. One vertex is distinguished as the *root* vertex. Each process is supposed to maintain information about a minimum-weight (also known as the "shortest") path from the root to itself. The specific information that the process keeps is the weight of the path plus its "parent" on the path. We assume that processes can fail and recover. Thus, minimum-weight paths may change over time, and the processes will need to adjust their information to accommodate these changes.

Such an algorithm could be used to construct and maintain a structure (a *shortest-paths tree*) that would allow the root process to perform efficient global broadcasts. To use the algorithm in this way, the processes would have to carry out additional work to construct child pointers as well as parent pointers.

8.1 The root process

Data types related to directed graphs and weighted directed graphs appear in the vocabulary defined in Figure 19. Namely, we use type *Index* to represent the vertices, and we define an *edge* to be an ordered pair of indices, called the *source* and *target* respectively. A *Graph* is then simply a set of edges. A *WeightedGraph* is a pair consisting of a *Graph* and a *weight* mapping, which maps edges to natural numbers.

Although we allow an *Edge* to be any pair of indices, we intend that a *Graph* should include only edges in which the source and target nodes are different. Properties such as these can be stated as

vocabulary *Nodes*
  types *Index*,
    Edge : **Tuple**[*source*: *Index*, *target*: *Index*],
    Graph : **Set**[*Edge*],
    WeightedGraph: **Tuple**[*G*: *Graph*, *weight*: **Map**[*Edge*, *Nat*]],
    Message : **Tuple**[*weight* : *Nat*, *destination*: *Index*]

  operators
    createedge: *Index*, *Index* →*Edge*,
    root: *Graph* →*Index*,
    innbrs, *outnbrs*: *Graph*, *Index* →**Set**[*Index*]

end

Code Sample 19: Vocabulary for dynamic Bellman-Ford

formal axioms, for example, $\forall e \in G$: *Graph* (*e.source* *neq e.target*). Such axioms are not part of the standard Tempo language, however, but rather, are part of the input to the theorem-prover used with Tempo.

We also require certain operators involving these types. For instance, *createedge* simply produces an edge from a pair of indices. Also, *root* is a mapping that selects a distinguished root vertex for a given graph. And *innbrs* and *outnbrs*, respectively, yield the set of incoming and outgoing neighbors of a given vertex of a given graph. Again, we would like some axioms to express properties of these data operators, for example, $innbrs(G,j) = \{\ e.source \mid e \in G \wedge e.target = j\ \}$ and $outnbrs(G,j) = \{\ e.target \mid e \in G \wedge e.source = j\ \}$.

Figure 20 contains the automaton for the root process. The parameter u represents the interval between successive times when the automaton sends information to all of its neighbors.

The root automaton has inputs *fail* and *restart*, outputs $send(m,i,j)$, and inputs $receive(m,i,j)$. In the *send* outputs, m is always 0, because it represents the distance from the root to itself. The second parameter i of *send* and *receive* actions is the index of the root automaton itself, which is a parameter of the automaton definition, and so this is preceded here by the keyword **const**. The third parameter j is the index of an outgoing neighbor of the root. The automaton also has an internal action *sendupdate*, by which it assembles messages to send to its outgoing neighbors.

The state contains a *sendbuffer*, which holds the messages to be sent to the outgoing neighbors, and a Boolean flag *failed* saying whether the node is currently failed. It also contains a real-time *clock* and a deadline variable *next_send* giving a bound on when the next messages should be sent.

A *sendupdate* may occur when the clock reaches the deadline *next_send*. Its effect is to add, to *sendbuffer*, messages to all of its outgoing neighbors, and to reset the *next_send* deadline. A *send* may occur whenever there is a message in the *sendbuffer*, and which point the message is removed. A *receive* input is simply ignored (since the root does not care about indirect paths to itself!). A *fail* results in setting the *failed* flag to *true* and setting the other variables (except for *clock*) to default values. A *restart* results is setting the *failed* flag to *false* and then setting *next_send* to the current time.

The trajectories allow time to pass at rate 1, stopping time when either the *next_send* deadline variable says it is time to put new messages in the *sendbuffer* or when the *sendbuffer* contains any messages to be sent to the neighbors. Thus, messages placed in the *sendbuffer* by a *sendupdate* are

```
imports Nodes

automaton BellmanFordRoot(W: WeightedGraph, u: Real, i: Index) where u > 0 ∧ i = root(W.G)

  signature
    input fail, restart
    output send(m: Nat, const i, j: Index) where m = 0 ∧ j ∈ outnbrs(W.G,i)
    input receive(m: Nat, j: Index, const i) where j ∈ innbrs(W.G,i)
    internal sendupdate

  states
    failed: Bool := false;
    sendbuffer: Set[Message] := ∅;
    clock: Real := 0;
    next_send: AugmentedReal := 0;

  transitions

    internal sendupdate
      pre clock = next_send;
      eff sendbuffer := sendbuffer ∪ {m:Message where m.weight = 0 ∧ m.destination ∈ outnbrs(W.G,i)};
          next_send := clock + u;

    output send(m, i, j)
      pre [m,j] ∈ sendbuffer;
      eff sendbuffer := sendbuffer − {[m,j]};

    input receive(m, i, j)
      eff

    input fail
      eff failed := true;
          sendbuffer := ∅;
          next_send := ∞;

    input restart
      eff failed := false;
          next_send := clock;

  trajectories

    trajdef traj
      stop when
        clock = next_send ∨ sendbuffer ≠ ∅;
      evolve
        d(clock) = 1;
```

Code Sample 20: Root process for dynamic Bellman-Ford

sent immediately, before any further time passes.

8.2 The non-root processes

Code for the non-root processes appears in Figures 21 22. Here, parameter u is the interval for sending notifications to outgoing neighbors, b is an assumed upper bound on message delay, and e is a small "epsilon' used for the timeout, as in the leader election example in Section 7.

The signature includes *fail*, *restart*, *send*, *receive*, and *sendupdate* actions, as for the root automaton. In addition, we now have a new *timeout* internal action, which the automaton uses as a signal to remove old path information.

The state variables include *sendbuffer*, *failed*, *clock*, and *next_send*, as for the root. In addition, we now have state variables *dist* and *parent*, which keep track of path information, namely, a natural-number distance estimate and a proposed parent on a minimum-weight path. Initially, *dist* and *parent* are *nil*, and *next_send* is set to *infty*, because the automaton has no information to send. The automaton also has a *timeout_deadline* variable, which keeps track of when any current information should be timed out.

Nearly all of the interesting work of this automaton is done by the *receive* transitions: A *receive*(*m*,*i*,*j*) transition begins with a **locals** declaration, which defines and initializes a local variable w to the weight of the edge incoming from the originator of the message, j, to i. This variable is local to the given transition definition, and is not regarded as part of the automaton's state.

In the effects, process i does nothing if it is currently failed. Otherwise, it considers the incoming information and decides whether it should accept it in preference to its current information. It accepts the new information in two situations: if it currently has no path information ($dist = nil$), or if the new information yields a strictly shorter distance than the current information ($dist \neq nil \wedge (m + w < (Nat)dist)$) In this second case, the process does not distinguish whether the new, shorter distance has arrived from the current parent or from a different incoming neighbor.

If the process does decide to accept the new information, it resets its *dist* variable to the new, improved distance, and its *parent* variable to the originator of the message. It sets *timeout_deadline* to an appropriate time in the future—calculated as $u+b$, the sum of the sending interval and an upper bound on the message delay. The idea here is that, if the current information remains correct, process i should receive it again within that amount of time. Also, if the process accepts the new information, it sends the new distance estimate to all of its neighbors (by putting the needed messages into *sendbuffer*), and resets its deadline variable for its next send, *next_send*, to time u in the future.

On the other hand, if the process decides not to accept the new information, if checks to see whether the message is in fact refreshing the path information that it previously had. In that case, it resets the *timeout_deadline* to the appropriate future time.

A *timeout* transition may occur anytime when the clock strictly exceeds the *timeout_deadline*. When it occurs, all path information is discarded, and *next_send* and *timeout_deadline* are set to their default value, *infty*.

A *sendupdate* transition may occur when $clock = next\_send$ and $dist \neq nil$. The effect is to put messages containing the current *dist* into the *sendbuffer* and reset the *next_send* deadline so it can send again, time u later.

You might think that the second clause of the precondition, $dist \neq nil$, could be omitted since it could be proved from the first clause, $clock = next\_send$. However, this is not quite correct. The fact that the first clause implies the second could in fact be proved as an invariant of this automaton,

```
imports Nodes

automaton BellmanFordNonRoot( W: WeightedGraph, u:Real, b:Real, e: Real, i: Index)
  where u > 0 ∧b ≥0 ∧e > 0 ∧i ≠root( W.G)

  signature
    input fail, restart
    output send(m: Nat, const i, j: Index) where j ∈ outnbrs( W.G,i)
    input receive(m: Nat, j: Index, const i) where j ∈ innbrs( W.G,i)
    internal sendupdate, timeout

  states
    failed: Bool := false;
    sendbuffer: Set[Message] := ∅;
    dist: Null[Nat] := nil;
    parent: Null[Index] := nil;
    clock: Real := 0;
    next_send: AugmentedReal := ∞;
    timeout_deadline: AugmentedReal := ∞;

  transitions

    input receive(m, j, i)
      locals
        w:Nat := W.weight[createedge(j,i)];
      eff
        if ¬failed then
            if dist =nil ∨(dist ≠nil ∧(m + w < (Nat)dist)) then
                dist := embed(m + w);
                parent := embed(j);
                timeout_deadline := clock + u + b;
                sendbuffer := sendbuffer ∪{m:Message where m.weight = (Nat)dist ∧m.destination ∈
outnbrs( W.G,i)};
                next_send := clock + u;
            else
                if (parent =embed(j) ∧dist =embed(m+w)) then
                    timeout_deadline := clock + u + b;
                fi;
            fi;
        fi;

    internal timeout
      pre timeout_deadline ≠∞∧clock > timeout_deadline;
      eff dist := nil;
          parent := nil;
          next_send := ∞;
          timeout_deadline := ∞;
```

Code Sample 21: Non-root process for dynamic Bellman-Ford

```
    internal sendupdate
      pre clock =next_send ∧dist ≠nil;
      eff sendbuffer := sendbuffer ∪{m:Message where m.weight =(Nat)dist ∧m.destination ∈
outnbrs(W.G,i)};
          next_send := clock + u;

    output send(m, i, j)
      pre [m,j] ∈ sendbuffer;
      eff sendbuffer := sendbuffer − {[m,j]};

    input fail
      eff failed := true;
          sendbuffer := ∅;
          dist := nil;
          parent := nil;
          next_send := ∞;
          timeout_deadline := ∞;

    input restart
      eff failed := false;

  trajectories

    trajdef traj
      stop when
        clock =next_send ∨clock =timeout_deadline + e ∨sendbuffer ≠∅;
      evolve
        d(clock) =1;
```

Code Sample 22: Non-root process for dynamic Bellman-Ford, continued

once the automaton is defined. However, in defining the automaton, we need to make sure that all the transition definitions make sense on their own. In this *sendupdate* transition definition, we convert *dist* from type **Null**[*Nat*] to type *Nat*, which presupposes that *dist* $\neq$*nil*. To avoid the circularity, we include the condition *dist* $\neq$*nil* in the precondition.

The *send* transitions are the same as for the root. The *fail* transitions are similar to those for the root, though now there are several more variable to set to default values. The *restart* transitions are exactly like those for the root. The trajectories allow time to pass until either a sending or timeout deadline is reached, or the *sendbuffer* is nonempty.

8.3 The complete Bellman-Ford system

The complete dynamic Bellman-Form algorithm is defined in Figure 23. The components are the root automaton, non-root automata for all the non-root indices, and timed channels connecting all of the neighboring indices in the graph. The channels we use here are essentially the same as *TimedChannel* from the timeout example, in Section 6. However, we need to modify *TimedChannel* slightly, to take two additional parameters representing the source and target vertices for the channel.

What could one prove about the behavior of this protocol? It is difficult to say much about what is happening during times when the system is changing, while processes fail and recover. However, we can describe guarantees for stable situations, which arise when failures and recoveries stop for a while. For example, consider the special case case where failures and recoveries stop from some point on, and where the root is non-failed after that point. In this case, one can show that eventually, every live process that has a path from the root ends up with its *dist* and *parent* variables set to the weight and parent for some correct minimum-weight path from the root. Furthermore, the time for this to happen can be bounded in terms of the difference between the maximum correct distance and the minimum incorrect distance estimate.

8.4 Discussion

This example illustrates how to model a simple, typical communication protocol for wired networks using Tempo. Other communication protocols for wired networks can be modeled similarly.

To model algorithms for wireless networks, we can again use process and channel automata. The processes are similar to those in a wired network, but now the channels are broadcast channels. Complications arise in describing characteristics of the broadcast channels, such as message losses and collisions and limited broadcast range. In order to describe these characteristics, the channel model needs information about the physical locations of the processes.

Mobility introduces additional complications: If the components are mobile, the channel model must maintain the location information so that it is relatively up-to-date.

9 Example 6: One-Shot Vehicle Controller

Our final example is intended for people interested in modeling and analyzing hybrid (continuous/discrete) systems, for example, systems in which real-world entities such as robots or vehicles are controlled by computer programs. Our example consists of only two components, a train, and a controller that may apply a brake to slow the train down. The train moves according to Newtonian laws of motion, with different accelerations based on whether its brake is currently being applied.

```
vocabulary TimedChannel2Types(M: Type)
   types Packet : Tuple[message: M, deadline: Real]
end

types Index

automaton TimedChannel2(b: Real, i,j: Index, M: type) where b ≥0
imports TimedChannel2Types(M)

   signature
      input send(m:M, i,j: Index)
      output receive(m:M, i,j: Index)

   states
      queue: Seq[Packet] : = ∅;
      now: Real : = 0;

   transitions
      input send(m,i,j)
         eff queue : = queue ⊢[m,now+b];
      output receive(m,i,j)
         pre queue ≠∅∧head(queue).message =m;
         eff queue : = tail(queue);

   trajectories
      trajdef traj
         stop when ∃p: Packet (p ∈ queue ∧now =p.deadline);
         evolve d(now) =1;


automaton BFSystem(W: WeightedGraph, u, b, e: Real)
  where u > 0 ∧b ≥0 ∧e > 0
  components
    BRoot: BellmanFordRoot(W, u, root(W.G));
    BNR[j: Index]: BellmanFordNonRoot(W, u, b, e, j) where j ≠root(W.G);
    TC[i,j: Index]: TimedChannel2(b,i,j,Nat) where createedge(i,j) ∈ W.G;
```

Code Sample 23: Complete dynamic Bellman-Ford system

While the brake is being applied, the train decelerates, following some negative acceleration in a known range [*amin*,*amax*]. While the brake is not being applied, the train continues at a steady speed.

This example is derived from one in H. B. Weinberg's M.S. thesis [10] and in the related paper [11]. The example illustrates how to model a combination of real-world and computer components. In fact, it is essentially the first example we are presenting that allows interesting state evolution between discrete events, that is, during trajectories. (We did briefly consider, in Sections 6 and 7, local clocks that progress during trajectories at rates that are approximately that of real time.)

9.1 The train

We model the train by a TIOA called *Train*, described in Figure 24. *Train* has a parameter *v0*, which represents its initial velocity. This velocity must be positive, indicating that the train is moving in the positive direction (left to right). We assume that the train starts at position 0 on the track. *Train* has two other parameters, *amin* and *amax*, which represent the minimum and maximum acceleration allowed while the train is braking. Both of these are negative reals, which indicates that the train actually decelerates when the brake is applied.

The only discrete actions are two inputs, *brakeOn* and *brakeOff*, which represent engagement and disengagement of the brake, respectively. The state variables include real values x, v, and a, which represent the current position, velocity, and acceleration of the train. The state also includes a Boolean flag b that says whether the brake is currently engaged or not; initially it is not. Finally, we have a real-time clock *now*.

There are two types of transitions. A *brakeOn* transition simply sets the brake flag to *true* and sets the acceleration to be any value between *amin* and *amax*. Similarly, a *brakeOff* transition sets the flag to *false* and sets the acceleration to be 0.

The automaton has two kinds of trajectories. The first kind, called *On* trajectories, capture the behavior of the train while the brake is on. During such a period, the acceleration is allowed to change, though it always remains between the specified minimum and maximum acceleration bounds; this limitation is expressed by an **invariant** clause. Besides remaining between these bounds, a is constrained to observe its dynamic type, which is the set of piecewise continuous real-valued functions. The velocity evolves in the usual way, with its derivative equal to the acceleration, and similarly, the position's derivative is equal to the velocity. That is, the velocity is derived from the acceleration, and the position is derived from the velocity, by integrating. Finally, *now* evolves at rate 1.

The second kind of trajectories are called *Off* trajectories, and they capture the behavior of the train while the brake is off. During such a period, the acceleration remains at 0; the velocity and position are expressed in terms of the acceleration as in the *On* trajectories. As usual, *now* evolves at rate 1.

As we noted above, this is the first example we are presenting that includes interesting state evolution during trajectories. Interesting state evolution is typical of hybrid system examples, since they generally include real-world components whose behavior is subject to quantifiable physical laws. Distributed algorithms and communication protocol examples, on the other hand, usually have rather simple state evolution. The most complicated type of evolution that we typically see in those examples is for *local clocks*, which may drift slightly as time passes, at a rate that may be constrained to remain within some known bounds. One type of communication setting that

automaton *Train*(*v0*, *amin*, *amax*: *Real*) **where** $v0 > 0 \wedge amin \leq amax \wedge amax < 0$

signature

input *brakeOn*, *brakeOff*

states

x: $Real := 0$;
v: $Real := v0$;
a: $Real := 0$;
b: $Bool := false$;
now: $Real := 0$;

transitions

input *brakeOn*

eff $b := true$;
$a :=$ **choose** k **where** $amin \leq k \wedge k \leq amax$;

input *brakeOff*

eff $b := false$;
$a := 0$;

trajectories

trajdef *On*

invariant
$b \wedge amin \leq a \wedge a \leq amax$;
evolve $d(v) = a$;
$d(x) = v$;
$d(now) = 1$;

trajdef *Off*

invariant $\neg b \wedge a = 0$;
evolve $d(v) = a$;
$d(x) = v$;
$d(now) = 1$;

Code Sample 24: The train

does have interesting state evolution is *mobile wireless communication*: complete models for such systems must take into account physical motion of the mobile devices.

9.2 The controller

In contrast to the real-world model, the controller is generally a discrete component, like the algorithmic components we have already been considering. In this case, we model a specific kind of controller—one that engages the brake just once and then disengages it once. Code for the controller appears in Figure 25

The given vocabulary defines a simple type *Phase*, defined as an enumeration of three phase values. The automaton has *phase* =*idle* before it has applied the brake, *phase* =*braking* while the brake is on, and *phase* =*done* after it has stopped braking.

The automaton is called *Oneshot*, because it engages and disengages the brake exactly once. The times at which it performs these two events are determined by parameters A, B, and C. In particular, A indicates the latest time when the brake might be applied, and B and C indicate the earliest and latest times, respectively, by which the brake might be disengaged. The only discrete actions are the outputs *brakeOn* and *brakeOff*.

The state contains a variable *phase*, which keeps track of the current controller phase; initially this is *idle*. Other variable keep track of the current time, a deadline for when the brake can be applied (initialized at A), and an earliest time and a deadline for when the brake can be disengaged.

A *brakeOn* transition is allowed to happen when *phase* =*idle*. Its effect is to reset *phase* to *braking* and to set the earliest-time and deadline variables for disengaging the brake (to B and C in the future, respectively). A *brakeOff* transition may happen when *phase* =*braking* and when the earliest-time constraint is satisfied, that is, when the current time is greater than or equal to the earliest time allowed for braking. The effect is to reset *phase* to *done*.

The controller has three kinds of trajectories, corresponding to the three phases. The *idle* trajectories must stop when time reaches the *last_on* deadline for braking, and the *braking* trajectories must stop when time reaches the *last_off* deadline for disengaging the brake.

9.3 The controlled train system

Now we combine the train and controller into a controlled system, *OneshotSys*, whose code appears in Figure 26. The system has six real-valued parameters. As in the *Train* automaton, *v0* indicates the train's initial velocity, and *amin* and *amax* indicate bounds on the acceleration. Parameter *xt* represents a target (destination) point on the track. Parameters *vtmin* and *vtmax* indicate lower and upper bounds on velocity; we assume that system is supposed to ensure that, if the train ever reaches the target point *xt*, then its velocity will be between *vtmin* and *vtmax*.

The main interesting part of the specification is the instantiation of the three parameters, A, B, and C, of the controller *OneShot*. These values are calculated specifically to ensure that the train's velocity indeed reaches the desired range by the time the train reaches the desired position *xt*.

The first parameter of *OneShot* represents the latest time at which the controller should engage the brake. The subtracted term within the brackets represents the remaining distance needed to be sure that the train can slow down from its initial velocity of *v0* to the maximum permissible velocity *vtmax*. We require (in the composed system's **where** clause) that this distance is in fact no greater than the available distance *xt*. The result of the subtraction represents the maximum distance that the train may travel while still allowing enough time to slow down sufficiently. Dividing

```
vocabulary Oneshot_types
    types Phase: Enumeration [idle, braking, done]
end

automaton Oneshot(A, B, C: Real)
imports Oneshot_types

    signature
        output brakeOn, brakeOff

    states
        phase: Phase := idle;
        now: Real := 0;
        last_on: Real := A;
        first_off: DiscreteReal := 0;
        last_off: AugmentedReal := ∞;

    transitions
        output brakeOn
            pre phase =idle;
            eff phase := braking;
                first_off := now + B;
                last_off := now + C;

        output brakeOff
            pre phase =braking ∧first_off ≤now;
            eff phase := done;

    trajectories
        trajdef idle
            invariant phase =idle;
            stop when now =last_on;
            evolve d(now) =1;

        trajdef braking
            invariant phase =braking;
            stop when now =last_off;
            evolve d(now) =1;

        trajdef done
            invariant phase =done;
            evolve d(now) =1;
```

Code Sample 25: The controller

```
automaton OneshotSys(v0, xt, vtmin, vtmax, amin, amax: Real)
  where v0 > 0 ∧amin ≤amax ∧amax < 0 ∧0 ≤xt ∧
        0 ≤vtmin ∧vtmin ≤vtmax ∧vtmax ≤v0 ∧
        xt ≥(vtmax * vtmax − v0 * v0) / (2 * amax)

components
   Train: Train(v0, amin, amax);
   Controller: Oneshot(1/v0 * (xt − (vtmax * vtmax − v0 * v0) / (2 * amax)),
                       (vtmax − v0)/amax,
                       (vtmin − v0)/amin);
```

Code Sample 26: The controlled system

by *v0* translates this maximum distance to the maximum time before braking. The second and third parameters represent the minimum and maximum amounts of braking time that are safe for ensuring that the velocity ends up in the allowed range. In reading these various bounds, recall that *amax* and *amin* are negative reals, and *vtmax* and *vtmin* are less than are equal to the initial velocity *v0*; thus, many of the quantities in these expressions are negative.

The main property to be proved may be written as the invariant $x = xt \Rightarrow vtmin \leq v \wedge v \leq vtmax$.

9.4 Discussion

This example illustrates how to model a simple control system, consisting of a software controller component interacting with a real-world vehicle component. Although this example is simple, it should give you a good idea of how to model other kinds of control systems, for example, robot control systems, air-traffic control systems, or factory control systems. The main difference among these is that the real-world portions of these systems involve other kinds of physical dynamics, which means that trajectories are described using different kinds of equations.

Another difference is that some of these systems involve *distributed controllers*. For example, an air-traffic control system may be implemented by separate controllers running on computers on board the aircraft. In this case, we would probably choose to model these controllers as separate TIOAs. In the air-traffic example, the physical system also consists of separate components (the physical aircraft). However, in this case, we would probably not choose to model the aircraft as separate automata, since we might want to talk about relationships (e.g., minimum separation distance) among the various aircraft. Such relationships are most easily described using a single "Aircraft" automaton.

Some control systems also include human components. For example, an aircraft collision-avoidance system might consist of physical aircraft, on-board controller software, a communication system connecting the controllers, and the pilots of the aircraft. Such controller software may play an *advisory* role, communicating recommended actions to the pilots, who make the final decisions. In such a case, we can model all the system components, including the human pilots, as TIOAs. We can model the expected behavior of the pilots formally, along with that of the other components. In this way, we can use a TIOA model to analyze the behavior of the entire aircraft control system, including human participants.

Other kinds of hybrid systems that could be modeled using Tempo include systems of biological cells, such as a heart muscle composed of interacting heart cells. However, note that Tempo

(and TIOA) are restricted to allow only discrete interaction among components; if we want to capture continuous interaction, we would have to use a more general model such as Hybrid I/O Automata [5, 8].

Part II
TIOA User Guide

10 User Guide Introduction

In Part I, we presented an example-based tutorial on the use of the Tempo language. Here, in Part II, we give a more systematic, though still informal, description of the language constructs. More formal descriptions of Tempo syntax and semantics appear in the reference manual, in Part III.

The longest section of this user guide, Section 11, describes the Tempo language constructs that are used to define Timed I/O Automata (TIOAs). Section 12 describes facilities for applying composition and hiding operations to TIOAs. Section 13 shows how to define invariant assertions and simulation relations within Tempo, both of which are useful in explaining the behavior of algorithms and in proving their correctness. Finally, Section 14 describes the use of data types in Tempo, including a catalog of primitive data types and type constructors.

11 Timed I/O Automata

This section contains descriptions of the Tempo language constructs that are used to define Timed I/O Automata (TIOAs). These include automaton names and parameters, action signatures, state variables, transitions, and trajectories. We begin with a review of the mathematical definition of a Timed I/O Automaton.

11.1 Mathematical definition of Timed I/O Automata

Mathematically, a *Timed Input/Output (I/O) Automaton* (TIOA) A is a tuple with the following elements:

- A set X of *variables*, each of which has a *static type* and a *dynamic type*. The static type describes the values that the variable may take on whereas the dynamic type describes the allowable ways in which a variable may evolve over time. For example, a variable may have *Real* as its static type and the piecewise continuous functions from time intervals to Reals as its dynamic type.
- A set Q of *states*, which is a subset of the set of all possible valuations of X (a *valuation* is a function f that assigns to each variable x in X a value $f(x)$ in the static type of x).
- A set Θ of *initial states*, which is a non-empty subset of Q.
- A triple A, consisting of three disjoint sets I, O, and H of discrete *input*, *output*, and *internal (hidden)* actions,
- A *transition relation* D, which is a subset of $Q \times A \times Q$. We say that an action a is enabled in a state q if there exits a transition of the form (q, a, q') in D.
- A set T of *trajectories* for X, which is a set of functions from intervals of time starting with 0 to valuations in Q.

An action of an automaton is called *external* if it is an input or output action, and *locally-controlled* if it is an output or internal action. A TIOA must satisfy the following two axioms:

- *Input action enabling:* Every input action is enabled in every state.
- *Time-passage enabling:* From every state time can advance either indefinitely, or for a finite interval at the end of which a locally-controlled action is enabled.

There are also axioms that must be satisfied by dynamic types and trajectories. These axioms capture some natural requirements such as the fact that a prefix or suffix of a trajectory must be a valid trajectory. For a complete and formal definition of a TIOA the reader is referred to [2].

Executions and traces An *execution fragment* of a TIOA is a finite or infinite sequence $\tau_0 a_1 \tau_1 a_2 \ldots$ of alternating trajectories and actions such that, if τ_i is not the last trajectory in the sequence, then its domain is a closed interval $[0, t_i]$ of time and there exists a discrete transition from the last state of τ_i to the first state of τ_{i+1}. An *execution* is an execution fragment such that the first state of τ_0 is an initial state. A state is *reachable* if it occurs in some execution.

The external behavior of an automaton is captured by the set of "traces" of its executions. The *trace* of an execution is a sequence that contains only the external actions and the amount of time passage during each trajectory. Formally, the amount of time passage during a trajectory is the projection of a trajectory onto an empty set of variables.

A property that is true for all reachable states of an automaton is called an *invariant assertion* or *invariant* for short.

The Tempo language provides notations for defining TIOAs either as primitive automata by specifying their names, signatures, state variables, transition relations, and trajectories, or as composite automata by specifying how they can be constructed from simpler TIOAs. The following subsections describe the notations for specifying primitive automata, and their relation to the mathematical model of TIOAs. Section 12 describes notations for composite automata.

11.2 Automaton names and parameters

The first line of an automaton description in Tempo consists of the keyword **automaton** followed by the name of the automaton. The name may be followed by a list of formal parameters enclosed within parentheses. For example, the automaton *Clock* (Code Sample 27) has no formal parameters, and the *TimedChannel* automaton (Code Sample 28 has two parameters: *r*, which represents the maximum delay for a message in the channel and *M*, which represents the type of message that can be communicated by the channel.

Just to provide some intuition, we describe what the *Clock* automaton does. It keeps track of the *nextHour* and *nextMinute*. which represent the next time that it will display. Initially, this next time is set to 12:00, and it can be reset by an external user at any time, to any legal hour and minute. The *Clock* displays the next time, via a *show* output, at time zero, and also immediately after a reset. Immediately after the *Clock* displays the new time, it increments its recorded next time, waits one minute, and (unless a new reset occurs) displays the new time. The *TimedChannel* automaton was also used in Part I, in Section 6.1; you can look there for an informal description of its behavior.

There are two kinds of automaton parameters. An individual parameter, such as *r*: *Real*, consists of an identifier and an associated type, and it denotes an element of that type. A type parameter,

```
let legalTime(hour, minute): Nat, Nat →Bool =minute < 60 ∧0 < hour ∧hour < 13;

automaton Clock

  signature
    output show(hour, minute: Nat) where legalTime(hour, minute)
    input set(hour, minute: Nat) where legalTime(hour, minute)

  states
    now: Real := 0;
    nextHour: Nat := 12;
    nextMinute: Nat := 0;
    timeToShow: DiscreteReal := 0;

  transitions

    input set(hour, minute)
      eff nextHour := hour;
          nextMinute := minute;
          timeToShow := now;

    output show(hour, minute)
      pre hour =nextHour ∧minute =nextMinute ∧now =timeToShow;
      eff nextMinute := mod(minute + 1, 60);
          if nextMinute =0 then nextHour := mod(hour + 1, 12); fi;
          timeToShow := now + 1;

  trajectories

    trajdef timePassage
      stop when now =timeToShow;
      evolve d(now) =1;
```

Code Sample 27: *Clock* component

```
vocabulary Message(M: Type)
        types
                Packet : Tuple[message: M, deadline: Real]
end

automaton TimedChannel(b: Real, M: Type) where b ≥0
        imports Message(M)

        signature
                input send(m:M)
                output receive(m:M)

        states
                queue: Seq[Packet] : = ∅;
                now: Real : = 0;

        transitions

                input send(m)
                        eff queue : = queue ⊢[m,now+b];

                output receive(m)
                        pre queue ≠∅∧head(queue).message =m;
                        eff queue : = tail(queue);

        trajectories
                trajdef traj
                        stop when ∃p: Packet (p ∈ queue ∧now =p.deadline);
                        evolve d(now) =1;
```

Code Sample 28: Time bounded FIFO communication channel

such as *M*: **type**, consists of an identifier followed by the keyword **Type**, and it denotes a type. Type parameters allow the specification of polymorphic automata. A type parameter such as *M* can be instantiated by importing a vocabulary at the very beginning of an automaton (see 14.5). The types and operators from a vocabulary can then be used in defining variables, actions, transitions, and trajectories.

An automaton can contain a clause that constrains the values of its individual parameters. For example, an automaton whose definition begins with

automaton *Swap*(*A*, *B*: **Set**[*Int*]) **where** $A \subset B$

is parameterized by two sets of integers, the first of which must be a proper subset of the second.

11.3 Action signatures

The signature for an automaton is declared using the keyword **signature** followed by lists of entries describing the automaton's input, internal, and output actions. Each entry contains a name and an optional list of parameters enclosed in parentheses. Varying parameters (such as *hour*, *minute*: *Nat* in Code Sample 27) consist of identifiers with associated types, and they denote arbitrary elements of those types. A fixed parameter (such as **const** *i* or **const** *j* in Code Sample 29) consists of the keyword **const** followed by a term denoting a fixed element of its type. Fixed parameters turn out to be necessary when we want to name actions based on automaton parameters. Neither kind of parameter can have **Type** as its type.

automaton *IndexedChannel*(*i*, *j*: *Nat*, *M*: **Type**)

signature
  input *send*(**const** *i*, **const** *j*, *m*: *M*)
  output *receive*(**const** *i*, **const** *j*, *m*: *M*)

states
  buffer: **Seq**[*M*] := $\emptyset$;

transitions

  input *send*(*i*, *j*, *m*)
    eff *buffer* := *buffer* $\vdash$ *m*;

  output *receive*(*i*, *j*, *m*)
    pre *buffer* $\neq\emptyset \wedge m$ = *head*(*buffer*);
    eff *buffer* := *tail*(*buffer*);

Code Sample 29: Indexed communication channel

Each entry in the signature denotes a set of actions, one for each assignment of values to its varying parameters. Thus, *IndexedChannel* has one input action *send*(*i*, *j*, *m*) for each value of its parameter *m*; the values of *i* and *j* in these actions are fixed by their values as parameters of the automaton. Hence, the automaton *IndexedChannel*(1,2,*String*) (the specification of an indexed channel between parties 1 and 2), does not have, for example, an action *send*(3, 4, *Hello*). All *send* actions are of the form (1, 2, *s*) where *s* is of type *String*.

It is possible to constrain the values of the varying parameters for an entry in the signature using the keyword **where** followed by a predicate. These predicates are arbitrary first-order boolean predicates that may involve formal parameters of the associated action and formal parameters of the automaton (see **??**). For example, the **where** clauses in Code Sample 27 constrain the values of the parameters *hour* and *minute*. Thus, the set of output actions for the *Clock* automaton contains one action *show*(*hour*, *minute*) for each pair of values of its parameters that satisfy the predicate *legalTime*(*hour*, *minute*).

11.4 State variables

As in the examples, state variables are declared in Tempo using the keyword **states** followed by a semicolon-terminated list of state variables and their static types.

11.4.1 Initial values

The initial values of state variables can be constrained using two methods. First, each individual variable can be initialized using the assignment operator followed by an expression that may refer to automaton parameters but not to other state variables. Note that the expression here may be a nondeterministic choice over a set.

For example, in the *Clock* automaton (Code Sample 27), the initial values of the state variables *now*, *nextMinute*, and *timeToShow* are all 0, and the initial value of *nextHour* is 12. Hence, there is a single initial state for this automaton. If, for example, the assignment *timeToShow* : =0 were omitted from the declaration of the state variable *timeToShow* in the *Clock* automaton (Code Sample 27), then it would have an infinite number of initial states, one for each real number. In this case, the clock would not be able to display anything until either a **set** action occurs or until the value of *now* has reached *timeToShow*.

The initial value of *timeToShow* can be constrained to ensure an upper bound on the amount of time we might need to wait until the clock displays something, for example:

timeToShow: *DiscreteReal* : =**choose** r **where** $0 \leq r \leq 10$

When such a nondeterministic **choose** clause is used to initialize a state variable, we intend that at least one value of the variable should satisfy the predicate following the **where** clause. If the predicate is true for all values of the variable, then the meaning is the same as if no initial value had been specified for the state variable.

It is also possible to constrain the initial values of all state variables taken together, whether or not initial values are assigned to any individual state variable. This can be done using the keyword **initially** followed by a predicate (involving state variables and automaton parameters). For example, we can allow the clock to display an arbitrary legal time of day as soon as it is turned on by constraining *now* and *timeToShow* to have the same unspecified value:

states
now: *Real*; *nextHour*: *Nat*; *nextMinute*: *Nat*; *timeToShow* : *DiscreteReal*;
initially *now* =*timeToShow*

Note that **initially** predicates are allowed to contain variables whose initial values are assigned nondeterministically, via **choose** expressions. Thus, we may write:

states
xcoord: *Real* : = **choose** x **where** $x \geq 0$;
ycoord: *Real* : = **choose** y **where** $y \geq 0$;

initially $x + y = 4$

In this case, we intend that at least one valuation of the set of variables should satisfy the **initially** predicate, as well as all of the **where** predicates.

The order in which state variables are declared makes no difference: they are initialized simultaneously.

11.4.2 Types

The static type of a variable is declared explicitly as we have seen above. The dynamic type, on the other hand, is implicit and inferred automatically from the static type.

In Tempo, variables are classified as *discrete* or *analog* with respect to their dynamic types. Suppose that v is a variable of static type $s(v)$. We say that v is a discrete variable if its dynamic type consists of piecewise constant functions from time intervals to $s(v)$, and analog if $s(v)$ is *Real* and its dynamic type consists of piecewise continuous functions from time intervals to Reals. For a formal definition of discrete and analog variables see [2].

More strongly, in Tempo, we require that the values of discrete variables can change only through discrete transitions and remain constant as time passes between discrete transitions. And we assume that analog variables evolve continuously as time passes.

All variables that are of simple built-in types other than *Real* are assumed to be discrete and variables of type *Real* are assumed to be analog. It is also possible to define discrete real variables, by qualifying the static type name with the keyword *Discrete*; more precisely, we write the type as *DiscreteReal*. We discuss how to infer the dynamic type for compound static types that are built by type constructors in Section **??**, where we present data types of Tempo in detail.

11.5 Transition relations

Transitions for the actions in an automaton's signature are defined following the keyword **transitions**. A transition definition consists of

- an action kind (**input**, **output**, or **internal**),
- an action name with optional formal parameters and an optional **where** clause constraining the values of the parameters (see Section 11.5.1),
- an optional local variables list (see Section 11.5.2),
- an optional function definitions list (see Section 11.7),
- an optional precondition (see Section 11.5.3), and
- an optional effect (see Section 11.5.4).

More than one transition definition can be given for a parameterized action. In such cases, **where** clauses can be used to partition the set of transition definitions according to predicates on formal parameters. The predicates associated with the **where** clauses are not required to be disjoint although in most common usages they are.[2]

[2]In a tool that involve the execution of an automaton such as a simulator, it turns out to be practical to require that the **where** clauses for different transition definitions for the same action be disjoint.

```
automaton Temp
  signature
    input read(i:Nat)

  states
    temp: Enumeration[low,medium,high];
    degree: Int := 0;

  transitions

    input read(i) where i > 60
      eff temp := high;
          degree := i;

    input read(60)
      eff temp := medium;
          degree := 60;

    input read(i) where i < 60
      eff temp := low;
          degree := i;
```

Code Sample 30: Tempo description of a temperature reader

The automaton *Temp* in Code Sample 30 has multiple transition definitions for the input action *read*(*i*) with disjoint where clauses. The definitions, respectively, describe the effects of reading a temperature greater than 60, exactly 60, and less than 60.

11.5.1 Transition parameters

The parameters that follow an action name in a transition definition must match those that follow the action name in the automaton's signature, both in number and in type. The simplest way to formulate parameters for a transition definition is to erase the keyword **const** and the type modifiers from the parameters given for the action in the automaton's signature; thus, in Code Sample 29, the parameters of the *send* action are given as (**const** *i*, **const** *j*, *m*: *M*) in the signature, but are shortened to (*i*, *j*, *m*) in the transition definition.

Action parameters and transition parameters differ slightly. Parameters in the action signature can be terms (identified by the keyword **const**) that denote fixed values or they can be (declarations for) variables. All parameters in transition definitions are variable identifiers or constants and the keyword **const** cannot appear. For example, the parameter in *read*(60) in *Temp* denotes a fixed value. If a transition definition contains other variables (such as *i* in *read*(*i*)), these variables can have arbitrary values.

11.5.2 Local variables

A transition definition may contain additional local variables, which are declared after the action name and transition parameters, and before the precondition. The syntax for listing and initializing

local variables follows the same rules as for state variables. except that a local variable list is identified by the keyword **locals**.

Local variables serve two purposes. First, they can be constrained by a transition's precondition and used in the effects, as in the following automaton, *PitchTwo*:

automaton *PitchTwo*(*s*: **Set**[*Nat*])
  signature
    output *pitch*(*n*: *Nat*)
  states
    left: **Set**[*Nat*] := *s*;
  transitions
    output *pitch*(*n*)
      locals *x* : *Nat*;
      pre $n \in left \wedge x \in left \wedge n < x$;
      eff *left* := *delete*(*n*, *delete*(*x*, *left*));

PitchTwo discards two numbers at a time from a set, but communicates only the smaller of the two when a transition occurs. When the effects clause in a transition definition does not assign any values to a local variable, as is the case here, the definition can be rewritten using explicit quantification instead of local variables, as in:

  transitions
    output *pitch*(*n*)
      pre $n \in left \wedge \exists x : Nat\ (x \in left \wedge n < x)$;
      eff *left* := **choose** *s* **where** $\exists x : Nat\ (x \in left \wedge n < x \wedge$
          $s = delete(n, delete(x, left)))$;

In general, to eliminate local variables to which no values are assigned, one quantifies them explicitly in the precondition for the transition, and then repeats the quantified precondition as part of the effects clause.

A second use for local variables of transition definitions is as temporary variables in the effects clause, as in the following definition of an automaton that sorts an array into ascending order by swapping pairs of incorrectly ordered elements.

  automaton *Arrange*
    signature
      output *swap*(*i*, *j*: *Nat*)
    states
      A: **Array**[*Nat*, *Nat*];
    transitions
      output *swap*(*i*, *j*: *Nat*)
        locals *temp*: *Nat*;
        pre $A[i] < A[j]$;
        eff *temp* := *A*[*i*]; $A[i] := A[j]$; $A[j] := temp$;

11.5.3 Preconditions

The precondition in a transition definition is a predicate (that is, a boolean-valued expression) on the state variables, automaton parameters, action parameters, and local variables, indicating the conditions under which the transition can occur. In Tempo, preconditions can be defined for transitions of output or internal actions using the keyword **pre** followed by one or more predicates (with a

semicolon-terminated list). If no precondition is present, it is assumed to be *true*. If a precondition contains more than one predicate, it is equivalent to the conjunction of those predicates. Thus, for example, the transition of *PitchTwo* could be rewritten equivalently as:

output *pitch*(*n*)
  locals x : *Nat*;
  pre $n \in$ *left*;
    $x \in$ *left*;
    $n < x$;
  eff *left* := *delete*(*n*, *delete*(*x*, *left*));

Mathematically, an action π is said to be enabled in a state s if there is a state s' such that the triple (s, π, s') is in the transition relation of the automaton. Since input actions are enabled in every state; i.e., automata are not able to "block" input actions from occurring, transitions of input actions cannot have preconditions.

11.5.4 Effects

The effects clause in a transition definition describes the changes that occur as a result of the action. The clause can be written either in the form of a simple imperative program or as a predicate relating the pre-state and the post-state (i.e., the states before and after the action occurs). However a transition is defined, it always happens instantaneously and indivisibly. By "instantanously", we mean that a transition takes no real time; the automaton state changes from s to s' by performing an action at a particular moment in time. By "indivisibly", we mean that transitions happen one at a time, with the effects of each transition being completed before another transition begins. In fact, even more is true: the combination of precondition and effects of each transition should be atomic. That is, the precondition of a transition should evaluate to "true" and then the entire effects part is executed, before the next transition begins.

In Tempo, the effect of a transition is defined following the keyword **eff**, using a (possibly nondeterministic) program that assigns new values to state variables. Tempo assumes that state variables do not change during a transition unless they are assigned to during the execution of the effects program. In particular, if a transition definition has no effects program. then it leaves the state unchanged. The amount of nondeterminism in a transition can also be limited by a predicate relating the values of state variables in the post-state to each other and to their values in the pre-state.

Using programs to specify effects A program is a semicolon-terminated list of statements. Statements in a program are executed sequentially and the computation of the whole program constitutes an atomic step.[3] There are three kinds of statements:

- assignment statements,
- conditional statements, and
- **for** statements.

[3]The fact that the effects of the transition happen atomically does not mean that the left-hand sides of all assignments use values from the pre-state. The effects code should be executed like ordinary sequential code, one statement after another, so each statement's left-hand side can use results of previous assignments.

Assignment statements: An assignment statement changes the value of a state variable or local variable. The statement consists of a state variable or local variable followed by the assignment operator $:=$ and an expression. When a state variable is an array (see Section 14.3.1) or a tuple (see Section **??**), then terms denoting its elements or its fields can also appear on the left-hand side (*lhs*) of the assignment operator, as in the automaton *Arrange* (see page 56).

The expression following the assignment operator must have the same type as the variable on the lhs of the assignment operator. The value of this expression is determined in the state in which the assignment statement is executed, and it becomes the value of the variable on the lhs in the subsequent state. Execution of an assignment statement does not have side effects; i.e., it does not change the value of any state variable or local variable other than the one on the left side of the assignment operator.

As illustrated in the discussion of the automaton *PitchTwo* (see page 57), the expression on the right side of an assignment statement can consist of a nondeterministic **choose** clause. The value of such a clause is constrained by a predicate following the keyword **where**. If the **choose** clause does not contain the keyword **where** (as in the statement $x :=$**choose**), then it is treated as if it contained **where** *true*, and it produces an arbitrary new value of the type of the lhs variable.

Conditional statements: A conditional statement selects one of several program segments to execute in a larger program. Each conditional statement starts with the keyword **if** followed by a predicate and a **then** clause. The **then** clause contains a program segment that is executed if the predicate is true. Each conditional statement ends with the keyword **fi**. As illustrated by

if $x < y$ **then** $x := x + y$ **fi**;
if $x < y$ **then** $x := x + y$ **else** $y := x + y$; $x := x + y$ **fi**;
if $x < y$ **then** $x := x + y$ **elseif** $y < x$ **then** $y := x + y$ **fi**;
if $x < y$ **then** $x := x + y$ **elseif** $y < x$ **then** $y := x + y$ **else** $y := x$ **fi**;
if $x < y$ **then** $x := x + y$ **elseif** $y < x$ **then** $y := x + y$ **elseif** $(x + y) < z$ **then** $y := x$ **fi**;

a conditional statement can contain any number of **elseif** clauses (each of which contains a predicate and a **then** clause) and/or a final **else** clause, which also contains a program segment. The effect of executing a conditional statement is that of executing the program segment in the first **then** clause, if any, for which the preceding predicate is true and otherwise is that of executing the program segment in the **else** clause, if one exists.

For statements: A **for** statement executes a program segment once for each value of a variable that satisfies a given condition. It starts with the keyword **for** followed by a variable, a clause describing a set of values for this variable, a **do** clause that contains a program segment, and the keyword **od**.

Code Sample 31 illustrates the use of a **for** statement in a high-level description of a multicast protocol that has no timing constraints. The figure begins with the definition of a vocabulary (i.e., a set of symbols) that can be used to describe packets sent by the protocol. Elements of the *Packet* data type (see Section 14.3.8) are triples [*contents*, *source*, *dest*], in which the *contents* field represents a message, the *source* field the *Node* sending the message, and the *dest* field the set of *Node*s to which the message should be delivered. The state of the multicast algorithm consists of a multiset *network*, which represents the packets currently in transit, and an array *queue*, which represents, for each *Node*, the sequence of packets delivered to that *Node*, but not yet read by the *Node*.

```
vocabulary Packet
   types Message, Node, Packet : Tuple [contents: Message, source: Node, dest: Set[Node]]
end

automaton Multicast
   imports Packet

   signature
     input mcast(m: Message, i: Node, I: Set[Node])
     internal deliver(p: Packet)
     output read(m: Message, j: Node)

   states
     network: Mset[Packet] := ∅;
     queue: Array[Node, Seq[Packet]];
     initially ∀i: Node (queue[i] =∅)

   transitions
     input mcast(m, i, I)
       eff network := insert([m, i, I], network);

     internal deliver(p)
       pre p ∈ network;
       eff for j: Node in p.dest do
                   queue[j] := queue[j] ⊢p;
               od;
           network := delete(p, network);

     output read(m, j)
       pre queue[j] ≠∅∧head(queue[j]).contents =m;
       eff queue[j] := tail(queue[j]);
```

Code Sample 31: Tempo description of a multicast protocol

The *mcast* action inserts a new packet in the network; the tuple data type provides the notation [*m*, *i*, *I*] and the multiset data type provides the *insert* operator (see Section 11.4). The *deliver* action, which is described using a **for** statement, distributes a packet to all nodes in its destination set (by appending the packet to the queue for each destination node and then deleting the packet from the network). The *read* action receives the contents of a packet at a particular *Node* by removing that packet from its queue of delivered packets.

There are two ways to describe the set of values for the control variable in a **for** statement. The first (shown in Code Sample 31) consists of the keyword **in** followed by an expression denoting a set or multiset of values of the appropriate type, in which case the program segment in the **do** clause is executed once for each value in the set or multiset. The second consists of the keyword **where** followed by a predicate, in which case the program is executed once for each value satisfying the predicate. These executions of the program occur in an arbitrary order, and Tempo requires that the effect of a **for** statement be independent of the order in which executions of its program occur.

Using predicates to constrain effects The results of a program in the effects clause can be constrained by a predicate relating the values of state variables after a transition has occurred to the values of state variables before the transition began. For example, the transition definition for the *swap* action in the *Arrange* automaton (see page 56) can be rewritten using assignment statements to indicate that the array *A* may be modified only for certain indices (*i* and *j*) and using an **ensuring** clause to constrain the modifications. A primed state variable in this clause (i.e., *A*') indicates the value of the variable in the post-state; an unprimed state variable (i.e., *A*) indicates its value in the pre-state. As shown below, this notation allows us to eliminate the local variable *temp* needed previously for swapping.

```
transitions output swap(i, j: Nat)
  eff A[i] := choose;
      A[j] := choose;
      ensuring A'[i] = A[j] ∧ A'[j] = A[i]
```

There are important differences between **where** clauses attached to nondeterministic **choose** operators and **ensuring** clauses. A **where** clause restricts the value chosen by a **choose** operator in a single assignment statement, and variables appearing in the **where** clause denote values in the state just before the assignment statement is executed. An **ensuring** clause can be attached only to an entire **eff** clause; unprimed variables appearing in an **ensuring** clause denote values in the state before the transition represented by the entire **eff** clause occurs, and primed variables denote values in the state after the transition has occurred.

Recall that Tempo assumes that state variables do not change during a transition unless they are assigned to during execution of the effects program. When assignments include **choose** clauses, the overall result of the program may be nondeterministic; that is, it may allow several different post-states to arise from the same pre-state. Adding an **ensuring** clause allows us to restrict this nondeterminism, by limiting which post-states can arise from each pre-state.

Thus, we may say that the **choose** statements in an effects program give the transition permission to change the values of the variables on the left-hand sides. For example, if we eliminated the **choose** statements from the *swap* program above, the transition would not be allowed to change any elements of the array *A*, and so would not be able to guarantee that the **ensuring** clause could be satisfied.

11.6 Trajectories

In Tempo, instantaneous changes to state variables upon the occurence of a discrete action are specified by transition definitions that follow the keyword **transitions**. Analogously, continous changes to state variables over time are specified by trajectory definitions that follow the keyword **trajectories**.

Each trajectory definition starts with the keyword **trajdef** and has the following components:

- a trajectory definition name,
- an optional list of formal parameters and a where-clause constraining the values of the parameters,
- optional function definitions,
- an optional **invariant**,
- an optional stopping condition, and
- an optional **evolve** condition, which is a collection of Differential and Algebraic Inequalities (DAIs).

A trajectory definition defines a set of trajectories. A trajectory belongs to the set defined by a trajectory definition if every state in the trajectory satisfies the predicates in the **invariant** condition, none of the states in the trajectory, except possibly the final state, satisfies the **stop when** condition, and the evolution of the state follows the algebraic and differential equations in the **evolve** condition. We examine each of these more closely below. As in the case of transition definitions, trajectory definitions can have parameters and the values of these parameters can be constrained by where-clauses.

An automaton with no trajectory definitions is the same as a basic untimed I/O automaton in which state can change only through discrete transitions.

Multiple **trajdef** clauses can be used to define subsets of the set of all trajectories of an automaton. For example, the automaton *Thermostat* in Code Sample 32 defines two sets of trajectories, one that describes how the temperature inside a room changes when a heater in the room is off and another that describes how the temperature changes when the heater is on. The set of all trajectories for *Thermostat* is then the union of the sets defined by trajectory definitions *heaterOff* and *heaterOn*.

The automaton *ClockSync* in Code Sample 33 describes how a collection of processes, indexed by natural numbers, exchanges the values of their separate physical clocks. The parameter u determines how often process i sends its value of *physclock* (its own physical clock), which may drift from real time with a rate bounded by r. The variable *maxother* records the largest physical clock value received from the other processes in the system, and the variable *nextsend* records when process i should send the value of its physical clock to the other processes. The logical clock, *logclock*, is defined to be the maximum of *maxother* and *physclock*.

The automaton *ClockSync* has a unique trajectory definition, which defines the entire set of trajectories of this automaton.

```
automaton Thermostat(low, high, initialTemp, ambientTemp, coolingRate, heatingRate: Real)
    signature
        output turnOn, turnOff
    states
        temp: Real := initialTemp;
        isOn: Bool := initialTemp < high;
    transitions
        output turnOn
            pre temp ≤low ∧¬isOn;
            eff isOn := true;
        output turnOff
            pre temp ≥high ∧isOn;
            eff isOn := false;
    trajectories
        trajdef heaterOff
            invariant ¬isOn;
            stop when temp =low
            evolve d(temp) =coolingRate∗(ambientTemp−temp);
        trajdef heaterOn
            invariant isOn;
            stop when temp =high
            evolve d(temp) =heatingRate;
```

Code Sample 32: Tempo description of a thermostat for controlling a heater

```
automaton ClockSync(u, r: Real, i: Nat) where u > 0 ∧0 ≤r ∧r < 1
  signature
    output send(m: Real, const i)
    input receive(m: Real, j: Nat, const i) where j ≠i
  states
    nextsend: DiscreteReal := 0;
    maxother: DiscreteReal := 0;
    physclock: Real := 0;
  let logclock() : →Real =max(maxother, physclock);
  transitions
    output send(m, i)
      pre m =physclock ∧physclock =nextsend;
      eff nextsend := nextsend + u;
    input receive(m, j, i)
      eff maxother := max(maxother, m);
  trajectories
       trajdef waitSend
           stop when physclock =nextsend;
           evolve (1 − r) ≤d(physclock);
              d(physclock) ≤(1 + r);
```

Code Sample 33: Tempo description of a clock synchronization algorithm

The following example illustrates a parametric trajectory definition[4]. It describes a vehicle controlled by an external *vel*(*v*) input. The input gets recorded in a *velocity* state variable. Trajectory definition *move*(*v*) describes motion at velocity *v*, and is allowed to occur when the *velocity* variable's value is *v*.

automaton *ControlledCar*
signature
  input *vel*(*v*: *Real*)
states
  velocity: *DiscreteReal* : = 0;
  position: *Real* : = 0;
transitions
  input *vel*(*v*)
    eff *velocity* : = *v*;
trajectories
  trajdef *move*(*v*: *Real*)
    invariant *velocity* =*v*;
    evolve
      d(*position*) =*v*;

11.6.1 Invariants

An invariant for a trajectory definition can be defined using the keyword **invariant** followed by a predicate or a semicolon-terminated list of predicates on the formal parameters and state variables. In the case of multiple predicates the invariant is the conjuction of these. If no **invariant** condition is given it is assumed to be *true*.

The **invariant** condition for a trajectory definition defines the set of states whose evolution is governed by the stopping condition and **evolve** condition. States that do not satisfy a trajectory definition's **invariant** condition cannot appear anywhere in a trajectory described by that definition.

The **invariant** conditions in trajectory definitions *heaterOff* and *HeaterOn* in *Thermostat* in Code Sample 32 are simple predicates involving a single variable *isOn*. When the value of *isOn* is false, the evolution of the analog variable *temp* is governed by the **evolve** and stopping conditions of the trajectory definition *HeaterOff*. Similarly, when *isOn* is true, the evolution of *temp* is governed by the conditions of the definition *heaterOn*.

11.6.2 Stopping conditions

A stopping condition for a trajectory definition can be defined using the keywords **stop when** followed by a predicate on the automaton parameters, trajdef parameters, and the state variables. If no stopping condition is given, it is assumed to be false.

If the stopping condition for a trajectory definition is satisfied at a given point in time t of a trajectory τ, then t must be the end-point of τ. This means that time must stop at that point, allowing enabled actions to occur. Note that time need not advance until the point where the stopping condition is true: The automaton can choose to perform an output or internal action at a certain point in time even though the stopping condition is not true at that time; moreover, the

[4]While Tempo supports the definition of parametric trajectories, none of the backends can currently take advantage of them. Future versions of the suite will improve the backends

trajectory can be interrupted at any time by an input action. What a stopping condition expresses is simply that time cannot advance beyond the point where it becomes true.

In automaton *Thermostat*, the stopping condition in trajectory definition *heaterOff* imposes the condition that time stops when the value of *temp* is equal to the value of *low*. This means time cannot advance any longer. Note that when this stopping condition becomes true and time stops, *Thermostat* will be able to perform a *turnOn* action. It is possible to modify the *Thermostat* automaton so that it allows a *turnOn* action to be performed any time when the value of *temp* is less than *high*, and analogously for *turnOff*:

output *turnOn*
  pre $temp < high \wedge \neg isOn$;
  eff $isOn := true$;
output *turnOff*
  pre $temp > low \wedge isOn$;
  eff $isOn := false$;

This modified automaton still does not allow *temp* to decrease below *low*, or increase above *high*.

In automaton *ClockSync* periodic sending of the value of *physclock* to other processes is enforced by the stopping condition: time is not allowed to pass beyond the point at which $physclock = nextsend$.

11.6.3 DAIs

The behavior of analog variables in a trajectory definition is specified using a semicolon-terminated list of Differential and Algebraic Inequalities (DAIs) following the keyword **evolve**. Discrete variables need not be mentioned in **evolve** conditions, because we assume that they remain constant during each trajectory. The DAIs constrain the set of trajectories, and thereby specify how the analog variables are allowed to evolve over time. If an analog variable does not occur in any DAI then it is assumed to evolve according to any arbitrary continuous function, constrained only by its dynamic type.

In automaton *Thermostat*, when the heater is off the rate of change of *temp* with respect to real time is $coolingRate*(ambientTemp-temp)$. The notation $d(temp)$ is used to specify rate with respect to real time. Here, the *ambientTemp* is the temperature outside the room; thus, the higher the temperature is initially, the faster it cools towards the ambient temperature. When the heater is on, the temperature rises linearly with elapsed time, as specified by the **evolve** condition $d(temp) = heatingRate$. In automaton *ClockSync* the **evolve** condition states that the variable *physclock* changes continuously, with a drift rate bounded by r.

Since **invariant** conditions, stopping conditions, and **evolve** conditions are optional it is possible for an automaton to have an empty trajectory definition. Recall that in this case the default **invariant** condition is *true* and the default stopping condition is *false*. Hence, an empty trajectory definition defines a set of trajectories in which any amount of time is allowed to pass, discrete variables remain constant and other variables are allowed to changed arbitrarily, according to their dynamic types.

11.7 User-defined functions

Tempo allows users to define their own functions to use in specifications. A function definition can occur as a stand-alone specification unit outside the automaton (a global function), in an automaton following state variables (an automaton-wide function), in a transition definition following

local variables and before the preconditions, or within a trajectory definition after its name and parameters.

A function definition begins with the keyword **let** and consists of a name, an optional list of formal parameters, and an expression defining the body of the function. In Tempo, the type of a function must be provided explicitly; it cannot be inferred automatically. Functions are allowed to be recursive. The following code fragment shows how the factorial function can be defined in Tempo. The type annotation shows that *fact* is a function from type *Nat* to type *Nat*. Individual type annotations for formal parameters are not required.

```
let fact(n): Nat →Nat =
    if n =0 then 1
    else n ∗ fact(n−1);
```

The scope of a function and the identifiers that can occur in a function naturally depend on where in the code the function is defined. Each constant, operator, or variable appearing in the expression defining the body of the function must be defined in the current context. For example, if a function is defined after the state variables of an automaton, then it can refer to functions defined outside the automaton, and to the automaton's formal parameters and state variables. If a function is defined within a transition definition, then it can refer to all of these and additionally to the parameters of that transition definition. Each transition definition or trajectory definition has a distinct scope, that is, functions that are defined in separate transition definitions or trajectory definitions can refer to global functions or automaton-wide functions but not to one another.

```
let dist(x1,x2,y1,y2): Real,Real,Real,Real→Real =sqrt((x2 − x1)∗(x2 − x1)+(y2 − y1)∗(y2 −y1));
automaton Track
  signature
    input TargetUpdate(x,y: Real)
  states
    pos: Tuple[x: Real, y: Real];
    togo: AugmentedReal := ∞;
  let distToTarget(x,y) : Real,Real→Real =dist(x,pos.x,y,pos.y);
  transitions
    input TargetUpdate(x,y)
        eff togo := distToTarget(x,y);
```

In the code fragment above, the variable *pos* represents the position of a vehicle on a plane. The evolution of *pos* is not of interest to us and hence is not shown. The vehicle receives information about the location of targets through the input action *TargetUpdate*(*x*,*y*) and updates the variable *togo* with the value of its current distance to the most recently detected target (*x*,*y*). The function *dist* returns the Euclidean distance between any two points (*x1*,*y1*) and (*x2*,*y2*). It is a global function and does not use any of the automaton variables. The function *distToTarget* returns the distance of any point (*x*,*y*) to *pos*. This function uses *dist* and the state variable *pos*. This is legal because its definition appears *after* the **states** section of the code and so the state variable *pos* is in its scope. The action *TargetUpdate*(*x*,*y*) updates the variable *togo* with the distance of *pos* to (*x*,*y*) as computed by *distToTarget*.

When a function definition occurs within an automaton definition, just after state variables, such as *distToTarget* in the above fragment, each formal parameter of the function must differ from the formal parameters and variables of the automaton. Functions that are defined within an automaton after state variables can refer to other functions that are defined outside the automaton. Likewise, when a function definition occurs within a transition or trajectory definition, each formal

parameter of the function must differ from any parameter of that transition or trajectory definition.

In the case of multiple function definitions within a given scope, it matters whether a function is textually defined before another. Consider the following automaton-wide functions.

let *addone*(n) : *Int* →*Int* =n+1;
let *bar*(m) : *Int* →*Int* =*div*(*addone*(m),2);

Here, the function *bar* can call *addone*, which is defined before it, but not vice versa. Mutual recursion is not supported in Tempo.

12 Operations on Automata

The Timed I/O Automata mathematical model includes two separate operations on automata: *parallel composition* and *action hiding*. See Chapter 7 of [2] for formal definitions of both operations.

The composition operation combines a collection of TIOAs into a single TIOA. The automata being combined must satisfy certain compatibility conditions, namely, that no action is an output of more than one automaton, and that no internal action of any automaton is shared with any other automaton. The set of states of the composition is the Cartesian product of the sets of states of the component automata, and likewise for the start states. Composition identifies actions with the same name in different component automata. That is, when any component automaton performs a step involving an action π, so do all component automata that have π in their signatures. Formally, a triple (s, π, s') is in the transition relation for the composite automaton if, for every component automaton C , (s_C, π, s'_C) is a transition of C when π is an action of C and $s_C = s'_C$ when π is not an action of C. An action is classified as an output action of the composition if it is an output action of some component automaton. An action is an input action of the composition if it is an input action of some component automaton, but not an output action of any component. Finally, an action is an internal action of the composition if it is an internal action of some component automaton.

The hiding operation "hides" output actions of a TIOA by reclassifying them as internal actions. This prevents them from being used for further communication and means that they are no longer included in traces.

In Tempo, we use just one construct to express both operations. This construct allows us to specify a collection of automata to be composed, and then to specify which (if any) output actions of the combination are to be hidden. We can use this combined construct to express parallel composition alone, by not hiding any output actions. We can also use it to express action hiding alone, by "composing" a collection consisting of just a single automaton before hiding actions.

An example of the use of this combined operation is the following, which is extracted from the leader-election example in Section 7.

automaton *LeaderSystem*(*delta*,*e*,*u*: *Real*) **where** *delta* $> 0 \wedge e > 0 \wedge u > 0$
  components
    E[*j*: *J*]: *Elect*(*j*, *delta*, *e*, *u*);
    FD: *FailureDetector*(*delta*);
  hidden
    status(*L*,*j*);

The *LeaderSystem* automaton consists of a collection of *Elect* automata, one for each j in a given set J of process identifiers, plus one *FailureDetector* automaton, which provides the *Elect* automata with information about process failures. The *Elect* automata are defined in Figure 15, and the

FailureDetector automaton in Figure 16. The *status*(*L*,*j*) actions, by which the failure detector notifies the processes about failures, are hidden in the final *LeaderSystem*.

In general, a Tempo specification of a composite automaton begins with the keyword **automaton** followed by the name of the automaton. This may in turn be followed by a list of formal parameters, and an optional **where** clause which may constrain the values of these parameters.

The *LeaderSystem* automaton takes three non-negative real numbers as arguments. *delta* is an upper bound on the amount of time that can elapse between two reports produced by the failure detector. u is an upper bound on the time between two "I'm the leader" announcements by an elected *Elect* automaton, and e is used to define deadlines for elections within an *Elect* automaton. The *LeaderSystem* automaton uses a **where** clause to ensure that all three constants are positive.

The Tempo specification of a composite automaton continues with the **components** keyword, followed by a semicolon-terminated list of the automata involved in the composition. Each element in this list can describe either a single automaton or an indexed collection of automata.

In the case of a single automaton, the list element contains a description of the component, in terms of a previously-defined automaton. The list element also contains a new name that can be used to refer to the component within the context of the composite automaton. This name is useful in carrying out proofs about the composite automaton, and in specifying schedules for simulating the composite automaton. The name precedes the component description, with a colon intervening. The case of an indexed collection of automata is similar: the list element contains a description of the collection of automata, and a collection of new names by which they can be referenced within the composite automaton.

In the *LeaderSystem* automaton, the list of components has two lines, one for all of the *Elect* automata and one for the single 'FailureDetector' automaton. The *LeaderSystem* requires one instance of *Elect* for each process identifier. The line

E[*j* : *J*] : *Elect*(*j*,*delta*,*e*,*u*);

creates a map E containing one automaton for each value j in the set J of process identifiers. These automata are defined after the colon. The automaton $E[j]$ is defined to be an instance of *Elect* with four arguments: the process identifier (j) and the three timing constants $delta, e, u$. For an example of how this can be used, suppose that $J = \{1, 2, 3, 4, 5\}$. Then the expression $E[3].live$ denotes the value of the *live* state variable, in the automaton *E*[3] =*Elect*(3,*delta*,*e*,*u*). This represents process 3's view of the set of live processes.

The line

FD: *FailureDetector*(*delta*);

defines the final component of *LeaderSystem*: an instance of the *FailureDetector* automaton, which is referred to as *FD*.

Tempo also allows a list entry for an indexed collection of automata to include a **where** clause; this enables selection of automata whose indices satisfy some property. Any boolean formula can be used in the **where** clause. For instance, if $J = \{1, 2, 3, 4, 5\}$, then

E[*j* : *J*] : *Elect*(*j*,*delta*,*e*,*u*) **where** *mod*(*j*,2) =0;

would populate the map E with two automata, with identifiers 2 and 4.

To express action hiding, Tempo uses an optional section that begins with the keyword **hidden**, and continues with a semicolon-terminated list of actions to be hidden. Each list entry must include an action name and may have a list of arguments. These arguments must be matched by number and position with the arguments of the same-named actions of the component automata.

The specification of *LeaderSystem* contains the fragment

hidden
status(*L*,*j*);

which says that the *status* actions of all the components should be made internal. The *status* actions are outputs of the *FailureDetector* component automaton, in which they are declared as follows:

output *status*(*L*: **Set**[*J*], *j*: *J*)

The two arguments for *status* in the *LeaderSystem* definition correspond with those in the *FailureDetector* definition. However, the actual names for the arguments in *LeaderSystem* are immaterial; for example, we could have written

hidden
status(*a*,*b*)

with the same meaning.

Tempo also allows a list entry for a parameterized collection of actions to include a **where** clause; this enables selection of certain instances of the action to be hidden. For instance, the statement

hidden
status(*L*,*j*) **where** $j \setminus not \in L$;

would hide the status action only in the special case when j is not a member of L, that is, when j is being notified that it is not live.

13 Invariants and Simulation Relations

Invariants and simulation relations are two of the most important concepts used in reasoning about TIOAs. Invariants are used in stating properties of the reachable states of a single automaton, and simulation relations are used in proving an implementation relationship between two automata, by relating the states of the two automata.

13.1 Invariants

An *invariant* of a Timed I/O Automaton is a property that is true in all reachable states of the automaton. In Tempo, it is possible to specify invariants for automata. An invariant of an automaton starts with the phrase **invariant of**, followed by the name of the automaton, followed by a semicolon-terminated list of first-order predicates that can refer to state variables and formal parameters of the automaton. For example,

invariant of *TimedChannel*:
$\forall i$: *Nat*: (*queue* $\neq \setminus$*emptyset* $\wedge 1 \leq i \leq$ *len*(*queue*) $\Rightarrow$
queue[*i*].*deadline* $\geq$ *now*);
$\forall i$: *Nat* $\forall j$: *Nat* ($1 \leq i \wedge i \leq j \wedge j \leq$ *len*(*queue*) $\Rightarrow$
queue[*i*].*deadline* $\leq$ *queue*[*j*].*deadline*);

specifies that the deadlines that occur in a packet queue of a *TimedChannel* (Figure 28) are all in the future, and that the deadlines occur in nondecreasing order.

Tempo also allows multiple invariants of the same automaton to be written separately (and possibly named), instead of using a list. For example, the invariant of TimedChannel above can be broken down into two separate invariants:

invariant *A1* **of** *TimedChannel*:
$\forall i$: *Nat*: (*queue* $\neq \setminus$*emptyset* $\wedge 1 \leq i \leq$ *len*(*queue*) $\Rightarrow$

queue[*i*].*deadline* ≥*now*);
invariant *A2* **of** *TimedChannel*:
∀*i*: *Nat* ∀*j*: *Nat* (1 ≤*i* ∧*i* ≤*j* ∧*j* ≤*len*(*queue*) ⇒
queue[*i*].*deadline* ≤*queue*[*j*].*deadline*);

When the effects of an automaton's transition are defined by a program that consists of more than one statement, then the states between the executions of those statements are intermediate states, which do not appear in the execution fragments of the automaton. Invariants are not required to be true in such intermediate states, (unless they are also reachable). For example, consider the automaton *fischer* from Figure 1 and the transition definition for *test*(*i*). The second invariant from Figure 5 need not hold in the intermediate state that appears after setting *pc*[*i*] to *pc_set* but before changing the value of *last_set*[*i*].

13.2 Simulation relations

An automaton A is said to *implement* an automaton B provided that A and B have the same input and output actions and that every trace of A is also a trace of B. In order to show that A implements B, one can use a *simulation relation* between states of A and states of B (see Section 3.3 for a high-level explanation of simulation relations and [2] for formal definitions).

Suppose that A and B have the same input and output actions. A relation R between the states of A and B is a *forward simulation* if

- every start state of A is related (via R) to some start state of B,
- for every state s of A and every state u of B such that $R(s, u)$, and for every discrete step (s, π, s') of A, there is an execution fragment α of B starting with u, that has the same trace as π and that ends with a state u' such that $R(s', u')$, and
- for every state s of A and every state u of B such that $R(s, u)$, and for every trajectory τ of A starting with s, there is an execution fragment α of B starting with u that has the same trace as τ and that ends with a state u' such that $R(s', u')$.

A general theorem is that A implements B if there is a forward simulation from A to B (see Chapter 4 of [2].

A Tempo specification of a forward simulation begins with the keywords **forward simulation**, followed by a name for the simulation relation. This name may be followed by a list of formal parameters and an optional **where** clause involving these parameters. It continues with descriptions of the two automata involved in the simulation. The "lower-level" automaton (A in the forward simulation definition above) is specified using the keyword **from**, followed by a short name for the automaton, followed by a colon and a description of the automaton. Similarly, the "higher-level" automaton (B above) is specified using the keyword **to**, followed by a short name, a colon, and a description of the automaton. We assume that the automata are defined elsewhere; their formal parameters are replaced by actuals that are computed from the formal parameters of the forward simulation.

Short names are used within the context of the forward simulation, for instance, in designating state variables of the two automata. This use of short names is similar to their use in composition (see Section 12).

The Tempo specification of a forward simulation then continues with the keyword **mapping**, followed by the definition of the actual mapping. This definition is a first-order predicate involving

the formal parameters of the forward simulation and the state variables of the two automata. Whenever a state variable appears in the mapping, it is prefaced by the short name of the automaton to which it belongs.

The mapping can also be a semicolon-terminated list of predicates; as elsewhere, this is interpreted as the conjunction of the individual predicates.

Consider two instances of *TimedChannel* (Figure 28), *TimedChannel*(2,*M*) and *TimedChannel*(3,*M*), where *M* is a predefined message alphabet. In a trace of *TimedChannel*(2,*M*), the time that elapses between the placement of a packet in the queue and its removal can be at most 2 time units. Clearly, any such trace would also be a trace of *TimedChannel*(3,*M*), which allows packets to remain in the queue for up to 3 time units. This implementation relationship can be proved by showing the existence of a forward simulation relation from *TimedChannel*(2,*M*) to *TimedChannel*(3,*M*). The following is the definition of a forward simulation relation F from *TimedChannel*(2,*M*) to *TimedChannel*(3,*M*):

forward simulation *F*
  from *TC1*: *TimedChannel*(2,*M*)
  to *TC2*: *TimedChannel*(3,*M*)
mapping
  TC1.now = *TC2.now*
  ∧ *len*(*TC1.queue*) = *len*(*TC2.queue*)
  ∧ ∀ *i*: *Nat*: 1 ≤ *i* ≤ *len*(*TC1*).*queue*:
    TC1.queue[*i*] = [*m*,*u1*] ⇒
      (∃ *u2*: *Nat* *TC2.queue*[*i*] = [*m*,*u2*] ∧ *u1* ≤ *u2*)
end

A state x of *TimedChannel*(2,*M*) is related to a state y of *TimedChannel*(3,*M*) by the relation F if all of the conditions in the mapping part are met. Namely, the value of the variable *now* must be the same in both x and y, the packet queues must be of the same length, and if a message m in position i is associated with a particular deadline $u1$ in x, then the ith position in the packet queue of y must contain the same massage m associated with a deadline $u2$ that is at least as large as $u1$. Note that we use *TC1* and *TC2* as short names for *TimedChannel*(2,*M*) and *TimedChannel*(3,*M*) in the mapping part.

We may also want to define a forward simulation from one automaton to another without fixing their parameters. For example, in the case of timed channels, it should be clear that a timed channel with maximum delay $r1$ should implement any other timed channel with the same message alphabet that has a maximum delay that is greater than or equal to $r1$. In other words, we may want to define a forward simulation from *TimedChannel*(*r1*,*M*) to *TimedChannel*(*r2*,*M*), for any case where $r1 \leq r2$. Here, we do not fix the parameters $r1$ and $r2$ as we did in the example above, but just constrain their values so that the forward simulation is defined only for those instances where $r1 \leq r2$.

forward simulation *F*(*r1*,*r2*)
  where *r1* ≤ *r2*
  from *TC1*: *TimedChannel*(*r1*,*M*)
  to *TC2*: *TimedChannel*(*r2*,*M*)
mapping
  TC1.now = *TC2.now*
  ∧ *len*(*TC1.queue*) = *len*(*TC2.queue*)
  ∧ ∀ *i*: *Nat*: 1 ≤ *i* ≤ *len*(*TC1*).*queue*,
    TC1.queue[*i*] = [*m*,*u1*] ⇒
      (∃ *u2*: *Nat* *TC2.queue*[*i*] = [*m*,*u2*] ∧ *u1* ≤ *u2*)

end

The forward simulation relation of 9, for the *TwoTaskRace* example, is also an example of this kind.

Defining a forward simulation relation in Tempo consitutes an important step toward proving an implementation relationship. After defining a forward simulation relation, one must still prove that the defined relation is indeed a simulation relation. This requires establishing the three conditions in the definition of a simulation relation. Specifically, one must show that the relation holds in the start state and is preserved by discrete transitions and trajectories.

The Tempo Toolset includes a simulator that can check empirically that a defined relation is actually a forward simulation relation. It does this by following a "paired simulation" strategy, executing two automata—one representing an implementation and the other the abstract specification—together. The Tempo language supports writing "schedules" and "proof blocks" that are used by the simulator in executing automata and in checking simulation relations. Proof blocks specify how the individual steps of an automaton representing an implementation can be matched by steps of an automaton representing the abstract specification, so that the conditions of the simulation relation can be satisfied. The details of this part of the language can be found in the documentation for the simulator.

14 Data types in Tempo

Tempo supports the notions of static type and dynamic type in tandem with the underlying theoretical framework. Each variable has a static type, which specifies the values that the variable can take on. This corresponds to the standard notion of data type in languages and in the rest of this section we often use the term data type and static type interchangeably. Dynamic types are functions that describe the evolution of a variable over time; for example, they specify whether a variable remains constant during trajectories or may change value continuously.

Tempo requires its users to declare the static types of variables explicitly. The dynamic type of a variable, on the other hand, is inferred from its static type. The semantic analysis of a TIOA specification by Tempo uses the static types of declared objects in the specification to verify its correctness. Formally, Tempo uses a *typing context* $\Gamma : \mathcal{I} \rightarrow \mathcal{T}$ that maps identifier to static types, i.e., Γ is of the form $\Gamma = [i_0 \mapsto T_0, i_1 \mapsto T_1, \cdots, i_n \mapsto T_n]$ where $i_0 \cdots i_n$ are type identifiers drawn from $\mathcal{I}$ while T_0 through T_n are static types drawn from $\mathcal{T}$. Γ contains, at the very least, all the primitive data types (i.e., *Bool*,*Nat*, *Int*, *Real*, *Char*, and *String*) defined in Subsection 14.1. Subsection 14.3 shows how to define additional types with constructors (i.e., **Array**, **Map**, **Mset**, **Null**, **Seq**, **Set**, **Enumeration**, **Tuple**, and **Union**) whereas Subsection 14.4 shows how to extend the typing context through *aliases*. Subsection 14.5 shows how abstract data types (the combination of static types and operations) can be defined with vocabularies.

Note that Tempo augments Γ with two other built-in data types: *DiscreteReal* whose *static type* is identical to *Real* but with a *dynamic type* equal to the set of piecewise continuous real- valued functions. *DiscreteReal* is typically used with discrete variables only. *AugmentedReal* simply adds two elements, $+\infty$ and $-\infty$ to the type *Real*. *AugmentedReal* variables are used, for example, as discrete *deadline variables*, which are used to impose upper bounds on the time of occurrence of actions. When the automaton performs discrete transitions, this upper bound may change. When we want to remove a deadline for an action we set the deadline variable for it to $+\infty$. *DiscreteReal* variables are used, for example, as *earliest time variables*, which impose lower bounds on the time of occurrence of actions. Since a default lower bound can be 0, we don't need to use *AugmentedReal*

variables for this purpose. See, for instance, the *fischer* example in Section 4) for such uses of deadline and earliest time variables.

14.1 Primitive data types

14.1.1 Booleans

The data type *Bool* has two elements, *true* and *false*, which are called *boolean* or *logical* values. The following notations may be used to denote boolean values or functions that can be applied to boolean values p and q. In the table, rows $3-7$ denote boolean operators. The column labeled **Alternate** gives the Tempo notation while columns **Prec** and **Assoc** respectively provide the operator precedence and associativity. Note that a higher precedence is denoted by a larger numerical value. For instance, the expression `a \/ b /\ c` will be interpreted as $a \vee (b \wedge c)$ as the disjunction has a lower precedence than the conjunction. These conventions will be used throughout the rest of the document.

Symbol	Alternate	Prec.	Assoc.	Sample use	Meaning
true				*true*	The logical value *true*
false				*false*	The logical value *false*
$\neg$	`~`	6		$\neg p$	Negation (not)
$\wedge$	`/\`	3	left	$p \wedge q$	Conjunction (and)
$\vee$	`\/`	2	left	$p \vee q$	Disjunction (or)
$\Rightarrow$	`=>`	1	left	$p \Rightarrow q$	Implication (implies)
$\Leftrightarrow$	`<=>`	1	left	$p \Leftrightarrow q$	Logical equivalence (if and only if)

The following operators can be used to denote boolean values that result from binary equality or disequality testing and apply to values x and y, both of which must have the same type. The table also illustrates how to define first-order boolean expressions with either universal or existential quantifiers.

Symbol	Alternate	Prec.	Assoc.	Sample use	Meaning
$=$		1	left	$x = y$	Equal to
$\neq$	`~=`	1	left	$x \neq y$	Not equal to
$\forall$	`\A`	0	prefix	$\forall n{:}Nat\ \neg(n < 0)$	For all
$\exists$	`\E`	0	prefix	$\exists i{:}Int\ (i < 0)$	There exists

Recall that the operator $\neg$ has higher precedence than the operators $\wedge$ and $\vee$, which themselves bind more tightly than $\Rightarrow$, which binds more tightly than $\Leftrightarrow$. Thus, $\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$ abbreviates the fully parenthesized expression $(\neg(p \wedge q)) \Leftrightarrow ((\neg p) \wedge (\neg q))$. Naturally, parentheses are required to distinguish $(p \Rightarrow p) \wedge q$, which always has the value q, from $p \Rightarrow p \wedge q$, which abbreviates $p \Rightarrow (p \wedge q)$ and always has the value *true*.

14.1.2 Natural numbers

The elements of the data type *Nat* are the non-negative integers 0, 1, 2, ..., which are called *natural numbers* or $\mathbb{N}$ for short. The following notations may be used to denote natural numbers, operators or functions that can be applied to natural numbers x, y, and z.

Symbol	Alternate	Prec.	Assoc.	Sample use	Meaning
0, 1, ...				123	Natural numbers
succ			fun	*succ*(x)	Successor (*succ*(x) $=x+1$)
pred			fun	*pred*(x)	Predecessor (*pred*(*succ*(x)) $=x$)
$+$		5	left	$x + y + z$	Addition
$-$	-	5	left	$x - y$	Subtraction (undefined if $x < y$)
$*$		6	left	$x * (y{**}z)$	Multiplication, exponentiation
$**$		7	right	$x ** y$	exponentiation x^y
min, *max*			fun	*min*(x, y)	Minimum, maximum
div, *mod*			fun	*mod*(x, y)	Quotient, modulus
$<$, $\leq$	$<$, <=	4	left	$x \leq y$	Less than (or equal to)
$>$, $\geq$	$>$, >=	4	left	$x > y$	Greater than (or equal to)
$=$, $\neq$	$=$, ~=	4	left	$x = y$	Equal to, not equal to

The values of the functions *div*(x, y) and *mod*(x, y) are defined when $y > 0$, in which case *mod*(x, y) $< y$ and $x = y*div(x, y) + mod(x, y)$. The binary operators follow the traditional precedence and associativity and $**$ has the highest precedence. Note also that the exponentiation operator is right-associative. For instance, the expression $x+y*z**w$ stands for

$$x + (y * z^w)$$

Parentheses can be used to override the default precedence and associativity rules. Some operators like *div*, *mod*, *min* or *max* use a prefixed functional notation and therefore require parentheses around the list of arguments.

14.1.3 Integers

The elements of the data type *Int* are the integers .., -2, -1, 0, 1, 2, ... All the notations applicable to natural numbers have the same meaning when those natural numbers are considered as integers, and they have suitably extended meanings when used with integers. In particular, the value of $x - y$ is always defined (and is an integer) if x and y are integers, but the value of $x**y$ is not an integer if y is negative.

The following additional notations may be used to denote functions that can be applied to integers.

Symbol	Alternate	Prec.	Assoc.	Sample use	Meaning
$-$	-	8		$-x$	Additive inverse (unary minus)
abs			fun	*abs*(x)	Absolute value

Syntactically, a numeric constant (e.g., 0 or 1) can denote either a natural number or an integer (or even a real number). In many cases, it makes no difference which of these it denotes, because Tempo treats the natural numbers as a subset (subtype) of the integers and the integers as a subset (subtype) of the real numbers. In other cases, it does make a difference. For example, the type of the expression $x**y$ will be a natural number if both x and y are natural numbers, an integer if x is an integer and y is a natural number, and a real number if x is not an integer or y is negative. Tempo usually determines the types of numeric constants from the contexts in which they appear.

14.1.4 Real numbers

The elements of the data type *Real* are real numbers and belong to $\mathbb{R}$. All notations (except *succ*, *pred*, *div*, and *mod*) that can be used with integers have the same meaning when those integers are considered as real numbers, and they have suitably extended meanings when used with real numbers. In addition, x/y denotes the value of x divided by that of y, with the exception that the value of $x/0$ is undefined, and *floor*(x) denotes the largest integer that is less than or equal to x.

Discrete vs. Analog In Tempo, the static type *DiscreteReal* is identical to the static type *Real*. The only difference is the *dynamic type* that Tempo associates with variables of this type. A variable with static type *DiscreteReal* is a discrete variable, whereas one with type *Real* is an analog variable.

Basic vs. Augmented The elements of the type *AugmentedReal* are the real numbers plus two additional elements $+\infty$ and $-\infty$. Tempo extends the numerical type hierarchy and considers that the *Real* type is a subtype (subset of values) of *AugmentedReal*.

Vocabulary definitions Tempo relies on two vocabularies to define the operations authorized on *Real* and *AugmentedReal*. The definition of the *Real* vocabulary is

vocabulary *Real*
  types *Real*
  operators
    $-\_\_$, *abs*: *Real* $\rightarrow$*Real*,
    $\_\_**\_\_$: *Real*, *Int* $\rightarrow$*Real*,
    floor: *Real* $\rightarrow$*Int*,
    $\_\_+\_\_$, $\_\_-\_\_$, $\_\_*\_\_$, $\_\_/\_\_$, *min*, *max*: *Real*, *Real* $\rightarrow$*Real*,
    $\_\_ < \_\_$, $\_\_\leq\_\_$, $\_\_ > \_\_$, $\_\_\geq\_\_$, $\_\_ =\_\_$, $\_\_\neq\_\_$: *Real*, *Real* $\rightarrow$*Bool*
end

and the definition of the *AugmentedReal* vocabulary is

vocabulary *AugmentedReal*
  types *AugmentedReal*
  operators
    ∞: $\rightarrow$*AugmentedReal*,
    $-\_\_$, *abs*: *AugmentedReal* $\rightarrow$*AugmentedReal*,
    $\_\_**\_\_$: *AugmentedReal*, *Int* $\rightarrow$*AugmentedReal*,
    floor: *AugmentedReal* $\rightarrow$*Int*,
    $\_\_+\_\_$, $\_\_-\_\_$, $\_\_*\_\_$, $\_\_/\_\_$, *min*, *max*: *AugmentedReal*, *AugmentedReal* $\rightarrow$*AugmentedReal*,
    $\_\_ < \_\_$, $\_\_\leq\_\_$, $\_\_ > \_\_$, $\_\_\geq\_\_$, $\_\_ =\_\_$, $\_\_\neq\_\_$: *AugmentedReal*, *AugmentedReal* $\rightarrow$*Bool*
end

Note that Tempo automatically imports the relevant built-in vocabularies as soon as the specification being checked refers to the corresponding type. The correspondence relies on the file naming convention. For instance, the *AugmentealReal* vocabulary defines the *AugmentedReal* type and is stored in a (built-in) file named `AugmentedReal.tioa`. Consequently, Tempo users should not use `Real,Int,Nat,AugmentedReal,Bool,Seq,Set,Mset,Char,String` as Tempo filenames as these would interfere with the automatic loading.

14.1.5 Characters

The elements of the data type *Char* are the characters for letters and digits (Future versions of Tempo may introduce other elements in this data type.) The following notations may be used to denote characters.

Symbol	Alternate	Sample use	Meaning
'A', ..., 'Z'		'J'	Uppercase letters
'a', ..., 'z'		'j'	Lowercase letters
'0', ..., '9'		'7'	Digits
$<$, $\leq$, $>$, $\geq$	<, <=, >, >=	'A' $<$ 'Z'	Alphabetic ordering

14.1.6 Strings

The elements of the data type *String* are sequences of characters. All notations that can be used with the data type **Seq**[*Char*] (see Section 14.3.5 can also be used with the *String* data type. In addition, the symbols $<$, $\leq$, $>$, and $\geq$ represent the lexicographic ordering. String literals can be specified as a sequence of characters enclosed in double quotes as in `"hello"`.

14.2 Casting

Each primitive type supported by Tempo is associated to a static type and is part of a simple type hierarchy shown in Figure 14.2 where the most general type is *AugmentedReal* and the upward arrows indicate sub-typing ($S <: T$ means S is a sub-type of T) relationships. Tempo uses a sub-typing subsumption rule to determine the validity of specific statements. The subsumption rule

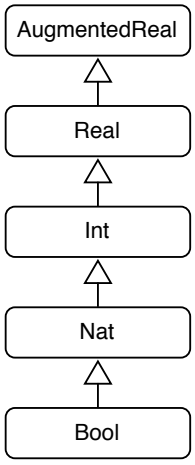

Figure 14.2: Primitive data types hierarchy.

$$\frac{\Gamma \vdash e : S, S <: T}{\Gamma \vdash e : T}$$

states that whenever the expression e has static type S and S is a sub-type of T within the typing context Γ, then e also has type T within the same context Γ. For instance, the fragment

x : *Real* : = 4;

is well-typed simply because 4 is an *Int* and by subsumption 4 is also *Real*, which matches the expected type of the left-hand side declaration of x, i.e., *Real*. The type conversion taking place here is an implicit upcast which is always safe. Sometimes, it is necessary to convert a value from an abstract to a more specialized numeric type. This operation is commonly known as a *downcast* and Tempo supports it. For instance, the statement

x : *Nat* : = (*Nat*)4.0;

first converts a value (4.0) from the static type *Real* to the static type *Nat* and assigns the resulting value to the variable x of type *Nat*. Thanks to the type cast operation, the statement is well typed. In general, the expression $(T)e$ denotes a type casting and must satisfy the typing rule

$$\frac{\Gamma \vdash e : S, T <: S}{\Gamma \vdash e : T}$$

which states that whenever expression e is of type S and T is a sub-type of S within the typing context Γ, the value denoted by e can be *converted* to a value of type T, although a loss of precision is possible. Clearly, casting 3.1415 to type *Nat* will result in an unspecified behavior (the current implementation rounds the value to the nearest natural number) and it would always be wise to safeguard the casting operation with tests or rounding operations as in

x : *Nat* : = (*Nat*)*floor*(3.1415);

Casting operators have the highest precedence among all Tempo operators and it is therefore advisable to use parentheses to ensure a suitable grouping. For instance, the statement

x : *Nat* : = (*Nat*)(5.0 + *floor*(3.1415));

first computes $\lfloor 3.1415 \rfloor$ and adds the real number 5.0 to obtain the real 8.0 before converting it to the natural 8. If the parentheses are omitted, the fragment is not well-typed. Indeed, 5.0 would first be converted to a *Nat*, then promoted to a *Real* given that the binary + operator is only defined for pairs of values of the same static type and *floor*(3.14) is of type *Real*. The result of the addition is therefore a *Real* and cannot be safely assigned to x that happens to be of type *Nat*.

14.3 Type constructors

The following type constructors and operators require no declaration.

14.3.1 Arrays

For each $n > 0$, the elements of the data type **Array**[*T1*,..,*Tn*, *E*] are the n-dimensional arrays of elements of type E indexed by elements of types *T1*, ..., *Tn*. The following notations may be used to denote arrays or functions that can be applied to arrays A and B of types **Array**[*I*, *E*] and **Array**[*I*, *J*, *E*].

Symbol	Sample use	Meaning
constant	*constant*(*e*)	Array with all elements equal to *e*
...[..., ..., ...]	*A*[*i*]	Element of *A* indexed by *i*:*I*
	B[*i*, *j*]	Element of *B* indexed by *i*:*I* and *j*:*J*
...[..., ..., ...] := e	*B*[*i*,*j*] : =*e*	Replaces *B* by *B*' where *B*' equal to *B* except that *B*'[*i*, *j*] =*e*

The dimension of the array denoted by *constant*(*e*) is inferred from its context. For instance, given the expression *constant*(*e*)[*i*], Tempo determines that the array is one-dimentional, is indexed by values drawn from the type of *i* and all its elements have the type of expression *e*, i.e., the static type of the array is derived according to the typing rule

$$\frac{\Gamma \vdash e : E, i : T}{\Gamma \vdash constant(e) : Array[T, E]}$$

which states that, within the typing context Γ, if *e* has type *E* and the index *i* has type *T*, then within the same context Γ, the expression *constant*(*e*) denotes a one-dimentional array and its type is **Array**[*T*,*E*]. Arrays and matrices can be modified easily. For instance the fragment

transitions output *write*(*i* : *Int*,*v* : *Int*)
eff *x*[*i*] : = *v*;

performs an assignment that modifies the entry *i* of the array *x* so that it now refers to value *v*. Note that the assignment can be thought of as a sequence of two operations: one that non-destructively creates a new array *x*' equal to *x* except that *x*'[i] = v and a second operation that assigns *x*' to *x*.

14.3.2 Finite sets

The elements of the data type **Set**[*E*] are finite sets of elements of type *E*. The following notations may be used to denote sets of type **Set**[*E*] or functions that can be applied to sets *s* and *s*' of type **Set**[*E*] and an element *e* of type *E*.

Symbol	Alternate	Sample use	Meaning
$\emptyset$	`{}`	$\emptyset$	Empty set
{...}		{*e*}	Singleton set containing *e* alone
$\{e_1, \cdots, e_n\}$		{1,2,3,4,5}	set of *n* elements given in extension
insert		*insert*(*e*, *s*)	Set containing *e* and all elements of *s*
delete		*delete*(*e*, *s*)	Set containing all elements of *s*, but not *e*
$\in$	`\in`	$e \in s$	Member of
$\backslash notin$	`\notin`	$e \backslash notin\ s$	Not a member of
$\cup$, I, $-$	`\U`, `\I`,`-`	($s \cup s$') − ($s \cap s$')	Union, intersection, difference
$\subset$, $\subseteq$	`\subset`, `\subseteq`	$s \subset s$'	(Proper) subset
$\supset$, $\supseteq$	`supset`, `\supseteq`	$s \supseteq s$'	(Proper) superset
size		*size*(*s*)	(Natural) number of elements in *s*

14.3.3 Finite mappings

The elements of the data type **Map**[*D1*, ..., *Dn*, *R*] are finite partial mappings with an *n*-dimensional domain of type $D1 \times ... \times Dn$ to elements of a range with type *R*. Mappings differ from arrays in that they are defined only for finitely many elements of their domains (and hence may not be totally defined). The following notations may be used to denote mappings or functions that can be applied to mappings *m* and *m*' of types **Map**[*I*,*R*] and **Map**[*I*, *J*, *R*].

Symbol	Alternate	Sample use	Meaning
$\emptyset$	`{}`	$\emptyset$	Empty mapping
...[..., ..., ...]		$m[i]$	Image of i:I under m (if defined)
		m'$[i, j]$	Image of i:I and j:J under m' (if defined)
defined		*defined*(m, i)	True if $m[i]$ is defined
update		*update*(m, i, r)	m" is equal to m except that m"`[i] =r`
remove		*remove*(m,i)	m" is equal to m except that m"`[i]` is undefined.

14.3.4 Finite multisets

The elements of the data type **Mset**[E] are finite multisets of elements of type E. All notations that can be used with sets of type **Set**[E] have suitably extended meanings when used with multisets of type **Mset**[E]. In addition, the symbol *count* denotes a binary function such that *count*(e, s) is the (natural) number of times an element e occurs in a multiset s.

14.3.5 Sequences

The elements of the data type **Seq**[E] are finite sequences of elements of type E. The following notations may be used to denote sequences of type **Seq**[E] or functions that can be applied to sequences s and s' of type **Seq**[E] and elements e of type E and n of type *Nat*. The first index in a sequence is supposed to be 0.

Symbol	Alternate	Sample use	Meaning
$\emptyset$	`{}`	$\emptyset$	Empty sequence
$\vdash$	$\vdash$	$s \vdash e$	Sequence with e appended to s
$\dashv$	$\dashv$	$e \dashv s$	Sequence with e prepended to s
$\|\|$		$s \,\|\| s$'	Concatenation of s and s'
$\in$	`\in`	$e \in s$	Member of
$\backslash notin$	`\notin`	$e \backslash notin\ s$	Not a member of
head, *last*		*head*(s)	First (last) element in s
init, *tail*		*tail*(s)	All but first (last) elements in s
len		*len*(s)	Length of s
...[...]		$s[n]$	n^th element in s

14.3.6 Extensions by nil

The elements of the data type **Null**[E] consist of a copy of each element of the underlying data type E, plus one additional element *nil*. The following notations may be used to denote elements of type **Null**[E] or functions that can be applied to an element e of type E or an element n of type **Null**[E].

Symbol	Sample use	Meaning
nil	*nil*	The additional element *nil*
embed	*embed*(e)	The element corresponding to e:E
val	*val*(n)	The e such that $n = embed(e)$; undefined if $n = nil$

14.3.7 Enumerations

The data type **Enumeration**$[e_1, \cdots, e_n]$ denotes a finite set with elements $e_1, \cdots, e_n$. The set is ordered where the successor of an element e is denoted by *succ*(e) and corresponds to the element textually following e in the enumeration. Enumeration can be used to index arrays or maps. For instance, the excerpt

types *Colors* : **Enumeration** [*red*, *green*,*blue*]

defines an enumerated type to represent the three basic colors whereas the statement

c : *Colors* : = *blue*;

declares a variable c of type *Colors* and initializes the variable to the value *blue* drawn from the *Colors* set. Note that the expression *succ*(*green*) would also initialize c to *blue*.

14.3.8 Tuples

The elements of the data type **Tuple**$[f_1 : T_1, ..., f_n : T_n]$ are n-tuples with fields $f_1, \ldots, f_n$ of types respectively $T_1, \ldots, T_n$. Tempo supports the standard notation for selecting a field of a tuple. For example if we have a variable p of type **Tuple**[*name*: *String*, *age*: *Int*] then *p.name* denotes the string associated with variable p.

14.3.9 Unions

Elements of the data type **Union**$[f_1 : T_1, ..., f_n : T_n]$ represent a value whose type T must be one of $T_1 \cdots T_n$. It is useful to represent an object whose type is not know a-priori but could be one of several alternatives. For instance, the fragment

MType : **Enumeration** [*DISCOVER*,*REQUEST*,*DECLINE*,*RELEASE*,*INFORM*,*OFFER*,*PACK*,*NAK*],
OptionValue : **Union** [*s*:*String*, *n*:*Int*, *m*:*MType*]

states that *OptionValue* could be a *String*, an *Int* or a *MType*. To assign an *OptionValue* variable, one must create a value of the union type. The following fragment

x : *OptionValue* : = s(`"Hello"`);
...
x : = m(*DHCPOFFER*);

declares x as as variable of type *OptionValue*, initializes it with the string "Hello" using the constructor s (the field name of type *String* in the **Union**) and later assigns x to a *MType* using the m constructor. When a **Union** is declared, Tempo automatically defines an enumeration type

OptionValue_tag : **Enumeration** [*s*,*n*,*m*]

that has one value for each constructor of the **Union** type. Finally, Tempo defines a function *tag*

tag : *OptionValue* $\rightarrow$ *OptionValue_tag*

that can be used on a variable of type *OptionValue* to find out the *kind* of value it currently holds. In the example above, a call *tag*(x) after the second assignment would return m, the third value of enumeration *OptionValue_tag*.

14.4 Type aliases

Tempo supports the definition of type aliases that can serve as shorthands when referring to complex types. For instance, consider the situation where a specification contains several automata that all receive as an argument a value drawn from an enumerated type, e.g., a type representing colors. The list of formal arguments of each automaton should define a formal whose type is exactly that enumeration. To avoid repetitions, the specification can start with

types *Colors* : **Enumeration** [*red*,*green*,*blue*]

to globally define *Colors* as an alias for the enumerated type containing the three primary colors. The rest of the specification can, from that point on, refer to the type *Colors*. More formally, from the current typing environment Γ, the type aliasing of n to the type T drawn from the set of static types $\mathcal{T}$ produces a new typing context Γ' used by the type analysis of the subsequent statements and differs from Γ in the following way

$$\Gamma' : Identifiers \rightarrow \mathcal{T} = \begin{cases} \Gamma'(i) = T \Leftrightarrow i = n \\ \Gamma'(i) = \Gamma(i) \ \ \forall i \in dom(\Gamma) : i \neq n \end{cases}$$

namely, both contexts are identical except for n which now maps to type T. Once a type alias is defined, one can write

automaton $A(c : Colors)$

to define an automaton A that expects a value from the *Colors* type. The type statement can be used to specify a comma-separated list of types and appears either in the global scope or within vocabularies. Note that, when identical type declarations are repeated, Tempo introduces multiple *distinct* types that are not equal, i.e., Tempo relies on referential equivalence rather than structural equivalence for type equality. Type aliases are therefore crucial when multiple automata must refer to the same type.

14.5 User-defined vocabularies

Users can define abstract data types through the association of types and operations within vocabulary definitions. Each vocabulary introduces notations for one or more types (following the keyword **types** or **defines**) and zero or more operators (following the keyword **operators**). Each operator has a signature that specifies the types of its arguments followed by the symbol → (which can be typed as `->`) and the type of its result. Infix, prefix, postfix, and mixfix operators are named by sequences of characters and are defined using placeholders _ (two underscores) to indicate the locations of their arguments. Functions (e.g., in *max*(*a*,*b*)) are denoted by simple identifiers.

Tempo relies on vocabularies for all its builtin data types. For instance, the Sequence abstract data type is specified with

vocabulary Seq defines Seq[*E*]
  operators
    ∅: →**Seq**[*E*],
    _⊢_: **Seq**[*E*], *E* →**Seq**[*E*],
    _⊣_: *E*, **Seq**[*E*] →**Seq**[*E*],
    _‖_: **Seq**[*E*],**Seq**[*E*] →**Seq**[*E*],
    _ ∈ _: *E*, **Seq**[*E*] →*Bool*,
    head, *last*: **Seq**[*E*] →*E*,

tail, *init*: **Seq**[*E*] →**Seq**[*E*],
len: **Seq**[*E*] →*Nat*,
__[__]: **Seq**[*E*], *Nat* →*E*

end

The vocabulary defines a parametric data type **Seq**[*E*] where *E* is a place-holder for a type parameter. The keyword **operators** is followed by a comma separated list of functions that can use prefix, infix, or postfix notation. The function ∅ is a constant function that returns an empty sequence of elements of type *E*. Tempo will infer the actual type to substitute for *E* based on the context in which the function is called. If the context is ambiguous, one can always use a type specifier as in ∅:**Seq**[*Nat*] to tell Tempo that the empty sequence is meant to hold values of type *Nat*. The second operator is an infix *append* operator. The **defines** keyword is used when defining parametric types. A vocabulary can also introduce type aliases with with the **types** keyword as in

vocabulary *Col*
  types *Colors* : **Enumeration** [*red*,*green*,*blue*]
  operators
    makeColor : *Nat*,*Nat*,*Nat* →*Colors*

end

Table 14.1 summarizes the ways in which operators can be described in vocabulary definitions and then used in various kinds of expressions.

Sample declaration	Form of expression	Sample use
`f: Int -> Int`	functional	$f(i)$
`min: Int, Int -> Int`	functional	$min(i, j)$
`0: -> Int`	zeroary	0
`__<__: Int, Int -> Bool`	infix	$i < j$
`-__: Int, Int -> Int`	prefix	$-i$
`__!: Int, Int -> Int`	postfix	$i!$
`__[__]: A, Int -> V`	mixfix	$a[i]$
`{__}: E -> Set[E]`	mixfix	$\{x\}$
`{__}: List[E] -> Set[E]`	mixfix	$\{x, y, z\}$
`{}: -> Set[E]`	mixfix	$\{\}$
`if__then__else__: Bool, S, S -> S`	mixfix	**if** $x < 0$ **then** $-x$ **else** x

Table 14.1: Sample operator declarations and use in terms

Functions that take no arguments can be invoked in one of two ways. Tempo authorizes to simply use the function name or to use the function name followed by an empty pair of parenthesis. Given the function definition

let *five*() : →*Int* =5;

the following two statements are correct

x : *Int* : = *five*;
y : *Int* : = *five*();

and both initialize the variable on the left-hand side with the value returned by the function *five*. The mixfix notation where the operands can appear before, between and after the symbols of the operator uses double underscores __ to indicate the position of each operand and the type of the operands are listed in left to right order after the colon. The declaration

$\{\_\_\}$: *List*[*E*] →**Set**[*E*]
is particularly interesting. It defines an operator that uses a pair of curly braces and receives its operand(s) in between the braces. The type of the operand is *List*[*E*], a builtin type of Tempo denoting a comma-separated list of values of type *E*. The operator can receive a list of values of type *E* of arbitrary length. This is particularly convenient to define a set in extension as in

w : **Set**[*Int*] : = {1,2,3,4,5}
Indeed, the curly brace operator is called on a list of 5 integers and therefore produces a set of integers that can be assigned to w. Finally, note that the vocabulary also includes an operator

$\{\_\_\}$: *E* →**Set**[*E*]
to build a singleton out of a single value of type *E*. Clearly, Tempo supports name overloading and is capable to disambiguate between the two curly brace operators based on the context.

14.5.1 Builtin Vocabularies

Vocabularies play a key role in Tempo itself as all the builtin abstract data types are actually defined in vocabularies. For example, the following vocabulary defines the *Nat* abstract data type.

vocabulary *Nat*
  types *Nat*
  operators
    succ, *pred*: *Nat* →*Nat*
    $\_\_+\_\_$, $\_\_-\_\_$, $\_\_*\_\_$, $\_\_**\_\_$, *min*, *max*, *div*, *mod*: *Nat*, *Nat* →*Nat*
    $\_\_ < \_\_$, $\_\_\leq\_\_$, $\_\_ > \_\_$, $\_\_\geq\_\_$, $\_\_ =\_\_$, $\_\_\neq\_\_$: *Nat*, *Nat* →*Bool*
end

In Tempo, an operator always denotes a total function, even if its values are not known for some elements in its domain. Thus, to say that *mod*(x, 0) is "undefined" means that its value is some fixed, but unknown element of *Nat*; it does not mean that *mod* is a partial function.

14.5.2 Parametric Vocabularies

A vocabulary definition can be parameterized and reused by importing it into other vocabularies, as in

vocabulary *Ordering*(*T*: **type**)
  types *T*
  operators $\_\_ < \_\_$, $\_\_\leq\_\_$, $\_\_ > \_\_$, $\_\_\geq\_\_$, $\_\_ =\_\_$, $\_\_\neq\_\_$: *T*, *T* →*Bool*
end

vocabulary *MyNat*
  types *MyNat*
  imports *Ordering*(**type** *MyNat*)
  operators
    succ, *pred*: *Nat* →*Nat*,$\_\_+\_\_$, $\_\_-\_\_$, $\_\_*\_\_$, $\_\_**\_\_$, *min*, *max*, *div*, *mod*: *Nat*, *Nat* →*Nat*
end

vocabulary *MyReal*
  imports *Ordering*(**type** *Real*)
  operators
    $-\_\_$, *abs*: *Real* →*Real*,

_--+--_, _--−--_, _--*--_, _--**--_, _--/--_, *min*, *max*: *Real*, *Real* →*Real*,*floor*: *Real* →*Int*

end

or into automaton definitions, as in

automaton *Arrange*(*T*: **type**)
  imports *Ordering*(**type** *T*)
  signature output *swap*(*i*, *j*: *Nat*)
  states *A*: **Array**[*Nat*, *T*];
  transitions output *swap*(*i*, *j*)
    locals *temp*: *T*;
    pre *A*[*i*] < *A*[*j*];
    eff *temp* := *A*[*i*]; *A*[*i*] := *A*[*j*]; *A*[*j*] := *temp*;

Import statements can even be used with builtin vocabularies to *explicitly* import them into a specification rather than relying on a declaration to trigger an automatic import. For instance, the fragment

automaton *Arrange*(*T* : **type**)
    imports Set(*Int*)
    ...

explicitly imports the **Set** vocabulary and instantiates it with the type *Int*. A declaration **Set**[*Int*] would achieve the same result and *implicitly* load and instantiate the vocabulary.

14.5.3 Vocabularies with Constructors

A vocabulary can also introduce a type constructor with the **defines** keyword. Consider, for instance, the builtin **Null** vocabulary

vocabulary Null defines Null[*T*]
  operators
    nil : →**Null**[*T*];
    embed : *T* →**Null**[*T*];
    val : **Null**[*T*] →*T*;

end

The identifier *T* in this vocabulary is a type parameter, which is bound to a type any time the constructor **Null** is used to provide operator appropriate for that use. Thus, if *x* is a variable of type *Int*, then one can write *embed*(*x*) to obtain a value of type **Null**[*Int*] and the boolean expression *val*(*embed*(*x*)) =*x* is always true.

14.5.4 User-defined Generic Vocabularies

To further illustrate user-defined vocabularies, consider the following fragment used to define directed graphs.

vocabulary *DirectedGraph*(*T*: **type**)
  types
    Edge : **Tuple**[*src*: *T*, *dst*: *T*],
    Digraph : **Tuple**[*vset*: **Set**[*T*], *eset*: **Set**[*Edge*]],
    Path : **Seq**[*T*]
  operators
    connected: *T*,*T* →*Bool*,
    addEdge: *Digraph*, *Edge* →*Digraph*

end

Here, the type parameter *T* represents the type of the vertices of a directed graph defined by the *directedGraphs* vocabulary. The vocabulary introduces the types *Edge*, *Digraph*, and *Path*. The type *Edge* is defined as an ordered pair of elements of type *T*. *Digraph* is defined as an ordered pair of sets —*vset* is a set of type *T* and *eset* is a set of *Edge*'s—. The **operators** section introduces two functions *connected* and *addEdge*. *connected* takes a pair of elements of type *T* and returns a boolean. *addEdge* takes a *Digraph* and an *Edge* and returns a *Digraph*.

If *DirectedGraph*(*Nat*) is imported into an automaton, then all the type and operator definitions in *DirectedGraph* are interpreted with *T* bound to *Nat*. For example, the initial graph *G* would be an arbitrary graph with a set of natural numbers as vertices, and a set of pairs of natural numbers as edges. The automaton *updateGraph* shown below imports *directedGraphs*(*V*), where *V* is a formal type parameter of the automaton which illustrates how to perform abstract operations without specifying a concrete type for the digraph.

automaton *updateGraph*(*V*: **type**)
  imports *DirectedGraph*(*V*)
    signature
      input *add*(*e*: *Edge*)
    states
      G: *Digraph*;
    transitions
      input *add*(*e*)
        eff *G* := *addEdge*(*G*,*e*);

Of course, for proving or model-checking properties of the automaton, for simulating it, or for generating executable code from the Tempo specification, it may become necessary to assert properties or to provide implementations of the types and operations. For instance, these may take the form of axioms stating key properties of the *connected* function for a theorem prover, Java implementation the *Digraph* of the data type for a code generator. These are provided as appropriate auxiliary files to the back-end tools independent of Tempo language.

14.5.5 Java Code Integration

To simulate a TIOA specification that relies on user-defined vocabularies, it is necessary to write and make available to Tempo a Java implementation of each such vocabulary. For instance, consider a TIOA model that uses the following vocabulary

vocabulary *Random*
  operators
    randomInt: *Int*, *Int* →*Int*,
    randomReal: *Real*, *Real* →*Real*,
    chooseRandom: **Set**[*Nat*] →*Nat*
end

To simulate the model, Tempo will search a Java JAR archive that contains an implementation of the *Random* abstract data type[5]. As a starting point, Tempo can generate the skeleton of the implementation from the vocabulary definition[6]. The Tempo simulator will create a skeleton Java for each vocabulary. For the Random vocabulary above, the skeleton will look like

[5]Tempo will search in a directory specified at runtime. In Eclipse this setting can be reached in the Simulator plugin page through the Preferences menu.

[6]To generate the skeleton, refer to the actual tool documentation. The Eclipse user interface has a checkbox option within the Simulator Preference.

```
package com.veromodo.tempo.simulator.runtime.voc;
public class TempoRandom {
   public Integer chooseRandom_fun(BasicSet arg0) {
      //TODO implement function body
      return null;
   }
   public Integer randomInt_fun(Integer arg0, Integer arg1) {
      //TODO implement function body
      return null;
   }
   public Double randomReal_fun(Double arg0, Double arg1) {
      //TODO implement function body
      return null;
   }
}
```

The Vocabulary is implemented by a class that has one method for each function and operator defined within the vocabulary. Scalar builtin types are mapped to their natural Java counterparts whereas type constructor (e.g., **Set**[*Int*]) are mapped to Java classes whose nomenclature is always `BasicXXX` where `XXX` is the name of the type constructor. Note that the Java code uses type erasures, i.e., the type parameters are absent from the definition. Hence a Tempo **Set**[*Int*] becomes a Java `BasicSet`. Implementation for all the builtin type constructors are provided in a `builtins.jar` Java archive which should be included in order to compile your own definition of `TempoRandom.java`. For reference, a possible implementation is shown below.

```
package com.veromodo.tempo.simulator.runtime.voc;
import java.util.Iterator;
import java.util.Random;
import com.veromodo.tempo.simulator.exception.SimulatorRuntimeException;
public class TempoRandom {
   private Random _gen;
   public TempoRandom(){  _gen = new Random(System.nanoTime()); }
   public Integer chooseRandom_fun(BasicSet arg0) throws SimulatorRuntimeException {
      int size = arg0.getSize();
      if(size <= 0)
         throw new SimulatorRuntimeException("can't choose from an empty set!");
      int index = _gen.nextInt(size);
      Iterator iter = arg0.iterator();
      Integer val = (Integer)iter.next();
      for(int x=0; x<index; x++)
         val = (Integer)iter.next();
      return val;
   }
   public Integer randomInt_fun(Integer arg0, Integer arg1) {
      Integer lowerBound,upperBound;
      if(arg0.compareTo(arg1) < 0){
          lowerBound = arg0;
          upperBound = arg1;
      } else if(arg0.compareTo(arg1) > 0){
          lowerBound = arg1;
          upperBound = arg0;
      } else return arg0;
      return _gen.nextInt(upperBound - lowerBound)+lowerBound;
   }
```

```
    public Double randomReal_fun(Double arg0, Double arg1) {
        Double lowerBound,upperBound;
        if(arg0.compareTo(arg0) < 0){
            lowerBound = arg0;
            upperBound = arg1;
        } else if(arg0.compareTo(arg0) > 0){
            lowerBound = arg1;
            upperBound = arg0;
        } else return arg0;
        return _gen.nextDouble()%(upperBound - lowerBound)+lowerBound;
    }
}
```

14.6 Type constraints

Tempo specifications often use very natural notations to specify construct or initialize variables of different types. For instance, the lines

x : **Mset**[*Int*] : = {1,2,3};
y : **Set**[*Int*] : = {1,2,3};
defines x as a multiset of integers initialized with $\{1, 2, 3\}$ and y as a set of integers initialized with $\{1, 2, 3\}$. Syntactically, both lines are identical and the initializations use a set notation that make the fragment very readable. Semantically though, the two lines are quite different as the first one invokes the {__} operator of the **Mset** vocabulary whereas the second invokes the {__} operator of the **Set** vocabulary. Clearly, Tempo supports function and operator overloading and its type inference engine is often capable to determine the right operator/function based on the type of the operands and the expected type of the result.

In some situation though, the type inference may yield an unexpected result or the construction may simply be ambiguous and prevent Tempo from choosing a suitable type for a sub-expression. To address this problem, Tempo supports *type constraints*, i.e., type annotations on sub-expressions that dictate the type that a sub-expression should have. For instance, the example above can be rewritten

x : **Mset**[*Int*] : = {1,2,3} : **Mset**[*Int*];
y : **Set**[*Int*] : = {1,2,3} : **Set**[*Int*];
where the sub-expression {1,2,3} is followed by :**Set**[*Int*] to require that the sub-expression yield an object of type **Set**[*Int*]. Generally, a type constraint has the form e:T where e is an expression and T is a type. Type constraints can appear deep within an expression as long as proper parenthesizing is used. For instance, the type constraint in (3:*Real*)+5.2 forces the sub-expression 3 to be typed as a *Real*. (Strictly speaking, the type annotation is not necessary in this example as *Nat* is a sub-type of *Real* and Tempo will automatically promote 3 to *Real*).

A type constraint can play a critical role in an expression such as

x < *size*({*a*,*b*,*c*,*d*})
where x as well as a, b, c, d are all variables of type *Int*. Indeed, if Tempo infers that {*a*,*b*,*c*,*d*} denotes a set, its cardinality will be anything between 1 and 4. However, if {*a*,*b*,*c*,*d*} is seen as a multiset, its cardinality will be exactly 4. If both operators are available, Tempo will report an ambiguity. However, if only one operator is available, Tempo will silently select that one, even if it does not correspond to the true intent of the modeler. A type constraint would solve the problem and explicitly state the type of the set. A *safer* fragment would therefore be

$x < size(\{a,b,c,d\} : \mathbf{Set}[Int])$

14.7 Dynamic Types

The dynamic type of a variable is declared implicitly; it is inferred automatically from the variable's static type. For variables of all built-in simple types except *Real* the dynamic type is the set of piecewise constant functions from the set of real numbers to the set denoted by the static type of the variable. On the other hand, the dynamic type of *Real* variables is the set of continuous functions from the set of real numbers to the set of real numbers. In order to to define a variable that takes on real numbered values but is not changed by trajectories one must use the type *DiscreteReal*.

If the type of a variable v is defined by one of the type constructors **Tuple** or **Array**, then its dynamic type $dtype(v)$ is defined as follows. The variable v is viewed as an ordered tuple of variables $\{v_1, \ldots, v_k\}$, for some finite k. and its static type, $type(v)$, as $type(v_1) \times \ldots \times type(v_k)$. The dynamic type of v is the set of functions f from intervals of time to $type(v)$ such that $f.v_i \in dtype(v_i)$ for each $i \in \{1, \ldots, k\}$.

If the type of v is defined by nesting the type constructors **Tuple** and **Array**, then $dtype(v)$ is defined recursively using the above rule. Variables of all other compound types and user-defined types are considered to be discrete.[7]

Consider the following vocabulary definition:

vocabulary *matrix*
 types
  T: **Tuple** [*a*: *Real*, *b*: *Nat*, *c*: *DiscreteReal*],
  Row: **Enumeration** [*p1*, *p2*, *p3*],
  Col: **Enumeration** [*q1*, *q2*, *q3*],
  matrix: **Array** [*Row*, *Col*, *Real*],
  intMatrix: **Array** [*Row*, *Col*, *Int*]
end

Suppose that variable v is declared to be of type T. Then $dtype(v)$ is the set of functions f from intervals of time to T such that $f.a$ is piecewise continuous with real values, $f.b$ is piecewise constant with natural values, and $f.c$ is piecewise constant with real values. The type *matrix* represents a 3×3 array of real numbers. A variable x declared to be of type *matrix* can be viewed as an ordered 9-tuple of reals. The dynamic type of x is the set of functions f from intervals of time to $\mathbb{R}^9$ ($\mathbb{R}$ stands for the set of real numbers) such that the restriction of f on each of the coordinates is a piecewise continuous function from intervals of time to reals. A variable y of type *intMatrix* can be viewed as an ordered 9-tuple of integers. And $dtype(y)$ is the set of functions f from intervals of time to $\mathbb{Z}^9$ ($\mathbb{Z}$ stand for the set of integers) such that the restriction of f on each coordinate is a piecewise constant.

[7] We assume finite maps and sequences to be discrete, regardless of the type of their constituents. We use this simple aprroach because we typically use variables of these types to hold values that remain unchanged by time passage. The fact that maps can be partially defined and that sequences can shrink and grow over time requires a careful definition for dynamic types associated with these types, which is not as straightforward as tuples and arrays.

Part III
Tempo Reference Manual

This part formally defines the Tempo language, including its grammar. Syntactic categories are indicated by *italic* type, and literal words and characters are indicated by `typewriter` type. The grammar's meta-syntax is shown in purple. Alternative constructs are separated by |. Optional constructs are followed by a ?. Parentheses are used for grouping. A construct followed by a * indicates zero or more instances of the construct (Kleene closure), and a construct followed by a + indicates one of more instances of the construct (positive closure).

15 Tempo Programs

spec ::=(*typeDecls* | *imports* | *include* | *funDecls* | *vocabDef* | *autoDef* | *invDef* | *simDef* | *simProg*)+ 'EOF'

A Tempo program consists of one or more of the following, in any order: type declarations, import statements, include statements, function declarations, vocabulary definitions, automata definitions, invariant definitions, simulation relations, and simulation programs. Vocabulary definitions are discussed in Section 16.2, automata definitions are discussed in Section 17, simulation relations are discussed in Section 21, and simulation programs are discussed in Section 20. The other constructs are discussed in the remainder of this section.

15.1 Type Declarations

typeDecls ::='types' *typeDecl* (, *typeDecl*)*
typeDecl ::=ID (: *typeRef*)?

Type declarations enable Tempo users to define their own data types or create aliases for existing data types. The type declarations begin with the keyword `types` followed by one or more individual type declarations, separated by commas. Each individual type declaration begins with an identifier, corresponding to the name of the new data type. For new data types, the identifier is all that is needed. For renaming (aliasing) an existing data type, the identifier is followed by a colon and the specification of the existing data type (see Section 16.1). More powerful data types are defined with vocabularies (see Section 16.2).

15.2 Import Statements

imports ::='imports' *vocabRef* (, *vocabRef*)*
vocabRef ::=ID ((*actual* (, *actual*)*))?
actual ::=(*expr* | 'Type' *typeRef*)

An import statement begins with the keyword `imports` followed by one or more vocabulary references, separated by commas. Each vocabulary reference consists of an identifier, corresponding to the name of the vocabulary, followed by an optional list of the actual parameters of the vocabulary, separated by commas and enclosed in parentheses. An actual parameter is either the

keyword `Type` followed by a data type, if the associated vocabulary formal parameter is a data type, or an arbitrary expression.

Each vocabulary is stored in a file with the same name as the vocabulary. For example, the `NewVocab` vocabulary is stored in a file named `NewVocab.tioa`. As a result of an import statement, the contents of the files for each of the vocabularies in the import list are included in the Tempo program, with each vocabulary formal parameter replaced with the corresponding actual parameter, as specified.

15.3 Include Statements

include ::='include' STRING

An include statement consists of the keyword `include` and a string corresponding to the name of a file. If the file is not in the current directory, the string begins with the relative path to the file followed by the file name. The effect of an include statement is to add the text of the named file to the Tempo program. In particular, include statements may be used to add automata defined in other Tempo programs to the current program.

15.4 Function Declarations

funDecls ::='let' (*funDecl* ;)+
funDecl ::=ID ((ID (, ID)*)?) : *typeSignature* = *expr*
typeSignature ::=*typeList*? - > *typeRef*

The function declarations begin with the keyword `let` followed by one or more individual function declarations, each terminated with a semicolon. A function declaration begins with an identifier, corresponding to the name of the function, a left parenthesis, and zero or more identifiers, separated by commas, corresponding the names of the function's formal parameters. These are followed by a right parenthesis, a colon, and the type signature of the function. The type signature consists of the data types of each of the function's parameters, separated by commas, followed by the symbol `->` and the data type of the value returned by the function. A function declaration ends with an equals sign (`=`) and the expression which is evaluated to determine the value of the function. The parameter identifiers are local to the function's expression; their initial values are the values of the function's actual parameters at the time the function is invoked.

15.5 Invariant Definitions

invDef ::='invariant' *idOrNumeral* ? 'of' ID : *exprTerminated*
exprTerminated ::=(*expr* ;)+

An invariant definition begins with the keyword `invariant` followed by an optional identifier or integer, which may be used as the name of the invariant. This is followed by the keyword `of` and an identifier, corresponding to the name of the automaton to which this invariant applies. An invariant definition ends with a colon and a list of one or more expressions, each ending in a semicolon. Each of these expressions should evaluate to `true` at the end of each transition and trajectory of the automaton (but need not evaluate to `true` in any intermediate states of a transition or trajectory).

An interactive theorem-prover, such as PVS, or a model checker, such as Uppaal, may be used to check the validity of the invariants.

15.6 Comments

The character `%`, appearing anywhere in a Tempo program, denotes the beginning of a comment. The rest of the line after the `%` is ignored by Tempo.

16 Data Types and Vocabularies

16.1 Data Types

typeRef ::=ID
    | ID [*typeList*]
    | 'Enumeration' [*idOrNumerals*]
    | 'Tuple' [*fieldDecls*]
    | 'Union' [*fieldDecls*]
fieldDecls ::=*fieldDecl* (, *fieldDecl*)*
fieldDecl ::=ID (, ID)* : *typeRef*
typeList ::=*typeRef* (, *typeRef*)*
idOrNumerals ::=*idOrNumeral* (, *idOrNumeral*)*
idOrNumeral ::=(ID | INT)

Tempo requires its variables to be declared with explicit data types. For many types of data, the data type specification is just an identifier, corresponding to the name of the data type. Data types of this form include the primitive data types provided by Tempo (`Bool`, `Nat`, `Int`, `Real`, `AugmentedReal`, `DiscreteReal`, `Char`, and `String`), and data types defined with a type declaration (see Section 15.1).

Other data type specifications consist of an identifier followed by a list of data type specifications, separated by commas and enclosed in square brackets. Data types of this form include the built-in data types of `Array`, `Set`, `Mset`, `Map`, `Seq`, and `Null`, and parameterized data types in vocabularies. For `Set`, `Mset`, `Seq`, and `Null`, the bracketed list consists of a single data type specification, corresponding to the data type of the elements of the collection. For `Array` and `Map`, the bracketed list contains at least two data specifications; the last data type specification is the data type of the elements of the collection, and the rest are the data types of the indices used to access the elements.

The data type specification for an enumeration data type consists of the keyword `Enumeration` followed by a list of string or integer constants, separated by commas and enclosed in square brackets, corresponding to the set of values available to a variable of this data type. The tuple and union data type specifications consist of the keyword `Tuple` or `Union`, respectively, followed by a list of member declarations, separated by commas and enclosed in square brackets. Each member declaration consists of one or more identifiers, separated by commas, followed by a colon and the data type associated with those identifiers. For a tuple, the identifiers are the names of the tuple's fields, and for a union, the identifiers are the tag names of the union's constructors.

16.2 Vocabulary Definitions

vocabDef ::='vocabulary' ID *formals*? *defines*? *importsAndTypes*? *operators*? 'end'
formals ::=(*formal* (, *formal*)*)
formal ::=ID (, ID)* : (*typeRef* | 'Type')
defines ::='defines' ID [*typeList*]
importsAndTypes ::=(*imports* | *typeDecls*)+

A vocabulary definition begins with the keyword `vocabulary` and an identifier, corresponding to the name of the vocabulary. Next is an optional list of the formal parameters of the vocabulary, separated by commas and enclosed in parentheses. Each formal parameter consists of an identifier, corresponding to the name of the parameter, followed by a colon and its data type or the keyword `Type`. If multiple, adjacent parameters are of the same data type, their identifiers may be separated by commas and followed by a single colon and their common data type. These are followed, in order, by an optional defines clause, optional import statements and type declarations, optional operator declarations, and the keyword `end`.

The defines clause consists of the keyword `defines`, an identifier, corresponding to the name of the data type defined by this vocabulary, and a list of data type specifications, separated by commas and enclosed in square brackets, corresponding to the data types of the parameters of the data type.

The optional import statements and type declarations of a vocabulary may be intermixed in any order. Import statements are discussed in Section 15.2, and type declarations are discussed in Section 15.1.

operators ::='operators' *opDecl* (, *opDecl*)*
opDecl ::=*rootOpName* (, *rootOpName*)* : *typeSignature*
rootOpName ::='if' __'then' __'else' __
    | *opName*
    | *idOrNumeral*
opName ::=*prefixSpec* | *infixSpec* | *mixfixSpec*
prefixSpec ::=(*opSym* | ~)__| . __| ID __
infixSpec ::=__(*opSym* __| . (__| ID)| ID __)
mixfixSpec ::=__? { (__(, __)*)? }
        | __? [(__(, __)*)?]
opSym ::=(OPERATOR | <=> | => | ∧ | ∨ | = | ~= | *plainOp*)
plainOp ::=(< | <= | > | >= | + | - | * | / | **)

The operator declarations begin with the keyword `operators` followed by one or more groups of operator declarations, separated by commas. Each group of operator declarations consists of a list of one or more operators, separated by commas, followed by a colon and the type signature for those operators. A type signature consists of the data types of each of the operands of the operation, separated by commas, followed by the symbol `->` and the data type of the result of the operation.

An operator is declared in one of five ways, depending upon its use. The simplest form of an operator is an identifier or integer, corresponding to the name of a function to be applied to the operands. In all the other forms, the symbol `__` is used as a placeholder within the operator declaration to represent the location of an operand. The operator may be denoted with a operator symbol. All of the Tempo symbols for logical, relational, and arithmetic operations may be used as operator symbols. In addition, a vocabulary may define its own operator symbol beginning with the character \.

A conditional operator consists of the keyword `if`, a placeholder for an operand, the keyword `then`, a placeholder for a second operand, the keyword `else`, and a placeholder for a third operand. A prefix operator consists of either an operator symbol, a `~`, a period, or an identifier, followed by a placeholder for an operand. An infix operator consists of two placeholders for operands separated by either an operator symbol, a period, or an identifier, or a placeholder for a single operand followed by a period and an identifier. Finally, a mixfix operator consists of an optional placeholder for a beginning operand, a left brace or left square bracket, zero or more operand placeholders separated by commas, and a matching right brace or right square bracket.

Mixfix operators with square brackets currently may not be used in Tempo expressions other than for accessing elements of an array and constructing tuples. Also, vocabulary-defined operator symbols currently may not be used as prefix operators.

17 Automaton Definitions

autoDef ::='automaton' ID (*formals where*?)? *imports*? *autoCore*
formals ::=(*formal* (, *formal*)*)
formal ::=ID (, ID)* : (*typeRef* | 'Type')
where ::='where' *expr*
autoCore ::=(*basicAutomaton* | *composedAutomaton*)

The definition of an automaton begins with the keyword `automaton` and an identifier, corresponding to the name of the automaton. Next is an optional list of the formal parameters of the automaton, separated by commas and enclosed in parentheses. Each formal parameter consists of an identifier, corresponding to the name of the parameter, followed by a colon and its data type or the keyword `Type`. If multiple, adjacent parameters are of the same data type, their identifiers may be separated by commas and followed by a single colon and their common data type. The range of parameter values may be constrained with an optional where clause, consisting of the keyword `where` followed by a predicate expression. Whenever an instance of a parameterized automaton is created, the values of the actual parameters must be such that the where clause expression evaluates to `true`.

The automaton definition may then contain an optional imports statement, consisting of the keyword `imports` followed by a list of one or more vocabularies to be imported (see Section 15.2). The rest of the body of an automaton definition varies depending on whether the automaton is a basic automaton or a composite automaton.

17.1 Basic Automaton Definitions

basicAutomaton ::=*actionSignature*? *states funDecls*? *transitions*? *trajectories*? *tasks*? *schedule*?

The body of a basic automaton consists of the following parts, in order: signature, states, function declarations, transitions, trajectories, tasks, and schedule. Only the states part is required; the rest are optional. Each of the parts is described, in turn, in the following sections.

17.1.1 Signature

actionSignature ::='signature' (*formalActions*)+
formalActions ::='input' *formalAction* (, *formalAction*)*
| 'output' *formalAction* (, *formalAction*)*
| 'internal' *formalAction* (, *formalAction*)*
formalAction ::=ID ((*sigFormal* (, *sigFormal*)*) *where*?)?
sigFormal ::='const' *expr*
| ID (, ID)* : *typeRef*

The signature part, if present, begins with the keyword `signature`. It is followed by a non-empty list of the signatures of the automaton's transitions. The signature of each transition begins with an identifier, corresponding to the name of the transition, followed by an optional list of the transition's formal parameters, separated by commas and enclosed in parentheses, whose values may be constrained by an optional where clause.

Each transition parameter may be either a constant or a variable. A constant parameter is denoted by the keyword `const` followed by an expression depending only on the values of the automaton's parameters. A variable parameter is denoted by an identifier followed by a colon and its data type. If multiple, adjacent parameters are of the same data type, their identifiers may be separated by commas and followed by a single colon and their common data type.

As in an automaton definition, a where clause consists of the keyword `where` followed by a predicate expression which constrains the range of acceptable parameter values. The values of the actual parameters must be such that the predicate expression evaluates to `true`. Typically, a signature where clause specifies the sets of values from which the actual parameter values are drawn.

Each transition signature must be preceded by a keyword denoting the type of the transition, namely `input`, `output`, or `internal`. Multiple transitions of the same type may be grouped together with a single instance of the word describing the type of the transitions followed by each of the transition signatures, separated by commas.

The following example illustrates many of the options for a basic automaton signature:

automaton *channel* (*i*:*Int*)
signature
input *send* (*m*:*Msgs*, **const** *i*, *j*:*Int*) **where** $j \in Nodes$,
fail
output *receive* (*m*:*Msgs*, *j*:*Int*, **const** *i*) **where** $j \in Nodes$

17.1.2 States

states ::='states' (*state* ;)+ *initially*?
| 'states'
state ::=ID : *typeRef* (:= *value*)?
initially ::='initially' *expr* ;

The states part of a basic automaton declares the automaton's state variables. It begins with the keyword `states` followed by zero or more state variable declarations and an optional initially expression.

Each state variable declaration consists of an identifier, corresponding to the name of the variable, followed by a colon and the data type of the variable. Optionally, the data type may be

followed by the assignment operator `:=` and an initial value for the variable. Each state variable declaration ends with a semicolon. The scope of a state variable declaration is the body of the automaton.

The initially expression, if present, begins with the keyword **`initially`** followed by a predicate expression. The initial values of the state variables must be selected in such a way so that the predicate expression evaluates to **`true`**. In particular, an initially expression restricts the values of variables initialized with a choose operator.

17.1.3 Function Declarations

Function declarations within an automaton definition are identical to function definitions outside of an automaton definition (see Section 15.4), except the scope of the functions is limited to the body of the automaton.

17.1.4 Transitions

```
transitions ::='transitions' (transition )+
transition ::='input' transitionCore
            | 'output' transitionCore
            | 'internal' transitionCore
transitionCore ::=ID (( actionActuals ) where? )? localVars? funDecls? precondition? urgency? effect?
actionActuals ::=expr (, expr )*
localVars ::='locals' (localDecl ; )+
precondition ::='pre' exprTerminated
urgency ::='urgent' 'when' expr ;
effect ::='eff' effProgram ('ensuring' expr ; )?
localDecl ::=ID : typeRef (:= value )?
exprTerminated ::=(expr ; )+
effProgram ::=effStmt+
effStmt ::=lvalue := value ;
          | 'print' value ;
          | 'if' effCondRec 'fi'
          | 'while' expr 'do' effProgram 'od'
          | 'for' ID 'in' expr 'do' effProgram 'od'
          | 'for' ID : typeRef 'in' expr 'do' effProgram 'od'
          | 'for' ID : typeRef 'where' expr 'do' effProgram 'od'
          | ;
effCondRec ::=expr 'then' effProgram ('elseif' effCondRec | 'else' effProgram )
```

The transitions part of an automaton definition, if present, consists of the keyword **`transitions`** followed by one or more transition definitions. Each transition definition begins with a keyword describing the type of the transition, namely **`input`**, **`output`**, or **`internal`**, and an identifier corresponding to the name of the transition. Next is an optional list of transition parameter expressions, separated by commas and enclosed in parentheses, possibly followed by the keyword **`where`** and a predicate expression constraining the set of parameter values. Since an automaton may contain more than one transition definition with the same name, the where clauses use the actual parameter values to determine which transition definition is applicable. A transition definition requires the actual parameter values to be such that its where clause expression evaluates to **`true`**.

The body of a transition definition consists of the following five optional parts, in order: local variable declarations, function declarations, a precondition, an urgency condition, and an effect. An input transition may not have a precondition or an urgency condition.

The local variable declarations, if any, begin with the keyword `locals` followed by one or more local variable declarations. Each local variable declaration begins with an identifier, corresponding to the name of the variable, followed by a colon and the data type of the variable. Optionally, the data type may be followed by the assignment operator `:=` and an initial value for the variable. Each local variable declaration ends with a semicolon. The scope of a local variable declaration is the body of the transition.

Function declarations within a transition definition, if any, are identical to function definitions outside of an automaton (see Section 15.4), except the scope of the functions is limited to the body of the transition.

The transition precondition, if any, begins with the keyword `pre` followed by one or more expressions, each ending with a semicolon. In order for the transition to be enabled, each of the expressions in the precondition must evaluate to `true`.

The urgency condition, if any, begins with the keywords `urgent when` followed by a predicate expression and a semicolon. Whenever the predicate expression evaluates to `true`, the currently executing trajectory must stop and an enabled transition must be fired. However, the transition that is fired need not be the transition with the true urgency condition.

The transition effect, if any, begins with the keyword `eff` followed by one or more statements specifying the behavior of the transition and an optional ensuring clause. The transition's effect statements are executed sequentially when the transition is fired. Statement types that are valid in a transition effect are assignment statements, print statements, if statements, while statements, for statements, and empty statements. An ensuring clause consists of the keyword `ensuring` followed by a predicate expression and ending with a semicolon. The ensuring expression must evaluate to `true` after the transition has fired, and thus may be used to restrict the nondetermistic values selected by choose operators in the effect statements.

17.1.5 Trajectories

trajectories ::=`'trajectories'` (*trajectory*)+
trajectory ::=`'trajdef'` ID (*formals where*?)? *funDecls*? *trajInvariant*? *stopCond*? *evolve*?
trajInvariant ::=`'invariant'` *exprTerminated*
stopCond ::=`'stop' 'when'` *expr* `;`
evolve ::=`'evolve'` *exprTerminated*
exprTerminated ::=(*expr* `;`)+

The trajectories part of an automaton definition, if present, consists of the keyword `trajectories` followed by one or more trajectory definitions. Each trajectory definition begins with the keyword `trajdef` followed by an identifier corresponding to the name of the trajectory. Next is an optional list of the formal parameters of the trajectory, separated by commas and enclosed in parentheses. Each formal parameter consists of an identifier, corresponding to the name of the parameter, followed by a colon and its data type or the keyword `Type`. If multiple, adjacent parameters are of the same data type, their identifiers may be separated by commas and followed by a single colon and their common data type. The range of parameter values may be constrained with an optional where clause, consisting of the keyword `where` followed by a predicate expression.

In order for the trajectory to be followed, the values of the actual parameters must be such that the where clause expression evaluates to `true`.

The body of the trajectory consists of four optional parts, namely function definitions, trajectory invariants, a stop condition, and evolve expressions. Function declarations within a trajectory definition, if any, are identical to function definitions outside of an automaton (see Section 15.4), except the scope of the functions is limited to the body of the trajectory.

The trajectory invariants, if any, begin with the keyword `invariant` followed by one or more predicate expressions, each ending in a semicolon. All of the trajectory invariant expressions must evaluate to `true` before the trajectory can begin to be followed, and they must remain `true` for the duration of the trajectory.

The stop condition, if any, consists of the keywords `stop when` followed by a predicate expression and a semicolon. The trajectory must stop as soon as the expression evaluates to `true`.

The evolve expressions, if any, consist of the keyword `evolve` followed by one or more expressions, each ending in a semicolon. The expressions specify how the values of the automaton's analog variables change with respect to time during the trajectory. Because of their unique role, these expressions are not arbitrary expressions, but instead are specialized expressions of the following form:

evolveExpr ::=*evolveLHS* (= | < | <= | > | >=)*expr*
evolveLHS ::=(ID .)* ID | 'd'((ID .)* ID) |

The left-hand side of each evolve expression must be either the identifier of a variable of type `Real` or `AugmentedReal` or the keyword `d`, denoting the derivative function with respect to time, followed by the identifier of a variable of type `Real` or `AugmentedReal`, enclosed in parentheses. Within a composite automaton (see Section 17.2), the identifier of the evolve variable may be preceded by the identifier(s) for its component automaton. The left-hand side is followed by any relational operator except not equal (`~=`) and an expression. Note that the derivative function cannot be part of the expression; it may be used only on the left-hand side of an evolve expression. Any analog variable whose behavior is not explicitly constrained by the evolve expressions is allowed to change arbitrarily during the trajectory. The automaton's discrete variables remain constant during the trajectory.

17.1.6 Tasks

tasks ::='tasks' (*task*)+
task ::={ *actionSet* (, *actionSet*)* } *forClause*?
actionSet ::=(*compInstance* .)? ID ((*expr* (, *expr*)*) *where*?)?
forClause ::='for' *varList* (, *varList*)* *where*?
varList ::=ID (, ID)* : *typeRef*

The tasks part of an automaton definition, if present, begins with the keyword `tasks` followed by one or more task lists. Each task list consists of one or more of the automaton's transitions, separated by commas and enclosed in braces, possibly followed by a for clause. The individual transition specifications begin with the name of the component in which the transition is located, for composite automata, and an identifier corresponding to the name of the transition. These are followed by expressions for each of the transition's parameters, if any, separated by commas and enclosed in parentheses, and an optional where clause. The where clause begins with the keyword

`where` followed by a predicate expression; the transition is included in the task list only when the where clause expression evaluates to `true`.

The optional for clause consists of the keyword `for` followed by one or more variable declarations, separated by commas, and an optional where clause. Each variable declaration consists of an identifier, corresponding to the name of the variable, followed by a colon and its data type. If multiple, adjacent variables are of the same type, their identifiers may be separated by commas and followed by a single colon and their common data type. The variables in the for clause may be used in the transition parameter expressions, with their values constrained by the where clause, if present.

None of the Tempo tools currently take advantage of task definitions.

17.1.7 Schedule

schedule ::='schedule' *states*? 'do' *basicProgram*? 'od'
states ::='states' (*state* ;)+ *initially*?
| 'states'
state ::=ID : *typeRef* (:= *value*)?
basicProgram ::=*basicStatement*+
basicStatement ::=*lvalue* := *value* ;
| 'print' *value* ;
| 'if' *basicCondRec* 'fi'
| 'while' *expr* 'do' *basicProgram* 'od'
| 'for' ID 'in' *expr* 'do' *basicProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *basicProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *basicProgram* 'od'
| 'fire' ;
| 'fire' 'input' ID ((*expr* (, *expr*)*))? ;
| 'fire' 'output' ID ((*expr* (, *expr*)*))? ;
| 'fire' 'internal' ID ((*expr* (, *expr*)*))? ;
| 'follow' ID 'duration' *expr* ;
| ;
basicCondRec ::=*expr* 'then' *basicProgram* ('elseif' *basicCondRec* | 'else' *basicProgram*)?

The schedule part of a basic automaton, if present, specifies the behavior of the automaton during simulation. It begins with the keyword `schedule` followed by optional variable declarations, the keyword `do`, zero or more statements, and the keyword `od`. The variable declarations begin with the keyword `states` and are the same as those in the states part of an automaton definition (see Section 17.1.2) except the scope of the variables is limited to the automaton's schedule.

The schedule's statements are executed sequentially when the automaton is run in a simulation. Statement types that are valid in a schedule are assignment statements, print statements, if statements, while statements, for statements, fire statements, follow statements, and empty statements.

17.2 Composite Automaton Definitions

composedAutomaton ::=*components hiddenActionSets*? *compSchedule*?

The body of a composite automaton consists of a list of the component automata followed by an optional list of hidden actions and an optional schedule.

17.2.1 Components

components ::='components' (*component* ;)+
component ::=ID ([*varList* (, *varList*)*])? (: *componentDef*)? *where*?
varList ::=ID (, ID)* : *typeRef*
componentDef ::=ID ((*actual* (, *actual*)*))?

The list of component automata begins with the keyword `components` followed by one or more component specifications, each ending with a semicolon. A component specification begins with an identifier, corresponding to the local name of the component within the automaton, and a list of its parameters, if any, separated by commas and enclosed in square brackets. Each parameter specification consists of an identifier, corresponding to the local name of the parameter, followed by a colon and its data type. If multiple, adjacent parameters are of the same data type, their identifiers may be separated by commas and followed by a single colon and their common data type. After the component name and parameters, there is an optional component designation, consisting of a colon, a second identifier, and a list of actual parameters, and an optional where clause.

If a component designation is present, its identifier specifies the name of the automaton, defined elsewhere, for which this component is an instance. If that automaton is parameterized, the automaton name is followed by a list of expressions for the values of the actual parameters for this instance, separated by commas and enclosed in parentheses. If a component designation is not specified, the local name of the component is the same as the name of the automaton for which it is an instance.

If the local name of the component is followed by a list of parameters, a separate component is created for each set of values in the parameter domains. Within the body of the composite automaton, each individual component is identified by its local name followed by expressions for the actual values of its parameters, separated by commas and enclosed in square brackets.

The optional where clause restricts the set of acceptable parameter values, and thus limits the number of components generated. The where clause begins with the keyword `where` followed by a predicate expression. The expression must evaluate to `true` for each component generated.

17.2.2 Hidden Actions

hiddenActionSets ::='hidden' (*actionSet* ;)+
actionSet ::=(*compInstance* .)? ID ((*expr* (, *expr*)*) *where*?)?

Within a composite automaton, the output transitions of the components are linked, to the extent possible, to input transitions of the same name in other components, and the combined transitions are performed as a single action. In order to reclassify these actions as internal transitions of the composite automaton, rather than leaving them as output transitions, it is necessary to hide them.

The list of hidden actions, if any, begins with the keyword `hidden` followed by one or more actions, each ending in a semicolon. Each hidden action is specified by the name of the component in which the action is located, if needed, and an identifier corresponding to the name of the action. These are followed by the action's parameters, if any, separated by commas and enclosed in parentheses, and an optional where clause. The where clause consists of the keyword `where` followed by a predicate expression; the action is hidden only when the where expression evaluates to `true`.

17.2.3 Schedule

```
compSchedule ::='schedule' states? 'do' compProgram? 'od'
states ::='states' (state ; )+ initially?
        | 'states'
state ::=ID : typeRef (:= value )?
compProgram ::=compStmt+
compStmt ::=lvalue := value ;
           | 'print' value ;
           | 'if' compCondRed 'fi'
           | 'while' expr 'do' compProgram 'od'
           | 'for' ID 'in' expr 'do' compProgram 'od'
           | 'for' ID : typeRef 'in' expr 'do' compProgram 'od'
           | 'for' ID : typeRef 'where' expr 'do' compProgram 'od'
           | 'fire' 'input' compInstance . ID (( expr (, expr)* ) )? ;
           | 'fire' 'output' compInstance . ID (( expr (, expr)* ) )? ;
           | 'fire' 'internal' compInstance . ID (( expr (, expr)* ) )? ;
           | 'follow' compTrajList 'duration' expr ;
           | ;
compCondRec ::=expr 'then' compProgram ('elseif' compCondRec | 'else' compProgram )?
compInstance ::=ID ([ expr (, expr)* ] )? (. ID ([ expr (, expr)* ] )? )*
compTrajList ::=componentTrajectory (, componentTrajectory )*
componentTrajectory ::=compInstance . ID (( expr (, expr)* ) )?
```

The schedule part of a composite automaton, if present, specifies the behavior of the automaton during simulation. It begins with the keyword `schedule` followed by optional variable declarations, an optional with block, the keyword `do`, zero or more statements, and the keyword `od`. The variable declarations begin with the keyword `states` and are the same as those in the states part of an automaton definition (see Section 17.1.2) except the scope of the variables is limited to the composite automaton schedule.

The schedule's statements are executed sequentially when the automaton is run in a simulation. Statement types that are valid in a schedule are assignment statements, print statements, if statements, while statements, for statements, fire statements, follow statements, and empty statements. A composite automaton's fire statements identify the transition to be fired by specifying the name of the component in which the transition is located, the parameters of that component if any, a period, the name of the transition within the component, and the parameters of the transition if any. The follow statements optionally specify a list of trajectories to be concurrently followed, separated by commas; each trajectory to be followed is identified by specifying the name of the component in which the trajectory is located, the parameters of that component if any, a period, the name of the trajectory within the component, and the parameters of the trajectory if any. For both the fire and follow statements, if the component automaton is itself a composite automaton, the component name consists of a sequence of identifiers for the local component names followed by their parameters if any, each separated by a period, to the level of composite nesting. Note that the parameters of a local component are enclosed in brackets, whereas the parameters of a transition or a trajectory are enclosed in parentheses.

18 Expressions

A Tempo expression may take on many forms, as described in the following sections.

18.1 Conditional Expressions

expr ::= 'if' *expr* 'then' *expr* 'else' *expr*

A conditional expression consists of the keyword `if`, an expression, the keyword `then`, a second expression, the keyword `else`, and a third expression. If the first expression evaluates to `true`, the value of the conditional expression is the value of the second expression. Otherwise, the value of the conditional expression is the value of the third expression.

18.2 Logical Expressions

expr ::= *expr* (<=> *expr*)+
| *expr* (=> *expr*)+
| *expr* (∨ *expr*)+
| *expr* (∧ *expr*)+
| ~ *expr*

A logical expression consists of two or more expressions separated by logical operators or a single expression preceded by a `~`. The value of a logical expression is a Boolean value. For two expressions separated by a logical operator, the value of the resulting expression is the equivalence of the two expressions for `<=>`, the implication of the second expression from the first expression for `=>`, the disjunction (or) of the two expressions for `\/`, and the conjunction (and) of the two expressions for `/\`. A `~` preceding an expression negates the value of the expression. All logical operators group left to right.

18.3 Relational Expressions

expr ::= *expr* (= *expr* | ~= *expr*)+
| *expr* (< *expr* | > *expr* | <= *expr* | >= *expr*)+

A relational expression consists of two or more expressions separated by relational operators. The value of the resulting expression is a Boolean value. For two expressions separated by a relational operator, the value of the resulting expression is `true` if the indicated relationship holds between the values of the two expressions, and `false` otherwise. All relational operators group left to right.

18.4 Arithmetic Expressions

expr ::= *expr* (+ *expr* | - *expr* | * *expr* | / *expr* | ** *expr*)+
| - *expr*

An arithmetic expression consists of two or more expressions separated by arithmetic operators or a single expression preceded by a `-`. For two expressions separated by an arithmetic operator, the value of the resulting arithmetic expression is the sum of the two expressions for `+`, the first expression minus the second expression for `-`, the product of the two expressions for `*`, the first expression divided by the second expression for `/`, and the first expression raised to the power of the second expression for `**`. A `-` preceding an expression negates the value of the expression. All the arithmetic operators except exponentiation group left to right; exponentiation groups right to left.

18.5 Expressions with Vocabulary-Defined Operator Symbols

expr ::=*expr* (*OPERATOR expr*)+

An expression may consist of two or more expressions separated by vocabulary-defined operator symbols. For two expressions separated by a vocabulary-defined operator symbol, the value of the resulting expression is the vocabulary-defined result of applying the operator to the left and right expressions. Vocabulary-defined operator symbols have the same precedence as addition and subtraction and group left to right.

18.6 Quantification Expressions

expr ::=(\A | \E)ID (: *type* | 'in' *expr*)*expr*

A quantification expression begins with either the universal quantification operator (denoted `\A`) or the existential quantification operator (denoted `\E`) and an identifier, corresponding to the name of the quantification variable. Next is either a data type specification for the quantification variable, consisting of a colon followed by the data type of the variable, or the keyword `in` followed by an expression for the domain set of the quantification variable. A quantification expression ends with the expression to be quantified. Note that unless the expression to be quantified is a single variable identifier or function invocation, it must be enclosed in parentheses.

The value of a universal quantification expression is true if the quantified expression evaluates to `true` for every value of the quantification variable, and `false` otherwise. The value of an existential quantification expression is true if the quantified expression evaluates to `true` for at least one value of the quantification variable, and `false` otherwise.

18.7 Constant Array Constructors

expr ::='constant' (*expr*)

A constant array constructor consists of the keyword `constant` followed by an arbitrary expression enclosed in parentheses. The value of the expression is a new array with the value of each element of the array initialized to the value of the parenthesized expression. The dimension of the array is inferred from the context of the expression.

18.8 Tuple Constructors

expr ::= [*expr* (, *expr*)*]

A tuple constructor consists of a list of one or more expressions, separated by commas and enclosed in square brackets. The value of the expression is a new tuple which has one field for each expression in the list. The initial values of the fields are the values of the associated expressions.

18.9 Set Constructors

expr ::= { ID : *type* 'where' *expr* } | { (*expr* (, *expr*)*)? }

Tempo provides two forms of set constructor expressions. The first begins with a left brace, an identifier, a colon, and a data type. The identifier is the name of a variable of the specified data type whose scope is local to the set constructor expression. These are followed by the keyword `where`, a predicate expression, and a right brace. The elements of the new set are all the values of the variable for which the predicate expression evaluates to `true`.

The second form of a set constructor expression consists of a left brace, a list of zero or more expressions separated by commas, and a right brace. The value of the expression is a new set with one element for each expression in the list. The value of each element is the value of the associated expression. An expression for the empty set consists of a left brace followed by a right brace.

18.10 Expressions with Vocabulary-Defined Mixfix Operators

expr ::= *expr* { (*expr* (, *expr*)*)? }

An expression with a vocabulary-defined mixfix operator consists of a beginning expression, a left brace, zero or more expressions separated by commas, and a right brace. The value of the expression is defined by the vocabulary.

18.11 Type Constraints

expr ::= *expr* : *type*

A type constraint consists of an expression followed by a colon and a data type. It is used to specify the data type of the expression if the type of the expression is not clear from its context.

18.12 Tuple, Union, and Array Elements

expr ::= *expr* (. ID | [*expr* (, *expr*)*])+

An expression for the value of the field of a tuple consists of an expression for the tuple followed by a period and an identifier corresponding to the name of the field within the tuple.

An expression for the value of a union element is restricted to one of the union's data types by following the expression for the union element with a period and the identifier corresponding to the name of the constructor for the element's data type.

An expression for the value of an array element consists of an expression for the array followed by one or more expressions, separated by commas and enclosed in square brackets, whose values are the indices of the element within the array.

18.13 Cast Operations

expr ::= (*type*) *expr*

A cast operation consists of a data type, enclosed in parentheses, followed by an expression. The cast operation converts the value of the expression to a value of the specified data type. As a result, some information may be lost, e.g., the value of (`Int`)5.6 is 5.

18.14 Function Invocations

expr ::=ID ((*expr* (, *expr*)*)?)

A function invocation consists of an identifier, corresponding to the name of the function, a left parentheses, zero or more expressions separated by commas, corresponding to the actual parameters of the function, and a right parenthesis. The value of a function invocation is the value returned by the function of that name when its formal parameters are replaced by the values of the expressions for the actual parameters.

18.15 Basic Expressions

expr ::=(INT | FLOAT | STRING | 'true' | 'false' | (*expr*) | ID | ID')

A basic expression is a constant, an arbitrary expression enclosed in parentheses, or an identifier. A constant is either an integer, consisting of one or more digits, a floating point number, consisting or one or more digits followed by a period and one or more digits, or a string, consisting of zero or more characters enclosed in quotation marks (").

The value of an expression enclosed in parentheses is the same as the value of the expression within the parentheses. Parentheses allow operations to be grouped in a different order than the order dictated by precedence.

The value of an identifier, when it is a basic expression, is the value of the state or local variable of that name within the current scope. In an ensuring clause, an identifier may be followed by an apostrophe; in that case the value of the expression is the value of the state or local variable after the transition is performed.

18.16 Values

value ::=(*expr* | *choice*)

A value is either an arbitrary expression or a choose operator. It is used to specify a new value for a variable in an assignment statement or to specify the value to be printed in a print statement.

18.17 Choose Operators

choice ::='choose' (*variable where*?)? *choiceNDR*?
variable ::=ID (: *typeRef*)?
choiceNDR ::='det' 'do' *ndrProgram*? 'od'
| *yield expr*

A choose operator begins with the keyword `choose`, followed by an optional constraint on the value chosen and an optional specification of the manner in which the value is chosen during a simulation. If a choose operator consists only of the keyword `choose`, its value is arbitrary and nondeterministic.

The value chosen may be limited to a value within a data type or to a value that satisfies a condition or both. The constraint specification, if any, begins with an identifier, corresponding to the name of a variable, local to the choose operator, whose value will be the value of the choose operator. The variable name may be followed by a colon and the data type of the variable. The constraint specification ends with an optional where clause, consisting of the keyword `where` and a predicate expression; the value chosen for the variable must be such that the predicate expression evaluates to `true`.

The value of a choose operator during a simulation may be specified in two ways. First, the value may be specified explicitly with the keyword `yield` and an expression whose value is the value of the choose operator. Second, a program may be provided to calculate the value of the choose operator. This is specified with the keywords `det do` followed by a nondeterminism resolution program and the keyword `od`.

18.18 Nondeterminism Resolution Programs

ndrProgram ::=*ndrStatement*+
ndrStatement ::=*lvalue* := *value* ;
| 'print' *value* ;
| 'if' *ndrCondRef* 'fi'
| 'while' *expr* 'do' *ndrProgram* 'od'
| 'for' ID 'in' *expr* 'do' *ndrProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *ndrProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *ndrProgram* 'od'
| 'yield' *expr* ;
| ;
ndrCondRec ::=*expr* 'then' *ndrProgram* ('elseif' *ndrCondRec* |'else' *ndrProgram*)?

A nondeterminism resolution program is a group of one or more statements, where valid statement types are assignment statements, print statements, if statements, while statements, for statements, yield statements, and empty statements. The first time a nondeterminism resolution program is executed, it begins execution with its first statement. Subsequent statements are executed sequentially until a yield statement is executed. At that time, execution of the program stops, and the value of the yield statement becomes the value of the associated choose operator. The next time the program is executed, it begins execution with the statement immediately following the yield statement where it stopped, and it continues executing sequential statements until the next

yield statement is executed. After the last statement in a nondeterminism resolution program is executed, execution continues with the first statement of the program.

19 Statements

effProgram ::=*effStmt*+
basicProgram ::=*basicStatement*+
compProgram ::=*compStmt*+
simulateProgram ::=*simulateStmt*+
ndrProgram ::=*ndrStatement*+
simProofProgram ::=*simProofStmt*+

Statements may appear in many places within a Tempo program. These places include transition effects, basic and composite automata schedules, simulation blocks, nondeterminism resolution programs, and simulation proofs. The Tempo grammar contains separate constructs for the valid statements in each of these locations. However, the form of each of the statement types is the same, regardless of the location. The discussions of the various statement types are combined by type and presented in the following sections.

19.1 Assignment Statements

effStmt ::=*lvalue* := *value* ;
basicStatement ::=*lvalue* := *value* ;
compStmt ::=*lvalue* := *value* ;
simulateStmt ::=*lvalue* := *value* ;
ndrStatement ::=*lvalue* := *value* ;
simProofStmt ::=*lvalue* := *value* ;

lvalue ::=ID ([*expr* (, *expr*)*] | . ID)*
value ::=(*expr* | *choice*)

An assignment statement changes the value of a state or local variable. The statement consists of the name of the state or variable to be changed, the assignment operator `:=`, the new value, and a semicolon. For primitive data types and complete objects of complex data types, the name of the state or local variable is just its identifier. For an element of an array, the name is the identifier of the array followed by one or more expressions, separated by commas and enclosed in square brackets, which evaluate to the indices of the element within the array. For a field of a tuple, the name is the identifier of the tuple, followed by a period and the identifier of the field within the tuple. Since an element of an array may be a tuple or another array and since a field of a tuple may be an array or another tuple, the array indices and tuple fields may be combined in any order and to any depth in order to specify the particular value to be changed.

The new value is specified with either an expression or a choose operator. The new value must be of the same data type as the state or local variable to be changed or a subtype of that data type.

19.2 Print Statements

effStmt ::=`'print'` *value* ;
basicStatement ::=`'print'` *value* ;
compStmt ::=`'print'` *value* ;
simulateStmt ::=`'print'` *value* ;
ndrStatement ::=`'print'` *value* ;
simProofStmt ::=`'print'` *value* ;

A print statement consists of the keyword `print` followed by either an expression or a choose operator and then a semicolon. The effect of a print statement is to print the value of the expression or choose operator on the standard output device.

19.3 If Statements

effStmt ::=`'if'` *effCondRec* `'fi'`
effCondRec ::=*expr* `'then'` *effProgram* (`'elseif'` *effCondRec* | `'else'` *effProgram*)?

basicStatement ::=`'if'` *basicCondRec* `'fi'`
basicCondRec ::=*expr* `'then'` *basicProgram* (`'elseif'` *basicCondRec* | `'else'` *basicProgram*)?

compStmt ::=`'if'` *compCondRed* `'fi'`
compCondRec ::=*expr* `'then'` *compProgram* (`'elseif'` *compCondRec* | `'else'` *compProgram*)?

simulateStmt ::=`'if'` *simCondRec* `'fi'`
simCondRec ::=*expr* `'then'` *simulateProgram* (`'elseif'` *simCondRec* | `'else'` *simulateProgram*)?

ndrStatement ::=`'if'` *ndrCondRef* `'fi'`
ndrCondRec ::=*expr* `'then'` *ndrProgram* (`'elseif'` *ndrCondRec* |`'else'` *ndrProgram*)?

simProofStmt ::=`'if'` *proofCondRec* `'fi'`
proofCondRec ::=*expr* `'then'` *simProofProgram* (`'elseif'` *proofCondRec* |`'else'` *simProofProgram*)?

An if statement selects from among one or more groups of instructions the particular group of instructions, if any, to be executed. The if statement begins with the keyword `if` followed by a predicate expression, the keyword `then`, and a group of one or more statements. Next are zero or more occurrences of the keyword `elseif` followed by another predicate expression, the keyword `then`, and another group of statements. These optionally may be followed by the keyword `else` and a group of statements. An if statement ends with the keyword `fi`.

The effect of executing an if statement is to execute the group of statements immediately following the first predicate expression that evaluates to `true`, or if none of the predicate expressions evaluate to `true`, to execute the group of statements following the `else`. If there is no `else` and none of the predicate expressions evaluate to `true`, no group of statements are executed as a result of the if statement.

19.4 While Statements

effStmt ::='while' *expr* 'do' *effProgram* 'od'
basicStatement ::='while' *expr* 'do' *basicProgram* 'od'
compStmt ::='while' *expr* 'do' *compProgram* 'od'
simulateStmt ::='while' *expr* 'do' *simulateProgram* 'od'
ndrStatement ::='while' *expr* 'do' *ndrProgram* 'od'
simProofStmt ::='while' *expr* 'do' *simProofProgram* 'od'

A while statement repeatedly executes a group of statements as long as a specified condition is satisfied. The while statement begins with the keyword `while` followed by a predicate expression, the keyword `do`, one or more statements, and the keyword `od`. The predicate expression is evaluated, and if it evaluates to `true`, the group of statements between the `do` and `od` are executed sequentially. The predicate expression is then reevaluated. The group of statements are repeatedly executed and the predicate expression is reevaluated until expression evaluates to `false`.

19.5 For Statements

effStmt ::='for' ID 'in' *expr* 'do' *effProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *effProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *effProgram* 'od'

basicStatement ::='for' ID 'in' *expr* 'do' *basicProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *basicProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *basicProgram* 'od'

compStmt ::='for' ID 'in' *expr* 'do' *compProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *compProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *compProgram* 'od'

simulateStmt ::='for' ID 'in' *expr* 'do' *simulateProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *simulateProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *simulateProgram* 'od'

ndrStatement ::='for' ID 'in' *expr* 'do' *ndrProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *ndrProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *ndrProgram* 'od'

simProofStmt ::='for' ID 'in' *expr* 'do' *simProofProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *simProofProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *simProofProgram* 'od'

A for statement executes a group of statements once for each value of a variable that satisfies a condition. The for statement begins with the keyword `for`, an identifier for the iteration variable, a colon, and the data type of the variable. These are followed by the condition to be satisfied, the keyword `do`, one or more statements, and the keyword `od`.

The condition to be satisfied may be specified in two ways. The first consists of the keyword `in` followed by an expression for a collection. The iteration variable is assigned to each member of the collection, in turn, and the group of statements are executed with each value. The order in which the elements of the collection are assigned to the variable is not specified. If the data type

of the variable can be determined from the data type of the collection, the colon and data type of the variable may be omitted from the for statement.

The second variation for the condition to be satisfied consists of the keyword `where` followed by a predicate expression. The expression is evaluated, and if it evaluates to `true`, the group of statements are executed. The predicate expression is repeatedly evaluated and the statements are executed until the expression evaluates to `false`.

19.6 Fire Statements

```
basicStatement ::='fire' ;
                 | 'fire' 'input' ID (( expr (, expr)* ) )? ;
                 | 'fire' 'output' ID (( expr (, expr)* ) )? ;
                 | 'fire' 'internal' ID (( expr (, expr)* ) )? ;

compStmt ::='fire' 'input' compInstance . ID (( expr (, expr)* ) )? ;
           | 'fire' 'output' compInstance . ID (( expr (, expr)* ) )? ;
           | 'fire' 'internal' compInstance . ID (( expr (, expr)* ) )? ;
compInstance ::=ID ([ expr (, expr)* ] )? (. ID ([ expr (, expr)* ] )? )*

simProofStmt ::='fire' msgInvoke ('using' proofUsings)? ;
msgInvoke ::='input' (compInstance . )? ID (( expr (, expr)* ) )?
            | 'output' (compInstance . )? ID (( expr (, expr)* ) )?
            | 'internal' (compInstance . )? ID (( expr (, expr)* ) )?
proofUsings ::=expr 'for' ID (, expr 'for' ID)*
```

A fire statement causes a transition to be fired in a simulation run. The fire statement consists of the keyword `fire` followed by the keyword for the type of the transition to be fired (`input`, `output`, or `internal`), the name of the transition, expressions for its actual parameters, if any, separated by commas and enclosed in parentheses, and a semicolon. For schedules and simulation relation proofs for basic automata, the name of the transition is its identifier. For schedules and simulation relation proofs for composite automata, the name of the transition is the name of the component automaton in which the transition is defined, followed by a period and the identifier for the transition. The component name is an identifier, corresponding to the local name of the component within the composite automaton, followed by expressions for the values of its actual parameters, if any, separated by commas and enclosed in brackets. If the component is itself a composite component, its local component name and parameters are followed by a period and the identifier and parameters of its component, continuing in this manner through the levels of composite nesting until the component in which the transition is defined is identified.

A fire statement within a simulation proof may have an optional list of substitutions just before the semicolon. This list consists of the keyword `using` followed by one or more substitutions, each of which is an expression followed by the keyword `for` and an identifier. The value of expression is substituted for the identifier when the transition is fired.

19.7 Follow Statements

```
basicStatement ::='follow' ID 'duration' expr ;
```

compStmt ::='follow' *compTrajList* 'duration' *expr* ;
compTrajList ::=*componentTrajectory* (, *componentTrajectory*)*
componentTrajectory ::=*compInstance* . ID ((*expr* (, *expr*)*))?
compInstance ::=ID ([*expr* (, *expr*)*])? (. ID ([*expr* (, *expr*)*])?)*

simProofStmt ::='follow' (*compInstance* .)? ID 'duration' *expr* ;

A follow statement causes a trajectory to be followed during a simulation run. The follow statement begins with the keyword `follow`, the name of the trajectory to be followed, and expressions for the trajectory's actual parameters, if any, separated by commas and enclosed in parentheses. Next are the keyword `duration`, an expression denoting the length of time the trajectory should be followed, and a semicolon. For follow statements in a schedule or simulation relation proof for a basic automaton, the name of the trajectory is its identifier. For schedules and simulation relation proofs for composite automata, the name of the trajectory is the name of the component automaton in which the trajectory is defined, followed by a period and the identifier for the trajectory. The component name is an identifier, corresponding to the local name of the component within the composite automaton, followed by expressions for the values of its actual parameters, if any, separated by commas and enclosed in brackets. If the component is itself a composite component, its local component name and parameters are followed by a period and the identifier and parameters of its component, continuing in this manner through the levels of composite nesting until the component in which the trajectory is defined is identified. A follow statement in a composite automaton optionally may specify a list of trajectories, separated by commas, to be followed concurrently.

19.8 Run Statements

simulateStmt ::='run' *componentDef* ;
componentDef ::=ID ((*actual* (, *actual*)*))?

A run statement consists of the keyword `run` followed by an identifier, corresponding to the name of the automaton to be run, a list of expressions for the values of the automaton's actual parameters, if any, separated by commas and enclosed in parentheses, and a semicolon. The effect of the run statement is to execute the statements in the schedule of the named automaton. Run statements may only appear in simulation blocks.

19.9 Yield Statements

ndrStatement ::='yield' *expr* ;

A yield statement consists of the keyword `yield` followed by an expression and a semicolon. The yield statement may only appear in a nondeterminism resolution program. The value of the expression becomes the value returned by the `choose` operator associated with this nondeterminism resolution program. The next time the `choose` operator is invoked, execution of the nondeterminism resolution program begins with the statement immediately after the yield statement.

19.10 Empty Statements

effStmt ::=;
basicStatement ::=;
compStmt ::=;
simulateStmt ::=;
ndrStatement ::=;
simProofStmt ::=;

An empty statement is just a semicolon. It has no effect.

20 Simulation Blocks

simProg ::='simulate' 'do' *simulateProgram* 'od'
simulateProgram ::=*simulateStmt*+
simulateStmt ::=*lvalue* := *value* ;
| 'print' *value* ;
| 'if' *simCondRec* 'fi'
| 'while' *expr* 'do' *simulateProgram* 'od'
| 'for' ID 'in' *expr* 'do' *simulateProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *simulateProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *simulateProgram* 'od'
| 'run' *componentDef* ;
| ;
simCondRec ::=*expr* 'then' *simulateProgram* ('elseif' *simCondRec* | 'else' *simulateProgram*)?

A simulation block begins with the keywords `simulate do` followed by one or more statements and the keyword `od`. Statement types that are valid in a simulation block are assignment statements, print statements, if statements, while statements, for statements, run statements, and empty statements. The statements are executed sequentially when the Tempo program is run in a simulation. A simulation block is required if more than one automaton is defined in the Tempo program of if the automaton to be simulated is parameterized.

21 Simulation Relations

simDef ::='forward' *simulationCore* 'end'
| 'backward' *simulationCore* 'end'
simulationCore ::='simulation' ID (*formals where*?)?
'from' ID : *componentDef*
'to' ID : *componentDef*
'mapping' (*expr*;)+ *imports*? *simProof*?
formals ::=(*formal* (, *formal*)*)
formal ::=ID (, ID)* : (*typeRef* | 'Type')
where ::='where' *expr*
componentDef ::=ID ((*actual* (, *actual*)*))?

A simulation relation expresses a relationship between the behaviors of two automata. It begins with the keywords `forward simulation` or `backward simulation`, describing the direction of simulation, followed by an identifier, corresponding to the name of the relation. The simulation

relation may have a list of the formal parameters, separated by commas and enclosed in parentheses. Each formal parameter consists of an identifier, corresponding to the name of the parameter, followed by a colon and its data type or the keyword `Type`. If multiple, adjacent parameters are of the same data type, their identifiers may be separated by commas and followed by a single colon and their common data type. The range of parameter values may be constrained with an optional where clause, consisting of the keyword `where` followed by a predicate expression. The simulation relation is valid only when the values of the actual parameters are such that the where clause expression evaluates to `true`.

Next, the two automata involved in the simulation relation are specified with the keyword `from`, a designation of the first automaton, the keyword `to`, and a designation of the second automaton. Each automaton designation consists of an identifier, corresponding to a local name for the automaton within the simulation relation, followed by a colon, a second identifier, corresponding to the name of the automaton, and a list of the actual parameters of the automaton, if any, separated by commas and enclosed in parentheses.

The mapping between the two automata is specified with the keyword `mapping` followed by one or more predicate expressions, each ending in a semicolon. Within these expressions, a state variable of either automaton is referenced by the local name of the automaton, followed by a period and the name of the variable within the automaton. The simulation relation holds if all the predicate expressions of the mapping evaluate to `true` in every pair of initial states of the two automata, and continue to be `true` after every transition and trajectory of the "from" automaton. Such a proof may be done by hand or with an interactive theorem-prover such as PVS.

A simulation relation ends with an optional import statement (see Section 15.2), an optional simulation relation proof, and the keyword `end`.

21.1 Simulation Relation Proof

simProof ::= `'proof'` *states*? *simProofStart*? *simProofEntries*?
simProofStart ::= `'start'` (*lvalue* `:=` *expr* `;`)+
simProofEntries ::= (*transEntry* | *trajEntry*)+
transEntry ::= `'for'` `'input'` (*compInstance* `.`)? ID (`(` ID (`,` ID)* `)`)? *simProofAction*
| `'for'` `'output'` (*compInstance* `.`)? ID (`(` ID (`,` ID)* `)`)? *simProofAction*
| `'for'` `'internal'` (*compInstance* `.`)? ID (`(` ID (`,` ID)* `)`)? *simProofAction*
trajEntry ::= `'for'` `'trajectory'` (*compInstance* `.`)? ID (`(` ID (`,` ID)* `)`)? `'duration'` *expr simProofAction*

A simulation relation proof begins with the keyword `proof` followed by an optional list of states (see Section 17.1.2), an optional initialization of variables, and an optional list of transitions and trajectories of the `from` automaton, each with an associated proof program. The initialization of variables, if present, begins with the keyword `start` followed by one or more assignment statements (see Section 19.1).

Each entry in the list of transitions and trajectories begins with the keywords `for` and either `input`, `output`, `internal`, or `trajectory`, depending on the type of the transition or trajectory. If the transition or trajectory is located in a composite automaton, the name of its component automaton is next. These are followed by an identifier, corresponding to the name of the transition or trajectory, and a list of identifiers, separated by commas and enclosed in parentheses, for its parameters, if any. Next, for trajectories, is the keyword `duration` followed by an expression for the length of time the trajectory should be followed. Finally, each entry ends with a simulation relation

proof program specifying the actions of the `to` automaton which correspond to this transition or trajectory.

21.2 Simulation Relation Proof Programs

```
simProofAction ::='ignore'
                | 'do' simProofProgram 'od'
simProofProgram ::=simProofStmt+
simProofStmt ::=lvalue := value ;
               | 'print' value ;
               | 'if' proofCondRec 'fi'
               | 'while' expr 'do' simProofProgram 'od'
               | 'for' ID 'in' expr 'do' simProofProgram 'od'
               | 'for' ID : typeRef 'in' expr 'do' simProofProgram 'od'
               | 'for' ID : typeRef 'where' expr 'do' simProofProgram 'od'
               | 'fire' msgInvoke ('using' proofUsings)? ;
               | 'follow' (compInstance . )? ID 'duration' expr ;
               | ;
proofCondRec ::=expr 'then' simProofProgram ('elseif' proofCondRec |'else' simProofProgram )?
msgInvoke ::='input' (compInstance . )? ID (( expr (, expr)* ) )?
            | 'output' (compInstance . )? ID (( expr (, expr)* ) )?
            | 'internal' (compInstance . )? ID (( expr (, expr)* ) )?
proofUsings ::=expr 'for' ID (, expr 'for' ID)*
```

A transition or trajectory proof program may consist only of the keyword `ignore`, in which case the associated transition or trajectory is not considered in the simulation relation proof. Otherwise, the proof program begins with the keyword `do` followed by one or more simulation proof statements and the keyword `od`. Statement types that are valid in a simulation proof program are assignment statements, print statements, if statements, while statements, for statements, fire statements, follow statements, and empty statements.

Appendices

A Tempo Keywords

A.1 Reserved Words

automaton	fi	mapping	then
backward	fire	od	to
choose	follow	of	trajdef
components	for	operators	trajectories
const	forward	output	trajectory
constant	from	pre	transitions
d	hidden	print	true
defines	if	proof	Type
det	ignore	run	types
do	imports	schedule	urgent
duration	in	signature	using
eff	initially	simulate	vocabulary
else	input	simulation	when
elseif	internal	start	where
end	invariant	states	while
ensuring	let	stop	with
evolve	locals	tasks	yield
false			

A.2 Built-in Data Types

Array	Enumeration	Nat	Set
AugmentedReal	Int	Null	String
Bool	Map	Real	Tuple
Char	Mset	Seq	Union
DiscreteReal			

A.3 Keywords for Built-in Data Types

abs	floor	max	size
count	head	min	succ
defined	init	mod	tag
delete	insert	nil	tail
div	last	pred	update
embed	len	remove	val

B Operator Symbols

Mathematical Symbol	Tempo Symbol	Meaning
$\leftarrow$	`:=`	Assignment
$+$	`+`	Addition
$-$	`-`	Subtraction or set difference
$\times$	`*`	Multiplication
$/$	`/`	Division
$\hat{}$	`**`	Exponentiation
$=$	`=`	Equal to
$\neq$	`~=`	Not equal to
$<$	`<`	Less than
$\leq$	`<=`	Less than or equal to
$>$	`>`	Greater than
$\geq$	`>=`	Greater than or equal to
∞	`\infty`	Infinity
$\neg$	`~`	Negation (not)
$\wedge$	`/\`	Conjunction (and)
$\vee$	`\/`	Disjunction (or)
$\forall$	`\A`	For all
$\exists$	`\E`	There exists
$\Rightarrow$	`=>`	Implication (implies)
$\Leftrightarrow$	`<=>`	Logical equivalence (if and only if)
$\in$	`\in`	Member of
$\notin$	`\notin`	Not a member of
$\subset$	`\subset`	Proper subset
$\subseteq$	`\subseteq`	Subset
$\supset$	`\supset`	Proper superset
$\supseteq$	`\supseteq`	Superset
$\cup$	`\U`	Union
$\cap$	`\I`	Intersection
$\emptyset$	`{}`	Empty set or sequence
$\vdash$	`|-`	Append to sequence
$\dashv$	`-|`	Prepend to sequence
$\|\|$	`||`	Concatenation
--	`__`	Vocabulary placeholder
$\rightarrow$	`->`	Operator result for a vocabulary
	`!`	Reserved for future use

C Tempo Grammar

Top-level

spec ::=(*typeDecls* | *imports* | *include* | *funDecls* | *vocabDef* | *autoDef* | *invDef* | *simDef* | *simProg*)+ 'EOF'
imports ::='imports' *vocabRef* (, *vocabRef*)*
vocabRef ::=ID ((*actual* (, *actual*)*))?
include ::='include' STRING
funDecls ::='let' (*funDecl* ;)+
funDecl ::=ID ((ID (, ID)*)?) : *typeSignature* = *expr*
invDef ::='invariant' *idOrNumeral*? 'of' ID : *exprTerminated*
idOrNumerals ::=*idOrNumeral* (, *idOrNumeral*)*
idOrNumeral ::=(ID | INT)
actual ::=(*expr* | 'Type' *typeRef*)

Type declarations

typeDecls ::='types' *typeDecl* (, *typeDecl*)*
typeDecl ::=ID (: *typeRef*)?
typeRef ::=ID
    | ID [*typeList*]
    | 'Enumeration' [*idOrNumerals*]
    | 'Tuple' [*fieldDecls*]
    | 'Union' [*fieldDecls*]
fieldDecls ::=*fieldDecl* (, *fieldDecl*)*
fieldDecl ::=ID (, ID)* : *typeRef*
typeSignature ::=*typeList*? - > *typeRef*
typeList ::=*typeRef* (, *typeRef*)*

Vocabulary definitions

vocabDef ::='vocabulary' ID *formals*? *defines*? *importsAndTypes*? *operators*? 'end'
formals ::=(*formal* (, *formal*)*)
formal ::=ID (, ID)* : (*typeRef* | 'Type')
defines ::='defines' ID [*typeList*]
importsAndTypes ::=(*imports* | *typeDecls*)+
operators ::='operators' *opDecl* (, *opDecl*)*
opDecl ::=*rootOpName* (, *rootOpName*)* : *typeSignature*
rootOpName ::='if' __'then' __'else' __
    | *opName*
    | *idOrNumeral*
opName ::=*prefixSpec* | *infixSpec* | *mixfixSpec*
prefixSpec ::=(*opSym* | ~)__| . __| ID __
infixSpec ::=__(*opSym* __| . (__| ID)| ID __)
mixfixSpec ::=__? { (__(, __)*)? }
        | __? [(__(, __)*)?]
opSym ::=(OPERATOR | <=> | => | ∧ | ∨ | = | ~= | *plainOp*)
plainOp ::=(< | <= | > | >= | + | - | * | / | **)

Automaton definitions

autoDef ::='automaton' ID (*formals* *where*?)? *imports*? *autoCore*
where ::='where' *expr*
autoCore ::=(*basicAutomaton* | *composedAutomaton*)
basicAutomaton ::=*actionSignature*? *states* *funDecls*? *transitions*? *trajectories*? *tasks*? *schedule*?
actionSignature ::='signature' (*formalActions*)+
states ::='states' (*state* ;)+ *initially*?
| 'states'
transitions ::='transitions' (*transition*)+
trajectories ::='trajectories' (*trajectory*)+
tasks ::='tasks' (*task*)+
schedule ::='schedule' *states*? 'do' *basicProgram*? 'od'
state ::=ID : *typeRef* (:= *value*)?
initially ::='initially' *expr* ;

formalActions ::='input' *formalAction* (, *formalAction*)*
| 'output' *formalAction* (, *formalAction*)*
| 'internal' *formalAction* (, *formalAction*)*
formalAction ::=ID ((*sigFormal* (, *sigFormal*)*) *where*?)?
sigFormal ::='const' *expr*
| ID (, ID)* : *typeRef*

Automaton transitions and trajectories

transition ::='input' *transitionCore*
| 'output' *transitionCore*
| 'internal' *transitionCore*
transitionCore ::=ID ((*actionActuals*) *where*?)? *localVars*? *funDecls*? *precondition*? *urgency*? *effect*?
actionActuals ::=*expr* (, *expr*)*
localVars ::='locals' (*localDecl* ;)+
precondition ::='pre' *exprTerminated*
urgency ::='urgent' 'when' *expr* ;
effect ::='eff' *effProgram* ('ensuring' *expr* ;)?
localDecl ::=ID : *typeRef* (:= *value*)?
exprTerminated ::=(*expr* ;)+
effProgram ::=*effStmt*+
effStmt ::=*lvalue* := *value* ;
| 'print' *value* ;
| 'if' *effCondRec* 'fi'
| 'while' *expr* 'do' *effProgram* 'od'
| 'for' ID 'in' *expr* 'do' *effProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *effProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *effProgram* 'od'
| ;
effCondRec ::=*expr* 'then' *effProgram* ('elseif' *effCondRec* | 'else' *effProgram*)?

trajectory ::='trajdef' ID (*formals* *where*?)? *funDecls*? *trajInvariant*? *stopCond*? *evolve*?
trajInvariant ::='invariant' *exprTerminated*
stopCond ::='stop' 'when' *expr* ;
evolve ::='evolve' *exprTerminated*
task ::={ *actionSet* (, *actionSet*)* } *forClause*?

actionSet ::=(*compInstance* .)? ID ((*expr* (, *expr*)*) *where*?)?
forClause ::='for' *varList* (, *varList*)* *where*?
varList ::=ID (, ID)* : *typeRef*

Composed Automaton

composedAutomaton ::=*components hiddenActionSets*? *compSchedule*?
components ::='components' (*component* ;)+
hiddenActionSets ::='hidden' (*actionSet* ;)+
compSchedule ::='schedule' *states*? 'do' *compProgram*? 'od'
component ::=ID ([*varList* (, *varList*)*])? (: *componentDef*)? *where*?
componentDef ::=ID ((*actual* (, *actual*)*))?

Simulation Relations

simDef ::='forward' *simulationCore* 'end'
| 'backward' *simulationCore* 'end'

simulationCore ::='simulation' ID (*formals where*?)?
'from' ID : *componentDef*
'to' ID : *componentDef*
'mapping' (*expr*;)+ *imports*? *simProof*?
simProof ::='proof' *states*? *simProofStart*? *simProofEntries*?
simProofStart ::='start' (*lvalue* := *expr* ;)+
simProofEntries ::=(*transEntry* | *trajEntry*)+
transEntry ::='for' 'input' (*compInstance* .)? ID ((ID (, ID)*))? *simProofAction*
| 'for' 'output' (*compInstance* .)? ID ((ID (, ID)*))? *simProofAction*
| 'for' 'internal' (*compInstance* .)? ID ((ID (, ID)*))? *simProofAction*
trajEntry ::='for' 'trajectory' (*compInstance* .)? ID ((ID (, ID)*))? 'duration' *expr simProofAction*
simProofAction ::='ignore'
| 'do' *simProofProgram* 'od'
simProofProgram ::=*simProofStmt*+

simProofStmt ::=*lvalue* := *value* ;
| 'print' *value* ;
| 'if' *proofCondRec* 'fi'
| 'while' *expr* 'do' *simProofProgram* 'od'
| 'for' ID 'in' *expr* 'do' *simProofProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *simProofProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *simProofProgram* 'od'
| 'fire' *msgInvoke* ('using' *proofUsings*)? ;
| 'follow' (*compInstance* .)? ID 'duration' *expr* ;
| ;
proofCondRec ::=*expr* 'then' *simProofProgram* ('elseif' *proofCondRec* |'else' *simProofProgram*)?
msgInvoke ::='input' (*compInstance* .)? ID ((*expr* (, *expr*)*))?
| 'output' (*compInstance* .)? ID ((*expr* (, *expr*)*))?
| 'internal' (*compInstance* .)? ID ((*expr* (, *expr*)*))?
proofUsings ::=*expr* 'for' ID (, *expr* 'for' ID)*

Expressions

expr ::='if' *expr* 'then' *expr* 'else' *expr*
| *expr* (<=> *expr*)+
| *expr* (=> *expr*)+
| *expr* (∨ *expr*)+
| *expr* (∧ *expr*)+
| *expr* (= *expr* | ~= *expr*)+
| *expr* (< *expr* | > *expr* | <= *expr* | >= *expr*)+
| *expr* (+ *expr* | - *expr* | * *expr* | / *expr* | ** *expr*)+
| *expr* (*OPERATOR expr*)+
| - *expr*
| ~ *expr*
| (\A | \E)ID (: *type* | 'in' *expr*)*expr*
| 'constant' (*expr*)
| [*expr* (, *expr*)*]
| { ID : *type* 'where' *expr* } | { (*expr* (, *expr*)*)? }
| *expr* { (*expr* (, *expr*)*)? }
| *expr* : *type*
| *expr* (. ID | [*expr* (, *expr*)*])+
| (*type*) *expr*
| ID ((*expr* (, *expr*)*)?)
| (INT | FLOAT | STRING | 'true' | 'false' | (*expr*) | ID | ID')
lvalue ::=ID ([*expr* (, *expr*)*] | . ID)*
value ::=(*expr* | *choice*)
choice ::='choose' (*variable where*?)? *choiceNDR*?
variable ::=ID (: *typeRef*)?
choiceNDR ::='det' 'do' *ndrProgram*? 'od'
| *yield expr*

Imperative Programs

basicProgram ::=*basicStatement*+
basicStatement ::=*lvalue* := *value* ;
| 'print' *value* ;
| 'if' *basicCondRec* 'fi'
| 'while' *expr* 'do' *basicProgram* 'od'
| 'for' ID 'in' *expr* 'do' *basicProgram* 'od'
| 'for' ID : *typeRef* 'in' *expr* 'do' *basicProgram* 'od'
| 'for' ID : *typeRef* 'where' *expr* 'do' *basicProgram* 'od'
| 'fire' ;
| 'fire' 'input' ID ((*expr* (, *expr*)*))? ;
| 'fire' 'output' ID ((*expr* (, *expr*)*))? ;
| 'fire' 'internal' ID ((*expr* (, *expr*)*))? ;
| 'follow' ID 'duration' *expr* ;
| ;
basicCondRec ::=*expr* 'then' *basicProgram* ('elseif' *basicCondRec* | 'else' *basicProgram*)?

compProgram ::=*compStmt*+
compStmt ::=*lvalue* := *value* ;
| *print value* ;
| 'if' *compCondRed* 'fi'

```
        | 'while' expr 'do' compProgram 'od'
        | 'for' ID 'in' expr 'do' compProgram 'od'
        | 'for' ID : typeRef 'in' expr 'do' compProgram 'od'
        | 'for' ID : typeRef 'where' expr 'do' compProgram 'od'
        | 'fire' 'input' compInstance . ID (( expr (, expr)* ) )? ;
        | 'fire' 'output' compInstance . ID (( expr (, expr)* ) )? ;
        | 'fire' 'internal' compInstance . ID (( expr (, expr)* ) )? ;
        | 'follow' compTrajList 'duration' expr ;
        | ;
compCondRec ::=expr 'then' compProgram ('elseif' compCondRec | 'else' compProgram )?
compInstance ::=ID ([ expr (, expr)* ] )? (. ID ([ expr (, expr)* ] )? )*
compTrajList ::=componentTrajectory (, componentTrajectory )*
componentTrajectory ::=compInstance . ID (( expr (, expr)* ) )?

ndrProgram ::=ndrStatement+
ndrStatement ::=lvalue := value ;
        | 'print' value ;
        | 'if' ndrCondRef 'fi'
        | 'while' expr 'do' ndrProgram 'od'
        | 'for' ID 'in' expr 'do' ndrProgram 'od'
        | 'for' ID : typeRef 'in' expr 'do' ndrProgram 'od'
        | 'for' ID : typeRef 'where' expr 'do' ndrProgram 'od'
        | 'yield' expr ;
        | ;
ndrCondRec ::=expr 'then' ndrProgram ('elseif' ndrCondRec |'else' ndrProgram )?
```

Simulation blocks

```
simProg ::='simulate' 'do' simulateProgram 'od'
simulateProgram ::=simulateStmt+
simulateStmt ::=lvalue := value ;
        | 'print' value ;
        | 'if' simCondRec 'fi'
        | 'while' expr 'do' simulateProgram 'od'
        | 'for' ID 'in' expr 'do' simulateProgram 'od'
        | 'for' ID : typeRef 'in' expr 'do' simulateProgram 'od'
        | 'for' ID : typeRef 'where' expr 'do' simulateProgram 'od'
        | 'run' componentDef ;
        | ;
simCondRec ::=expr 'then' simulateProgram ('elseif' simCondRec | 'else' simulateProgram )?
```

References

[1] Stephen J. Garland, Nancy A. Lynch, Joshua A. Tauber, and Mandan Vaziri. *IOA User Guide and Reference Manual*. MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, 2003. Available at `http://theory.csail.mit.edu/tds/ioa/manual.ps`.

[2] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan-Claypool Publishers, May 2006. Also, revised and shortened version of Technical Report MIT-LCS-TR-917a (from 2004), MIT Laboratory for Computer Science, Cambridge, MA.

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

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

[5] N.A. Lynch, R. Segala, and F.W. Vaandrager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003. Also Technical Report MIT-LCS-TR-827d, MIT Laboratory for Computer Science.

[6] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC 1987)*, pages 137–151, Vancouver, British Columbia, Canada, August 1987.

[7] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[8] Sayan Mitra. *A Verification Framework for Ordinary and Probabilistic Hybrid Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2007. To appear.

[9] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer Verlag, 1996.

[10] H. B. Weinberg. Correctness of vehicle control systems: A case study. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, February 1996. Also, MIT/LCS/TR-685.

[11] H. B. Weinberg and Nancy Lynch. Correctness of vehicle control systems: A case study. In *17th IEEE Real-Time Systems Symposium (RTSS 1996)*, pages 62–72, Washington, D. C., December 1996.

6.852J / 18.437J Distributed Algorithms
Fall 2009