Using Timed Input/Output Automata for Implementing Distributed Systems

Peter M. Musial
CSAIL, MIT, MA, USA
pmmusial@csail.mit.edu

Abstract

The objective of this work is the derivation of software that is verifiably correct. Our approach is to abstract system specifications and model these in a formal framework called Timed Input/Output Automata, which provides a notation for expressing distributed systems and mathematical support for reasoning about their properties. Although formal reasoning is easier at an abstract level, it is not clear how to transform these abstractions into executable code. During system implementation, when an abstract system specification is left up to human interpretation, then this opens a possibility of undesirable behaviors being introduced into the final code, thereby nullifying all formal efforts. This manuscript addresses this issue and presents a set of transformation methods for systems described as a network to timed automata into Java codes for distributed platforms. We prove that the presented transformation methods preserve guarantees of the source specifications, and therefore, result in code that is correct by construction.

Keywords: Model Driven Development, Verifiable Distributed Code, Code Generation
Contents

1 Introduction ........................................ 4

2 TEMPO Mathematical Framework .................. 5
   2.1 Execution of Timed I/O Automata ............... 5
   2.2 Proof Techniques ................................ 5
   2.3 Using Timed I/O Automata ....................... 6
   2.4 Successes with the I/O Automaton Model ....... 7

3 Prior development and recent progress .......... 7
   3.1 Related Frameworks .............................. 7
   3.2 IOA Toolkit ..................................... 7

4 TEMPO to Java Plug-in ............................ 8
   4.1 Connecting programs to system services ....... 9
   4.2 Modeling procedure calls ....................... 9
   4.3 Composing automata ............................ 10
   4.4 Resolving nondeterminism and liveness ....... 10
   4.5 Implementing datatypes ......................... 11

5 TEMPO 2 Java Transformation ................... 11
   5.1 Translation Layout .............................. 11
   5.2 Illegal and Unsupported TEMPO Constructs .... 12
   5.3 Data Types and Variables ....................... 12
   5.4 State, Schedule and Local Variables .......... 13
   5.5 Transitions ..................................... 15
       5.5.1 Transition Output Parameters ............... 16
   5.6 Trajectories .................................... 17
   5.7 Schedule and Action Matching ................. 19
   5.8 Translating MPI and TCP Transitions .......... 20

6 Translation’s Correctness ......................... 25

7 How to Use the Translation Module ............... 29

8 Evaluation ......................................... 31
   8.1 The Paxos Algorithm ............................ 31
   8.2 Clock Synchronization ........................... 32
   8.3 Partial Reversal Algorithm ..................... 34

9 Future Work ....................................... 38

10 Summary .......................................... 39
### A Modeling TCP channel

- A.1 Sender mediator ........................................ 42
- A.2 Receive mediator ........................................ 42
- A.3 Channel mediator ......................................... 43
- A.4 Complete channel system ................................. 43
- A.5 Channel proof ............................................ 43

### B TCP Lossy Channel Abstraction

- B.1 TCP Send Mediator Automaton .......................... 47
- B.2 TCP Receive Mediator Automaton ....................... 48
- B.3 TCP Channel Mediator Automaton ....................... 50
- B.4 TCP Channel Vocabulary ................................ 54

### C Paxos Specialized with TCP Channels

- C.1 The Paxos Node ........................................... 56
- C.2 The Starter Algorithm .................................... 60
- C.3 The Detector Algorithm .................................. 62
- C.4 The Leader Election Algorithm ......................... 64
- C.5 The Paxos Leader Algorithm ............................ 65
- C.6 The Paxos Agent Algorithm ............................. 69
- C.7 The Paxos Success Algorithm ......................... 70
- C.8 The Vocabulary for Paxos ............................... 73

### D Paxos Specialized with MPI Channels

- D.1 Paxos Node ................................................... 75
- D.2 The Starter Algorithm .................................... 78
- D.3 The Detector Algorithm .................................. 79
- D.4 The Leader Election Algorithm ......................... 81
- D.5 The Paxos Leader Algorithm ............................ 82
- D.6 The Paxos Agent Algorithm ............................. 85
- D.7 The Paxos Success Algorithm ........................... 87
- D.8 The Vocabulary for Paxos ............................... 90
- D.9 MPI Send Mediator Automaton ........................... 90
- D.10 MPI Receive Mediator Automaton ..................... 91
1 Introduction

Developing dependable distributed systems for modern computing platforms continues to be challenging. While the availability of distributed middleware makes feasible the construction of systems that run on distributed platforms, ensuring that the resulting systems satisfy specific safety, timing, and fault-tolerance requirements remains problematic. The middleware services used for constructing distributed software are specified informally and without precise guarantees of efficiency, timing, scalability, compositionality, and fault tolerance. Current software-engineering practice limits the specification of such requirements to informal descriptions. When formal specifications are given, they are typically provided only for the system interfaces. (Middleware interface syntax is usually strongly defined, where computational semantics are often defined superficially.) The specification of interfaces alone stops far short of satisfying the needs of users of critical systems. Such systems need to be equipped with precise specifications of their semantics and guaranteed behavior. When a system is built of smaller components, it is important to specify the properties of the system in terms of the properties of its components.

In designing distributed systems, the best practice today involves a patchwork of specifications, including graphical object modeling tools, manual documentation of component interaction, formal specification of interfaces, descriptions of algorithms and protocols at varying degrees of formalism, and specifications of distributed system configuration and deployment. Even when services and algorithms are specified formally, rigorous reasoning about the specifications is often left out of the development process. Without a comprehensive design framework, it is very difficult to ensure that all necessary types of specifications are produced within the design effort. It is usually the case that when a distributed system is first deployed, numerous forgotten or underspecified aspects of the system begin to surface. Granted that one is able to amass all necessary specifications, it is extremely difficult to deal with the dissimilar kinds of specifications and specification formats and media. During the development of a system, it becomes difficult to maintain traceability between the specifications and the resulting implementation. This problem is further aggravated when an existing distributed system needs to be refined, optimized, extended or redeployed. Many of these problems remain outside of the realm of academic research.

We view formal specification and analysis as valuable tools that should be at the disposal of the developers of distributed systems. However, theoretically sound specifications have a limited impact, unless tools exist that automatically transform these specifications from high level notation to executable code. Only if such tools are formally scrutinized, then the resulting executable code can be deemed as reliable and verifiably correct.

At the core of this work is the TEMPO framework and its toolkit. TEMPO is derived from the formal mathematical modeling framework called Timed Input/Output Automata (Timed I/O Automata) [23]. Timed I/O Automata framework is well established in the theoretical distributed computing research community and is used to specify and reason about distributed and concurrent algorithms. Systems in this framework are specified in terms of Timed I/O Automata that are interacting state machines. The TEMPO language closely matches semantics of the Timed I/O Automata modeling framework and hence it inherits the rich set of capabilities for system modeling and analysis.

The TEMPO toolkit developed by VeroModo, Inc. contains tools to support analysis of systems such as, a compiler that checks syntax and performs static semantic analysis; a simulator to produce and explore execution traces for an automaton; a translation module to the UPPAAL model-checker [29]; and a translation module to the PVS interactive theorem prover [39]. A goal of this proposal is to turn the TEMPO toolkit into a practical tool by extending it with the TEMPO-to-Java translator.

Our contribution is developing a strategy for generating distributed executable Java programs from sys-
tems specified in the TEMPO notation. With some restrictions on the properties of the source specifications, that will be discussed in this paper.

2 TEMPO Mathematical Framework

As aforementioned, the TEMPO framework was derived from Timed Input/Output Automata [23] model, which in turn has its roots in Input/Output Automata model [35]. In the nutshell, Timed Input/Output Automata (Timed I/O Automata) provide a mathematical basis for modeling and reasoning of distributed systems (both without and with timed components). Flexibility and power of this framework come from two of its properties. (i) System designer has the flexibility of using nondeterminism to allow multiple correct specifications of its system, hence relaxing assumptions on behaviors of the environment in which the system operates (at least at certain parts of the execution). (ii) Complex systems can be decomposed into subsystems, where composition of these subsystems yields the unified complex system. Such structured design enables one to view the specification at multiple levels of abstraction.

Timed I/O Automata model behavior of systems of interacting components (i.e., automata), where system components operate in discrete steps, and timing-related components whose behavior includes continuous transformation over time. For the timed components, behaviors involve continuous transformations over time. However, when the transformation reaches its stopping condition the time stops to allow one or more discrete interactions which are assumed to be instantaneous. This is an obvious departure from our intuitive understanding of physical time.

A timed automaton is a labeled state transition system. It consists of a (possibly infinite) set of states (including a nonempty subset of start states); a set of discrete actions (classified as input, output, or internal); and a transition relation, consisting of a set of (state, action, state) triples (transitions specifying the effects of the discrete automaton’s actions). Finally, as set of trajectories describing state evolution over time.

2.1 Execution of Timed I/O Automata

The operation of a timed I/O automaton is described by its executions, which are alternating sequences of trajectories and actions. The external behavior of a timed automaton is captured by the set of traces of its execution fragments, which record external actions and the trajectories that describe the intervening passage of time (external actions are those with input and output labels). Timed I/O automata can be composed together via the composition operation. In a composition of two or more automata external actions are identified and matched with those that have the same name in different component. When any automaton performs a discrete step involving an action, then so do all components that have a matching action in their external signature. In case of trajectories, the variable involved in the trajectory has to be external and if any component follows a particular trajectory for that external variable, then so do all component automata with the matching external variable.

2.2 Proof Techniques

The Timed I/O automaton model supports a rich set of proof techniques. Invariant assertion techniques are used to prove that properties of automata are true in all reachable states [38]. Compositional reasoning where properties of a system are evaluated by reasoning about each of its components individually [32]. Another important proof strategy is hierarchical proofs [31] where comparable automata are tested whether one implements the other; automata are comparable if they have the same external interface. Ideally, this
is done at different levels of abstraction. One automaton is said to implement another if all of its traces are also traces of the other automaton. Pairs of automata can be related using various forms of simulation relations. A simulation relation is a mapping between the states of two automata that is maintained in all reachable states while preserving the external behavior of the automata. To prove a relation is a simulation relation, one demonstrates a correspondence between states and a step correspondence between the two automata, that is, one shows that for every state transition of the implementation automaton there is an equivalent (possibly empty) sequence of state transitions that the specification automaton can take that will maintain the simulation relation [25, 31, 9]. Interestingly, simulations can also be used while reasoning about liveness properties of the implementation [14] which we will use to claim that the resulting executable codes do not add any additional safety guarantees of the source specifications. We will forgo further details on specifics of simulation relations, and leave the subject with the mention that there are two basic simulation relations (i) a forward simulation, and (ii) a backward simulations. There exist other types, which are derived from the aforementioned simulation relations. Simulations are useful tools that demonstrate correctness of specification, and are subject of implementation in the TEMPO toolkit (i.e., the simulator). However, the simulation constructs will not be part of the code generation since we are interested in the implementation of already proved correct specifications.

### 2.3 Using Timed I/O Automata

This section is best presented in terms of an example. Suppose that we are interested in developing a software system for the next age helicopter. At the highest level the helicopter can be represented as a single automaton. Its main functions can be captured using most general description of the externally desirable behaviors. These can for example capture the interaction of the environment with the system. To begin, one may define simple helicopter controls such as lift up and down, forward, backward, left and right motion. At this high level of abstraction it is useful to take advantage of nondeterministic choices if possible. In order to guarantee safety of its crew, the key properties of the system are defined using invariants. For instance, limits on maximum altitude, rate of helicopter descent, or maximum tilt change from the horizontal, etc.

A process of successive refinement then follows to describe the system as made up of lower-level services that are themselves modeled as automata. These lower-level automata may be simpler to understand (individually), a more realistic depiction of a real system, or a model of an existing system we wish to use. Two orthogonal methods of refinement apply. First, levels of abstraction may be used to define interfaces between low-level services and high-level applications. A navigation system of a helicopter depends on inputs from the low level hardware giro device that describes the precise tilt and rotation of machine with respect to the horizontal position. Second, one can apply parallel decomposition to describe a service or application as a collection of components. Given our example, navigation software may be dependent on a collection of computing devices that form a network, such as giro mechanism, speed measuring mechanism, atmospheric pressure sensor, etc. The result is a set of lower-level I/O automata that are intended to implement the previously specified high-level service. Having refined the high-level, global system specification into a low-level distributed system description, one wishes to prove that the low-level description implements the high-level specification. Doing so shows that all behaviors of the low-level system could be interpreted as valid behaviors of the specification. So, for example, the pilot of the helicopter is not allowed to put the machine in an unsafe state. To perform this proof, one applies the composition operator to the various pieces of the low-level, distributed system specification. To complete the proof, it suffices to demonstrate that a simulation relation between the resulting automaton and the high-level, global system specification is always preserved.
2.4 Successes with the I/O Automaton Model

I/O automata [32, 31] have been used to successfully model and verify a wide variety of distributed systems and algorithms [32, 33, 40, 10] and to express and prove several impossibility results. The model was developed for reasoning about theoretical distributed algorithms, but has since been applied to many practical services. For example, I/O automata have been applied to distributed shared memory [11, 34, 16], group communication [12], and standard networking [10]. The resulting expositions and proofs have resulted from a structured, rigorous approach that has resolved ambiguities and uncovered errors. Logical errors have been found in algorithms underlying Orca [3] and Ensemble [18] while unexpected behavior was found in T/TCP [5].

However, I/O automata model limits specifications to those that are purely nondeterministic. Noteworthy, a powerful modeling framework such as TEMPO can become even more attractive if it is supported by a comprehensive set of practical tools.

3 Prior development and recent progress

Automated code generation is a well established research subject, but in the recent years it has received increased interest. In specific to the manuscript, we will discuss two most closely related activities: the IOA toolkit [46], and TEMPO toolkit [23].

3.1 Related Frameworks

Several formal frameworks exist for modeling and reasoning about complex systems [19, 37, 7, 6, 26] (to name a few) and for which software support was developed [17, 36, 20, 8, 28]. These frameworks provide a high level notation that can be used to express concurrent systems (resp. distributed algorithms) at various levels of abstraction, and the mathematical support to reason about their properties. However, for the aforementioned and other frameworks we found that automated software development to be very limited or nonexistent. There is a documented success for automated code generation for embedded systems [22]. A natural question to ask if the same can be repeated for distributed systems in general or under what constraints. This work answers this question in part.

3.2 IOA Toolkit

Input/Output (I/O) Automata [31, 32] is the predecessor of Timed Input/Output Automata model and allows modeling of untimed systems. The IOA toolkit [21] has been developed as support for the I/O Automata model. Using the Input/Output Automata (IOA) notation wide range of systems can be specified and reasoned about and the IOA toolkit supports the design, development, testing, and formal verification of concurrent untimed systems.

There are few downsides to the existing IOA toolkit. Besides the obvious fact that it does not support modeling of timed systems, its code depends on libraries that are now poorly supported (such as PolyJ [4], which was needed since once upon a time Java did not provide constrained parametric polymorphism). Another down side is the decision to translate IOA language into an intermediate representation that is not very well documented and which is used by the various functional components of the toolkit. Therefore, any changes or extensions to the base language are not easily incorporated into the intermediate language representation (for example, how to extend the intermediate language to support trajectories). Also the
code generator for the IOA toolkit supports only communication through MPI, which limits its practicality. Finally, the IOA toolkit is now unsupported and is becoming badly out of date in respect to the new technologies.

Despite the above shortcomings, the theoretical work developed during creation of the IOA toolkit is instrumental to the activities of this work. Specifically, it defines the limitations on types of specifications that are allowed for compilation to executable code. These limitations will apply in our work as well.

TEMPO framework [30] has been developed from the Timed Input/Output Automata model [23]. The corresponding language provides mathematical notations for describing systems, their intended properties and relations between their descriptions at various levels of abstraction. The TEMPO toolkit is a suite of tools that support a range of validation methods for descriptions of system and their properties, included static analysis, simulation, and machine-checked proofs. Its implementation provides a robust development environment where new functionality is written in form of plug-ins. TEMPO toolkit is distributed under VeroModo, Inc. license, which is free for academic and research use.

Architecturally the TEMPO toolkit is build as a front-end supporting various plugins. The front-end performs static type checking of the input specification and based on it creates an Abstract Syntax Tree (AST) representation. The AST is then passed in as input to the supported plug-ins for further processing. New plug-ins can be developed and easily incorporated into this architecture.

Interaction with the toolkit is possible via graphical and command line interfaces, where the graphical interface is implemented on the Eclipse Rich Client Platform that provides the familiar development environment with error messaging and syntax highlighting.

4 TEMPO to Java Plug-in

The key challenge is to create a system with the correct externally visible behavior without using any synchronization between processes running on different machines. This goal is achieved by matching the formal specification of the distributed system to the target architecture of running systems. That is, we restrict the form of the TEMPO programs admissible for compilation and require the programmer rather than the compiler to decide on the distribution of computation. Meaning that the programs submitted for compilation are restricted to a node-channel form that reflects the message-passing architecture of the collection of networked processing units and properties of the communication medium. During the translation process, the send and receive interfaces will be automatically matched to the appropriate channel implementations. Currently the plug-in only supports channels implemented over MPI and TCP/IP sockets, providing more diversity is a subject of future work.

The above achieves communication in a networked setting. An alternative approach is to allow programs to communicate via shared-memory system, in that case we would require TEMPO programs to be written in that style. Currently this type of communication is not supported and is also subject of future work.

Given the generality of the TEMPO framework, each networked component can run a different node algorithm, hence in that case compilation proceeds on the node-by-node basis. Alternatively, a node algorithm could be general enough to allow it to be executed on any number of nodes, in this case it should either provide parameters or internal configuration to give it a unique label (for example, an IP address, MPI rank, or user specified label). If the specification is for a distributed system then all node algorithm must use some specialization of the underlying communication channel, such as MPI or TCP/IP. If TCP is used, then all nodes must include a predetermined server port number that allows for establishing of point-to-point connections.

Since we use Java as the target programming language, the above activity suffices as portability of Java
programs (through specialized Java Virtual Machines) hides specifics of the deployment. In order to further promote portability we have chosen MPI implementation presented in [2] (the MPJ Express).\footnote{The IOA toolkit depends on C based MPI libraries which require specific MPI versions and hence limit its portability.}

Additionally, we restrict input specifications to those that allow time to diverge from the physical time. This restriction originates from the challenge of modeling timed systems where time can progress in ways that conflict with the intuition of physical time. For example, TEMPO framework may force the time to stop, in addition it may allow infinitively many discrete actions to happen in a finite amount of time.

### 4.1 Connecting programs to system services

TEMPO node programs need to interact with external services such as communication networks and console inputs. The compiler needs to generate only the specialized code necessary to implement an algorithm at each node in the system. At runtime, each node connects to the external system services it uses. Our plug-in is capable of connecting to MPI network primitives and to the Java TCP sockets (an extension of [15]), hence supporting a cluster to WAN deployment platforms.

Connecting to any external service should bring that service into the TEMPO model by creating automata that make explicit all assumptions about its interfaces and about all its externally visible behaviors. This approach is documented in [46] where two automata are introduced that model a subset of MPI and facilitate send and receive actions between the algorithm automata and the channel automata (implemented using MPI).

The challenge is to prove the correctness of systems dependent on the external services. Distributed system designers can produce conditional proofs of correctness for an entire system even though nodes communicate via some form of network channel. The correctness of the systems accessing other external system services can be verified by following the same general approach. We can model an external service by writing a TEMPO specification. Subsequently, proofs of correctness about programs that use the service must consider the entire system including the modeled service. Such proofs are conditioned on the assumption that the external service behaves as described in our model. However, requiring the programmer to write code specifically to model the particulars of procedure calls to specific external services is more restrictive than necessary. Writing programs at such a low level complicates system designs unnecessarily and, thus, makes verifying the correctness of systems harder. We avoid this complexity by specifying abstract services designers want to use (e.g., point-to-point, reliable, FIFO channels, or point-to-point lossy channels) and then implementing these abstract services by combining our model of the external service with auxiliary mediator automata. We then verify that this design implements the desired abstract service. Such a proof does involve just the sort of details about low-level interactions with the external service we wish to avoid. However this proof need only be performed once to verify the compiler design. Programmers may then assume the existence of the simpler abstract service in any proofs about their programs. Thus, our strategy for verifying access to an external service is a four step process. (i) Model the external service as a TEMPO automaton. (ii) Identify the desired abstract service programmers would like to use and specify the TEMPO automaton to be compatible with the interface of the chosen abstract service. (iii) Import mediator automata such that the composition of the mediator automata and the external service automaton implements the abstract service automaton. (iv) Prove that implementation relationship.

### 4.2 Modeling procedure calls

Modeling procedure calls requires careful reasoning about behaviors of the call and proofs verifying the interface correct. Proving a model of a system call can be tricky, but is easier than the alternative. If
system calls are directly invoked from the TEMPO specification, then the state of the automaton will have
to include procedure call stacks, which complicates correctness proofs. Also, when one considers procedure
calls as the interface between interacting components in a concurrent setting, the procedure call and the
procedure return should often be considered to be two separately visible events across the interface. The
Java interface to an external service is defined in terms of method invocations (procedure calls). In the
specification of the IOA compiler [46] models of services are treated as method invocations and method
returns as distinct behaviors of the external service. When procedure calls may block, these are described
as handshake protocols to model such blocking. Although this approach is valid in for blocking system
calls, it substantially complicates specification, scheduling, and the resulting code. Therefore, in our work
we take a different approach and use TEMPO functions to model invocations to system calls that are non-
blocking. Blocking calls should be avoided since use of such calls will void properties of the underlying
Timed Input/Output Automata model.

4.3 Composing automata

In the TEMPO model all auxiliary mediator automata created to implement abstract system services are com-
bined with the source automaton prior through composition. The IOA compiler composes these automata
to form a single automaton that describes all the computation local to a single node in the system. The
compiler assumes that the composition across nodes is performed as part of proofs concerning the behavior
of an entire distributed system but not as part of compiling such a system. We further extend this assumption
and leave components in their decomposed state (unless this is carried out manually). This means that com-
position happens at runtime, where we match all automata interfaces and pass parameters in the process.
This approach leaves resulting code readable, modular, and manageable. Since at the point of compilation
the specification passed model checking and it was proved correct under assumption of composition, this
implies that the automata interfaces are safe to compose dynamically and the process is analogous to manual
composition. Specifically, our prototype performs action matching of the composed automata from within
its components. Action matching (a.k.a., action hiding) is restricted to actions that have fixed and a priori
known parameters with strict one-to-one type correlation.

4.4 Resolving nondeterminism and liveness

The TEMPO framework is inherently nondeterministic. Translating programs written in TEMPO into an
executable language such as Java requires resolving all nondeterministic behaviors. This process is called
scheduling an automaton. Developing a method to schedule automata is the key conceptual challenge in
the initial design of our code generator. In general, it is computationally infeasible to schedule TEMPO
programs automatically. Instead, TEMPO provides constructs specified in the nondeterminism resolution
language (NDR) [44] that enable resolution of nondeterminism through which developers can schedule
automata directly and safely.

The challenge with NDR schedules is that they are a meta structure appended to the automaton and are
not used in the proofs, so they are not verified to ensure liveness. NDR projects actions of a node and without
any synchronization with NDR segments at other nodes. Due to natural processing and communicating
delays, collective application of NDR schedules may result in a different execution (different external trace)
each time the system is executed. It is up to the programmer to ensure that (i) all possible executions
are safe, and that (ii) all possible executions satisfy system liveness conditions. Therefore, our translation
ensures liveness only if the NDR schedules satisfy system liveness conditions. Also, since compilation
proceeds on the node-by-node basis, either a single general schedule that governs actions across all nodes is provided or each node is equipped with its own schedule.

4.5 Implementing datatypes

The TEMPO framework provides a set of built-in data types. However, the meaning of these types changes during the implementation, and any correctness reasoning must incorporate this change. Specifically, implementations are bounded by physical constraints of the deployment architecture, hence numeric types are bound in size and precision.

User data types are allowed and during translation appropriate code is generated, however, body for user specified operators must be provided by the developer since there is not specific mechanism in TEMPO that allows such complete specification.

5 TEMPO 2 Java Transformation

In this section we present in detail how allowed TEMPO constructs are mapped into Java code. We begin with the Java code structure of the translated TEMPO model, and then proceed with a description of how each TEMPO construct is translated into the corresponding Java code.

5.1 Translation Layout

For presentation’s sake, let us use an example of the Paxos [41]. Specification for the two versions of Paxos, one specialized to use TCP and another to MPI, appear in the Appendix C and D respectively. Abstractly Paxos participants have access to unreliable communication channels, where in this work these are specialized TCP channels. We model Paxos as a composition of automata, each modeling a specific function – such as leader election component, ballot trigger component, etc. It is possible to place all automata specifications along with the needed vocabulary descriptions in a single file. However, this is discouraged. Instead each component is placed in its own file and all source files are linked together using the imports key word. In this example the main file is called TCPPaxos.tioa Let us also assume that the automaton modeling a complete Paxos system (i.e., the composition of all component automata) is called paxos and is located in TCPPaxos.tioa.

Note that the translation must be initiated with the appropriate flag indicating that the type of communication channel to be used will be TCP (i.e., -comm tcp or -comm mpi if one wishes to use MPI, but this requires import of the MPI channel model and appropriate vocabularies). If translation is successful, then the generated code will be placed in the directory called paxos.java which will be created in the same path location where paxos.tioa resides, let us call this a project root directory (or simply a root directory). It is possible for the main TIOA file to include more than one automaton that are not components of any other automaton. In this case each of these automata will be translated into a Java class and placed in the root directory.

Each component automaton (if any) will be placed in the package called Components in a class file derived from the name of the source automaton. For example, automaton called leaderelector which implements the leader election protocol is translated as Components/leaderelector.java.

All data types, which includes the native and the user specified types, are translated as Java class files and placed in the Datatypes package. For example, the Int type is translated as Datatypes/Int.java. (More on that later.)
Static sets, such as an empty set, that are used in the model for comparison, are included as Java classes and placed in the Macros package. (More on that later.)

Finally, a Utils package is created that includes Java classes commonly used by translated code. This package includes TJMath.java class that implements all TEMPO operations on the basic native data types, for example, test if two variables are equal is implemented as an EQ(Object p1, Object p2) method. Other classes in this package implement object copy mechanism. This is needed to ensure that during assignment objects are copied to a location separate in memory (Java uses shallow object references in assignments). Hence a deep bit-wise object copy is sometimes used to ensure variable independence and hence ensuring translation safety.\(^2\)

5.2 Illegal and Unsupported TEMPO Constructs

Currently the translator does not support quantified expressions, however, this should not be a limiting factor for most systems. Also, since TEMPO supports sets and multi-sets with 'there exists' tests and in conjunction with looping constructs, it should be possible to rewrite some types of quantified expressions into an equivalent imperative code. Smart fire of enabled actions is not supported, since this may be prohibitively expensive especially in presence of action parameters.

Simulation constructs and invariants are not translated. The meaning of translating a simulation and running two systems side by side may be very difficult to implement and will require costly synchronization. However, the TEMPO toolkit includes a simulator plugin which is best suited for such task. Implementation of invariants is subject to future work.

There are some additional syntax restrictions on the use of Java reserved key words in the source TEMPO specification. For instance, int can be used as a variable name in TEMPO spec, but cannot be in Java since it has a specific meaning. Therefore, input specifications are screened for use of Java reserved key words.

Finally, we state a caution on the use of guard statements that involve infinite precision numerical values. To give an example, a stop when condition that uses equality that involves an infinite precision number will not translate correctly due to the fact that Java types are finite precision and array/set/map sizes are bound. Hence, the source model has to be augmented in a way to support greater (less) than or equal guards with appropriately adjusted constant values.

5.3 Data Types and Variables

As aforementioned all data types are translated as Java classes in the Datatypes package. The TEMPO data types are closely matched with the corresponding Java types. However, it is important to distinguish the difference between TEMPO types that have a mathematical meaning and the implementation. For example, the TEMPO Nat and Int numeric types are implemented as Java’s BigInteger, and Real types are implemented as BigDecimal\(^3\). Therefore the translator makes best effort to accommodate large number and high precession. However, the usual hardware limitations apply during execution. This means that liveness reasoning for the source model specifications must include use of subsets of the data types rather than their true mathematical meaning. Safety of the model cannot be violated due to use of final precision

---

\(^2\)TEMPO notation does not have any notion of a reference or memory, all assignments and operations have the implied mathematical meaning.

\(^3\)BigInteger and BigDecimal scale the memory footprint based on size of the value. However, operations on these unrestricted numeric types have higher latency then the corresponding bounded types. However, user may always replace our implementation of the built-in types with their own, which are more efficient or suitable for their system deployment.
types. As it will be explained fully in the translator correctness section, the generated code is a specialization of its source model, which by use of finite types can only restrict the legal executions of its source model.

The data type classes contain implementations of the data type supported operations. For example given a variable of type Set\(<E>\), say \(s\), one may insert elements into it using operation \(\text{insert}(e, \ s)\), where \(e\) is an element of type \(E\) that specializes \(s\). The class Set includes a member \(\text{insert}(E \ e)\). Hence:

\[s := \text{insert}(e, \ s);\]

in Java translation becomes:

\[s . \text{insert}(e);\]

In addition, each data type implementation includes the necessary constructor methods and an overwrite of the \(\text{equals}\) method.

Data types such as tuples, arrays, sets, and maps are all translated as templates, hence these implementations are specialized during the runtime and further promote abstraction and reusability of the data type implementations. Also, a user may overwrite implementation of any data type or its representation, as long as the class public interface is not augmented. Semantics of the external methods can also be changed; however, this is discouraged for apparent reasons.

Remark 5.1 Currently Java only supports indexing of vectors and lists using \(\text{int}\) type values. Since the TEMPO arrays, sequences, and sets type translations use lists and vectors, this limits the range of indexes in the source specifications.

Since the simple native TEMPO types are implemented as custom classes, these must provide methods that return values that are understood by Java in looping and conditional statements. Therefore, each data type has a method called \(\text{value}\) that returns the Java equivalent type if there is one. For example let \(b: \text{Bool}\) then if \(b\) is to be used as the condition to an if-statement, then the TEMPO code:

\[\text{if } b \text{ then ... fi}\]

in Java translation becomes:

\[\text{if}(\ b . \text{value}()); \{\ \text{...}\} \]

Hence, \(b . \text{value}()\) returns a boolean (true or false) which Java recognizes and applies during the evaluation the if-statement. Similarly, let \(n\) be of type \(\text{Int}\) then a for statement:

\[\text{for } n: \text{Int where } n < 20 \text{ do ... od}\]

in Java translation becomes:

\[\text{for}(\text{Int } n = \text{new } \text{Int}(0); \text{TJMath.LT}(n, \text{new } \text{Int}(20)).\text{value}();\]

\[n . \text{setValue}(n . \text{intValue}() + 1)) \{\ \text{...}\} \]

One addition explanation is required of the above Java code. The “<” operand is translated as \(\text{TJMath.LT}\) which is a method in the aforementioned TJMath package and returns a \(\text{Bool}\) value representing its outcome.

5.4 State, Schedule and Local Variables

State variables are translated as protected data members of the automaton class to which they belong to. Uninitialized variables remain uninitialized and are essentially \textbf{null}. In the model state variables may be assigned some initial value when declared, and this initialization is preserved in the Java translation. However, when state variables are initialized with the automaton parameters, then such assignment must
be placed in the *initially* clause of the model. This restriction is due to the order in which Java constructs the object on its instantiation. Following demonstrates how the model should handle initialization of automaton state variables with automaton formal parameters.

```plaintext
automaton A(param: Int)  
  signature  
    ...  
  states  
    statevar1:Int := 12;  
    statevar2:Int;  
  initially  
    statevar2 = param;  
```

Listing 1: Automaton with state variables.

In the Java translation, protected variables are created that represent the automaton parameters, initially these are uninitialized. The constructor method includes assignment statement so that the variables representing automaton formals are assigned the passed in values. The very last statement of the constructor method is a call to the private `initially` method that performs the final state variable assignments.

```plaintext
...  
public class A implements Serializable {  
  ...  
  protected Int param;  
  ...  
  protected Int statevar1 = new Int( 12 );  
  protected Int statevar2;  
  ...  
  public A(Int param) {  
    this.param = new Int( param );  
    ...  
    initially();  
  }  
  private void initially() {  
    statevar2 = new Int( param );  
  }  
  ...  
}
```

Listing 2: Java translation of Listing 1

Note that if variables are left uninitialized the translation makes no attempt at assigning random values or instantiating such variables, since when unintended the randomly assigned values may result in unsafe specifications. However, execution of programs with uninitialized variables may result in exceptions being thrown by the Java Virtual Machine and program termination. Therefore, we require that all variables (state and local) are initialized in the source model. However, if user does not want to assign any specific value to a variable, then the unconstrained `choose` statement should be used, such as:

```plaintext
automaton A(param: Int)  
  ...  
  states  
    statevar1:Int := choose;  
  ...  
```

Listing 3: Example use of the choose construct.
Schedule and local variables are handled similarly, where these become local variables to the schedule method and respectively to the method representing transition that uses local variables. We will talk about transitions next.

### 5.5 Transitions

In TEMPO transition definition consists of a list of parameters, a where clause, a precondition, and effects. There may be more than one transition associated with a single action. However, in this case the where clauses should make in unambiguous as to which transition should be performed. Such check is not performed neither by the TEMPO front-end nor the translator. Following illustrates this translation process:

```plaintext
automaton A(param : Int)
    signature
        output a1(p : Int)
        output a2(p : Int)
    ...
    transitions
        output a1(p) where p < 0
        pre cond1
        eff stmt1
        output a1(p) where p > 0
        pre cond2
        eff stmt2
        output a1(p) where p = 0
        pre cond3
        eff stmt3
        output a2(p)
        pre cond4
        eff stmt4
```

Listing 4: An automaton with transitions.

is translated as the following Java code:

```java
public class A implements Serializable {
    ...
    public boolean Output_a1(Int p) {
        if (TJMath.EQ(TJMath.LT(p, 0), new Bool(true)).value()) {
            if (TJMath.EQ(cond1, new Bool(false)).value()) {
                return false;
            }
            stmt1;
            return true;
        }
        if (TJMath.EQ(TJMath.GT(p, 0), new Bool(true)).value()) {
            if (TJMath.EQ(cond2, new Bool(false)).value()) {
```
Listing 5: Java translation of Listing 4

In Listing 5 the transition which is translated first will be the first to be evaluated and executed. Notice that once the transition is chosen in this way no other transition can execute, since program exits via the return statement either when its preconditions evaluate to false or once the statements have been executed.

Regarding the return statement, the method representing the translated transition returns false if the effect is not executed and true otherwise. This return value is used in the schedule during action pairing (the online automata composition). We will return to this subject when talking about schedules.

The effect statements are an imperative sequence of assignment, loop, and condition statements. These are translated into a corresponding imperative sequence of Java statements. Since any conditional and looping construct in TEMPO has a direct relative in Java there is not much to explain here. However, it is important to note that some assignment statements in TEMPO are replaced with method calls of the data type. For example,

\[
\text{AnIntVar := 123;}
\]

will be translated as:

\[
\text{AnIntVar.setValue( new Nat( 123 ) );}
\]

This is done to ensure that the value of the variable being assigned is copied to a separate location in the memory, since the alternative would be a shallow copy (reference only) which could result in variable values being changed externally.

5.5.1 Transition Output Parameters

TEMPO notation does not have a notion of a return mechanism. However, in a way it has a way to bind values to the transition parameters. Consider this example:

\[
\text{automaton A}
\]

\[
\text{states}
\]

\[
\text{statevar: Int := 8;}
\]
The way we interpret the above model is that the condition \( p = \text{statevar} \) is not a predicate, but it can be viewed as an assignment of \( \text{statevar} \) to \( p \). Such interpretation departs from the mathematical TEMPO model, but it allows scheduling of automaton’s actions and does not violate safety properties. The above code will translate to:

```java
public class A implements Serializable {
    ...
    public static class VarJarOutput_a1 {
        public static p;
    }
    public VarJarOutput_a1 _VarJarOutput_a1 = new VarJarOutput_a1();
    public boolean Output_a1(Int a) {
        // Output variable assignments, please verify.
        // Note that these assignments hold in the
        // state prior to execution (pre-state).
        VarJarOutput_a1.p = new Int(p);
        p = VarJarOutput RECEIVE.p;
        // Transition statements
        ...
    }
}
```

Listing 7: Java translation of Listing 6

The translator detects an output variables by examining all statements in the precondition of an output action. If it is observed that a parameter is on the left hand side of the equality, then it considers it to be an output parameter. Passing variables as output is accomplished by generating a container class with output parameters, which as assigned values prior to the effects being executed. How the output variables are used will be discussed in Section 5.7.

### 5.6 Trajectories

Trajectories are translated as classes and become member classes of the translated specifying automaton. This approach is required since the trajectory is defined in terms of an evolving variable (which is a member of the specifying automaton) and the rate, which is part of the trajectory specification. Hence the trajectory class encapsulates the rate variable and has access to any state variables that belong to the specifying automaton. Hence

```java
automaton A
    states
        clock : AugmentedReal := 0;
        stopClock : AugmentedReal := 100;
        dowork : Bool := true;
    ...
    trajectories
```
is translated as the following Java code:

```java
public class A implements Serializable {
    protected AugmentedReal clock = new AugmentedReal( new Nat(0) );
    protected AugmentedReal stopClock = new AugmentedReal( new Nat(100) );
    protected Bool dowork = new Bool( true );

    public v() {
        super();
    }

    public void setPrime() {
        _prime.setValue( clock );
    }

    public Bool evolveTill( AugmentedReal UB ) {
        if( !(TJMath.EQ( dowork, new Bool( true ) )).value() )
            return new Bool( true );
        if( TJMath.EQ( clock, TJMath.ADD(_prime, UB)).value() )
            return new Bool( true );
        if( TJMath.GE( clock, stopClock).value() )
            return new Bool( true );
        clock.setValue( TJMath.ADD( clock, _rate ) );
        return new Bool( false );
    }
}
```

Listing 9: Java translation of Listing 8

The generated class includes two private variables and three public methods. The variable called _rate is used to store the value at which the evolve variable is evolved by. Currently, we only support constant rates. The variable called _prime is used to record the value of variable being evolved prior to the trajectory being evolved. This is necessary to ensure that the trajectory is scheduled properly (more on that in Section 5.7).

The public methods are the constructor, a method for setting value of the prime variable, and the method implementing the evolve statement. The evolve method in our example includes three if-statements. First if-statement captures the single invariant, next if-statement captures the stopping condition of the evolve clause as specified by the schedule, and final if-statement implements the stop-when clause of the trajectory. The remaining observation is how the evolve statement terminates, which is either by returning true or false. It returns false when none of the stopping conditions or invariants are satisfied and hence we continue to
evolve the specified variable. Otherwise, true is returned indicating that the evolve process has stopped. The return value is used in the schedule of the top level automaton which is discussed next.

5.7 Schedule and Action Matching

Schedule is a mechanism for resolving nondeterminism of the TEMPO models. Technically, the schedule is an auxiliary structure and is not part of safety reasoning. The schedule block can be defined for a composed or basic automata. Within the framework of the compiler the schedule must be provided for each node. The schedule could be general if possible or specialized based on the role and function of the node automaton.

A schedule consists of state variables, and an imperative sequence of statements that may include control and conditional statements, calls to user defined functions, and fire-statements that specify the order in which actions occur in the automaton’s execution. A schedule however should not have a direct access to automata’s state variables since such changes result in modification of state variable members outside of any transition, hence leading to possible violations of safety.

The translation of imperative aspect of the schedule has the same direct property as the translation of transition effect code. However, there are two interesting items to talk about, one is translation of the fire statements, especially the output fire statements, and translation of the follow statements.

Since in our approach we do not pre-compose automata into a single automaton, the composition process has to happen during execution. Composition is a process where all input actions are matched to the output actions with the same name. To this end each automaton is equipped with a public method named `matchMaker` which takes as parameters name of the action and a list of its parameters. In this method if-statements are used to call the appropriate transition method based on its name. Note that each action can have one or more transitions which are selected using the `where-clause`. However, recall that all transitions are translated into a single method and the appropriate transition will be executed as described in Section 5.5. Let us assume that automata A and B are components of automaton C, where these are:

```plaintext
automaton A
  signature
    output act1(p: Int)
    output act2(p: Nat)
    input act3(b: Bool, n: Nat)
    input act4(d: DiscreteReal)

automaton B
  signature
    input act1(p: Int)
    input act2(p: Nat)
    output act3(b: Bool, n: Nat)
    output act4(d: DiscreteReal)

automaton C
  components
    comp1 : A;
    comp2 : B;
  schedule do
    fire comp2.act3(true, 7);
  od
```

Listing 10: Automata in composition.

then the `matchMaker` for the translated A will look like (matchMaker of B will look similar):
public void matchMaker(String name, Object ... params) {
    if (name.equals("Input_act3")) {
        Input_in1((Bool) params[0], (Nat) params[1]);
    } else if (name.equals("Input_act4")) {
        Input_in2((DiscreteReal) params[0]);
    }
}

Listing 11: A's matchMaker method, corresponding to the translation of Listing 10

Hence, with this mechanism only one of the input actions is matched with appropriate parameters passed. The schedule of C will contain:

    if (_comp2.Output_act3(true, 7)) {
        _comp1.matchMaker("Input_act3", true, 7);
        _comp2.matchMaker("Input_act3", true, 7);
    }

Hence, accomplishing our goal. Observe that the output of the output method is in the if-statement. Recall from Section 5.5 that if a transition gets to execute its effect code, then it will return with the value true. Hence, we ensure that the matching only occurs if the output action actually was executed.

The remaining interesting aspect of schedules is the handling of the follow-statement. Let us assume that both A and B include trajectory v and that follow _comp1.v _comp2.v duration 10; appears in the schedule of C then the follow-statement will be translated as:

    trajectoryStop = false;
    _test.setValue(new AugmentedReal(new Nat(10)));
    _comp1.v.setPrime();
    _comp2.v.setPrime();
    do {
        trajectoryStop = _comp1.v.evolveTill(_test).value() ? true : trajectoryStop;
        trajectoryStop = _comp2.v.evolveTill(_test).value() ? true : trajectoryStop;
    } while (!trajectoryStop);

For models that do not involve composition translation is similar, where the difference is that methods are invoked directly and not via the matchMaker method. Noteworthy, the schedule of C will include additional variables that are used to pass the duration and test the stopping point. Meaning, if any of the trajectories stops evolving, then once the end of the do-while-statement is reached the we can safely exit.

5.8 Translating MPI and TCP Transitions

As aforementioned, TEMPO programs can use network services and we allow programmers to assume the existence of two types of point-to-point channels: (1) a reliable, FIFO channel with static node membership, and (2) a lossy channel where channels can be created and destroyed at any point in the execution to any participating in the system node. We are able to provide programmers this convenience by following our

---

4We are currently working on a composer tool that will validate a submitted system if it passes all of the requirements for composition.
four step strategy for connecting to external services. First, we model specific channel implementation as a TEMPO automaton, where these are provided to the programmer. Second, the provided channel automaton is composed with the source model. Third, the actions of the channel are invoked through the top level automaton schedule. Fourth, we prove that this composite channel definition implements the desired abstract channel.

**Abstract lossy channel.** The implementation of an abstract lossy channel using TCP/IP protocol is loosely based on [15]. However, that specification was designed for a single point-to-point connection and a statement was made that the model can be extended to support multiple connections to multiple nodes, which we have done in this work.\(^5\)

In order to facilitate management of multiple connections that are in various states and allowing bi-directional connectivity, we structure our model as a composition of three automata: sender, receiver, and an automaton emulating an interface to the TCP protocol. Before we describe these automata in detail let us first introduce the data types required for proper emulation of a TCP connection. User specification will have to include the following vocabulary specifications. At compile time these vocabularies are translated into Java classes with appropriate implementations of operations and data type representations.

```plaintext
vocabulary TCPObjectsVoc
  types
    IPv4 : Tuple [one : Nat, two : Nat, three : Nat, four : Nat],
    JVMError : String
end

vocabulary TCPNodeVoc
  imports TCPObjectsVoc
  types
    Node : IPv4
   operators
    GT : Node, Node -> Bool,
    EQ : Node, Node -> Bool,
    LT : Node, Node -> Bool
end
```

**Listing 12:** A vocabulary for TCP objects.

TCPObjectsVoc introduces the IP addresses as tuples of natural numbers, where address such as 192.168.1.10 in TIOA specification will be represented as \([192,168,1,10]\). The TCPNodeVoc abstracts physical addressing to simply a Node. Since all IP addresses of networked nodes must be unique, it we provide operations that allow ordering of these. For example \(\text{EQ}([192,168,1,10],[192,168,1,2])\) will return a \text{Bool} with a value \text{false}.

```plaintext
vocabulary JVMSocket
  imports TCPObjectsVoc, TCPNodeVoc, tcp_specific_voc
  imports alg_specific_voc
  types JVMSocket
  operators
    JVM_TCPSocketOpen : Node, Nat, Nat -> Null [JVMSocket],
```

\(^5\)A keen reader will be alarmed to see lossy and TCP used in the same context. However, after close examination of the TCP/Java socket integration one will find that it is possible for messages to be dropped, where this fact is acknowledged by Java specification (although under very special circumstances). The primary reason for message loss is due to limited buffering space at the various levels of operating system and the JVM, where messages may be refused or dropped when memory is limited.
In Java a socket is represented as a `Socket` class. The above vocabulary `JVMSocket` models the `Socket` class along with the minimal set of necessary operations. The `JVMSocket` vocabulary requires the previously introduced vocabularies and the `AlgorithmVoc` which must include the definition of a `Message` type. The supported operations include socket open and close request, given a socket a local and remote IP addresses can be extracted, a socket can be tested for connectivity, and finally we support read and write operations to a socket. Notice that the return types for many of the operations return a `Null[ ... ]` type, which is compatible with the behavior of Java `Socket` class, where a value `null` may be returned. Also, if an error occurred during any of the operations, a `JVMError` is returned which contains the error string message from JVM. Similarly, we also introduce a vocabulary that mimics Java’s `ServerSocket`.

The final vocabulary is that for the channel itself. This vocabulary captures the states of a TCP channel, message being transported, and a representation of the channel itself.

Now we will briefly present the automata, where their full specification along with correctness proofs is included in the Appendix. We begin with the automaton that implements the send mediator.
### Send Mediator Automaton

**Signature**

- **Input:** `SEND(m: Null[Chan_message])`
- **Output:** `TCP_senderOpen(remote:Node, port: Nat)`
- **Input:** `TCP_respSenderOpen(remote:Node, port: Nat, resp: Bool)`
- **Output:** `TCP_senderClose(remote:Node)`
- **Input:** `TCP_respSenderClose(remote:Node, resp: Bool)`
- **Output:** `TCP_write(m: Null[Chan_message], s, r: Node)`

**States**

- `sendBuffer : Seq[Chan_message] := { };`
- `remoteStatus : Array[Node, Status] := constant(idle);`
- `clocks : Array[Node, AugmentedReal] := constant(\infty);`
- `clock : AugmentedReal := 0;`

Listing 16: Send mediator automaton, signature and state.

The automaton two parameters `port` and `timeout` represent the port to which this automaton will send messages to and the timeout on message delivery. There are four state variables, `sendBuffer` contains all messages to that have been delivered by the algorithm automaton for sending, but have not yet been sent onto the network, `remoteStatus` used to keep track of connection status with other nodes, `clocks` which records the time when a message is written to a network and together with `timeout` it allows modeling of network delay, and `clock` which is used in a trajectory that simply emulates channel delay.

The actions of `SendMed` are, an input `SEND` action that is paired with the like named output action of the algorithm automaton and by its effect a message is handed to the `SendMed` for sending. The remaining actions are paired with the `TCP_ChanMed` automaton and facilitate opening and closing of a connection to the remote node, and writing to that connection. Success or failure of these operations can be queried to the `TCP_ChanMed` (presented later). Next we present the `RecvMed` automaton.

### Receive Mediator Automaton

**Signature**

- **Output:** `RECEIVE(m: Null[Chan_message])`
- **Output:** `TCP_read`
- **Input:** `TCP_respRead(m : Null[Chan_message])`
- **Output:** `TCP_bind(local : Node)`
- **Input:** `TCP_respBind(error : Null[JVMError], local : Node)`
- **Output:** `TCP_accept`
- **Input:** `TCP_respAccept(error : Null[JVMError])`
- **Output:** `TCP_stopAccepting`
- **Output:** `TCP_stopListening(remote : Node)`
- **Output:** `TCP_rCloseStream(remote : Node)`
- **Output:** `TCP_rClose(remote : Node)`
- **Input:** `TCP_getError(e : Null[JVMError], remote : Node)`

**States**

- `recvBuffer :Seq[Chan_message] := { };`
- `recvErrors :Map[Node, Null[JVMError]];`
- `remoteStatus :Map[Node, Status];`
- `localStatus : Status := idle;`
- `localError : Null[JVMError] := nil();`

Listing 17: Receive mediator automaton, signature and state.

This automaton is also parametrized with port and timeout values. The state variables are as follows. `recvBuffer` holds all of the messages that arrived through the network, but have not yet been delivered to the algorithm automaton, `recvErrors` which records all connection related error messages,
remoteStatus which records the status of a remote connection, localStatus which indicates the local status, and localError that records any errors resulting from any locally performed actions.

The only action of RecvMed that pairs with the algorithm automaton is RECEIVE by which a message is delivered (if there is one). The remaining actions pair with the TCP ChanMed and when performed in the right order result in a server socket being configured and connected. These actions are named as to reveal their function with respect to the TCP protocol stages, such as bind, accept, etc.

We are now ready to present the TCP ChanMed automaton which mediates between the sender and receiver automata and the TCP protocol.

**automaton** ChanMed( port : Nat, timeout : Nat)

**signature**
- input TCP_read
- output TCP_respRead(m : Null[Chan_message])
- input TCP_bind(local : Node)
- output TCP_respBind(error : Null[JVMError], local : Node)
- input TCP_accept
- output TCP_respAccept(error : Null[JVMError])
- input TCP_stopAccepting
- input TCP_stopListening(remote : Node)
- input TCP_rClose(remote : Node)
- input TCP_senderOpen(remote : Node, port : Nat)
- input TCP_senderClose(remote : Node)
- input TCP_write(m : Null[Chan_message], s, r : Node)
- internal TCP_senderClosing(remote : Node)
- output TCP_getError(e : Null[JVMError], remote : Node)

**states**
- SSocket : Null[JVMServerSocket] := nil();
- acceptStatus : Status := idle;
- SError : Null[JVMError] := nil();
- AError : Null[JVMError] := nil();
- tcpChannel : Seq[Channel] := { };
- recvBuffer : Seq[Chan_message] := { };

Listing 18: Automaton modeling interaction with the TCP channel, signature and state.

The state variables are SSocket which models a server socket, acceptStatus, send and accept error which are SError and AError respectively, tcpChannel which is a sequence all instantiated channels, and finally the message buffer called recvBuffer for messages received from the TCP protocol but not yet passed to the RecvMed automaton. Actions of TCP ChanMed are paired with the send and receive mediator automata.

**Reliable FIFO channel.** In [46] an alternative communication method is proposed, and it uses MPI at its core. Automata are derived that collectively model a subset of MPI behaviors. Although MPI supports broadcast, in this implementation all to all communication is modeled as a collection of point-to-point channels connecting \( n \)-nodes (hence \( n^2 \) of such channels are needed). MPI is an excellent communication protocol in settings where failures are infrequent and performance is valued. However, in systems where connectivity is susceptible to delays and even failures, TCP/IP sockets are a better fit.

In our work we use MPJ Express [2] as an implementation of MPI for Java. This library implements all of the MPI interfaces and provides a comparable performance.
Connecting to MPI and TCP Methods. Invocation of the native MPI and Java calls is facilitated through the vocabulary and the associated operations on the defined types. The auto generated translations for the provided vocabularies provide appropriate implementations. Notice, that it is necessary for all native MPI and Java calls to be non-blocking, otherwise input actions may be blocked indefinitely which is not allowed under the TEMPO model.

6 Translation’s Correctness

Throughout this section we refer to the source model as a parent specification (or simply parent) and the abstracted Java code derived by translating the parent specification as a child implementation (or simply child).

The proof of the translator’s correctness is based on [24], where it presents a framework for incremental proof technique for untimed systems that was applied, among other things, to a translation of a group communication service into its implementation using C++. Introduced in [24] are specialization and extension operations on the parent which result in the child implementation, where it can be shown that following application of these operations on the parent the derived child can be used anywhere the parent can be used.

We extend these operations to support timed systems. We begin with a formal definition of Timed I/O Automata as presented in [23]. (Omitted proofs are found in Appendix E.)

Definition 6.1 A timed automaton $A$ is defined as $A = (X, Q, \Theta, E, H, D, T)$, s.t.:

- $X$ is a set of internal variables.
- $Q \subseteq \text{val}(X)$ is a set of states, where informally $\text{val}(X)$ represents the set of values over types of $X$.
- $\Theta \subseteq Q$ is a nonempty set of start states.
- $E$ is a set of external actions and $H$ is a set of internal actions, where $E \cap H = \emptyset$.
- $D \subseteq Q \times (E \cup H) \times Q$ is a set of discrete transitions.
- $T \subseteq \text{trajs}(Q)$ is a set of trajectories, where informally $\text{trajs}(Q)$ represents the set of all trajectories for variables in $X$.

Trajectory axioms: For some $\tau \in T$ we associate states, such that $\tau.fstate = x \in Q$ at the start of $\tau$. If $\tau$ is closed, then $\tau.lstate = x' \in Q$ where $x'$ is the state of $A$ at the end of $\tau$ and $\tau.ltime$ is the duration of $\tau$. When $\tau.fstate = x$ and $\tau.lstate = x'$ and $x, x' \in Q$, then the following axioms must hold:

1. If $x \in Q$ then there exists a point trajectory $\tau$ such that duration of $\tau$ is zero and $\tau.fstate = \tau.lstate = x$.
2. For every $\tau \in T$ and every $\tau' \leq \tau$, $\tau' \in T$, asserting a prefix closure.
3. For every $\tau \in T$ and every $t$ in the domain of $\tau$, $\tau \supseteq t \in T$, asserting a suffix closure (i.e., that the remainder is also a trajectory).
4. Let $\tau_0 \tau_1 \tau_2 \ldots$ be a sequence of trajectories in $T$ such that, for each nonfinal index $i$, $\tau_i$ is closed and $\tau_i.lstate = \tau_{i+1}.fstate$. Then $\tau_0 \prec \tau_1 \prec \tau_2 \ldots \in T$, asserting a concatenation closure.

Following is a definition of a forward simulation relation relating states of two automata (from [23]).

Definition 6.2 Let $A$ (child) and $S$ (parent) be two automata with the same external signature. A relation $R \subseteq Q_A \times Q_S$ is a forward simulation from $A$ to $S$ if it satisfies the following three conditions:

1. If $t \in \Theta_A$, then there exists state $s \in \Theta_S$ such that $(t, s) \in R$.
2. If $(t, s) \in R$ and $\alpha$ is an execution fragment of $A$ consisting of an action surrounded by two point trajectories, with $\alpha.fstate = t$, then $S$ has a closed execution fragment $\beta$ with $\beta.fstate = s$, $\text{traces}(\beta) = \text{traces}(\alpha)$, and $(\alpha.lstate, \beta.lstate) \in R$. 

25
3. If \((t, s) \in R\) and \(\alpha\) is an execution fragment of \(A\) consisting of a single closed trajectory, with \(\alpha.fstate = t\), then \(S\) has a closed execution fragment \(\beta\) with \(\beta.fstate = s\), \(\text{traces}(\beta) = \text{traces}(\alpha)\), and \((\alpha.lstate, \beta.lstate) \in R\).

Definition 6.2 asserts that each discrete transition (resp. trajectory) of \(A\) can be simulated by a corresponding execution fragment of \(S\) with the same trace and duration, and leads to the trace inclusion property.

The specialization operation presented in [24] is a construct for creating a child by specializing the parent. This operation is designed to capture the notion of subtyping in I/O automata in the sense of trace corresponding execution fragment of \(A\). We extend the construct to operate on the parent and the following additional parameters: a state extension, the new state components, an initial state extension, the initial values of the new state components, and a transition restriction, specifying the child’s addition of new preconditions and effects (modifying new state components only) to parent transitions. We extend the specialize construct with a trajectory restriction, specifying the child’s restrictions on stopping conditions of parent’s trajectories. Formally stated:

**Definition 6.3 (Specialization)** Let \(\mathcal{A} = (X, Q, \Theta, E, H, D, T)\) be an automaton; \(X_n\), a set of variables and \(N \subseteq \text{val}(X_n)\) be any set of states, called a state extension; \(N_0\) be a non-empty subset of \(N\), called an initial state extension; and \(TR \subseteq (\text{states}(\mathcal{A}) \times N) \times \text{sig}(\mathcal{A}) \times N\) be a relation called a transition restriction. Then \(\text{specialize}(\mathcal{A})(X_n, N, N_0, TR)\) defines \(\mathcal{A}' = (X', Q', \Theta', E', H', D', T')\):

- \(X' = X \cup X_n\), \(Q' = Q \times N\), \(\Theta' = \Theta \times N_0\), \(E' = E\) and \(H' = H\)
- \(D' = \{(\langle t_p, t_n \rangle, \pi, (t'_p, t'_n)) : (t_p, \pi, t_p') \in D \land (\langle t_p, t_n \rangle, \pi, t'_n) \in TR\}\), where \(\langle t_p, t_n \rangle\) is a state in \(Q'\)
- \(T'\) the set of all trajectories for variables in \(X'\).

For an automaton \(\mathcal{A}'\) as defined above, given \(t \in Q'\), we write \(t|_p\) denotes its parent component and \(t|_n\) its new component. If \(\alpha\) is an execution fragment of \(\mathcal{A}'\), then \(\alpha|_p\) and \(\alpha|_n\) denote sequences obtained by replacing each state \(t\) in \(\alpha\) with \(t|_p\) and \(t|_n\), respectively. Restating the specialization operation result from [24]:

**Corollary 6.4** If \(\mathcal{A}'\) is a specialization of \(\mathcal{A}\), then: \(\alpha \in \text{execs}(\mathcal{A}') \implies \alpha|_p \in \text{execs}(\mathcal{A}) \text{ and } \beta \in \text{traces}(\mathcal{A}') \implies \beta \in \text{traces}(\mathcal{A})\)

**Proof.** Let: \(\mathcal{A} = (X, Q, \Theta, E, H, D, T), \mathcal{A}' = (X', Q', \Theta', E', H', D', T')\), and \(\alpha = \tau_0 a_0 \tau_1 a_1 \tau_2 \ldots\) be a closed or an admissible execution\(^6\) of \(\mathcal{A}'\).

1. By the meaning of \(\alpha\) it begins with some initial state \(t_0\) and for every discrete transition and trajectory appearing in \(\alpha\) there must exist two states \(t \land t' \in Q'\) such that either \(\tau \xrightarrow{\pi} t'\) or \(\tau \xrightarrow{\pi} t'\), where \(\pi \in \{E' \cup H'\}\) and \(\tau \in T'\). By Definition 6.3, \(t_0|_p\) is an initial state of \(\mathcal{A}\), for each \(\tau \xrightarrow{\pi} t'\) we know that \(t|_p \xrightarrow{\pi} t'|_p \in D\), and that for each \(\tau \xrightarrow{\pi} \tau'\) we know that \(t|_p \xrightarrow{\pi} t'|_p \tau \in T\). From this it follows that the sequence obtained by replacing each state \(t\) in \(\alpha\) with \(t|_p\) is an execution of \(\mathcal{A}\). Since this sequence is \(\alpha|_p\), we conclude that \(\alpha|_p\) is an execution of \(\mathcal{A}\). (Note that per restrictions of Definition 6.3 any trajectory in \(\alpha\) must also be a trajectory of \(\mathcal{A}\).)

2. From part (1) and the fact that \(E' = E\).

We now reintroduce the specialized extension operation from [24]. This operation is similar to inheritance, where the child cannot overwrite parent’s behaviors, but can extend these with new types of behaviors. Specialized extension is performed in two steps, first via signature extension and then by the application of the previously presented specialization operation.

\(^6\)If \(\alpha\) is closed then the there exists some \(n\) such that \(\tau_n\) is the final trajectory in \(\alpha\) and \(\tau_n\) is right closed. An admissible execution \(\alpha\) is one where \(\alpha.ltime = \infty\).
The signature extension operation extends parent with new actions, where these are enabled in every state and do not modify the state, where specialization gives meaning to these actions. The result is a child automaton with extended signature, but same states and same start states. Additionally, state extension operation allows action renaming, which is specified by a signature-mapping function that maps child’s actions to parent’s actions. This mapping can be many-to-one, onto, and is undefined for new actions added. For example, let \( f \) be a signature-mapping then for each action in parent’s signature there is at least one corresponding \( f(\cdot) \), and if \( \pi \) is a new action under signature extension then \( f(\pi) = \bot \).

**Definition 6.5** Assume \( \mathcal{A} = (X, Q, \Theta, E, H, D, T) \), \( S' \) to be some signature. Let \( f \) be a partial function, called signature-mapping, from \( S' \) to signature of \( \mathcal{A} \) such that \( f \) is onto and preserves the classification of actions. Then, \( \text{extend}(\mathcal{A})(S', f) \) is defined to be the following automaton \( \mathcal{A}' = (X', Q', \Theta', E', H', D', T') \):

1. \( X' = X, \ Q' = Q, \ \Theta' = \Theta, \ \{E' \cup H'\} = S', \ T' = T \)
2. \( D' = \{ (t, \pi, t') \in Q' \times S' \times Q': ((f(\pi) = \bot) \land (t = t')) \lor ((f(\pi) \neq \bot) \land ((t, f(\pi), t') \in D)) \} \)

Hence, \( \mathcal{A}' \) is a signature extension of \( \mathcal{A} \) with signature-mapping \( f \) if \( \mathcal{A}' \) is such that \( \mathcal{A}' = \text{extend}(\mathcal{A})(\text{sig}(\mathcal{A}'), f) \) for some signature-mapping \( f \) from signature of \( \mathcal{A}' \).

**Definition 6.6** Automaton \( \mathcal{A}' \) is called a specialized extension of \( \mathcal{A} \) if \( \mathcal{A}' \) is a specialization of a signature extension of \( \mathcal{A} \).

If \( \mathcal{A}' \) is a signature extension of \( \mathcal{A} \) with a signature-mapping \( f \), then if \( \alpha \) is an execution fragment of \( \mathcal{A}' \), then \( f(\alpha) \) denotes an execution fragment obtained by replacing each discrete action \( \pi \) in \( \alpha \) with \( f(\pi) \) and then collapsing every triple \( \tau \perp \tau' \) to \( \tau \sim \tau' \).

**Lemma 6.7** Let \( \mathcal{A}' \) be a signature extension of \( \mathcal{A} \) with a signature-mapping \( f \), then: \( \alpha \in \text{execs}(\mathcal{A}') \iff f(\alpha) \in \text{execs}(\mathcal{A}) \) and \( \beta \in \text{traces}(\mathcal{A}') \iff f(\beta) \in \text{traces}(\mathcal{A}) \)

**Proof.** Let \( \mathcal{A} = (X, Q, \Theta, E, H, D, T) \) and \( \mathcal{A}' = (X', Q', \Theta', E', H', D', T') \).

(1) \( \Rightarrow \): Let \( \alpha \) be a closed or admissible execution of \( \mathcal{A}' \). By definition of execution, \( \alpha \) begins in some initial state \( t_0 \), and for each discrete transition \( \pi \) appearing in \( \alpha \) there must exist two states \( t_i \wedge t_{i+1} \in Q' \) such that \( t_i \stackrel{f(\pi)}{\rightarrow} t_{i+1} \in D' \). From this and Definition 6.6, \( t_0 \) is an initial state of \( \mathcal{A} \), and for each \( t_i \stackrel{f(\pi)}{\rightarrow} t_{i+1} \) in \( \alpha \), either \( t_i \stackrel{f(\pi)}{\rightarrow} t_{i+1} \in D \) when \( f(\pi) \in \{E \cup H\} \), or \( t_i = t_{i+1} \) when \( f(\pi) = \bot \). In the case when \( f(\pi) = \bot \), from the properties of \( \alpha \) we know that triple \( (t_i, \pi, t_{i+1}) \) appears in \( \alpha \) and that \( t_i, \text{state} = t_i \) and that \( t_i+1, \text{state} = t_{i+1} \). Hence we truncate \( (t_i \perp t_{i+1}) \) to \( t_i \sim t_{i+1} \). Fortunately from T3 we know that since \( t_i = t_{i+1} \) the trajectory \( t_i \sim t_{i+1} \in T' \) and from Definition 6.6 also in \( T \). Therefore, by definition of execution, the sequence obtained by replacing every \( t_i \stackrel{f(\pi)}{\rightarrow} t_{i+1} \) with \( t_i \stackrel{f(\pi)}{\rightarrow} t_{i+1} \) when \( f(\pi) \neq \bot \), or \( t_i \sim t_{i+1} \) otherwise is an execution of \( \mathcal{A} \). Since this sequence if \( f(\alpha) \), we conclude that \( f(\alpha) \in \text{execs}(\mathcal{A}) \).

(2): From (1) and the fact that \( f \) preserves the classification of actions.

The following captures application of specialization to signature-extension.

---

\(^7\) Actions are classified as *Input*, *Output*, and *Internal* (a.k.a., kinds).
Corollary 6.8 If \( \mathcal{A}' \) is a specialized extension of \( \mathcal{A} \) with a signature-mapping \( f \), then: \( \alpha \in \text{execs}(\mathcal{A}') \Rightarrow f(\alpha|_p) \in \text{execs}(\mathcal{A}) \) and \( \beta \in \text{traces}(\mathcal{A}') \Rightarrow f(\beta) \in \text{traces}(\mathcal{A}) \)

Proof. Follows directly from Colorary 6.4 and Lemma 6.7.

We are now ready to re-state the final result from [24] in terms of timed automata. In the theorem that follows assume that each automaton \( \text{Aut} \) is defined as \( \text{Aut} = (X_{\text{Aut}}, Q_{\text{Aut}}, \Theta_{\text{Aut}}, E_{\text{Aut}}, H_{\text{Aut}}, D_{\text{Aut}}, T_{\text{Aut}}) \).

Theorem 6.9 Let \( \mathcal{A}' \) be a specialized extension of \( \mathcal{A} \) with a signature-mapping \( f \). Let \( S' \) be a specialized extension of \( S \) with a signature-mapping \( g \), such that \( S' = \text{specialize}(\text{extend}(S(G, g)))(N, N_0, TR) \). Assume that \( \mathcal{A} \) and \( S \) have the same external signatures and that \( \mathcal{A} \) implements \( S \) via a simulation relation \( R_p \). Assume further that \( \mathcal{A}' \) and \( S' \) have the same external signatures, and that, for every external action \( \pi \in \mathcal{A}' \), \( g(\pi) = f(\pi) \).

A relation \( R_c \subseteq Q_{\mathcal{A}'} \times Q_{S'} \), defined in terms of relation \( R_p \) and a new relation \( R_n \subseteq Q_{\mathcal{A}'} \times N \) as \( \{(t, s): (t|_p, s|_p) \in R_p \land (t, s|_n) \in R_n\} \), is a simulation from \( \mathcal{A}' \) to \( S' \) if \( R_c \) satisfies the following two conditions:

1. For every \( t \in \text{start}(\mathcal{A}') \), there exists a state \( s|_n \in R_n(t) \) such that \( s|_n \in N_0 \).
2. If \( t \) is a reachable state of \( \mathcal{A}' \), \( s \) is a reachable state of \( S' \) such that \( s|_p \in R_p(t|_p) \) and \( s|_n \in R_n(t) \), and \( \alpha \) is an execution fragment of \( \mathcal{A}' \) where \( \alpha.lstate = t' \) and consisting of one action surrounded by two point trajectories, \( \tau_0 \) and \( \tau'_0 \) (\( \tau_0.lstate = s_i \) and \( \tau'_0.fstate = s_{i+1} \)), or a single closed trajectory \( \tau \), then there exists a closed execution fragment \( \beta \) of \( S' \) beginning from \( s \) and ending at some state \( s' \), and satisfying the following conditions:

   (a) \( \beta|_p \) is an execution fragment of \( S \).
   (b) For every step \( (s_i, \sigma, s_{i+1}) \) in \( \beta \), \( (s_i, \sigma, s_{i+1}, s'|_n) \in TR \).
   (c) \( s'|_p \in R_p(t'|_p) \).
   (d) \( s'|_n \in R_n(t') \).
   (e) \( \beta \) has the same trace as \( \tau \Rightarrow \tau' \).
   (f) If \( \alpha = \tau \), then for each \( \tau_i \in \beta \), \( \sum \tau_i.ltime = \tau.ltime \).

Proof. We show that \( R_c \) satisfies conditions of Definition 6.2.

(1) Consider an initial state \( t \) of \( \mathcal{A}' \). By assumption that \( \mathcal{A}' \) is a specialized extension of \( \mathcal{A} \) and hence by Definition 6.5 we know that \( t|_p \in \Theta_{\mathcal{A}} \). By assumption that \( \mathcal{A} \) implements \( S \) we know that there must exist \( s|_p \in R_p(t) \) such that \( s|_p \in \text{start}(S) \). By condition (1) of this corollary, there must exist a state \( s|_n \in R_n \) such that \( s|_n \in N_0 \). Consider state \( s = (s|_p, s|_n) \). State \( s \) is in \( R_c(t) \) by definition. Also by Definition 6.3, \( \text{start}(S') = \text{start}(S) \times N_0 \); therefore, \( s = (s|_p, s|_n) \in \text{start}(S) \times N_0 = \text{start}(S') \).

(2) First, notice that the definitions of state \( s \) and the relation \( R_c \) imply that \( s \in R_c(t) \); also, notice that conditions 2c and 2d imply that \( s' \in R_c(t') \). Next, we show that \( \beta \) is an execution fragment of \( S' \) with the right index. Indeed, every discrete step of \( \beta \) is consistent with \( D_S \) and every trajectory appearing in \( \beta \) is consistent with \( T_S \) (by 2a) and is consistent with \( TR \) (by 2b). Therefore, by definition of \( D_{S'} \) and \( T_{S'} \) (Definition 6.5), every step of \( \beta \) is consistent with \( \text{traces}(S') \). In other words, \( \beta \) is an execution fragment of \( S' \) that starts with state in \( R_c(t) \), ends with state \( R_c(t') \), and has the same trace as \( \alpha \) (by 2e) and same duration (by 2f). \( \square \)

The above constitutes the core result that we use to prove our translator's correctness. However, we emphasize that this result is general and it can be used beyond our work.

Translation soundness. We would like to remind the reader that specifications submitted for translation should be in the node-channel form, where the node component is modeled either as a single automaton or
a composition of automata that interface with the channel, where the channel is also modeled as an automaton. Hence, in the correctness reasoning it suffices to only consider translation of each automaton (since composition happens on-line and during runtime). In the following discussion we will refer to Section 5 that outlines the translation steps. Since we cannot use Java’s API in proofs, we have to bring the translator generated codes into the TEMPO framework by providing an abstraction of the executable code. In [46] techniques are presented that demonstrated how to abstract the generated Java code by the IOA translator and prove that the resulting code has the same externally observable behaviors as the source specification, where this is done by first reasoning about translation of the individual node automata (see Theorems 7.1 [46]), and then at the composition level, i.e. system wide, (see Theorems 7.2 in [46]). Same techniques can be adapted to this work and used to arrive at the same conclusion for the codes generated by the TEMPO-to-Java translator. To avoid tedious repetition we forgo presentation of this detail and refer the interested reader to [46].

In the context of translating timed automata models into executable code, the input model is $S$ and its implementation is $A'$. In Section 5 we introduced the translation guidelines. From these it is easy to see that we perform specialization and signature extension on the source specification. It remains to show that these are consistent with the proceeding results.

**Definition 6.10** Let $A$ be an abstracted automaton from the Java code generated by the direct translation of the allowed TEMPO syntax into the Java code.

**Definition 6.11** Let $A' = \text{specialize}(\text{extend}(A(G,f)))(N,N_0,TR)$, where $f$, $TR$, and specialization be defined according to Section 5.

**Theorem 6.12** The TEMPO-to-Java translator preserves behaviors of the source specification that adhere to the limitations presented in Section 5.

**Proof.** It follows from Definitions 6.10 and 6.11, Theorem 7.2 of [46], and Theorem 6.9. \hfill $\square$

### 7 How to Use the Translation Module

The Tempo toolkit can be upload from the VeroModo, Inc. web site [1]. Installation and configuration information can also be found there. A link to a video demonstrating a project setup is located under a tab called Videos. Once the Tempo toolkit is installed and configured, a project can be created and populated with model files. Each file must have an extension tioa. When a model is loaded the front end will parse it and check for syntactic correctness and if there are no errors, then the view will look something like the one depicted in Fig. 1.

If the source model will be used to generate networked code, then prior to generating Java code we must configure the translation module with type of the channel we wish to use. This can be accomplished by opening the translator presence window: Tempo $\rightarrow$ Preferences, then select Tempo Plugins $\rightarrow$ Java Generator, and finally select the appropriate communication mode, see Fig. 2. Click OK.

At the top of the Tempo window plugin icons can be found, select the one called tempo2java (circled in Fig. 1).
Figure 1: Tempo toolkit graphical interface.

Figure 2: Tempo to Java translation module preference configuration.
8 Evaluation

8.1 The Paxos Algorithm

The consensus problem addresses the situation in which there is a set of processes; each process can propose a value, but in order for the system to reach a consensus state, every process must decide on the same value. In particular three conditions must hold: Agreement: all (correct) processes agree on the same value. Validity: the agreed value was among the ones proposed by the processes. Termination: eventually each (correct) process decides. The first two conditions are safety conditions, that is, they must hold at all times. The third one is a liveness condition and it can only be met under the assumption of partial-asynchrony.

In brief, Paxos [27, 41] is a quorum system based algorithm and works as follows: a leader starts ballots, tries to associate a value to each ballot, and tries to collect approval from some quorum for each ballot to use the value of that ballot as the decision value. The leader bases its choice of a value to associate with a ballot on the information returned by a quorum. Once the value is associated with the ballot, the leader tries to collect approval from a quorum of processes: if it succeeds, the ballot value becomes the final consensus decision value. In general, several leaders may operate at the same time and may interfere with each others work. However, under a stable state only one leader operates and ensures that a ballot completes. We now outline the main phases of Paxos.

1. The leader starts a new ballot and informs the others about it.
2. A process that learns about the new ballot abstains from any earlier ballot for which it has not voted for. In response, a process replies to the leader with the value of the ballot for which it last voted for.
3. Once the leader receives responses from a quorum, it chooses a value for the ballot that is based on the received values and announces that value to others.
4. A process that learns about a new value may vote for the ballot, if it has not already abstained. If the process votes, then it informs the leader and others about its vote.
5. The leader decides on the ballots value once it receives messages from a quorum with a vote for that value. In case that the leader has failed, a separate leader election service is used to elect a new one. Timeouts are used to determine which processes are operational, and among these, the one with the highest id is elected as the leader. After the election, the new leader starts a new ballot.
6. Timeouts are also used for the leader to decide when it should start new ballots (that is, there is a limit on how long it takes for a given ballot to be accepted by a quorum of processes).

Observe from the above description that there are two timing-dependent components: the leader-election service that determines when a new election should be triggered, and the mechanism that determines when a leader should trigger a new ballot.

Our specification of the Paxos algorithm is based on [41]. The source is presented in the Appendix D, which is specialized to use MPI, and in the Appendix C, which is specialized to use TCP.

Empirical Results Here we present, as a proof-of-concept implementation, some experimental results we have obtained after applying our method on the Paxos system presented in Section 5. The LAN experiments were carried out on the target platform that consists of a cluster of nine machines, hosted at the University of Cyprus. Each machine is powered by an AMD Opteron 2.5GHz (single or dual) CPU and is running Linux (CentOS v5.5). The WAN experiments were carried out on the Planet-Lab [45] platform.

The Java codes were generated using the specification found in the Appendix (see Section D). The specification was compiled using an appropriate translator flag value for the -comm parameter mpi. To ensure portability of the generated code we opted to use implementation of MPI also in Java, called MPJ
<table>
<thead>
<tr>
<th>Version</th>
<th>Network</th>
<th>Nodes</th>
<th>Timeout</th>
<th>% msg. loss</th>
<th>Avg. run time</th>
<th>Avg. msg. sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>LAN</td>
<td>9</td>
<td>5</td>
<td>0%</td>
<td>2335.861 msec</td>
<td>14.333</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCP</td>
<td>LAN</td>
<td>9</td>
<td>40</td>
<td>0%</td>
<td>2274.622 msec</td>
<td>14.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WAN</td>
<td>9</td>
<td>100</td>
<td>dynamic</td>
<td>3398.611 msec</td>
<td>14.333</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WAN</td>
<td>30</td>
<td>100</td>
<td>dynamic</td>
<td>43718.733 msec</td>
<td>35.833</td>
</tr>
</tbody>
</table>

Table 1: Evaluation of Paxos.

Express [2]. This library must be installed and configured. Running MPI implementation requires a deployment platform that is MPI friendly (minimum security, etc.) MPJ as MPI requires a machines file to be provided which includes IP addresses (or hostnames/aliases) of participating nodes. Compile the code with the MPJ library and run on each machine using the following command line directive: mpjrun.sh -np 9 -dev nodev -ea paxos 22 200 400 24.

From Table 1 we see that MPI and TCP perform about equally. On the Planet-Lab network (the WAN setting) understandably we get higher consensus latency, but ones that are resonably close to the LAN experiments. Since Planet-Lab offers more machines we also tested Paxos with 30 machines where the operation latency scales linearly with number of machines.

### 8.2 Clock Synchronization

A simple clock synchronization algorithm is used as another test case. Goal of this algorithm is achieving an agreement and validity among the logical clock values. Specification of this algorithm can be found in [23] on page 33. Since the algorithm is not terribly challenging to understand and more its only use is as a test case we point the interested reader to [23]. Listing 19 depicts specification of the clock synchronization algorithm for a single node.

```plaintext
1 automaton ClockSync(u, r : DiscreteReal) where u >= 0 /\ (0 <= r /\ r <= 1);

    signature
    output SEND(m : Null[mpi_message])
    input RECEIVE(m : Null[mpi_message])
    internal prepmessages, init

6 states
    nextsend : DiscreteReal := 0;
    maxother : DiscreteReal := 0;
    physclock : DiscreteReal := 0;
    tosend : Seq[Null[Chan_message]] := {};

11 rates : Array[Nat, DiscreteReal] := constant(1);
    index : Nat := 0;

11 transitions
    output SEND(m) where len(tosend) <= 0
    pre m = head(tosend);
    eff nextsend := nextsend + u;
    tosend := tail(tosend);

    input RECEIVE(m)
    eff if (m = nil() /\ val(m).destination = MPI_Rank()) then
        maxother := max(maxother, val(m).data);

    fi
    internal prepmessages
    pre len(tosend) = 0;
    eff for n : Nat where n < MPI_Size() do
```

32
to send := to send | ¬ embed((physclock, n));

od
index := mod(index + 1, 100);
internal init
locals v : DiscreteReal := 0;
b : Bool := false;
pre true:
eff for n : Nat where n < 100 do
  v := choose n; b := choose n;
  if b then rates[n] := rates[n] + v;
  else rates[n] := rates[n] - v; fi
od;
trajectories
trajdef T stop when physclock > nextsend;
evolve d(physclock) = rates[index];


List. 20 depicts specification of the clock synchronization system for a single node, where the individual components are composed together yielding a new automaton called ClockSyncSys. ClockSyncSys represents a composed clock synchronization algorithm and the automata modeling communication channels in this specific instance MPI channels are used.

include "MPIVocabs.tioa"
imports mpi_message_voc , mpi_request_voc , mpi_status_voc , mpi_voc . alg_specific_voc
include "ClockSync.tioa", "ReceiveMediator.tioa", "SendMediator.tioa"
automaton ClockSyncSys(u , r : DiscreteReal, l : Nat)
6 components
CS : ClockSync(u , r);
SM : SendMediator;
RM : ReceiveMediator;
schedule
11 states
m : Null[mpi_message] := nil();
runs : Nat := l;
do
fire internal CS.init;
16 for i : Nat where i < runs do
  follow CS.T duration \infty ;
  fire internal CS.prepmessages;
  for j : Nat where j < MPI_Size() do
    fire output CS.SEND(m);
  od
  follow SM.DELAY duration 10;
  for j: Nat where j < MPI_Size() do
    fire input RM.probe( j );
    fire output RM.RECEIVE( m );
21 od od od

Listing 20: Specification for a single process in a clock synchronization system.

A node, \( i \), participating in the clock synchronization system is defined as the automaton ClockSyncSys(\( u , r \)), and will behave according to its schedule. Node to node interaction is performed through point-to-point channels where messages are deposited into the channel via the SEND event and removed from the channel via the RECEIVE event. This model makes use of channels implemented over MPI. Our translation allows use of TCP channels that are more forgiving to network fluctuations, as opposed to MPI. However, the specification is slightly longer due to additional steps required in the schedule.

We now outline the main steps of the clock synchronization algorithm:
MPI implementation: value of max(physclock, otherclock) at termination.

<table>
<thead>
<tr>
<th>node 1</th>
<th>node 2</th>
<th>node 3</th>
<th>node 4</th>
<th>node 5</th>
<th>node 6</th>
<th>node 7</th>
<th>node 8</th>
<th>node 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>59401.32</td>
<td>59401.32</td>
<td>59401.32</td>
<td>59401.80</td>
<td>59400.26</td>
<td>59401.17</td>
<td>59401.17</td>
<td>59400.58</td>
<td>59402.29</td>
</tr>
</tbody>
</table>

TCP implementation: value of max(physclock, otherclock) at termination.

<table>
<thead>
<tr>
<th>node 1</th>
<th>node 2</th>
<th>node 3</th>
<th>node 4</th>
<th>node 5</th>
<th>node 6</th>
<th>node 7</th>
<th>node 8</th>
<th>node 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>59400.37</td>
<td>59400.29</td>
<td>59400.31</td>
<td>59401.08</td>
<td>59400.85</td>
<td>59400.85</td>
<td>59400.85</td>
<td>59400.29</td>
<td>59400.85</td>
</tr>
</tbody>
</table>

Figure 3: Empirical results for the clock synchronization system.

1. Evolve physclock variable at the rate chosen from the interval \([1 - r, 1 + r]\) until its value exceeds that of nextsend.
2. Increment nextsend by \(u\), and send messages to all system participants with the value of physclock.
3. Receive messages and update maxother with a maximum of maxother and a clock value contained in the message.

**Empirical Results** The target testing platform is as in the previous experiment (with Paxos). The clock synchronization automaton at each node is initialized with two parameters, \(u\) a delay between send events, and \(r\) a clock drift bound. In our experiments we choose \(u = 600\) (milliseconds) and \(r = 0.6\). Each node runs the main schedule loop 100 times.

Results in Fig. 3 represent the average of final values of physclock at each system participant and for both version of the implementation. A first observation is that the values for both systems are roughly similar, which is exactly what we expect. A second observation is that for each system, the clocks of individual nodes are approximately equal, where the difference between the maximum and minimum is 2.03 for MPI and 0.79 for TCP. Ideally this difference should be within 1.2 of each other. This discrepancy can be explained by the fact that the translation does not make any attempt at synchronization across the nodes, since the source specification does not require it. Per step 1. of the algorithm and natural differences in the speed of the physical machines, some nodes inevitably execute faster than others, terminate sooner, and stop sending messages, therefore, making it impossible for the slower nodes to synchronize with the fast ones and vice versa.

### 8.3 Partial Reversal Algorithm

Link reversal algorithms were first introduced in [13] to provide an efficient graph structure for routing. The main goal of link reversal algorithms is to ensure that all the nodes in a DAG have paths to a destination node or nodes. These algorithms can be used to solve problems such as leader election, mutual exclusion, scheduling and resource allocation [47].

```bash
% Given a graph and a particular node, this functions computes the incoming neighbors of that node.
let innbrs (g, n, u, c, w): Graph, Nodes, Node, Nat, Nodes -> Nodes =
  if c = len(w) then n
  else if (get(w, c), u) \in g then innbrs (g, (n|-get(w, c)), u, succ(c), w)
  else innbrs (g, n, u, succ(c), w);

% Given a graph and a particular node, this functions computes the outgoing neighbors of that node.
let outnbrs (g, n, u, c, w): Graph, Nodes, Node, Nat, Nodes -> Nodes =
  if c = len(w) then n
  else if (u, get(w, c)) \in g then outnbrs (g, (n|-get(w, c)), u, succ(c), w)
  else outnbrs (g, n, u, succ(c), w);

% Computes the set of neighbors of a given node in a given graph.
```
let nbs(g,u,w): Graph, Node, Nodes -> Nodes =
  innbrs(g,{},u,0,w) || outnbs(g,{},u,0,w);
% A node is a sink when all its incident edges are incoming.
let sink(g,u,w): Graph, Node, Nodes -> Bool =
  if innbrs(g,{},u,0,w) = nbs(g,u,w) then true
  else false;
% Partial reversal algorithm.
automaton pr(id: Node)
  signature
  input RECEIVE(m : Null[Chan_message])
  output SEND(m : Null[Chan_message])
  internal reverse, initialize(w:Nodes, g: Graph, dn: Node), ifsink
states
  parity : Nat := 0; % 0 -> the number of reversals performed so far is even, 1 -> odd
  initial : Graph := { };% initially g -> DAG, dn -> the sink node
  current : Graph := { };% initially g
  dest : Node := id;
  initialized : Bool := false;
  sendBuffer : Seq[Chan_message] := { };
  counter : Nat := 1;
  counters : Map[Node,Nat] := empty();% for n: Nat where n < len(inseq) do
  WORLD : Nodes := { };% compute the incoming set according to 'initially'
transitions
  internal initialize(w,g,dn) % g -> DAG, dn -> the sink node
    pre initialized = false;
  eff
    WORLD := w;
    dest := dn;
    initial := g;
    current := initial;
  initialized := true;
  internal reverse % the graph is initialized and the node is a sink
  locals
    inseq : Nodes := { };
    outseq : Nodes := { };
    tcnt : Nat := 0;
pre
  id "= dest; % destination does not perform reversal
  initialized := true;
  sink(current, id, WORLD) = true;
  counter := counter + 1;
  sendBuffer := { };% 'sendBuffer' will be populated with new reversal msgs.
  if parity = 0 then
    for n: Nat where n < len(inseq) do
      tcnt := 0;
      if defined(counters, get(inseq,n)) then
        tcnt := counters[get(inseq,n)];
      fi
      current := insert([id, get(inseq,n)], current); % reverse the edge to v
      current := delete([get(inseq,n), id], current);
      sendBuffer := sendBuffer |- [[reverse, counter, tcnt], id, get(inseq,n)];
    od
    parity := 1; % flip the parity bit
  else
    outseq := outnbs(initial,{},id,0,WORLD);% compute the outgoing set according to 'initially'
    for n: Nat where n < len(outseq) do
      tcnt := 0;
      if defined(counters, get(outseq,n)) then
        tcnt := counters[get(outseq,n)];
      fi
      current := insert([id, get(outseq,n)], current); % reverse the edge to v
      current := delete([get(outseq,n), id], current);
      sendBuffer := sendBuffer |- [[reverse, counter, tcnt], id, get(outseq,n)];
  fi

parity := 0; % flip the parity bit

internal ifsink
pre initialized;

if outnbrs(current,{},id,0,WORLD) = {} then
  print "sink";
  print id;
  print current;

fi

output SEND(m) where sendBuffer = {} 
pre m = nil();
output SEND(m) where sendBuffer ={}
pre m = embed(head(sendBuffer));

% messages remain in the buffer as long as they are not acknowledged
sendBuffer := sendBuffer - head(sendBuffer);
sendBuffer := tail(sendBuffer);

input RECEIVE(m)

locals index : Nat := 0;

if initialized /\ m = nil() /\ val(m).destination = id then
  while index < len(sendBuffer) do
    if get(sendBuffer, index).data.scnt <= val(m).data.rcnt then
      sendBuffer := eject(sendBuffer, index);
    fi
    index := succ(index);
  od

if defined(counters, val(m).sender) /\
  (counters[val(m).sender] < val(m).data.scnt /\
  counter <= val(m).data.rcnt)
then
  if val(m).data.tag = reverse then
    current := insert([val(m).sender, id], current);
    current := delete([id, val(m).sender], current);
    counters := update(counters, val(m).sender, val(m).data.scnt);
    % add acknowledgment message
    sendBuffer := sendBuffer - [ack, counter, val(m).data.scnt], id, val(m).sender];
  fi
fi


components
  PR : pr([myip1,myip2,myip3,myip4]);
  SM : SendMed(port,timeout);
  RM : RecvMed(port,timeout);
  CM : ChanMed(port,timeout);

schedule

world : Nodes := { };
graph : Graph := { };

m : Null[Chan_message] := nil();

wltman : Bool := false;
error : Null[JVMError] := nil();
isconn : Bool := false;

% insert ip addresses to the 'world'
world := world - [10.0.0.110];
world := world - [10.0.0.111];
world := world - [10.0.0.112];
world := world - [10.0.0.113];
world := world - [10.0.0.114];
The partial reversal (PR) algorithm presented here is an extension of the one in [42] and its complete specification can be found in Listing 21. In PR nodes maintain two lists, in-nbrs which is a list of nodes with incoming edges to it, and out-nbrs which is a list of nodes with outgoing edges from it. Based on the initial in-nbrs and out-nbrs sets a node determines which edges to reverse in each step. Whenever a node is a sink, it reverses either its in-nbrs or out-nbrs set, alternating between the two. In addition each node maintains a count variable that keeps track of the number of steps it has taken so far, and a derived variable parity representing the parity of count. The precondition for a node to perform a reverse action is that it is a sink. The effect of the reversal is that, depending on the value of parity, either its edges corresponding to in-nbrs or out-nbrs are reversed. Also, count is incremented. The number of such iterations is quadratic in the number of participants. In [42, 43] the PR algorithm is presented as a composition, and it had to be
Table 2: Partial reversal algorithm empirical results.

<table>
<thead>
<tr>
<th>Version</th>
<th>Avg. run time</th>
<th>Correct</th>
<th>Avg. msg. sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>6358.667 msec</td>
<td>93.33%</td>
<td>258.53</td>
</tr>
<tr>
<td>TCP</td>
<td>11299.967 msec</td>
<td>96.66%</td>
<td>263.73</td>
</tr>
</tbody>
</table>

decomposed into the node-channel form, a requirement of the translator (see Appendix B for the full code specification).

For the purpose of evaluation, the initial DAG is one that maximizes the number of link reversals, and this is when nodes are arranged in a line with a “loop” at its end ($n_1 \rightarrow \ldots n_k \rightarrow n_{k+1} \rightarrow n_{k+2} \rightarrow n_k$, where $n_i$’s are some node labels and $n_1 \rightarrow n_2$ are graph edges).

Note that the specification of the PR algorithm used in this work assumes reliable channels. Message loss can result in safety conditions not being satisfied, where nodes may incorrectly believe themselves to be sinks, and cycles may be formed. Although this algorithm can be augmented to support lossy channels by use of extra bookkeeping and communication rounds, we intentionally chose not to do it. This is because we are interested in observing all behaviors including unwanted ones and conditions under which these occur. Another reason for incorrect final state is that nodes may become out of sync with respect to each other hence not waiting long enough to receive or send the link reversal messages.

Empirical Results  Again the same deployment setting was utilized to collect empirical data.

The PR algorithm was specialized to channels implemented using MPI and TCP. During execution we utilize nine machines and run each version of the algorithm thirty (30) times. The data collected includes average runtime to termination, whether a correct sink node was selected or not, and the average number of messages sent. The sink node was changed from execution to execution. The only runtime performance optimization performed was the choice of the message timeouts parameter which was adjusted so that messages had enough time to be delivered, and it was 500 msec. The system-wide average number of sent events per node is 260. Our results appear in Table 2.

From Table 2 we observe, as expected, that MPI had a much better performance. Both versions were able to correctly identify the sink in almost all runs, where the MPI code failed twice, and TCP code failed once out of the 30 runs to terminate with the correct final state.

9 Future Work

The only advantage that the old IOA toolkit has over the one presented in this manuscript is the model of the input console. This however is only a minor functionality increment, which can be implemented as outlined in Tauber’s thesis [46].

More interesting work is planned in the expansion of the communication alternatives, such as one-to-many, many-to-many communication channels via use of group communication services and radio/wireless protocols. We are also interested in providing other models that are useful to system developer such as access to information storage such as SQL databases. Finally, we are interested in providing implementations for quantified expressions.
10 Summary

In this manuscript we present a new TEMPO to Java translation module that generates code deployable on distributed platforms with two different methods of communication. We formally demonstrate that this translation module preserves properties of its source specifications under stated restrictions.

References


Appendix

A Modeling TCP channel

In Section 5.8 provides a brief introduction into the model of the TCP channel. The omitted details will be discussed in the remainder of this section. The complete code for the automata modeling the sender and receiver components and the automaton modeling interaction with the TCP channel are found in Listings 22, 23, and 24 respectively.

A.1 Sender mediator

Sender mediator is the simplest of the three automata comprising our channel model. Interestingly, is the fact that messages may be sent by the external automaton anytime, even before any actual connections have been established. These messages will be deposited in the sendBuffer set where there they will await being passed to the channel automaton. In order to send messages, a connection must be made with the receiver, via TCP_senderOpen and TCP_respSenderOpen actions. With TCP_senderOpen the sender indicates a request to open a communication channel with the receiver, the channel responds with TCP_respSenderOpen which informs the sender automaton if the requested connection was established or not. If successful, messages from the sendBuffer can be written to the channel via TCP_write action. This action moves messages from sendBuffer to the channel mediator via an action pair (i.e., CHANMED automaton has the corresponding input TCP_write). A connection that is open can be closed at any time via TCP_senderClose and TCP_respSenderClose actions.

A noteworthy observation is that this specification is that the only timing behaviors are dictated by the SENDMED automaton, which specifies a single trajectory which models message deliver interval and is used to throttle the sender and hence prohibits from channel flooding.

A.2 Receive mediator

An automaton responsible for mediating message deliver from the channel to the higher level system is the RECEVMed. As it is in the case of the send mediator, the RECEIVE action is enabled as long as there are messages to be delivered and is completely independent of the state of the connection with any sender.

Of course there will be no messages to deliver unless the receiver demonstrates its willingness to receive these in the first place. This is analogous to establishing a JVM ServerSocket. In fact the receive mediator will emulate the steps of creating and configuring a server socket, which include binding (via TCP_bind and TCP_respBind) and listening/accepting (via TCP_accept and TCP_respAccept).

Receiver may opt to stop listening on a server socket and then subsequently stop accepting any connections, where this is done by first initiating TCP_stopListening and TCP_stopAccepting actions. Alternatively, select connections may be closed by the receiver. Receiver initiates TCP_rClose and TCP_rCloseStream.

During normal operation and after the receiver enters the accepting mode, messages may be read out of the channel, via the TCP_read and TCP_respRead actions. Specifically the TCP_read action will scan all of the connections and extract pending messages (one per connection). Received messages are added to the recvBuffer. The response action to read returns messages from the recvBuffer, if there are no more messages to be delivered a null message is returned.
A.3 Channel mediator

All of the actions pair either with the \texttt{SENDMED} or \texttt{RECVMED} automata and become hidden under composition. Since these actions have been already discussed in the light of the aforementioned automata, we will point out only the relevant issues pertaining to the interaction of \texttt{CHANMED} with the TCP protocol.

Per the rules of the TEMPO framework actions cannot be blocking. To avoid blocking all of the calls to \texttt{Socket} and \texttt{ServerSocket} objects use timeouts. Furthermore, all calls to these objects are wrapped in the vocabulary operators found in \texttt{TCPObjectVoc} and their implementation is provided during translation. Of course, the timeout values are parameters and can be statically or dynamically specified by the specification.

A.4 Complete channel system

Recall that model submitted for compilation are in the node-channel form. This means that each node will have an algorithm automaton which then connects to the channel. The channel consists of component running on the end points of the channel and the network medium. Our automata are created in a way to support multiple connections concurrently, see Figure 4, where this is done to ensure efficiency, but abstractly one can pretend that there exists a one-to-one communication channel between any pair of nodes as depicted in Figure 5.

![Figure 4: JVM Channel System connecting multiple nodes.](image)

![Figure 5: Point-to-point JVM Channel connecting two nodes.](image)

For the purpose of the proof we will consider each connection independently as in Figure 5, which is composition of six automata. We will refer to such channel as JVMCHAN.

A.5 Channel proof

To prove correctness of our channel model we could again use the approach presented in [24], however, to avoid repetition we will use a different approach. Again we will use a relation between two system, our channel specification (i.e., a child) and a parent specification which is that of an abstract lossy channel (i.e., a parent), depicted in Figure 6.

![Figure 6](image)

Again, we require that the two automata to be compatible, meaning that their external signatures must be same, i.e., $\text{sig} \left( \text{ABSCHAN} \right) = \text{sig} \left( \text{JVMCHAN} \right)$. This condition is easily verified per inspection and from the above description we know it to be true.

For simplicity we consider the scenario depicted in Figure 5 where node $i$ sends a message to node $j$. Correctness proof will establish trace inclusion between JVMCHAN and ABSCHAN. To this end we will show that:
Signature:

**Input:** SEND($m$)$_{i,j}$, where $i, j \in I$ and $m \in \text{msgs}$

**Output:** RECEIVE($m$)$_{i,j}$, where $i, j \in I$ and $m \in \text{msgs}$

**Internal:** lose($m$)$_{i,j}$, where $i, j \in I$ and $m \in \text{msgs}$

State:

transSeq, a sequence that is initially empty

Transitions:

<table>
<thead>
<tr>
<th>Effect</th>
<th>Precondition</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>input send($m$)$_{i,j}$</td>
<td></td>
<td>output receive($m$)$_{i,j}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>internal lose($m$)$_{i,j}$</td>
</tr>
<tr>
<td>$\text{messages} := \text{messages} \uplus m$</td>
<td>$\text{messages} = {} / m = \text{head}($\text{messages}$)$</td>
<td>$\text{messages} := \text{tail}($\text{messages}$)$</td>
</tr>
<tr>
<td></td>
<td>$m \in \text{messages}$</td>
<td>remove $m$ from $\text{messages}$</td>
</tr>
</tbody>
</table>

Figure 6: Signature, state, and transitions of the ABSCHAN automaton, the abstract asynchronous, lossy channel from $i$ to $j$.

Where, $cs$ and $cr$ correspond to the channel end points assigned for the connection between $i$ and $j$, and $i$ and $j$ respectively. Also, use notation $\text{socket}.\text{stream}$ to designate messages in transit over the network, and assume that streams behave like TEMPO sequences. Since TCP guarantees reliable service these messages will exist in these streams until their delivery (i.e., are read from the stream).

We present the JVMCHAN and the ABSCHAN into predicative style, and begin with the definition of the various places where messages may appear.

$$a \triangleq \text{SendMed}(i).\text{sendBuffer}$$

$$b \triangleq \text{ChanMed}(i).\text{tcpChannel}[cs].\text{socket}.\text{stream}$$

$$c \triangleq \text{ChanMed}(j).\text{tcpChannel}[cr].\text{socket}.\text{stream}$$

$$d \triangleq \text{ChanMed}(j).\text{recvBuffer}$$

$$e \triangleq \text{RecvMed}(j).\text{recvBuffer}$$

$$x \triangleq \text{AbsChan}(i,j).\text{inTransit}$$

$$\varphi_{\text{init}} \triangleq a = \{\} \land b = \{\} \land c = \{\} \land d = \{\} \land e = \{\} \land x = \{\}$$

Given the properties of the TCP protocol we can simplify the above definition since we know that $b = c$, hence since the two are equal we will forgo use of $c$.

The last definition depicts the initial state where all sequences are empty. Next, we define what is the state of these sequences after specific events, which are (A) SEND($m, i, j$), (B) write($m, i, j$), (C) read($j, i$), (D) respRead($m, j, i$), and (E) RECV($m, j, i$).
ϕ_A(m) ≜ a' = a \vdash m \land b' = b \land d' = d \land e' = e
ϕ_B(m) ≜ a \neq \{\} \land m = \text{head}(a) \land a' = \text{tail}(a) \\
\land b' = b \vdash m \land d' = d \land e' = e
ϕ_C(m) ≜ a' = a \\
\land b \neq \{\} \land m = \text{head}(b) \land b' = \text{tail}(b) \\
\land d' = d \vdash m \land e' = e
ϕ_D(m) ≜ a' = a \land b' = b \\
\land d \neq \{\} \land m = \text{head}(d) \land d' = \text{tail}(d) \\
\land e' = e \vdash m
ϕ_E(m) ≜ a' = a \land b' = b \land d' = d \\
\land e \neq \{\} \land m = \text{head}(e) \land e' = \text{tail}(e)

Similarly, we derive like definitions for the ABSCHAN.

Ψ_A(m) ≜ x' = x \vdash m \\
Ψ_E(m) ≜ x \neq \{\} \land m = \text{head}(x) \land x' = \text{tail}(x)

Next we define our relationship more formally and in the predicative style:

\rho ≜ e || d || b || a = x

Where || is the concatenation operator on sequences.

**Theorem A.1** JVMCHAN implements ABSCHAN.

**Proof.** We establish:

1. \( \varphi_{\text{init}} \land \rho \Rightarrow \Psi_{\text{init}} \)
2. \( \varphi_A(m) \land \rho \land \rho' \Rightarrow \Psi_A(m) \)
3. \( \varphi_B(m) \land \rho \land \rho' \Rightarrow x' = x \)
4. \( \varphi_C(m) \land \rho \land \rho' \Rightarrow x' = x \)
5. \( \varphi_D(m) \land \rho \land \rho' \Rightarrow x' = x \)
6. \( \varphi_E(m) \land \rho \land \rho' \Rightarrow \Psi_D(m) \)

We will now demonstrate each of the above points.

(1): Assume that \( \varphi_{\text{init}} \land \rho \) is true. Then \( x = e || d || b || a = \{\} || \{\} || \{\} || \{\} \).

(2): Assume that \( \varphi_A(m) \land \rho \land \rho' \) is true. Then \( x' = e' || d' || b' || a' = e || d || b || (a \vdash m) = (e || d || b || a) \vdash m = x \vdash m. \)

(3): Assume that \( \varphi_B(m) \land \rho \land \rho' \) is true. Then \( x' = e' || d' || b' || a' = e || d || (b \vdash \text{head}(a)) || \text{tail}(a) = (e || d || b || a) = x. \) (use \( a \neq \{\}. \))

(4): Assume that \( \varphi_C(m) \land \rho \land \rho' \) is true. Then \( x' = e' || d' || b' || a' = e || (d \vdash \text{head}(b)) || \text{tail}(b) || a = (e || d || b || a) = x. \) (use \( b \neq \{\}. \))
(5): Assume that $\varphi_D(m) \land \rho \land \rho'$ is true. Then $x' = e'||d'||b'||a' = (e \leftarrow head(d))||tail(d)||b||a = (e||d||b||a) = x$. (use $d \neq \{\}$.)

(6): Assume that $\varphi_E(m) \land \rho \land \rho'$ is true. Since we know that $e \neq \{\}$, then $x = e||d||b||a$ and $m = head(e) = head(e||d||b||a) = head(x)$. Finally, $x' = e'||d'||b'||a' = tail(e)||d||b||a = tail(e||d||b||a) = tail(z)$.

Per inspection of the SENDMED, RECVMED, and CHANMED we can see that the only actions that modify locations where messages may appear are only: (A) $\text{SEND}(m,i,j)$, (B) $\text{write}(m,i,j)$, (C) $\text{read}(j,i)$, (D) $\text{respRead}(m,j,i)$, and (E) $\text{RECV}(m,j,i)$. Per this observation and the above reasoning, we arrive at the hypothesis. \hfill \Box
B TCP Lossy Channel Abstraction

The TCP lossy channel is abstracted to the TIOA model as a composition of automata. Each automaton hosts three automata: SENDMED, RECVMED, CHANMED automata, and requires specific vocabulary definitions.

B.1 TCP Send Mediator Automaton

```plaintext
automaton SendMed(port: Nat, timeout: Nat)

signature

input SEND(m: Null[Chan_message])
output TCP_senderOpen(remote: Node, port: Nat)
input TCP_respSenderOpen(remote: Node, port: Nat, resp: Bool)
output TCP_senderClose(remote: Node)
input TCP_respSenderClose(remote: Node, resp: Bool)
output TCP_write(m: Null[Chan_message], s, r: Node)

states

sendBuffer : Seq[Chan_message] := { };
remoteStatus : Array[Node, Status] := constant(idle);
clocks : Array[Node, AugmentedReal] := constant(\text{infty});
clock := AugmentedReal := 0;

let
getMessage(s, r, index) : Node, Node, Nat -> Null[Chan_message] =
if index = len(sendBuffer) then
  nil() : Null[Chan_message]
else
  if sendBuffer[index].sender = s \&\& sendBuffer[index].destination = r
  then
    embed(sendBuffer[index])
  else
    getMessage(s, r, index + 1);

transitions

input SEND(m)
  eff
  if m ≠ nil() then
    sendBuffer := sendBuffer \&\& val(m);
  fi

output TCP_senderOpen(remote, port)
  pre
    remoteStatus[remote] = closed \&\& remoteStatus[remote] = idle;

input TCP_respSenderOpen(remote, port, resp)
  eff
  if resp then
    remoteStatus[remote] := connected;
  fi

output TCP_senderClose(remote)
  pre
    remoteStatus[remote] = connected;

input TCP_respSenderClose(remote, resp)
  eff
  if resp then
    remoteStatus[remote] := closed;
  fi

output TCP_write(m, s, r) where len(sendBuffer) = 0
  pre
    m = nil();
```

output TCP_write(m, s, r) where len(sendBuffer) \neq 0
locals
    tempSendBuffer : Seq[Chan_message] := \{\};
    msg : Null[Chan_message] := nil();
pre
    m = getMessage(s, r, 0);
eff
    msg := getMessage(s, r, 0);
if msg \neq nil() then
    for n:Nat where n < len(sendBuffer) do
        if sendBuffer[n] = val(msg) then
            clocks[r] := 0;
        else
            tempSendBuffer := tempSendBuffer \|- sendBuffer[n];
        fi
    od
    sendBuffer := tempSendBuffer;
fi
\hfill Trajectory modeling the delay needed for a message to be delivered to the remote node.
\algstruc{trajectories}
\begin{align*}
    \text{trajdef } & \text{DELAY evolve } d(\text{clock}) = 1; \\
    \text{trajdef } & \text{v}(n:Node) \\
    \text{invariant } & \text{len}(sendBuffer) \neq 0; \\
    \text{stop when } & \text{clocks}[n] >= \text{timeout}; \\
    \text{evolve } & d(\text{clocks}[n]) = 1;
\end{align*}
\hfill Listing 22: Send mediator automaton.

B.2 TCP Receive Mediator Automaton

\algstruc{automaton}RecvMed(port:Nat, timeout:Nat)
\algstruc{signature}
\begin{align*}
    & \text{output } \text{RECEIVE}(m: Null[\text{Chan_message}]) \\
    & \text{output } \text{TCP_read} \\
    & \text{input } \text{TCPRespRead}(m: Null[\text{Chan_message}]) \\
    & \text{output } \text{TCP_bind}(local:Node) \\
    & \text{input } \text{TCPRespBind}(error: Null[\text{JVMError}], local:Node) \\
    & \text{output } \text{TCP_accept} \\
    & \text{input } \text{TCPRespAccept}(error: Null[\text{JVMError}]) \\
    & \text{output } \text{TCP_stopAccepting} \\
    & \text{input } \text{TCP_getError}(e: Null[\text{JVMError}], remote:Node) \\
    & \text{output } \text{TCP_stopListening} \\
    & \text{output } \text{TCP_rCloseStream}(remote:Node) \\
    & \text{output } \text{TCP_rClose}(remote:Node)
\end{align*}
\algstruc{states}
\begin{align*}
    & \text{recvBuffer} : \text{Seq}[\text{Chan_message}] := \{\}; \\
    & \text{recvErrors} : \text{Map}[\text{Node}, Null[\text{JVMError}]] := \text{empty}() \\
    & \text{remoteStatus} : \text{Map}[\text{Node, Status}] := \text{empty}() \\
    & \text{localStatus} := \text{idle}; \\
    & \text{localError} : Null[\text{JVMError}] := nil(); \\
    & \text{noop} := \text{Bool} := \text{true};
\end{align*}
\algstruc{transitions}
\begin{align*}
    & \text{output } \text{RECEIVE}(m) \text{ where } \text{len}(\text{recvBuffer}) = 0 \\
    & \text{pre} \hspace{2cm} m = \text{nil}() \\
    & \text{output } \text{RECEIVE}(m) \text{ where } \text{len}(\text{recvBuffer}) \neq 0 \\
    & \text{pre} \hspace{2cm} m = \text{embed}(\text{head}(\text{recvBuffer})) \\
    & \text{eff} \hspace{2cm} \text{recvBuffer} := \text{tail}(\text{recvBuffer});
\end{align*}
output TCP_read
pre
    localStatus = idle;
  eff
    recvErrors := empty();

input TCP_respRead(m)
  eff
    if (m = nil())
    then
        recvBuffer := recvBuffer | val(m);
    fi

output TCP_bind(local)
pre
    localStatus = idle;
  eff
    localStatus := connecting;
    localError := nil();

input TCP_respBind(error, local)
locals
    ftoss : JVMError := "FailedToOpenServerSocket";
  eff
    if (error = nil())
      if (localStatus = connecting) then
          localStatus := accepting;
      fi
    else
      localError := error;
      if (val(error) = ftoss) then
          localStatus := idle;
      fi
    fi

output TCP_accept
pre
    localStatus = accepting;
  eff
    localStatus := waiting;

input TCP_respAccept(error)
  eff
    if (localStatus = waiting) then
        localStatus := accepting;
    fi
    localError := error;

output TCP_stopAccepting
pre
    localStatus = notAccepting;
  eff
    localStatus := idle;

output TCP_stopListening
pre
    localStatus = accepting;
  eff
    localStatus := closed;

output TCP_rClose(remote)
pre
    true;
  eff

noop := true;

output TCP_rCloseStream(remote)
pre
  true;
post
  noop := true;

input TCP_getError(e, remote) 
eff
  if (e != nil())
  then
    recvErrors := update(recvErrors, remote, e);
  fi

Listing 23: Receive mediator automaton.

B.3 TCP Channel Mediator Automaton

automaton ChanMed (port: Nat, timeout: Nat)
signature
  input TCP_read
  output TCP_respRead(m : Null[Chan_message])
  input TCP_bind(local : Node)
  output TCP_respBind(error : Null[JVMError], local : Node)
  input TCP_accept
  output TCP_respAccept(error : Null[JVMError])
input TCP_stopAccepting
input TCP_stopListening
input TCP_rClose(remote : Node)
input TCP_rCloseStream(remote : Node)
input TCP_senderOpen(remote : Node, port : Nat)
output TCP_respSenderOpen(remote : Node, port : Nat, resp : Bool)
input TCP_senderClose(remote : Node)
output TCP_respSenderClose(remote : Node, resp : Bool)
invariant TCP_write(m : Null[Chan_message], s, r : Node)
internal TCP_senderClosing(remote : Node)
output TCP_getError(e : Null[JVMError], remote : Node)

states
  SSocket := Null[JVMServerSocket] := nil();
  acceptStatus := Status := idle;
  SError := Null[JVMError] := nil();
  AError := Null[JVMError] := nil();
  tcpChannel := Seq[Channel] := {};
  recvBuffer := Seq[Chan_message] := {};

let
  getError(r, index) : Node, Nat -> Null[JVMError] =
    if index = len(tcpChannel) then
      nil() : Null[JVMError]
    else
      if tcpChannel[index].node = r then
        tcpChannel[index].error
      else
        getError(r, index+1);
    fi

  isConnected(r, index) : Node, Nat -> Bool =
    if index = len(tcpChannel) then
      false
    else
      if tcpChannel[index].status = connected then

50
true
else
  isConnected(r, index+1);
else
  isClosed(r, index) : Node, Nat -> Bool =
  if index = len(tcpChannel) then
    false
  else
    if tcpChannel[index].status = closed \ tcpChannel[index].status = idle
      then
        true
    else
      isConnected(r, index+1);

transitions
input TCP_read
locals
msg : Null[Chan_message] := nil();
eff
  for n: Nat where n < len(tcpChannel) do
    if tcpChannel[n].socket ≠ nil() \ tcpChannel[n].status = connected
      then
        tcpChannel[n].status := reading;
        msg := JVM_read_TCP(socket(tcpChannel[n].socket));
        if msg ≠ nil() then
          tcpChannel[n].error := embed("TimeoutOnRead");
        else
          recvBuffer := recvBuffer | val(msg);
          tcpChannel[n].error := nil();
        fi
  fi
od

output TCP_respRead(m) where len(recvBuffer) = 0
pre
  m = nil();
eff
  for n: Nat where n < len(tcpChannel) do
    if tcpChannel[n].status = reading then
      tcpChannel[n].status := connected;
  fi
od

output TCP_respRead(m) where len(recvBuffer) ≠ 0
pre
  m = embed(head(recvBuffer));
eff
  recvBuffer := tail(recvBuffer);
  for n: Nat where n < len(tcpChannel) do
    if tcpChannel[n].status = reading then
      tcpChannel[n].status := connected;
  fi
od

input TCP_bind(local)
eff
acceptStatus := connecting;
SSocket := JVM_TCPServerSocketOpen(local, port, timeout);
if SSocket = nil() then
  SError := embed("FailedToOpenServerSocket");
fi

output TCP_respBind(error, local)
pre
  acceptStatus = connecting;
  error = SError;
\textbf{input} TCP_accept \\
\textbf{locals} \\
socket : Null[JVMSocket] := nil(); \\
found : Bool := false; \\
\textbf{diff}
\begin{align*}
\text{if} &\ \text{acceptStatus} = \text{accepting} \text{ then} \\
&\text{acceptStatus} := \text{waiting}; \\
&\text{if} \ socke t \neq \text{nil}() \land \text{JVM_TCPSocketIsConnected}(\text{socket}) \text{ then} \\
&\text{for} \ n : \text{Nat} \text{ where} \ n < \text{len}(\text{tcpChannel}) \land \text{found} \text{ do} \\
&\text{if} \ \text{tcpChannel}[n].\text{node} = \text{val}(\text{JVM_TCPSocketGetRemoteIP}(\text{socket})) \text{ then} \\
&\text{found} := \text{true}; \\
&\text{if} \ \text{GT}(\text{val}(\text{JVM_TCPSocketGetRemoteIP}(\text{socket})), \ \text{val}(\text{JVM_TCPSocketGetLocalIP}(\text{socket}))) \land \\
&\text{tcpChannel}[n].\text{status} = \text{connected} \land \\
&\text{JVM_TCPSocketIsConnected}(\text{tcpChannel}[n].\text{socket}) \text{ then} \\
&\text{tcpChannel}[n].\text{socket} := \text{socket}; \\
&\text{tcpChannel}[n].\text{status} := \text{connected}; \\
&\text{tcpChannel}[n].\text{emptying} := \text{false}; \\
&\text{tcpChannel}[n].\text{error} := \text{nil}(); \\
&\text{fi} \\
&\text{fi} \\
&\text{od} \\
&\text{if} \ \text{found} = \text{false} \text{ then} \\
&\text{tcpChannel} := \text{tcpChannel} \mid \\
&\{ \text{val}(\text{JVM_TCPSocketGetRemoteIP}(\text{socket})), \text{socket, connected, false, nil}() \}; \\
&\text{fi} \\
&\text{else} \\
&AError := \text{embed}(\text{"NoConnectionOnAccept"}); \\
&\text{fi} \\
&\text{fi} \\
\textbf{output} TCP\_respAccept(error) \\
\textbf{pre} \\
error = AError; \\
\textbf{diff}
\begin{align*}
\text{if} &\ \text{acceptStatus} = \text{waiting} \text{ then} \\
&\text{acceptStatus} := \text{accepting}; \\
&AError := \text{nil}(); \\
&\text{fi} \\
\textbf{diff}
\begin{align*}
\textbf{input} &\ \text{TCP\_stopAccepting} \\
\textbf{diff}
\begin{align*}
\textbf{if} &\ \text{acceptStatus} = \text{stopping} \text{ then} \\
&\text{acceptStatus} := \text{idle}; \\
&\text{SError} := \text{JVM_TCPServerSocketClose}(\text{val}(\text{SSocket})); \\
&\text{if} \ \text{SError} \neq \text{nil}() \text{ then} \\
&\text{print} \ \text{SError}; \\
&\text{fi} \\
&\text{fi} \\
\text{input} &\ \text{TCP\_stopListening} \\
\text{diff}
\begin{align*}
\textbf{if} &\ \text{acceptStatus} = \text{idle} \text{ then} \\
&\text{acceptStatus} := \text{stopping}; \\
&\text{fi} \\
\text{fi} \\
\textbf{input} &\ \text{TCP\_rClose}(\text{remote}) \\
\textbf{diff}
\begin{align*}
\textbf{for} &\ y : \text{Nat} \text{ where} \ y < \text{len}(\text{tcpChannel}) \text{ do} \\
&\text{fi} \\
\end{align*}
\end{align*}
if tcpChannel[y].node = remote then
tcpChannel[y].status := rClosing;
fi

% input TCP_rCloseStream(remote)

for y: Nat where y < len(tcpChannel) do
if tcpChannel[y].node = remote /
 tcpChannel[y].status = rClosing /
 tcpChannel[y].emptying = false
then
tcpChannel[y].status := closed;
fi
od

% internal TCP_senderClosing(remote)
pre
len(tcpChannel) > 0;

for y: Nat where y < len(tcpChannel) do
if tcpChannel[y].status = emptying /
 tcpChannel[y].node = remote
then
tcpChannel[y].status := closed;
fi
od

% output TCP_getError(e, remote)
pre
e = getError(remote, 0);

for y: Nat where y < len(tcpChannel) do
if tcpChannel[y].socket = nil() /
 tcpChannel[y].error = nil() /
 tcpChannel[y].node = remote /
 tcpChannel[y].status = reading
then
tcpChannel[y].emptying := true;
fi
od
socket : Null[JVMSocket] := nil();
error : Null[JVMError] := nil();
eff
for n : Nat where n < len(tcpChannel) \ "found do
  if tcpChannel[n].node = remote then
    found := true;
    if tcpChannel[n].socket = nil() \ tcpChannel[n].status = closed then
      tcpChannel[n].socket := JVM_TCPSocketOpen(remote, port, timeout);
    fi
  fi
od
if (found = false) then
  socket := JVM_TCPSocketOpen(remote, port, timeout);
  if socket = nil() then
    tcpChannel := tcpChannel \ [remote, socket, connected, false, nil()];
  else
    tcpChannel := tcpChannel \ [remote, socket, closed, false, nil()];
  fi
fi
%
output TCP_respSenderOpen(remote, port, resp)
pre
  resp = isConnected(remote, 0);
%
input TCP_senderClose(remote)
locals
  error : Null[JVMError] := nil();
eff
for n : Nat where n < len(tcpChannel) do
  if tcpChannel[n].node = remote then
    tcpChannel[n].emptying := true;
    error := JVM_TCPSocketClose(val(tcpChannel[n].socket));
    if error = nil() then
      tcpChannel[n].error := error;
      print val(error);
    fi
  fi
od
%
output TCP_respSenderClose(remote, resp)
pre
  resp = isClosed(remote, 0);
Listing 24: Channel automaton.

B.4 TCP Channel Vocabulary

vocabulary TCPObjectsVoc
  types
    IPv4 : Tuple[one : Nat, two : Nat, three : Nat, four : Nat],
    JVMError : String
end
%
vocabulary TCPNodeVoc
9 imports TCPObjectsVoc
types
  Node : IPv4
operators
  GT : Node, Node \rightarrow Bool,
  EQ : Node, Node \rightarrow Bool,
LT : Node, Node -> Boolean

vocabulary alg, specific_voc

imports TCPObjectsVoc
imports TCPNodeVoc

Data : Tuple [ SomeDataFields : Int ]

types Chan_message : Tuple [data:Data, sender:Node, destination:Node],
Status : Enumeration [closed, notAccepting, opening, emptying, connecting, reading, rClosing, sConnected, connected, accepting, waiting, stopping, idle]

vocabulary tcp, specific_voc

imports alg, specific_voc

Chan_message : Tuple [data:Data, sender:Node, destination:Node],
Status : Enumeration [closed, notAccepting, opening, emptying, connecting, reading, rClosing, sConnected, connected, accepting, waiting, stopping, idle]

vocabulary JVMSocket

imports TCPObjectsVoc, TCPNodeVoc, tcp, specific_voc

operators JVM_TCPServerSocketOpen : Node, Nat, Nat -> Null [JVMSocket],
JVM_TCPServerSocketClose : JVMSocket -> Null [JVMError],
JVM_TCPServerSocketGetLocalIP : Null [JVMSocket] -> Null [Node],
JVM_TCPServerSocketGetRemoteIP : Null [JVMSocket] -> Null [Node],
JVM_read_TCPSocket : Null [JVMSocket] -> Null [Chan_message],
JVM_write_TCPSocket : JVMSocket, Chan_message -> Null [JVMError],
JVM_TCPServerSocketIsConnected : Null [JVMSocket] -> Boolean

vocabulary JVMServerSocket

imports TCPObjectsVoc, JVMSocket, TCPNodeVoc

operators JVM_TCPServerSocketOpen : Node, Nat, Nat -> Null [JVMServerSocket],
JVM_TCPServerSocketClose : JVMServerSocket -> Null [JVMError],
JVM_TCPServerSocketAccept : JVMServerSocket -> Null [JVMSocket]

This type provides sugar for the actual types and provides declaration for types in the specification of the JCP channel.

vocabulary ChannelVoc

imports JVMSocket, TCPObjectsVoc, tcp, specific_voc

Channel : Tuple [node : Node, socket:Null [JVMSocket],
status : Status, emptying: Boolean, error:Null [JVMError]]

operators empty_channel : -> Channel

Listing 25: TCP Channel Vocabulary.
C Paxos Specialized with TCP Channels

C.1 The Paxos Node

include "components/TCPVocabs.tioa"
imports JVMSocket
imports JVMServerSocket
imports TCPObjectsVoc
imports TCPNodeVoc
imports ChannelVoc

include "components/TCPRecvMed.tioa"
include "components/TCPSendMed.tioa"
include "components/starteralg.tioa"
include "components/bpleader.tioa"
include "components/bpagent.tioa"
include "components/bpsuccess.tioa"
include "components/leaderelector.tioa"
include "components/detector.tioa"

% % % meaning of automata parameters
% % % L: Int : upper bound on time to execute any enabled action
% % % D: Int : upper bound on message deliver time
% % % C: Int : time interval between checking if alive status of other nodes
% % % Z: Int : time interval between sending of alive message
% : argument :.
% for example > java paxos 192 168 2 2 8002 2000 700 500
% IP of the local node [n1.n2.n3.n4], Server Port is n5, Timeout is n6 (a.k.a., D)


components
  A_starteralg : starteralg([n1,n2,n3,n4],timeout,timeout,cas,asd); % parameters are —> L, D, C, Z
  A_detector : detector([n1,n2,n3,n4],timeout,timeout,cas,asd);
  A_bpleader : bpleader([n1,n2,n3,n4],timeout,timeout,cas,asd);
  A_bpagent : bpagent([n1,n2,n3,n4],timeout,timeout,cas,asd);
  A_bpsuccess : bpsuccess([n1,n2,n3,n4],timeout,timeout,cas,asd);
  A_leaderelector : leaderelector([n1,n2,n3,n4]);
  S : SendMed(port,timeout);
  R : RecvMed(port,timeout);
  C : ChanMed(port,timeout);

schedule

states
  myIP : Node := [n1,n2,n3,n4]; % IP from parameters
  world : Seq[Node] := { };
  D : AugmentedReal := timeout;
  Z : AugmentedReal := asd;
  dummy : Null[ tcp.message ] := nil();
  ms : Null[ tcp.message ] := nil();
  mr : Null[ tcp.message ] := nil();
  leaderIP : Node := [n1,n2,n3,n4];
  decision : Null[Int] := nil();
  value : Int := 0;
  exitloop : Bool := false;
  dowhile : Bool := true;
  isconn : Bool := false;
  error : Null[JVMError] := nil();

% — begins paxos schedule

do

% % % % % % % INITIALIZE

world := world \ [128,30,51,90];
world := world \ [128,30,51,97];
if [n1,n2,n3,n4] \notin world then
  world := world \ [n1,n2,n3,n4];
fi

% — hard–code any other seed nodes as world members
% world := world |− ...;
%
% — Trigger initialization of all components
fire output A_starteralg.systeminitialize(world);
65 % % % % % % % BIND TO SERVER SOCKET
% — Everyone sets up their server socket.
fire output R.TCP_bind(myIP);
fire output C.TCP_respBind(error,myIP);
% % % % % % % CONNECT TO EACH OTHER
70 % — create connections to the server
for n : Nat where n < len(world) do
  fire output S.TCP_senderOpen(world[n],port);
  follow S.DELAY duration 200;
  % — accept only on new connections
  fire output R.TCP_accept;
  % — listen and accept
  fire output C.TCP_respAccept(error);
  if (error ~ nil()) then print val(error); fi
  isconn := false;
  fire output C.TCP_respSenderOpen(world[n],port,isconn);
  if isconn then
    fire output A_detector.InformAlive(world[n]);
  fi
od
85 % % % % % % % Run the leader election protocol.
% — prep and send alive messages
fire internal A_detector.PrepAliveMessages;
for y : Nat where y < len(world) do
  fire internal A_detector.Check(world[y]);
  ms := nil();
  fire output A_detector.SEND(ms);
  if (ms ~ nil()) then
    print val(ms);
    fire output S.TCP_write(ms,val(ms).sender,val(ms).receiver);
  fi
  % — gives time for messages to arrive and be responded to
  follow A_detector.v duration \infty();
  % — extract messages from channel if there are any
  mr := nil();
  fire output R.TCP_read;
  fire output C.TCP_respRead(mr);
  if (mr ~ nil()) then
    fire output R.RECEIVE(mr);
    fire output A_detector.InformAlive(val(mr).sender);
  fi
od
for y : Nat where y < len(world) do
  fire internal A_detector.Check(world[y]);
  fire output A_detector.InformStopped(world[y]);
od
% % % % % % % Announce the leader (locally).
fire output A_leaderelector.Leader(leaderIP);
print leaderIP;
% % % % % % % RUN PAXOS
115 if (EQ(leaderIP,myIP)) then
  %% PAXOS LEADER ALGORITHM
  % — create a value to vote for and initialize
  value := choose x;
  fire input A_bpleader.Init(value);
  % gives time for messages to arrive and be responded to
  exitloop := false;
  while ~exitloop do
    % — leader starts a new round
    fire output A_starteralg.NewRound;
  fi
% — prep collect messages
fire internal A_bpleader.Collect;
% — send collect messages
dowhile := true;
while( dowhile = true ) do
  ms := nil();
  fire output A_bpleader.SEND( ms );
  if ms := nil() then
    fire output S.TCP_write( ms, val(ms).sender, val(ms).receiver );
    print val(ms);
  else
    dowhile := false;
fi
dowhile := true;
while( dowhile = true ) do
  fire output C.TCP_respyRead( mr );
  if ( mr := nil() ) then
    fire output R.RECEIVE( mr );
    print val(mr);
  else
    dowhile := false;
fi
do:
% — extract messages from channel if there are any
fire output R.TCP_read;
dowhile := true;
while( dowhile = true ) do
  mr := nil();
  fire output C.TCP_respyRead( mr );
  if ( mr := nil() ) then
    fire output R.RECEIVE( mr );
    print val(mr);
  else
    dowhile := false;
fi
od:
% — gather last messages
fire internal A_bpleader.GatherLast;
fire internal A_bpleader.Continue;
fire output A_bpleader.NextPhase( beginicast, exitloop );
fire internal A_starteralg.CheckRndSuccess;

od:
exitloop := false;
while ~exitloop do
% — prep and send beginicast messages
fire output A_bpleader.BeginCast;
for y:Nat where y < len( world ) do
  ms := nil();
  fire output A_bpleader.SEND( ms );
  if ( ms := nil() ) then
    fire output S.TCP_write( ms, val(ms).sender, val(ms).receiver );
    print val(ms);
  fi
  ms := nil( );
fi
% — gives time for messages to arrive and be responded to
fire output R.TCP_read;
dowhile := true;
while( dowhile = true ) do
  mr := nil();
  fire output C.TCP_respyRead( mr );
  if ( mr := nil() ) then
    fire output R.RECEIVE( mr );
    print val(mr);
  else
    dowhile := false;
fi
od:
% — process accept messages
fire internal A_bpleader.GatherAccept;
fire output A_bpleader.NextPhase( decided, exitloop );
fire internal A_starteralg.CheckRndSuccess;

od
% — reached decision

fire output A_bpleader.RndSuccess( decision );

fire internal A_starteralg.CheckRndSuccess;

fire internal A_bpleader.GatherOldRound;

exitloop := false;

while "exitloop do
% — prep and send announce success

fire internal A_bpsuccess.SendSuccess;

for y: Nat where y < len( world ) do
ms := nil();

fire output A_bpsuccess.SEND( ms );

if (ms = nil()) then

fire output S.TCP_write(ms, val(ms).sender, val(ms).receiver);

print val(ms);
fi
fi
follo

A_starteralg.v, A_bpsuccess.v, A_bpleader.v duration \infty();

% — extract messages from channel if there are any

fire output R.TCP_read;

dowhile := true;
while( dowhile = true ) do
mr := nil();

fire output C.TCP_respRead( mr );

if (mr = nil()) then

fire output R.RECEIVE( mr );

print val(mr);
else
dowhile := false;
fi
od:

fire internal A_bpsuccess.GatherAck;

fire output A_bpsuccess.HasEnoughAcks( exitloop );

od:
else

********* PAXOS AGENT ALGORITHM

exitloop := false;

while "exitloop do

follow A_starteralg.v, A_bpsuccess.v, A_bagent.v duration \infty();

% agents collect
% — extract messages from channel if there are any

fire output R.TCP_read;

dowhile := true;

while( dowhile = true ) do
mr := nil();

fire output C.TCP_respRead( mr );

if (mr = nil()) then

fire output R.RECEIVE( mr );

print val(mr);
else
dowhile := false;
fi
od:

% — three stages of agent, preconditions should ensure that
% only the proper one is executed

fire internal A_bagent.LastAccept;

fire internal A_bagent.Accept;

fire internal A_bpsuccess.GatherSuccess;

% — send response
dowhile := true;

while dowhile do
ms := nil();

fire output A_bagent.SEND( ms );

if (ms = nil()) then

59
Listing 26: Paxos: the node automaton.

C.2 The Starter Algorithm

automaton starter alg (IP: Node, L: Int, D: Int, C: Int, Z: Int)

signature
3
input Recover, Leader(r : Node), RndSuccess(v : Null[Int]), BeginCast
internal CheckRndSuccess
output NewRound, systeminitialize(w : Seq[Node])

states
1
clock : AugmentedReal := 0;
status : NodeMode := live;
iamleader : Bool := false;
sstart : Bool := false;

fire output S.TCP_write(ms, val(ms).sender, val(ms).receiver);
print val(ms);
else
dowhile := false;
fi
do;
fire output A.bpsuccess.NextPhase(exitloop);
fire internal A.bpsuccess.SendSuccess;
for y:Nat where y < len(world) do
ms := nil();
fire output A.bpsuccess.SEND(ms);
if ms = nil() then
fire output S.TCP_write(ms, val(ms).sender, val(ms).receiver);
print val(ms);
fi
do;

fire output A.bpsuccess.Decide(decision);
if (decision = nil()) then
print val(decision);
fi
% clean up any delayed messages (for MPI performance)
% —— extract messages from channel if there are any
fire output R.TCP_read;
dowhile := true;
while (dowhile = true) do
mr := nil();
fire output C.TCP_respRead(mr);
if (mr = nil()) then
fire output R.RECEIVE(mr);
print val(mr);
else
dowhile := false;
fi
od;
% close connections
for j : Nat where j < len(world) do
fire output S.TCP_senderClose(world[j]);
follow S.DELAY duration 10;
fire output C.TCP_respSenderClose(world[j], dowhile);
od
% close server socket
fire output R.TCP_stopListening;
fire output R.TCP_stopAccepting;
od % end schedule
phase : NodeMode := live;
phasename : AugmentedReal := ∞();
deadline : AugmentedReal := 0;
start : AugmentedReal := ∞();
slast : AugmentedReal := ∞();
rndsuccess : Bool := false;
world : Seq[Node] := { };

transitions
  output systeminitialize(w)
    pre true;
    eff
      world := w;
  input Stop
    eff
      status := stopped;

input Leader(r)
  eff
    if status = live then
      if EQ(r, IP) then
        iamleader := true;
      else
        iamleader := false;
      fi
      if rndsuccess = false then
        deadline := 0;
        sstart := true;
        start := clock + L;
      fi
    fi

input BeginCast
  eff
    if status = live then
      deadline := clock + 3 * L + 2 * len(world) * L + 2 * D;
    fi

input RndSuccess(v)
  eff
    if (status = live /\ v = nil()) then
      rndsuccess := true;
      slast := ∞();
    fi

output NewRound
  pre
    status = live;
    iamleader;
    sstart;
  eff
    sstart := false;
    start := ∞();

internal CheckRndSuccess
  pre
    status = live;
    iamleader;
    deadline := 0;
    clock > deadline;
  eff
    slast := ∞();
    if (rndsuccess = false) then
      sstart := true;
      start := clock + L;
Listing 27: Paxos: the starteralg automaton (starteralg.tioa).

C.3 The Detector Algorithm

```tioa
signature
  internal Check(j : Node), PrepAliveMessages
output SEND(m : Null[Chan_message]), InformStopped(n : Node), InformAlive(n : Node)
input Stop, Recover, RECEIVE(m : Null[Chan_message]), Leader(r : Node)
output HasEnough(b : Bool)
input systeminitialize(w: Seq[Node])
states
  leaderIP : Node := IP;
  clock : AugmentedReal := 0;
  status : NodeMode := live;
  world : Seq[Node] := \{\};
  alive : Set[Node] := \{\};
  prevrec : Map[Node, AugmentedReal] := empty();
  lastinform : Map[Node, AugmentedReal] := empty();
  lastsend : Map[Node, AugmentedReal] := empty();
  lastcheck : Map[Node, AugmentedReal] := empty();
  outmsgs : Seq[Chan_message] := \{\};
let
  lb(value, index) : AugmentedReal, Nat => AugmentedReal =
    if index > len(world) then
      value
    else if (world[index] \in alive \&\& world[index] = IP) then
      lb(min(value, min(lastinform[world[index]]),
        min(lastsend[world[index]], lastcheck[world[index]])), succ(index))
    else
      lb(value, succ(index))
transitions
input systeminitialize(w)
eff
  world := w;
  alive := insert(IP, alive);
  for n : Nat where n < len(world) do
    prevrec := update(prevrec, world[n], clock);
    lastinform := update(lastinform, world[n], clock);
    lastsend := update(lastsend, world[n], clock);
    lastcheck := update(lastcheck, world[n], clock);
  od
input Leader(r)
eff
  leaderIP := r;
input Stop
eff
  status := stopped;
```

62
\textbf{input} Recover  \\
\textbf{eff}  \\
status := live;  \\
\% output SEND( m )  \\
\textbf{pre}  \\
status = live;  \\
len(outmsgs) := 0;  \\
m = embed(head(outmsgs));  \\
\textbf{eff}  \\
lastsend := update(lastsend, val(m).destination, clock + Z);  \\
outmsgs := tail(outmsgs);  \\
\% input RECEIVE( m )  \\
\textbf{eff}  \\
if status = live /\ m \neq \text{nul}() then  \\
\quad if val(m).data.M = live then  \\
\quad \quad alive := insert(val(m).sender, alive);  \\
\quad \quad lastcheck := update(lastcheck, val(m).sender, clock + C);  \\
\quad \quad prevrec := update(prevrec, val(m).sender, clock);  \\
\quad fi  \\
\fi  \\
\% \textbf{internal} Check(j)  \\
\textbf{pre}  \\
status = live;  \\
\textbf{eff}  \\
lastcheck := update(lastcheck, j, clock + C);  \\
if defined(prevrec, j) then  \\
\quad if clock > (prevrec[j] + Z + D) then  \\
\quad \quad alive := delete(j, alive);  \\
\quad fi  \\
\fi  \\
\% \textbf{internal} PrepAliveMessages  \\
\textbf{pre}  \\
status = live;  \\
\textbf{eff}  \\
for n: Nat where n < len(world) do  \\
\quad outmsgs := outmsgs \leftarrow [[live, [0,0,0,0]], [0,[0,0,0,0]], 0], IP, world[n]];  \\
\textbf{od}  \\
\% output InformStopped(n)  \\
\textbf{pre}  \\
status = live;  \\
\quad n \not\in\text{alive};  \\
\textbf{eff}  \\
lastinform := update(lastinform, n, clock + L);  \\
\% output InformAlive(n)  \\
\textbf{pre}  \\
status = live;  \\
\textbf{eff}  \\
if (n \notin \text{alive}) then  \\
\quad alive := insert(n, alive);  \\
\fi  \\
\textbf{eff}  \\
lastinform := update(lastinform, n, clock + L);  \\
\% output HasEnough(b)  \\
\textbf{let}  \\
majority(num) : Nat \rightarrow \text{Bool} =  \\
\quad \text{if num} \geq \text{floor}((1+size(alive))/2) \text{ then true else false};  \\
\textbf{count}(i, num) : Nat,Nat \rightarrow \text{Bool} =
if \((i+1) = \text{len}(\text{world})\) then \text{majority}(\ num\ )
else \text{count}(\ \text{succ}(i), \text{if clock < prevrec[world[i]] + D + Z then num + 1 else num};

\begin{verbatim}
115  \text{pre}
   \text{status} = \text{live};
   b = \text{count}(0, 0);
\text{eff}
   \text{print count}(0, 0);
\end{verbatim}

\begin{verbatim}
120  \text{trajectories}
\text{trajdef} \gamma
   \text{invariant} \text{status} = \text{live};
\text{stop when} \text{clock} \geq \text{lb}([\text{infty}, 0]);
\text{evolve} d(\text{clock}) = 1;
\end{verbatim}


C.4 The Leader Election Algorithm

\begin{verbatim}
1 \text{automaton} \text{leaderElector}(\text{IP} : \text{Node})
\text{signature}
   \text{input} \text{InformStopped}(j : \text{Node}), \text{InformAlive}(j : \text{Node}), \text{Stop}, \text{Recover}
   \text{output} \text{Leader}(r : \text{Node})
   \text{input} \text{systeminitialize}(w : \text{Seq[Node]})
\text{states}
   \text{status} : \text{NodeMode} := \text{live};
   \text{world} : \text{Seq[Node]} := \{ \};
   \text{alive} : \text{Set[Node]} := \{ \};
   \text{leaderIP} : \text{Node} := [0, 0, 0, 0];
\text{transitions}
   \text{input} \text{systeminitialize}(w)
   \text{eff}
      \text{status} := \text{live};
      \text{world} := w;
      \text{alive} := \text{insert}(\text{IP}, \text{alive});
   \text{input} \text{Stop}
   \text{eff}
      \text{status} := \text{stopped};
   \text{output} \text{Leader}(r)
\text{pre}
   \text{status} = \text{live};
   r = \text{leaderIP};
\text{input} \text{InformStopped}(j)
\text{locals}
   \text{maxip} : \text{Node} := \text{IP};
   \text{if} (\text{status} = \text{live}) \text{then}
      \text{alive} := \text{delete}(j, \text{alive});
      \text{for } y : \text{Nat} \text{ where } y < \text{len}(\text{world}) \text{ do}
         \text{if } (\text{world}[y] \\notin \text{alive} \land \text{LT}(\text{maxip}, \text{world}[y])) \text{ then}
            \text{maxip} := \text{world}[y];
   \text{fi}
\text{od}
   \text{leaderIP} := \text{maxip};
\text{fi}
\text{input} \text{Recover}
   \text{eff}
      \text{status} := \text{live};
\text{input} \text{InformAlive}(j)
\text{locals}
   \text{maxip} : \text{Node} := \text{IP};
   \text{if} \text{status} = \text{live} \text{then}
      \text{if} (j \\notin \text{alive}) \text{ then}
\end{verbatim}

64
alive := insert(j, alive);
fi
if (j \notin world) then
  world := world \setminus j;
fi
for y:Nat where y < len(world) do
  if (LT(maxip, world[y])) then
    maxip := world[y];
  fi
od;
leaderIP := maxip;
fi

Listing 29: Paxos: the leader elector automaton (leaderelector.tioa).

C.5 The Paxos Leader Algorithm

automaton bpleader(IP:Node, L:Int, D:Int, C:Int, Z:Int)
signature
3 input RECEIVE(m:Null[Chan,message]), Init(v:Int), NewRound, Stop
input Recover, Leader(r:Node)
internal Collect, GatherLast, Continue, GatherAccept, GatherOldRound
output SEND(m:Null[Chan,message]), BeginCast, RndSuccess(v:Null[Int])
output NextPhase(m:NodeType, b:Boolean)
8 input systeminitialize(w:Seq[Node])
states
world :Seq[Node] := { }; NIL : Int := 0;
status :NodeType := live;
iamleader :Boolean := false;
mode :NodeType := rnddone;
Initvalue :Null[Int] := nil();
decision :Null[Int] := embed(0:Int);
current :Round := [0, {0,0,0}]; highestround :Round := [0, {0,0,0}];
rndvalue :Int := 0;
rndvfrom :Round := [0, {0,0,0}];
rndinfquo :Set[Node] := { };
rndacqou :Set[Node] := { };
phased end : AugmentedReal := \infty;
transitions
28 input systeminitialize(w)
  eff
  world := w;
  %
input Stop
  eff
  status := stopped;
  %
input Leader(r)
  eff
  if status = live then
    if EQ(r,IP) then
      iamleader := true;
      phased end := clock + L;
    else
      iamleader := false;
    fi
  fi
output SEND(m)

pre
  status = live;
iameleader;
len(outmsgs) = 0;
m = embed(head(outmsgs));
eff
  outmsgs := tail(outmsgs);

input Recover

eff
  status := live;

input RECEIVE(m)

locals
  mval : Chan_message;
eff
  if status = live /\ m = nil() then
    mval := val(m);
    if (mval.data.M = last \/ mval.data.M = accept \/
      mval.data.M = success \/ mval.data.M = oldround)
    then
      inmsgs := inmsgs |- mval;
    fi
  fi

input Init(v)
eff
  if status = live /\ iameleader then
    initvalue := embed(v);
  fi

input NewRound

eff
  if status = live /\ iameleader then
    currn.d.C := highestrnd.C + 1;
    currn.d.O := IP;
    highestrnd := currn.d;
    mode := collect;
    phasend := clock + 2 * D + 5 * L;
  fi

internal Collect

pre
  status = live;
iameleader;

eff
  rndvfrom := [0:Int, IP];
  rndinfquo := { };
  rndaccquo := { };
  for y: Nat where y < len(world) do
    if "EQ(world[y], IP) then
      outmsgs := outmsgs |- [[collect, currnd, currn, NIL], IP, world[y]]:
    fi
  od
  mode := gatherlast;
  phasend := clock + 2 * D + 5 * L;

internal GatherLast

locals
  data : Data;
  t.inmsgs : Seq[Chan_message] := { };

pre
status = live;
iamleader;
mode = gatherlast;

\textbf{eff}
\begin{align*}
\text{for } y : \text{Nat where } y < \text{len}(\text{inmsgs}) \text{ do} \\
\text{data} := \text{inmsgs}[y].\text{data}; \\
\text{if } (\text{data}.M = \text{last}) \text{ then} \\
\text{if } (\text{data}.\text{RP} = \text{currnd}) \text{ then} \\
\text{if } (\text{inmsgs}[y].\text{sender} \notin \text{rndinfquo}) \text{ then} \\
\text{rndinfquo} := \text{insert}(\text{inmsgs}[y].\text{sender}, \text{rndinfquo}); \\
\text{fi} \\
\text{if } (\text{rndvfrom}.\text{C} < \text{data}.\text{RP}.\text{C} \land (\text{rndvfrom}.\text{C} = \text{data}.\text{RP}.\text{C} \lor \\
\text{LT}(\text{rndvfrom}.\text{O}, \text{data}.\text{RP}.\text{O}) \lor \text{data}.V = \text{NIL}) \text{ then} \\
\text{rndvalue} := \text{data}.V; \\
\text{rndvfrom} := \text{data}.\text{RP}; \\
\text{fi} \\
\text{if } \text{size}(\text{rndinfquo}) >= (\text{div}(\text{len}(\text{world}) - 1.2) + 1 \text{ then} \\
\text{if } \text{rndvalue} = \text{NIL} \land \text{initvalue} = \text{nil()} \text{ then} \\
\text{rndvalue} := \text{val}(\text{initvalue}); \\
\text{fi} \\
\text{if } \text{rndvalue} = \text{NIL} \text{ then} \\
\text{mode} := \text{begincast}; \\
\text{phaseend} := \text{clock} + 2 * \text{D} + 5 * \text{L}; \\
\text{else} \\
\text{mode} := \text{wait}; \\
\text{fi} \\
\text{phaseend} := \text{\infty();} \\
\text{fi} \\
\text{fi} \\
\text{else} \\
\text{t_inmsgs} := \text{t_inmsgs} \setminus \text{inmsgs}[y]; \\
\text{fi} \\
\text{od}; \\
\text{inmsgs} := \text{t_inmsgs}; \\
\%\textbf{internal} Continue \\
\textbf{pre} \\
\text{status} = \text{live}; \\
\text{iamleader}; \\
\text{mode} = \text{wait}; \\
\text{initvalue} = \text{nil();} \\
\textbf{eff} \\
\text{if } \text{rndvalue} = \text{NIL} \text{ then} \\
\text{rndvalue} := \text{val}(\text{initvalue}); \\
\text{mode} := \text{begincast}; \\
\text{phaseend} := \text{clock} + 2 * \text{D} + 5 * \text{L}; \\
\%\textbf{output} BeginCast \\
\textbf{pre} \\
\text{status} = \text{live}; \\
\text{iamleader}; \\
\text{mode} = \text{begincast}; \\
\textbf{eff} \\
\text{for } y : \text{Nat where } y < \text{len}(\text{world}) \text{ do} \\
\text{if } \text{EQ}(\text{world}[y], \text{IP}) \text{ then} \\
\text{outmsgs} := \text{outmsgs} \setminus [\text{begin, currnd, currnd, rndvalue}, \text{IP}, \text{world}[y]]; \\
\text{fi} \\
\text{od} \\
\text{mode} := \text{gatheraccept}; \\
\text{phaseend} := \text{clock} + 2 * \text{D} + 5 * \text{L}; \\
\%\textbf{internal} GatherAccept \\
\textbf{locals}
data : Data;
t_inmsgs : Seq[Chan_message] := { };

pre
  status = live;
iамleader;
  mode = gatheraccept;

eff
  for y: Nat where y < len(inmsgs) do
    data := inmsgs[y].data;
    if (data.M = accept) then
      if data.R = currnd then
        rndacq := insert(inmsgs[y].sender, rndacq);
      fi
    fi
    if size(rndacq) >= (div(len(world) - 1, 2)) + 1 then
decision := embed(rndvalue);
    mode := decided;
    phaseend := clock + 2 * D + 5 * L;
  else
    t_inmsgs := t_inmsgs | - inmsgs[y];
  fi;
  od;
inmsgs := t_inmsgs;

% output RndSuccess(v)
pre
  status = live;
iамleader;
  mode = decided;
  v = decision;

eff
  mode := rnddone;
  phaseend := clock + 2 * D + 5 * L;

% internal GatherOldRound

locals
data : Data;
t_inmsgs : Seq[Chan_message] := { };

pre
  status = live;

eff
  for y: Nat where y < len(inmsgs) do
    data := inmsgs[y].data;
    if (data.R.C < currnd.C /
      (data.R.C = currnd.C /
       LT(data.R.O, currnd.O) )) /
      data.M = oldround)
then
  highestnd := data.RP;
else
  t_inmsgs := t_inmsgs | - inmsgs[y];
fi
  od;
inmsgs := t_inmsgs;

% output NextPhase(m, b)
let
test2(m) : NodeMode -> Bool = if m = mode then true else false;

pre
  b = test2(m);
  trajectories

trajdef v
  invariant status = live;
  stop when clock >= phaseend;
  evolve d(clock) = 1;
C.6 The Paxos Agent Algorithm

```plaintext
automaton bagent(IP : Node, L : Int, D : Int, C : Int, Z : Int)

signature
  input RECEIVE(m : Null[Chan_message]), Init(v : Int), Stop, Recover
  internal LastAccept, Accept
  output SEND(m : Null[Chan_message])
  input systeminitialize(w: Seq[Node])

states
  status : NodeMode := live;
  lastr : Round := [0 : Int,[0,0,0,0] : Node];
  lastv : Int := 0;
  commit : Round := [0 : Int,[0,0,0,0] : Node];
  inmsgs : Seq[Chan_message] := { }; 
  outmsgs : Seq[Chan_message] := { }; 
  clock : AugmentedReal := 0;
  phaseend : AugmentedReal := \infty;

transitions
  input systeminitialize(w)
  eff
    clock := 0;

  input Stop
  eff
    status := stopped;

  output SEND(m)
  pre
    status = live;
    len(outmsgs) \neq 0;
    m = embed(head(outmsgs));
  eff
    outmsgs := tail(outmsgs);

internal LastAccept

locals
  data : Data;
  t_inmsgs : Seq[Chan_message] := { }; 

pre
  status = live;
  len(inmsgs) \neq 0;

eff
  for y : Nat where y \leq len(inmsgs) do 
    data := inmsgs[y].data;
    if (data.M = collect) then
      if (commit.C < data.R.C \/
        (commit.C = data.R.C \/
        LT(commit.O, data.R.O))) then
        commit := data.R;
        outmsgs := outmsgs |− [[last, lastr, data.R, lastv], IP, inmsgs[y].sender];
        phaseend := clock + 2 * D + 5 * L;
      else
        outmsgs := outmsgs |− [[oldround, commit, data.R, lastv], IP, inmsgs[y].sender];
        phaseend := clock + D + 5 * L;
      fi
    else
      t_inmsgs := t_inmsgs |− inmsgs[y];
    fi
  od
  inmsgs := t_inmsgs;
```

Listing 30: Paxos: the leader automaton (bpleader.tioa).
Listing 31: Paxos: the agent automaton.

C.7 The Paxos Success Algorithm
clock  : AugmentedReal := 0;
status : NodeMode := live;
decision : Int := 0;
leaderIP : Node := [0,0,0,0];
iamleader : Bool := false;
15
acked : Set[Node] := { };
prevsend : AugmentedReal := 0;
lastsend : AugmentedReal := ∞;
lastwait : AugmentedReal := ∞;
lastg : AugmentedReal := ∞;
lastss : AugmentedReal := ∞;
inmsgs : Seq[Chan_message] := { };

transitions
input systeminitilize(w)
eff
   leaderIP := IP;
   world := w;
input InformAlive(n)
eff
   if (n \not in world) then
      world := world |− n;
   fi
input Stop
eff
   status := stopped;
input Leader(r)
eff
   if status = live then
      if (EQ(r,leaderIP)) then
         iamleader := true;
      else
         iamleader := false;
   fi
   leaderIP := r;
output SEND(m) where len(outmsgs) = 0
pre
   status = live;
   m = nil();
eff
   lastsend := ∞;
output SEND(m) where len(outmsgs) ≠ 0
pre
   status = live;
   m = embed(head(outmsgs));
eff
   lastsend := clock + L;
   outmsgs := tail(outmsgs);
input RndSuccess(v)
eff
   if status = live /\ v = nil() then
      decision := val(v);
   lastss := clock + L;
fi
input Recover
eff
   status := live;
input RECEIVE(m)
locals
   mval : Chan_message;
eff
   if status = live /\ m = nil() then
mval := val(m);
if (mval.data.M = ack \ mval.data.M = success) then
  inmsgs := inmsgs \ mval;
if (mval.data.M = ack \ lastga = \infty()) then
  lastga := clock + L;
fi
if mval.data.M = success \ lastgs = \infty() then
  lastgs := clock + L;
fi
fi

internal SendSuccess
pre
  status = live;
  iamleader = true;
  prevsend = 0;
  prevsends = 0;
  decision := 0;
  prevsend := 0;
  t_inmsgs := t_inmsgs := \infty();

for y: Nat where y < len(world) do
  if (world[y] \ notin acked) \ EQ(world[y], IP) then
    outmsgs := outmsgs := \success, [0, [0, 0, 0, 0]], [0: Int, [0, 0, 0, 0]], decision], IP, world[y]:
    od
  prevsend := clock + L;
  lastsend := clock + L + 2 * len(world) + 2 * L + 2 * D + L;
  lastss := \infty();

internal GatherSuccess
locals
  data: Data;
  t_inmsgs := t_inmsgs := \{\};
pre
  status := live;
  foundit := false;
  prevsend := 0;
  prevsends := 0;
  decision := 0;
  prevsend := 0;
  t_inmsgs := t_inmsgs := \{\};
output Decide(v)
pre
  status := live;
  decision := 0;
  v := embed(decision);
  t_inmsgs := t_inmsgs := \{\};

internal GatherAck
locals
  data: Data;
  t_inmsgs := t_inmsgs := \{\};
pre
  status := live;
  foundit := false;
  prevsend := 0;
  prevsends := 0;
  decision := 0;
  prevsend := 0;
  t_inmsgs := t_inmsgs := \{\};

for y: Nat where y < len(inmsgs) do
  data := inmsgs[y].data;
  if data.M = success then
    decision := data.V;
    outmsgs := outmsgs := \ack, [0, [0, 0, 0, 0]], [0: Int, [0, 0, 0, 0]], decision], IP, inmsgs[y].sender:
  else
    t_inmsgs := t_inmsgs := inmsgs[y]:
  od
  inmsgs := t_inmsgs:
output Decide(v)
pre
  status := live;
  decision := 0;
  v := embed(decision);
  t_inmsgs := t_inmsgs := \{\};

for y: Nat where y < len(inmsgs) do
  data := inmsgs[y].data;
  if (data.M = ack) then
    if (inmsgs[y].sender \ notin acked) then
      acked := insert(inmsgs[y].sender, acked);
    fi
    foundit := true;
  else
  fi
  od
\[ t_{\text{inmsgs}} := t_{\text{inmsgs}} \leftarrow \text{inmsgs}[y]; \]
\[ \text{fi} \]
\[ \text{od}; \]
\[ \text{inmsgs} := t_{\text{inmsgs}}; \]
\[ \text{if} \text{foudit then} \]
\[ \text{lastga} := \text{infty}(); \]
\[ \text{else} \]
\[ \text{lastga} := \text{clock} + L; \]
\[ \text{fi} \]
\[ \text{output} \text{HasEnoughAcks}(b) \]
\[ \text{pre} \]
\[ \text{status} = \text{live}; \]
\[ b = \text{(size}(\text{acked}) \geq (\text{div}(\text{len}(\text{world}) - 1, 2) + 1); \]
\[ \text{internal} \text{Wait} \]
\[ \text{pre} \]
\[ \text{status} = \text{live}; \]
\[ \text{prevsend} = 0; \]
\[ \text{clock} > \text{prevsend} + (4 * L + 2 * \text{len}(\text{world}) * L + 2 * D); \]
\[ \text{eff} \]
\[ \text{prevsend} := 0; \]
\[ \text{lastwait} := \text{infty}(); \]
\[ \text{output} \text{NextPhase}(b) \]
\[ \text{let} \]
\[ \text{test1()} : \rightarrow \text{Bool} = \text{if} \text{decision} = 0 \text{then} \text{true} \text{else} \text{false}; \]
\[ \text{pre} \]
\[ b = \text{test1}(); \]
\[ \text{trajectories} \]
\[ \text{trajdef v} \]
\[ \text{stop when} \text{clock} \geq \text{lastsend} /\text{clock} \geq \text{lastwait} /\text{clock} \geq \text{lastss} /\text{clock} \geq \text{lastgs} /\text{clock} \geq \text{lastga}; \]
\[ \text{evolve d(clock)} = 1; \]

Listing 32: Paxos: the success automaton (bpsuccess.tioa).

### C.8 The Vocabulary for Paxos

```plaintext
classification vocabulary TCPObjectsVoc
types
  IPv4 = Tuple [one: Nat, two: Nat, three: Nat, four: Nat],
  IPv6 = Tuple [one: Nat, two: Nat, three: Nat, four: Nat, five: Nat, six: Nat],
  JVMError = String.
end
classification vocabulary TCPNodeVoc
imports TCPObjectsVoc

types
  Node = IPv4
operators
  GT : Node, Node \rightarrow Bool,
  EQ : Node, Node \rightarrow Bool,
  LT : Node, Node \rightarrow Bool
end
classification algorithm vocabcs :
classification vocabulary alg_specific_voc
imports TCPObjectsVoc
imports TCPNodeVoc
types
  NodeMode = Enumeration [live, stopped, begin, last, accept, success, oldround, collect, gatherlast, wait, begincast, gatheraccept, decided, rnddone, ack],
  Round = Tuple [C:Int, O:Node],
  Data = Tuple [M:NodeMode, R:Round, RP:Round, V:Int],
  Mode = Enumeration [done, working, leader, notleader]
```
vocabulary tcp_specific_voc
  imports alg_specific_voc
  types
    Chan_message : Tuple [ data : Data , sender : Node , destination : Node ] ,
    Status : Enumeration [ closed , notAccepting , opening , emptying ,
      connecting , reading , rClosing , sConnected , connected ,
      accepting , waiting , stopping , idle ]
end

%% JVM Socket types and operations
vocabulary JVMSocket
  imports TCPObjectsVoc , TCPNodeVoc , tcp_specific_voc
  imports alg_specific_voc
  types JVMSocket
    operators
      JVM_TCPStartSocketOpen : Node , Nat , Nat -> Null [ JVMSocket ] ,
      JVM_TCPStartSocketClose : JVMSocket -> Null [ JVMError ] ,
      JVM_TCPStartSocketGetLocalIP : Null [ JVMSocket ] -> Null [ Node ] ,
      JVM_TCPStartSocketSend : JVMSocket , Chan_message -> Null [ JVMError ] ,
end

vocabulary JVMServerSocket
  imports TCPObjectsVoc , JVMSocket , TCPNodeVoc
  types JVMServerSocket
    operators
      JVM_TCPStartServerSocketOpen : Node , Nat , Nat -> Null [ JVMServerSocket ] ,
      JVM_TCPStartServerSocketClose : JVMServerSocket -> Null [ JVMError ] ,
      JVM_TCPStartServerSocketAccept : JVMServerSocket -> Null [ JVMSocket ]
end

%% This type provides sugar for the actual types and provides
%% declaration for types in the specification of the JCP channel.

vocabulary ChannelVoc
  imports JVMSocket , TCPObjectsVoc , tcp_specific_voc
  types
    Channel : Tuple [ node : Node , socket : Null [ JVMSocket ] ,
      status : Status , emptying : Bool , error : Null [ JVMError ] ]
  operators
    empty_channel : -> Channel
end

Listing 33: Vocabulary (TCPVocabs.tioa).
D Paxos Specialized with MPI Channels

D.1 Paxos Node

%%% Paxos implementation based on DePrisco thesis, by Peter M. Musial

#include "myvocabs.tioa"

imports mpi_message_voc
imports mpi_request_voc
imports mpi_status_voc
imports mpi_voc
imports paxos_voc

%%% : MPI mediator automata:

#include "ReceiveMediator.tioa"
#include "SendMediator.tioa"

%%% : Paxos automata

#include "stateralg.tioa"

#include "bpleader.tioa"
#include "bpagent.tioa"
#include "bsuccess.tioa"
#include "leaderselector.tioa"
#include "detector.tioa"

%%% meaning of automata parameters
%%% L: Int :: upper bound on time to execute any enabled action
%%% D: Int :: upper bound on message deliver time
%%% C: Int :: time interval between checking if alive status of other nodes
%%% Z: Int :: time interval between sending of alive message

automaton paxos

components
A_stateralg : stateralg(5,100,510,500); % parameters are \rightarrow L, D, C, Z
A_detector : detector(5,100,510,500);
A_bpleader : bpleader(5,100,510,500);
A_bpagent : bpagent(5,100,510,500);
A_bsuccess : bsuccess(5,100,510,500);
A_leaderselector : leaderselector;
SM : SendMediator;
RM : ReceiveMediator;

schedule

states
D: AugmentedReal := 500;
Z: AugmentedReal := 20;
dummy: Null[mpi_message] := embed([live, [0,0], [0,0], 0, 0, 0, 0]);

ms: Null[mpi_message] := nil();
mr: Null[mpi_message] := nil();
leader: Nat := 0;
decision: Null[Int] := embed(0: Int);
value: Int := 0;
exitloop : Bool := false;
gotmsg : Bool := true;
waited : Bool := false;
do

% First run leader election protocol
% prep and send alive messages

fire output A_stateralg.systeminitialize;

fire internal A_detector.PrepAliveMessages;

for y: Nat where y < MPI_Size() do

fire internal A_detector.Check( y );

fire output A_detector.InformStopped( y );

fire output A_detector.SEND( ms );

if (ms := nil() ) then print val(ms);
fi
ms := nil();

od

exitloop := false;

while "exitloop do

% gives time for messages to arrive and be responded to
follow A_detector.v duration \infty();
% checks if any messages are ready to be received
for y:Nat where y < MPI_Size() do
    mr := dummy;
    while (mr ≠ nil()) do
        fire input RM.probe(y);
        fire output RM.RECEIVE(mr);
        if (mr ≠ nil()) then print val(mr); fi
    od;
    fire output A_detector.InformAlive(y);
od;
fire output A_leader_elector.Leader(leader);
print leader;

%%%% RUN PAXOS
if (leader = MPI_Rank()) then
    %% LEADER ALGORITHM
    % create a value to vote for and init
    value := choose x;
    fire input A_bpleader.Init(value);
    % gives time for messages to arrive and be responded to
    exitloop := false;
    while ¬exitloop do
        % leader starts a new round
        fire output A_starteralg.NewRound;
        % prep collect messages
        fire internal A_bpleader.Collect;
        % send collect messages
        for y:Nat where y < MPI_Size() do
            fire output A_bpleader.SEND(ms);
            if (ms ≠ nil()) then print val(ms); fi
            ms := nil();
        od
        follow A_starteralg.v, A_bpsuccess.v, A_bpleader.v duration \infty();
        % checks for messages
        for y:Nat where y < MPI_Size() do
            mr := dummy;
            while (mr ≠ nil()) do
                fire input RM.probe(y);
                fire output RM.RECEIVE(mr);
                if (mr ≠ nil()) then print val(mr); fi
            od;
            % gather last messages
            fire internal A_bpleader.GatherLast;
            fire internal A_bpleader.Continue;
            fire output A_bpleader.NextPhase(begincast, exitloop);
            fire internal A_starteralg.CheckRndSuccess;
        od;
        exitloop := false;
        while ¬exitloop do
            % prep begincast messages
            fire output A_bpleader.Begincast;
            % send begincast messages
            for y:Nat where y < MPI_Size() do
                fire output A_bpleader.SEND(ms);
                if (ms ≠ nil()) then print val(ms); fi
                ms := nil();
            od
            % gives time for messages to arrive and be responded to
        follow A_starteralg.v, A_bpsuccess.v, A_bpleader.v duration \infty();
        % collect responses

76
for y: Nat where y < MPI_Size() do
  mr := dummy;
  while (mr = nil()) do
    fire input RM.probe(y);
    fire output RM.RECEIVE(mr);
    if (mr = nil()) then print val(mr);
  od;
  od

% process accept messages
fire internal A_bpleader.GatherAccept;
fire output A_bpleader.NextPhase(decided, exitloop);
fire internal A_starteralg.CheckRndSuccess;

% reached decision
fire output A_bpleader.RndSuccess(decision);
fire internal A_starteralg.CheckRndSuccess;
fire internal A_bpleader.GatherOldRound;
exitloop := false;
while ~exitloop do
  % announce success
  fire internal A_bpsuccess.SendSuccess;
  for y: Nat where y < MPI_Size() do
    fire output A_bpsuccess SEND(ms);
    if (ms = nil()) then print val(ms);
    ms := nil();
  od;
  follow A_starteralg.v, A_bpsuccess.v, A_bpleader.v duration \infty();
for y: Nat where y < MPI_Size() do
  mr := dummy;
  while (mr = nil()) do
    fire input RM.probe(y);
    fire output RM.RECEIVE(mr);
    if (mr = nil()) then print val(mr);
  od;
  od
  fire internal A_bpsuccess.GatherAck;
  fire output A_bpsuccess.HasEnoughAcks(exitloop);
od;
else
  % AGENT ALGORITHM
  exitloop := false;
  while ~exitloop do
    follow A_starteralg.v, A_bpsuccess.v, A_bpagent.v duration \infty();
    % agents collect
    for y: Nat where y < MPI_Size() do
      mr := dummy;
      while (mr = nil()) do
        fire input RM.probe(y);
        fire output RM.RECEIVE(mr);
        if (mr = nil()) then print val(mr);
      od;
    od
    fire internal A_bpagent.LastAccept;
    fire internal A_bpagent.Accept;
    fire internal A_bpsuccess.GatherSuccess;
    % send response
    gotmsg := true;
    while gotmsg do
      fire output A_bpagent SEND(ms);
      if (ms = nil()) then
        print val(ms);
        ms := nil();
      else
        gotmsg := false;
      fi
    od;
  else
    gotmsg := true;
    while gotmsg do
      fire output A_bpagent SEND(ms);
      if (ms = nil()) then
        print val(ms);
        ms := nil();
      else
        gotmsg := false;
      fi
    od
  fi
}
Listing 34: Paxos: the node automaton (paxos.tioa).

D.2 The Starter Algorithm

```tla
automaton starter alg (L: Int ,D: Int ,C: Int ,Z: Int)

signature
  input Stop , Recover , Leader ( r: Nat ) , RndSuccess ( v: Null [ Int ] ) , BeginCast
  internal CheckRndSuccess
  output NewRound , system initialize

states
  clock : AugmentedReal := 0 ;
  status : NodeMode := live ;
  leader : Nat := MPI_Rank ( ) ;
  iamleader : Bool := false ;
  sstart : Bool := false ;
  phase : NodeMode := live ;
  phaseend : AugmentedReal := 0 ;
  deadline : AugmentedReal := 0 ;
  lastnr : AugmentedReal := \infty ( ) ;
  last : AugmentedReal := \infty ( ) ;
  rndsuccess : Bool := false ;

transitions
  output system initialize
  input Stop
  eff
    status := stopped ;

  input Leader ( r )
  eff
    if status = live then
      if ( r = MPI_Rank ( ) ) then
        iamleader := true ;
      else
        iamleader := false ;
      fi
    if ( rndsuccess = false ) then
      deadline := 0 ;
      sstart := true ;
      lastnr := clock + L ;
```
f i
leader := r;
input BeginCast
ef
if (status = live) then
deadline := clock + 3 * L + 2 * MPI_Size() * L + 2 * D;
fi
input RndSuccess(v)
ef
if (status = live \ v = nil()) then
rndsuccess := true;
slast := \infty();
fi
output NewRound
pre
status = live;
iamleader;
sstart;

ef
sstart := false;
lastnr := \infty();
internal CheckRndSuccess
pre
status = live;
iamleader;
deadline = 0;
clock > deadline;

ef
sstart := \infty();
if (rndsuccess = false) then
sstart := true;
lastnr := clock + L;
fi
input Recover
ef
status := live;
trajectories

D.3 The Detector Algorithm

automaton detector(L: Int, D: Int, C: Int, Z: Int)
signature
  internal Check(j: Nat), PrepAliveMessages
  output SEND(m: Null [ mpi_message ]), InformStopped(n: Nat), InformAlive(n: Nat)
  input Stop, Recover, RECEIVE(m: Null [ mpi_message ]), Leader(r: Nat)
  output HasEnough(b: Bool)
  input systeminitialize
states
  leader : Nat := MPI_Rank();
clock : AugmentedReal := 0: AugmentedReal;
status : NodeMode := live;
alive : Array[Nat, AugmentedReal] := constant(true);
prevrec : Array[Nat, AugmentedReal] := constant(0: AugmentedReal);
lastinform : Array[Nat, AugmentedReal] := constant(0: AugmentedReal);
lastsend : Array[Nat, AugmentedReal] := constant(0: AugmentedReal);
lastcheck : Array[Nat, AugmentedReal] := constant(0: AugmentedReal);
outmsgs : Seq[Null [ mpi_message ]] := { };
let
    lb(index, value) : Nat, AugmentedReal \rightarrow AugmentedReal =
    if index <= 0 then
        value
    else
        lb((Nat)(index - 1), \min(value, \min(lastInform[index], \min(lastSend[index], lastCheck[index]))))
transitions
input sysMinitalize
    eff
        prevRec := constant(clock);
        lastInform := constant(clock);
        lastSend := constant(clock);
        lastCheck := constant(clock);
    input Leader(r)
        eff
            leader := r;
    input Stop
        eff
            status := stopped;
    input Recover
        eff
            status := live;
output SEND(m)
    pre
            status = live;
            outmsgs := \{\};
            m = head(outmsgs);
    eff
            lastSend[val(m).destination] := clock + Z;
            outmsgs := tail(outmsgs);
input RECEIVE(m)
    if (status = live /\ m = nil()) then
        if val(m).data.M = live then
            alive[val(m).data.sender] := true;
            lastCheck[val(m).data.sender] := clock + C;
        fi
        prevRec[val(m).data.sender] := clock;
    fi;
internal Check(j)
    pre
        status = live;
        alive[j];
    eff
        lastCheck[j] := clock + C;
        if clock > (prevRec[j] + Z + D) then
            alive[j] := false;
        fi
internal PrepAliveMessages
    locals
        mn: Null[mpi_message];
    pre
        status = live;
    eff
        for n:Nat where n < MPI_Size() do
            mn := embed([\{live, [0.0], [0.0], 0, MPI_Rank(), n\}]);
            outmsgs := outmsgs |- mn;
        od
    output InformStopped(n)
    pre
        status = live;
    eff
        if \not alive[n]) then
            lastInform[n] := clock + L;
        fi
output InformAlive(n)
pre
  status = live;
  if (alive[n] = true) then
    lastInform[n] := clock + L;
  fi
output HasEnough(b)

let
  majority(num) : Nat -> Bool =
    if num >= (Nat)floor((1+MPI_Size())/2) then true else false;
count(i, num) : Nat, Nat -> Bool =
  if (i+1) = MPI_Size() then majority(num)
  else count(succ(i), if clock < prevrec[i] + D + Z then num + 1 else num);

pre
  status = live;
  b = count(0, 0);
trajectories
trajdef v
  invariant status = live;
  stop when clock > lb((Nat)(MPI_Size() - 1), (AugmentedReal)\infty);
evolve d(clock) = 1;

Listing 36: Paxos: the detector automaton (detector.tioa).

D.4 The Leader Election Algorithm

1 automaton leaderlector
signature
  input InformStopped(j: Nat), InformAlive(j: Nat), Stop, Recover
  output Leader(r: Nat)
  input systeminitialize
6 states
  status : NodeMode := live;
  pool : Array[Nat, Bool] := constant(false);
  leader : Nat := MPI_Rank();
transitions
  input systeminitialize
eff
    pool[MPI_Rank()] := true;
  input Stop
eff
  status := stopped;
  output Leader(r)
pre
    status = live;
  r = leader;
16 input InformStopped(j)
llocals
  maxid : Nat := MPI_Rank();
eff
  if status = live then
    pool[j] := false;
    for y:Nat where y < MPI_Size() do
      if (y < maxid) then y := maxid; fi
      if pool[y] then maxid := max(maxid, y); fi
    od
  fi
  input Recover
eff
    status := live;
\textbf{D.5 The Paxos Leader Algorithm}

\begin{verbatim}
input InformAlive(j)
locals
  maxid: Nat := MPI_Rank();
eff
  if status = live then
    pool[j] := true;
    for y: Nat where y < MPI_Size() do
      if pool[y] then
        maxid := \max(maxid, y);
      fi
    od
    leader := maxid;
  fi

Listing 37: Paxos: the leader election automaton (leaderelector.tioa).
\end{verbatim}
m = head(outmsgs);

**Input** Recover

```
status := live;
```

**Input** RECEIVE(m)

```
47 outmsgs := tail(outmsgs);
```

```
52 mval := val(m);
```

```
56 mval := mval.data.M;
```

```
57 if (mval.data.M = last \or mval.data.M = accept \or mval.data.M = success \or mval.data.M = oldround) then
  inmsgs := inmsgs | − mval;
fi
```

```
62 fi
```

**Input** Init(v)

```
67 initvalue := embed(v);
```

**Input** NewRound

```
72 currnd.C := highestnd.C + 1;
currnd.O := MPI_Rank();
highestnd := currnd;
mode := collect;
phaseend := clock + 2 * D + 5 * L;
```

**Internal** Collect

```
locals mn: Null[mpi_message];
pre
status = live;
iاملeader;
mode = collect;
```

```
81 rndvfrom := [0 : Int, MPI_Rank()];
rndinfquo := constant(false);
rndaccquo := constant(false);
for y : Nat where y < len(inmsgs) do
  data := inmsgs[y].data;
  if (data.M = last) then
     outmsgs := outmsgs | − mn;
   fi
od
```

```
87 phaseend := clock + 2 * D + 5 * L;
```

**Internal** GatherLast

```
locals
data: Data;
t_inmsgs : Seq[mpi_message] := {};
rndinfquosize: Int := 1;
```

```
92 status = live;
iاملeader;
mode = gatherlast;
```

```
97 data := inmsgs[y].data;
if (data.M = last) then
   outmsgs := outmsgs | − mn;
fi
```

```
107 for y: Nat where y < len(inmsgs) do
  data := inmsgs[y].data;
  if (data.M = last) then
```
if (data.RP = currnd) then
  rndinfoquos[ data.sender ] := true;
  if ((( rndvfrom . C < data . RP . C ) \ ( rndvfrom . C = data . RP . C \ ( rndvfrom . C < data . RP . C ) ) \ ( data . V = data . RP ) ) \ data . V ) = true
    rndvalue := data . V;
  rndvfrom := data . RP;
fi
if rndinfoquos [ data . sender ] then
  rndinfoquosize := rndinfoquosize + 1;
fi
if rndinfoquosize > ( div( MPI . Size ( ), 2 ) ) then
  if rndvalue = NIL \ initvalue = nil then
    rndvalue := val ( initvalue );
  fi
  if rndvalue ≠ NIL then
    mode := begincast;
    phaseend := clock + 2 * D + 5 * L;
  else
    mode := wait;
  fi
  phaseend := \infty ( );
fi
else
  t . inmsgs := t . inmsgs — inmsgs [ y ];
fi;
odo;
inmsgs := t . inmsgs :
\textbf{internal} Continue
\textbf{pre}
status = live;
iamleader;
mode = wait;
initvalue ≠ nil ( );
\textbf{eff}
if rndvalue = NIL then
  rndvalue := val ( initvalue );
  mode := begincast;
  phaseend := clock + 2 * D + 5 * L;
fi
\textbf{output} BeginCast
\textbf{locals} mn: Null [ mpi . message ];
\textbf{pre}
status = live;
iamleader;
mode = begincast;
\textbf{eff}
for y: Nat where y < MPI . Size ( ) do
  % leader does not run the agent algorithm
  if y ≠ MPI . Rank ( ) then
    mn := embed ( [ [ begin , currnd , currnd , rndvalue , MPI . Rank ( ) ] , y ] );
    outmsgs := outmsgs — mn;
  fi
odo;
mode := gatheraccept;
phaseend := clock + 2 * D + 5 * L;
\textbf{internal} GatherAccept
\textbf{locals}
data : data ;
rndaccquosize : Int := 1;
t . inmsgs : Seq [ mpi . message ] := {};
\textbf{pre}
status = live;
iamleader;
mode = gatheraccept;
\textbf{eff}
84
\begin{verbatim}
for y: Nat where y < len(inmsgs) do
    data := inmsgs[y].data;
    if (data.M = accept) then
        if data.R = currnd then
            rndaccquo[data.sender] := true;
        fi
    fi
    if rndaccquo[data.sender] then
        rndaccquosize := rndaccquosize + 1;
    fi
    if rndaccquosize > (div(MPI_Size(), 2)) then
        decision := embed(rndvalue);
        mode := decided;
        phaseend := clock + 2 * D + 5 * L;
    fi
    else
        t.inmsgs := t.inmsgs |− inmsgs[y];
    fi;
od;
output RndSuccess(v)

pre
    status = live;
    iamleader;
    mode = decided;
    v = decision;

 eff
    mode := rnddone;
    phaseend := clock + 2 * D + 5 * L;

internal GatherOldRound

locals
    data : Data;
    t.inmsgs : Seq[mpi_message] := { };

pre
    status = live;

 eff
    for y: Nat where y < len(inmsgs) do
        data := inmsgs[y].data;
        if ((data.R.C < currnd.C) /
            (data.R.C = currnd.C) /
            (data.R.O < currnd.O)) /
            (data.M = oldround)) then
            highestrnd := data.RP;
        else
            t.inmsgs := t.inmsgs |− inmsgs[y];
        fi
    od;

output NextPhase(m, b)

let test2(m) : NodeMode -> Bool = if m = mode then true else false;

pre
    b = test2(m);

trajectories
trajdef v

invariant status = live;
stop when clock >= phaseend;

 evolve d(clock) = 1;
\end{verbatim}

Listing 38: Paxos: the leader automaton (bpleader.tioa).

\section{The Paxos Agent Algorithm}

\texttt{automaton bagent(L: Int, D: Int, C: Int, Z: Int)}
signature
  input RECEIVE(m: Null[mpi_message]), Init(v: Int), Stop, Recover
internal LastAccept, Accept
output SEND(m: Null[mpi_message])

states
  status : NodeMode := live;
  last : Round := [0: Int, 0: Int];
  lastv : Int := 0;
  commit : Round := [0: Int, 0: Int];
  inmsgs : Seq[mpi_message] := {};
  outmsgs : Seq[Null[mpi_message]] := {};
  clock : AugmentedReal := 0: AugmentedReal;
  phaseend : AugmentedReal := 0: AugmentedReal;

transitions
input Stop
  eff
    status := stopped;
output SEND(m)
pre
    status = live;
    outmsgs := {};
    m = head(outmsgs);
  eff
    outmsgs := tail(outmsgs);
internal LastAccept
locals
  data : Data;
  mn : Null[mpi_message];
t_inmsgs : Seq[mpi_message] := {};
pre
    status = live;
  eff
    for y: Nat where y < len(inmsgs) do
      data := inmsgs[y].data;
      if (data.M = collect) then
        if (commit.C < data.R.C \ (commit.C = data.R.C \ commit.O < data.R.O)) then
          commit := data.R;
          mn := embed([last, last, data.R, lastv, MPI:Rank(), data.sender]);
          outmsgs := outmsgs \ mn;
          phaseend := clock + 2 * D + 5 * L;
        else
          mn := embed([oldround, commit, data.R, lastv, MPI:Rank(), data.sender]);
          outmsgs := outmsgs \ mn;
          phaseend := clock + D + 5 * L;
        fi
      else
        t_inmsgs := t_inmsgs \ inmsgs[y];
      fi
    od;
  inmsgs := t_inmsgs;
input Recover
  eff
    status := live;
input RECEIVE(m)
  eff
    if (status = live) then
      if (m = nil()) then
        if (val(m).data.M = collect \ val(m).data.M = begin) then
          inmsgs := inmsgs \ val(m);
        fi
      fi
    fi
internal Accept
locals
  data : Data;
Listing 39: Paxos: the agent automaton (bpagent.tia).

D.7 The Paxos Success Algorithm

```plaintext
automaton bpsuccess(L: Int ,D: Int ,C: Int ,Z: Int )
signature
  input RECEIVE(m: Null [ mpi_message ]) , Stop , Recover , Leader(r: Nat) , RndSuccess(v: Null [ Int ])
  internal SendSuccess , GatherSuccess , GatherAck , Wait
  output NextPhase(b: Boolean) , Decide(v: Null [ Int ]), SEND(m: Null [ mpi_message ])
states
  clock : AugmentedReal := 0;
  status : NodeMode := live;
  decision : Int := 0;
  leader : Nat := MPI.Rank();
  iamleader : Boolean := false;
  acked : Array[ Nat , Boolean ] := constant( false );
  prevsend : AugmentedReal := 0;
  lastsend : AugmentedReal := \infty();
  lastwait : AugmentedReal := \infty();
  lastg : AugmentedReal := \infty();
  lastss : AugmentedReal := \infty();
  inmsgs : Seq[ mpi_message ] := { };
transitions
  input Stop
  eff
```
status := stopped;
input Leader(r) eff
  if status = live then
    if (r = MPI_Rank()) then
      iamleader := true;
    else
      iamleader := false;
  fi
leader := r;
output SEND(m) where len(outmsgs) = 0 pre
  status = live;
  m = nil();
output SEND(m) where len(outmsgs) ^= 0 pre
  status = live;
  m = head(outmsgs);
output RndSuccess(v) eff
  if status = live /\ v ^= nil() then
    decision := val(v);
    lastss := clock + L;
  fi
input Recover eff
  status := live;
input RECEIVE(m) locals
  mval: mpi_message;
  if status = live /\ m ^= nil() then
    mval := val(m);
    if (mval.data.M = ack /\ mval.data.M = success) then
      inmsgs := inmsgs | − mval;
    fi
    if mval.data.M = ack /\ lastga = \infty() then
      lastga := clock + L;
    fi
    if mval.data.M = success /\ lastgs = \infty() then
      lastgs := clock + L;
  fi
internal SendSuccess locals
  mn: Null[mpi_message];
  if status = live;
  iamleader = true;
  decision ^= 0;
  prevsend = 0;
for y: Nat where y < MPI_Size() do
  if ^acked[y] /\ y ^= MPI_Rank() then
    mn := embed([success, [0: Int, 0: Int]: Round, [0: Int, 0: Int]: Round, decision, MPI_Rank()]:Data, y):mpi_message
  outmsgs := outmsgs | − mn;
  od
prevsend := clock;
lastsend := clock + L;
lastwait := clock + (4 * L + 2 * MPI_Size() * L + 2 * D) + L;
lastss := \infy();

internal GatherSuccess

locals
data : Data;

i : Seq[mpi_message] := \{ \};

pre
status = live;

for y : Nat where y < len(inmsgs) do
  data := inmsgs[y].data;
  if data.M = success then
data := data.V;
  mn := embed([ack, [0: Int, 0: Int]: Round, [0: Int, 0: Int]: Round, decision, MPI_Rank()]: Data, data.send);

outmsgs := outmsgs |− mn;
else
  t_inmsgs := t_inmsgs |− inmsgs[y];

fi

inmsgs := t_inmsgs;

output Decide(v)

pre
status = live;
decision ^= 0;
v = embed(decision);

internal GatherAck

locals
data : Data;

i : Seq[mpi_message] := \{ \};

pre
status = live;

for y : Nat where y < len(inmsgs) do
  data := inmsgs[y].data;
if (data.M = ack) then
  acked[y] := true;
  foundit := true;
else
  t_inmsgs := t_inmsgs |− inmsgs[y];

fi

inmsgs := t_inmsgs;

if !foundit then
  lastg := \infy();
else
  lastg := clock + L;

fi

output HasEnoughAcks(b)

let majority(num) : Nat → Bool =
  if num >= (Nat)floor((1 + MPI_Size())/2) then true else false;

count(i, num) : Nat, Nat → Bool =
  if (i+1) = MPI_Size() then majority (num)
else count(succ(i), if acked[i] then num + 1 else num);

pre
status = live;
b := count(0, 0);

internal Wait

pre
status = live;
present ^= 0;
clock > present + (4 * L + 2 * MPI_Size() * L + 2 * D + L);
D.8 The Vocabulary for Paxos

%%:Paxos vocabs:.
vocabulary paxos_voc
types NodeMode : Enumeration [live, stopped, begin, last, accept, success, oldround, collect, gatherlast, wait, begincast, gatheraccept, decided, rnddone, ack],
Round : Tuple [C: Int, O: Int],
end

%%:MPI Channel vocabs:.
vocabulary mpi_status_voc
types mpi_status
operators
MPI_Iprobe : Nat -> Null[mpi_status],
MPI_Test : mpi_status -> Bool
end

vocabulary mpi_message_voc
imports paxos_voc, mpi_status_voc
types mpi_message : Tuple [data: Data, destination: Nat]
operators
MPI_Irecv : mpi_status, Nat -> mpi_message
end

vocabulary mpi_request_voc
imports mpi_message_voc
types mpi_request
operators
MPI_Isend : mpi_message, Nat -> Null[mpi_request],
MPI_BARRIER : -> Bool
end

vocabulary mpi_voc
operators
MPI_Rank : -> Nat,
MPI_Size : -> Nat
end

vocabulary Mode
types Mode : Enumeration [done, working, leader, notleader]
end

Listing 41: Paxos: algorithm vocabulary (myvocabs.tioa).

D.9 MPI Send Mediator Automaton
automaton SendMediator
signature
input SEND(m: Null[mpi_message])
states
status : Array[Nat, Null[mpi_request]] := constant(nil());
clock : AugmentedReal := 0;
transitions
input SEND(m)
eff
  if (m = nil()) then
    status[val(m).destination] := MPI_Isend(val(m), val(m).destination);
  fi
trajectories
trajdef DELAY evolve d(clock) = 1;
Listing 42: Paxos: the sender automaton (SendMediator.tioa).

D.10 MPI Receive Mediator Automaton

automaton ReceiveMediator
signature
  output RECEIVE(m: Null[mpi_message])
  input probe(s: Nat)
states
toRecv : Seq[mpi_message] := {};
transitions
  output RECEIVE(m) where len(toRecv) = 0
  pre
    m = nil();
  output RECEIVE(m) where len(toRecv) ^= 0
  pre
    m = embed(head(toRecv));
eff
toRecv := tail(toRecv);
input probe(s)
locals
  status : Null[mpi_status] := nil();
eff
  status := MPI_Iprobe(s);
  if (status = nil()) then
    if (MPI_Test(val(status))) then
      toRecv := toRecv |− MPI_Irecv(val(status), s);
    fi
  fi
Listing 43: Paxos: the receive automaton (ReceiveMediator.tioa).