

ABSTRACT

The term automated implementation of distributed algorithms refers to the process during which an algorithm written in Input/Output Automata (IOA) and/or Timed IOA is automatically translated to executable code. The Tempo toolkit provides a connection with the IOA compiler and using particular plug-ins makes the above feasible. In that way, simple algorithms such as LCR Leader Election or even more complex such as Lamport's Paxos algorithm for the consensus problem can be translated very easily from TIOA to Java executable code using the toolkit, with the generated code preserving and keeping the correctness of the specification. The mediator between IOA compiler and Tempo is the Eclipse environment, which under the certain configuration and setup, is used for the desired translation.

So far, the generated code by the toolkit was able to run on several workstations that were communicating via the Message Passing Interface. Even though MPI is very powerful, it has certain limitations and restrictions such as that is suitable for WANs, nor it supports dynamic ad hoc connections. Hence, there was an urgency of finding a mechanism to overcome those limitations and enhance the toolkit with more capabilities.

In this Thesis we enhance the Tempo toolkit to support Java TCP connections between the communicating nodes. For this purpose, we created several classes imitating the behavior of the .Net package of Java. For certain reasons, which will be mentioned later, classes such as Socket and ServerSocket could not be used directly and hence we ended up creating our own classes.

To provide evidence of the correctness of our implementation we firstly tested the model using a very simple algorithm that uses two communicating nodes. The first machine, A, was a Sender and the other one, B, was the Receiver, managing successfully to send a message from A to B and print that in B. Then, we tested our implementation with the well known Paxos algorithm. The algorithm was translated and run correctly suggesting that our implementation does what it should; this is a strong indication that Tempo has been successfully enhanced to support Java/TCP Sockets.

**ENCHANCING THE TEMPO COMPILER TO SUPPORT
JAVA-SOCKETS/TCP-BASED COMMUNICATION**

Christos C. Ploutarchou

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

June, 2011

APPROVAL PAGE

Master of Science Thesis

ENHANCING THE TEMPO COMPILER TO SUPPORT JAVA-SOCKETS/TCP-BASED COMMUNICATION

Presented by

Christos C. Ploutarchou

Research Supervisor

Chryssis Georgiou

Committee Member

Anna Philippou

Committee Member

Georgia Kapitsaki

University of Cyprus

June, 2011

ACKNOWLEDGEMENTS

First of all I am heartily thankful to my project advisor Dr. Chryssis Georgiou for trusting me from the very beginning and giving me the opportunity to work on this project. He was the one who gave me the motivation to expand my knowledge in distributed systems after attending his postgraduate course. Moreover, he was always willing to assist me and encourage me, whenever his help was needed.

Furthermore, I will always be grateful to Dr. Peter M. Musial who was also supportive of me during all this period, spending many hours emailing and chatting with me in an attempt to successfully complete the project. Because of him I managed to overcome many difficulties starting from the Eclipse configuration and ending with the successful completion of the project.

Finally, I offer my regards and blessings to my fiancé and my family for supporting me all these years in every possible way. Without their patience and love I would have never been able to complete this thesis.

TABLE OF CONTENTS

Chapter 1 Introduction.....	1
1.1 Motivation.....	1
1.2 Contribution	3
1.3 Document Structure	3
Chapter 2 Background and Related Work.....	4
2.1 Input/Output Automata	5
2.2 Timed Input/Output Automata.....	8
2.3 Tempo toolkit.....	12
2.4 Message Passing Interface	13
2.5 Java TCP Sockets.....	15
2.6 IOA Compiler	17
2.7 The Consensus Problem and the Paxos Algorithm.....	20
2.8 Other Formal Methods	22
Chapter 3 Enhancing Tempo with TCP/Java Sockets.....	25
3.1 Tempo Language.....	26
3.1.1 <i>Booleans</i>	26
3.1.2 <i>Natural Numbers</i>	27
3.1.3 <i>Integers</i>	27
3.1.4 <i>Characters</i>	28
3.1.5 <i>Extensions by nil</i>	28
3.2 Java TCP Sockets Integration	29
3.2.1 <i>JVMError</i>	34
3.2.2 <i>JVMServerSocket</i>	34

3.2.3	<i>JVMSocket</i>	36
3.2.4	<i>JVMStream</i>	37
3.2.5	<i>TCPNodeVoc</i>	37
3.3	Problems Faced and How They Were Solved.....	38
Chapter 4 Proof-of-concept Implementation.....		41
4.1	A First Implementation of an Algorithm Using a TCP Channel.....	42
4.2	An Implementation of Paxos Algorithm using TCP Sockets.....	45
4.3	Automated translation procedure	50
Chapter 5 Conclusions and Future Work.....		51
5.1	Conclusions.....	51
5.2	Future work.....	52
Bibliography		54
APPENDIX A		57
APPENDIX B		62
APPENDIX C		76

LIST OF TABLES

Table 1. Java Client-Server Behavior

Table 2. Boolean supported notations

Table 3. Boolean supported operators

Table 4. Natural Number supported operators

Table 5. Integer additional supported operators

Table 6. Characters supported operators

Table 7. Nil supported notations

Table 8. Data types translation mapping

LIST OF FIGURES

Figure 1. A communication channel modeled as a Timed I/O Automaton

Figure 2. TIOA description of Alarm Component

Figure 3. Invariant

Figure 4. Hello World Program using MPI

Figure 5. Auxiliary automata mediate between MPI and algorithm automata to yield a reliable FIFO channel

Figure 6. Node automata

Figure 7. Java TCP Sockets import statements

Figure 8. Java TCP Sockets Input parameters

Figure 9. Java TCP Sockets data types declaration

Figure 10. Specifying the communication type in Tempo

Figure 11. Part of JVMChanTest.tioa's schedule

Figure 12. Schedule's automated translated code

Chapter 1

Introduction

1.1 Motivation

Nowadays, the need for direct communication from a point to another is more urgent than ever before. As time goes by newer and more complex systems are constantly implemented, aiming towards the same goal, the distribution of information around the world. The field of distributed systems is nowadays more dominating than ever before with distributed systems being all around us. A *distributed system* is a collection of independent computers that appear to the users of the system as a single coherent system. [1]. People are continuously interacting with such systems in their daily routine without even noticing. Such systems may include online airline reservation systems, telephone networks and of course, the world wide web.

For a system to be distributed, it means that one or more distributed algorithms are concurrently running on different machines of the system. Because of their nature, their scale and complexity, both distributed systems and distributed algorithms are difficult to comprehend. Several formal methods, have been implemented by researchers in an attempt to understand, analyze and implement such algorithms. Those methods include process algebras [2] and Input/Output automata [3].

Even if one would rigorously specify and verify distributed systems and algorithms, still it would need to write code and implement them from scratch. This process could jeopardize the correctness of the implementation. The IOA compiler [4], is a concrete tool supporting algorithm design, development, testing, and formal verification using automated tools. Through the compiler, programmers are allowed to specify an algorithm in an IOA form, ensuring that way that all its characteristics will be preserved and then by using the toolkit this algorithm is *automatically* translated into Java executable code.

The compiler has been used widely for modeling and automatically implementing many distributed algorithms during the past years. [5, 6, 7] All these implementations share one common characteristic: they use the Message Passing Interface as a communication protocol, as this was the only communicating mechanism supported by the toolkit.

Tempo is a formal language for modeling distributed systems as collections of interacting state machines called Timed Input/Output automata [8]. It was created by VeroModo [9] providing modeling and machine-checked proofs for distributed algorithms. Moreover, it provides a connection with the IOA compiler and hence timing is also taken into consideration when modeling distributed algorithms.

Even though MPI [10] is really important in parallel computing, it has however some limitations, concerning dynamicity and scalability. All participating nodes should be defined in advance and no new node can join at a later stage. Moreover, MPI can only be implemented

in LANs. Therefore, we should use a different approach aiming to overcome the above limitations; one approach is to use of Java TCP Sockets [11].

1.2 Contribution

In this Thesis an enhancement of the Tempo Compiler is presented that supports Java TCP Sockets [11]. What makes this very important is that in this way we offer a considerable extension to the toolkit thus eliminating and overcoming many of the MPI's limitations.

An automated implementation of an algorithm using Java TCP Sockets supports dynamic creation and tearing down of communication links between participating network nodes [12]. Moreover, Java Sockets provide the ability for a global execution of the algorithm. All the above suggest that our enhancement advances significantly the usability and importance of the Tempo Compiler.

1.3 Document Structure

The rest of this document is organized as follows. In Chapter 2, essential concepts and ideas are presented to ease comprehension. Moreover, previous work is presented. Chapter 3 is making a much deeper reference to the Tempo Compiler that has been used for the implementation, starting from some basic information, and ending with how the integration of Java TCP Sockets into it was achieved. Chapter 4 makes a reference in two implementation examples that suggest that our implementation is correct. We conclude in Chapter 5.

Chapter 2

Background and Related Work

We begin by first describing the Input/Output Automata framework. All presented algorithms are specified within this framework. Then, we present an extension of the model, the Timed IOA, which as its name suggests imports time into the model. The IOA Compiler, presented next, it is embedded in the Tempo toolkit is the mechanism which makes possible the automated implementation of complex algorithms. Later on, we present the MPI and Java TCP Sockets communication mediums. Finally, we briefly compare the IOA and Process algebra frameworks, and we overview the Paxos algorithm, which is one of the algorithms we implemented.

2.1 Input/Output Automata

The appearance of Input / Output Automata, or simply IOA, dates back to 1988 and it was introduced by Nancy A. Lynch and Mark R. Tuttle [13]. This model, which may be considered as an improvement of Communicating Sequential Processes (CSP) [14] it was partially based on Dijkstra's "guarded commands" and was intended to be used in modelling concurrent and distributed discrete event systems which in those days was a newly appearing field in Computer Science. Such systems can be used in modeling network resource allocation algorithms, communication algorithms, database systems, as well as in shared atomic objects and dataflow architectures. More specifically, the model performs better when it is used in systems whose components operate asynchronously. Systems with the characteristics described above, continuously receive input from and react to their environment.

For a system to be modeled using IOA, all its consisting components have to somehow be transformed as a separate I/O automaton using a unique language which as it has already been mentioned above, is very similar to the Dijkstra's "guarded commands". What has to be specified first defining an IOA is a suitable name and a list of optional input parameters. For example if we were defining an automaton taking two natural numbers as input parameters, the declaration would look as follows:

automaton A(i, j: Nat)

After defining the name and parameters of the automaton, it is necessary to list the set of its actions which are classified either as *input*, *output*, or *internal* and can be thought as a connection between the automaton and the external environment.

This set of actions is called action *Signature*, or S , and is a partition of the actions set, $act(S)$, which is divided into $in(S)$, $out(S)$, and $int(S)$ for each one of the classifications mentioned above. Input actions are those actions which are generated by the external environment and are transmitted to the automaton, with this transmission being instantaneous.

On the other hand, internal and output actions are generated autonomously by the automaton and the result is transmitted to the environment. Another distinction between internal and the rest of actions is that no restrictions can be established on them whereas output and internal actions can be blocked or restricted using several *preconditions*. That way the automaton has the ability to handle both "bad" and "good" input exhibiting the appropriate behavior each time. Hence its correct behavior depends on the nature and type of the input. The union of input and output actions forms the set of external actions, $ext(S)$, containing those actions which are visible to the external environment. When having a system in IOA with no input actions we refer to it as a "closed" system.

What has to be specified next to fully describe an automaton A , is a set of state variables, Av , which can be thought like system variables that are visible only to the automaton. Moreover, we need a set of states As which is a subset of all possible assigns to the state variables of the automaton, a nonempty set containing the start states, $start(A)$, a transition relation specifying what will happen when a particular event is fired, $step(A)$, and an equivalence relation, $part(A)$, which it is used to identify the primitive components of the system being modeled by the automaton [4]. A *step* is defined as a tuple of three components (s', π, s) where an action π is enabled in state s' and the fire of that action leads to a new state s . *Transitions* are usually accompanied with preconditions defining under which conditions the transition can be enabled. If no preconditions exist means that the transition will always be enabled. Finally, is important to mention that input actions are always enabled.

For better comprehension of the above, consider an example where there are two nodes (A, B) that are communicating through a common channel C, and A wants to send a message to B. The signature and one of the transitions of this model can be expressed in IOA as below:

signature**input** send(m: Message, j,j:Nat)**output** recv(m: Message, j,i:Nat)**transitions****input** send(m, i, j)

An *execution* is a finite or infinite sequence of alternating actions (input, internal, output) and states and it represents a computation of the system. The set containing all automaton's executions is denoted by $execs(A)$. In particular, an execution is of the form $S_0, \pi_1, S_1, \pi_2, \dots$ (where S is a state and π is an action) and leads to what we call a fair execution. More precisely, for an IOA to solve a problem P , the set of its fair, behaviors, that is the set of executions that lead to a solution of the problem, should be a subset of P . Since the automaton cannot block the input actions, this subset cannot be empty. As we mentioned earlier, an execution is composed both from actions and states. If we ignore the states and focus only to the actions we then have an automaton's *schedule* where the set containing all schedules is denoted as $scheds(A)$. More precisely, β is a schedule of an automaton A if β is the schedule of an execution of A [13]. A schedule might look as follow:

schedule**states**

%Definition of all required states

do

%Action's executions

fire output A;**fire** input B;**od**

The term *fire* means that an action can be executed and hence the automaton will proceed after the execution to a new state. Another powerful operation supported by the model is the *composition*. Different IOA can be used to compose other IOA enhancing that way the behavior and capabilities of the model since simple automata can result to more complex ones. One of the key ideas of the composition is that if an action π appears as output action in

one automaton, then that action has to be input action in all the rest. The result of the output action is transmitted to all other automata which they have that action as input and behave accordingly. In that way we establish synchronization between the automata or in other words we define a way of communication between them. Two automata A, B, can only form a composition if their internal and external actions are unique. That means:

$$\text{int}(A) \cap \text{acts}(B) = 0, \quad \text{int}(B) \cap \text{acts}(A) = 0, \quad \text{out}(A) \cap \text{out}(B) = 0.$$

Finally, abstraction mapping at different levels is also allowed in the model and that aids in correctness proofs of algorithms. That is, if we have a problem A that solves a particular problem P, and B is an image of A, then B also solves B. Additionally, the model is *non-deterministic*, which results in having many different executions of an algorithm, since many actions may be enabled at any given time.

2.2 Timed Input/Output Automata

IOA model is used to express distributed algorithms but it lacks in terms of timing issues. Timed Input/Output Automata [3], or just TIOA is an extension of the IOA model and it is addressed to systems which their correctness and performance is highly correlated with timing events such as real-time operating systems.

Such systems usually exhibit very complex behaviors and it is therefore needed to have a framework that is able to model them adequately. A system in TIOA is expressed in a similar way as in IOA meaning that it is translated to a nondeterministic state machine with possibly infinite-states. Like IOA, TIOA is expressed by firstly defining a set of *state variables* that are only visible within the automaton and not to the external environment. The values of those variables affect the state of the automaton at any time and therefore the set containing all possible states can be thought as a subset of all possible valuation of state variables.

Moreover, another set contains all starting states which is a subset of all states. Actions in TIOA are divided into *external* and *internal (or hidden)* where external action contains the union of input and output and internal those actions that are visible only within the automaton.

A fundamental difference between IOA and TIOA is that the state of the latter does not only depend on discrete transitions but on *trajectories* as well, which are either continuous or discontinuous functions enclosed in a left-closed time interval and describe how the values of state variables are changing and affected within specific intervals of time. Based on that, TIOA supports both *static* and *dynamic* variable types. The static type simply describes the set of values that the variable may take on. The dynamic type, on the other hand, describes the acceptable ways in which a variable may evolve [3] and they are usually used for constraining the values that the variable may take during trajectories.

More specifically, a variable might have a specific static data type and a dynamic type which will be equal to a set of values of a specific function. Figure 1 illustrates how a simple communication channel could be modeled in the form of a Timed I/O Automaton. In this example, a sender appends a message m into the channel by firing the input $send(m)$ action and the receiver gets the message from the channel by using the output action $receive(m)$. In this example we do not take into consideration other factors such as state variables, trajectories or whatever else is needed for a TIOA to be fully defined.

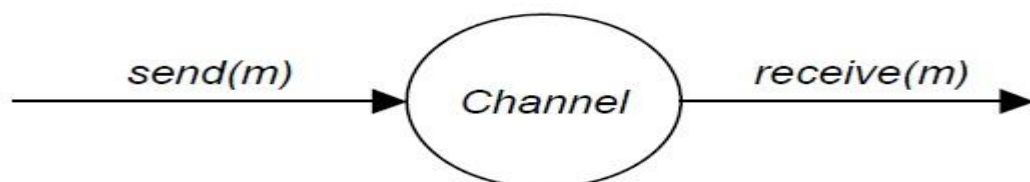


Figure 1: A communication channel modeled as a Timed I/O Automaton [7]

Since TIOA can be used for model checking we need a mechanism that will allow us to check whether specific properties are satisfied by the algorithm that is expressed using TIOA.

This mechanism offered by TIOA is called *invariant*, and it is a property which is true in every reachable state, starting from an initial one. Further details about the idea of invariant as well as an example will be provided in section 2.3.

Going back to trajectories, and in order to define such a function, we need to combine together algebraic and differential equations as well as stopping conditions. First of all, for each trajectory we are going to use, we need to specify a name. Secondly, we have the option to specify a list of formal parameters and impose restrictions on their possible values by making use of the *where* clause. Moreover, we can specify, if needed, functions definitions, stopping conditions, evolve conditions and invariants which are used to check whether specific properties are satisfied by an algorithm, such as mutual exclusion.

If a trajectory T satisfies the stopping conditions of the automaton A , then we can say that T belongs to set of trajectories of A . A trajectory begins with the *evolve* clause and terminates with the *stop when* clause. That way, the automaton is not allowed to continue its execution after a specific value of time. It is also very important to mention that when several TIOAs are composed, then the trajectory of any of them may be interrupted by a discrete function of one of the other composed automata.

Since the main difference between IOA and TIOA is the presence of trajectories, what needs to be added in the definition of a system in the form of a timed input/output automaton is a set T which will be a subset of all trajectories, $T \subseteq \text{trajs}(Q)$.

The following Figure represents how an Alarm could be defined using the TIOA model.

```

let legalTime(hour, minute: Nat) = minute < 60  $\wedge$  hour < 24

automaton Alarm
  signature
    input showTime(hour, minute: Nat) where legalTime(hour, minute),
      setAlarm(hour, minute: Nat) where legalTime(hour, minute),
      toggleAlarm
    output ring
  states
    alarmTime: Nat := 0,
    turnedOn: Bool := false,
    ringNow: Bool := false
  transitions
    input setAlarm(hour, minute)
      eff alarmTime := (60*hour) + minute
    input showTime(hour, minute)
      eff ringNow := turnedOn  $\wedge$  alarmTime = (60*hour) + minute
    input toggleAlarm
      eff turnedOn :=  $\neg$ turnedOn
    output ring
      pre ringNow
      eff ringNow := false

```

Figure 2: TIOA description of Alarm Component [3]

This automaton has three input actions and one output. Both the two input actions are parameterized with two natural numbers that are used to display the time and set an alarm time respectively. The `let` statement is used to define a predicate *legalTime* used to constraint the values of these action parameters. Moreover, the automaton has three state variables of type natural with initial value 0, and boolean initialized as false, used for representing the time, whether the automaton is turned on or off, and whether the alarm should be ringing. The values of these state variables can only change by the occurrence of a discrete transition.

There are no preconditions for the input actions, meaning that they are always enabled and ready to be fired. The effect of the first input action, *setAlarm*, is to set the *alarmTime* to the time which the alarm should ring. *showTime* effect is to set the *ringNow* to true, if the alarm is

on and should ring. Finally, *toggleAlarm* results in turning on, or off the alarm depending on its current state. Finally, the *ring* action can only occur if the alarm is enabled and the *ringNow* state variable is true.

Even though the above example is a simple one, TIOA can be used to express algorithms with any level of complexity. This is because its language supports many statements including assignments of the form $:=$, conditional like if $x < y$ then, and for loops. Furthermore, the model is fitted with a large number of primitive types like *Bool*, *Nat*, *Int*, *Real*, *AugmentedReal*, *Char*, *String* and many others. In addition, other data types can be defined by the user by creating what we call a *vocabulary*.

2.3 Tempo toolkit

Tempo [8] is an implementation of Timed Input/Output Automata created by VeroModo Inc [9] providing computer aid for describing and checking properties of distributed algorithms using several tools such like PVS and UPPAAL. What makes Tempo very powerful and fully compatible with TIOA is that they use almost the same language and syntax. Hence any algorithm defined in TIOA can be very easily analyzed and validated using the tools that come along with Tempo.

What is needed when analyzing an algorithm is first to have a way of checking that it is syntactically and semantically written correctly and that is why Tempo uses a checker. Secondly, we need to know if the algorithm runs correctly and gives the results it was supposed to. For that purpose, Tempo recommends its simulator.

As in TIOA, Tempo uses *vocabularies* in order to declare data types that are going to be used by the algorithm and can be imported later in the main automata by using the *imports*

clause. States, Actions and Transitions are specified exactly like in TIOA. Moreover, invariants in Tempo are checked by either PVS, UPPAAL, or by running simulations of the algorithm and observing its execution behavior. An invariant checking that no more than one process will be in its critical section at any time, is defined as shown in the following figure.

invariant of fischer:
 $\forall i: process \ \forall j: process$
 $(i \neq j \Rightarrow (pc[i] \neq pc_crit \vee pc[j] \neq pc_crit));$

Figure 3: Invariant [8]

In the previous sections we explained that an execution of a TIOA is an alternating sequence of actions and states. Furthermore, it is worth mentioning that Tempo supports *Simulations*, meaning that by the use of a schedule and a loop, an automaton may run several times with different parameter values.

For all the above, Tempo is enriched with a very user friendly interface which is implemented on the Eclipse Rich Client Platform [15] and can be used for easier TIOA writing and checking but also for step-by-step debugging with the usage of breakpoints.

2.4 Message Passing Interface

One of the most important aspects when running parallel algorithms is the way in which communication between the participating processes is established. We want a mechanism that will be reliable, efficient and robust ensuring that almost all messages will be delivered within a reasonable time period and without the occurrence of drops of connections.

The Message Passing Interface (MPI) [10] provide us with several predefined methods that allows process communications via message exchange, in the sense that one processor

sends a message and another processor receives it. MPI dates back to 1993 and has nowadays more than 40 different organizations participating in its forum, including IBM. Even though it includes several of ready-made functions, MPI is considered to be a specification rather than a library. There are many reasons why MPI has become a standard [10] and some of them are listed below.

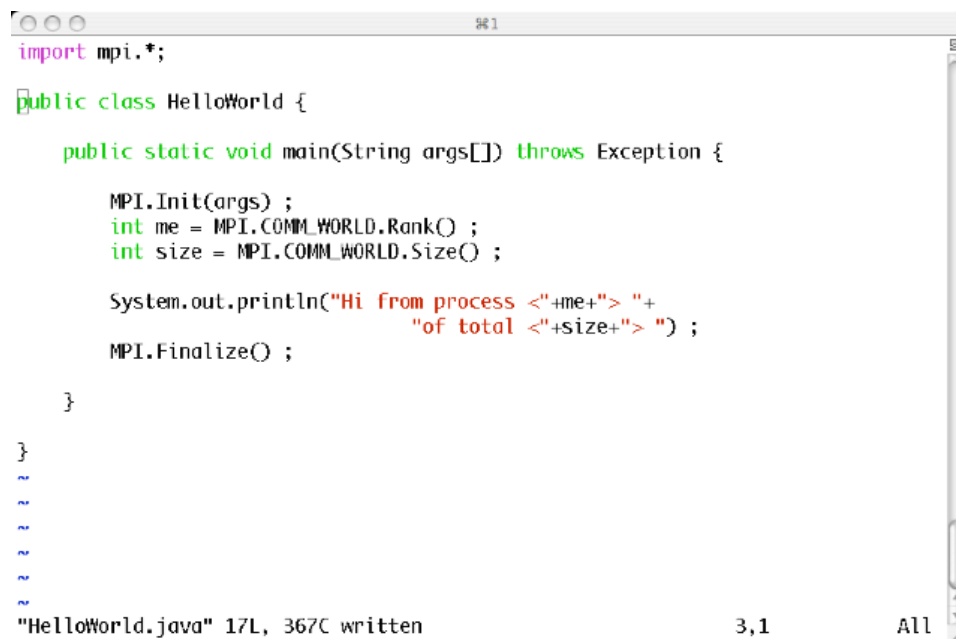
First of all, MPI is supported by almost all platforms and that makes it a kind of a standard. Moreover, it allows portability of code since a program written with MPI standard in a specific platform can be very easily migrated to a completely different platform. Furthermore, performance and functionality are the other two factors that aid in the wide use of the interface. MPI is enriched with more than 115 routines available for use without the need for any modification. In the following paragraphs we give a brief description of how a program could be written using MPI for process communication.

What a programmer has to do first for writing an MPI compatible program, is to download the MPJ library that is freely available from the internet and import it to the header declarations of the source code. After that, the *MPI.Init(args)* statement has to be fired and that initializes the MPI environment. This method is called in the program once and from that point and on, the programmer starts writing parallel code. Finally, *MPI.Finalize()* identifies that the MPI environment terminates and therefore no other MPI routines can be called or parallel code can be written.

A communication in MPI might be either point-to-point or collective, including broadcasting, all-to-all, and other. Point-to-point communication refers to the case where a communication is between only two processes with one sending a message and the other receiving it. A message in MPI may be either of specific data type supported by the interface, like float, integer and double, or it may be an object. At any case, the communication could be Blocking or Non-blocking. What distinguishes these two is that in the first case, when process

sends or receives a message, using *Send()* or *Recv()*, the methods do not terminate until the message has physically been sent or received. On the other hand, non-blocking uses *Isend()* or *Irecv()* and these terminate immediately. If there is a need to wait, other methods can be used, such as *Test()* or *Wait()*.

Operations like knowing the current number of participating processes or getting the unique identifier, or "task ID" of any communicating parties at each time, can be done by simply using the *MPI.COMM_WORLD.Size()* and *MPI.COMM_WORLD.Rank()* respectively. As already mentioned, many other methods exist and can be found on the net. The following figure illustrates how the well known "Hello World" program can be written in MPI.



```

import mpi.*;

public class HelloWorld {

    public static void main(String args[] throws Exception {

        MPI.Init(args) ;
        int me = MPI.COMM_WORLD.Rank() ;
        int size = MPI.COMM_WORLD.Size() ;

        System.out.println("Hi from process <"+me+"> "+
                           "of total <"+size+"> ") ;
        MPI.Finalize() ;

    }

}
~
~
~
~
~
~
"HelloWorld.java" 17L, 367C written          3,1          All

```

Figure 4: Hello World Program using MPI [16]

2.5 Java TCP Sockets

As time goes by, new technologies and methods arise. Therefore, even though MPI is adequate for process communications for all the reasons mentioned in the previous sections, it nevertheless has some limitations when we are talking about distributed and parallel

computing. A major limitation of MPI is that it is suitable for use in a local area network (LAN) rather than in a WAN. Furthermore, in MPI the number of participating nodes should be defined in advanced and hence an additional node cannot be added later on.

It is therefore obvious that an alternative candidate for process communication should exist. In this thesis, we enhanced Tempo Toolkit to support Java TCP Sockets [11] making it more powerful since many of the MPI limitations can be eliminated. In the following paragraphs the client-server model will be presented in order to provide a basic understanding.

As it is implied by the name of the model, for two nodes to communicate, a Transport Control Protocol socket should be used ensuring that way the presence of a reliable point-to-point connection-oriented communication channel. This model is also referred as Client-Server since one node is acting like a serving machine and the other as a client making requests to the server. A socket, which is a combination of an IP address and a port bounded to that address, provides a bi-directional link between two entities enabling them to read from it and to write on it. The Java language provides the `java.net` package which makes the writing of a client-server application very simple and easy, and of course, in a platform-independent fashion.

Since client and server behave differently, in the sense that the first one is the requester and the other the provider, when writing a client-server program we should separate into two different classes the communicating entities. What a server machine has to do first is to create a `ServerSocket` instance at a specific port. A `ServerSocket` is a class included in the `java.net` package and corresponds to a server machine. That instance will be the gateway through which a client could connect to.

After creating a `ServerSocket`, the server invokes the `accept()` method which indicates its readiness and willingness to accept clients. Hence it stays at that state until a client establishes

a connection or until the `ServerSocket`'s timeout period expires. On the other hand, a client trying to connect to a server should first of all create a `Socket` instance using the `java.net.Socket` class and identifying the IP address (or host name) of the server and the port number that socket corresponds to. When the connection is established both machines continue their execution in parallel.

If they want to exchange messages, both client and server should declare a `BufferedReader` and a `PrintWriter` for reading and writing to the socket respectively. This process of sending and receiving messages between server and client could last until one of them closes the connection by invoking the `close()` method. The following table depicts how a server and a client behave under normal circumstances.

	Client	Server
1	Open a socket	Opens a server socket, waiting for connections
2	Open input, output stream to the socket	Accept a connection and return a socket
3	Read from and write to the stream	Open input and output stream to the socket
4	Close the streams	Read from and write to the stream
5	Close the socket	Close the streams and the socket
6		Either close the server socket or wait for a new connection

Table 1. Java Client-Server behavior

2.6 IOA Compiler

The need for having automated implementation of Complex Distributed algorithms specified in the IOA Language, has led researchers in trying to creating an IOA Compiler capable of translating IOA specifications to executable code. Several attempts, including Goldman's Spectrum System [17], Goldman's Programmer's Playground [18], and Cheiner

and Shvartsman [19] experiments, and others, were done, before the creation of Josh's IOA Compiler [4], with no significant success. The IOA Compiler manages to turn IOA imperative constructs into Java executable code with the resulting code to be able to run on workstations supporting Java. The first way of communication between all participating nodes running the generated code was, in those days, via MPI.

What makes the IOA compiler really important and what magnifies its abilities is that the generated code preserves the properties of the algorithm which were proved formally to be correct during its definition as an automaton [4]. Hence, a programmer can write a specification in IOA language, and then use the compiler for validation and automated translation of it into Java executable code preserving all the correctness properties of the IOA [6] under the assumptions that no other factors such as the network behavior and programmer annotations may affect correctness.

For an IOA program to be able for compilation it must firstly comply with several syntax and semantics constraints imposed by the compiler's nature. Therefore, a programmer writing an IOA for compilation should of first all combine the automaton with some other auxiliary automata and then provide additional annotations for resolving the nondeterminism [6]. Moreover, all IOA should be structured in a node-channel form reflecting the architecture of the target systems regarding the communication method. In that way, the generated code is consisted not only of the algorithm automata, which is an algorithm implementation at a node, but with the communication protocol as well. Moreover, in that way, we are not concerned about synchronization issues between processes running on different workstations.

The first implementation of the IOA compiler was supporting only Message Passing Interface for process communication using *Isend*, *test*, *Iprobe*, and *recv*. Additionally, communication between nodes in the system was using, (and still does) asynchronous, reliable, one-way, FIFO channels [6]. Those channels are implemented by combining the

communication protocol that will be applied in the algorithm, and the *mediator automata*, *SendMediator* and *ReceiveMediator*, which are composed with the algorithm automata we have mentioned above. Figure 5 illustrates how a node (algorithm automaton) is able of communicating with another node.

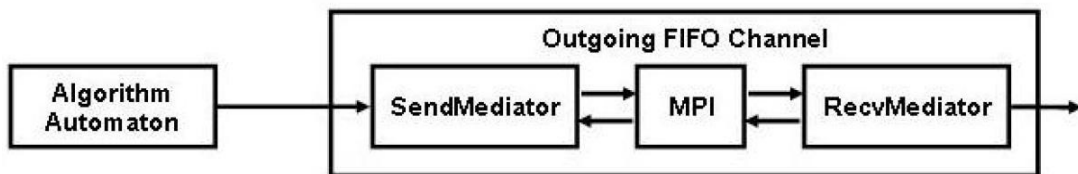


Figure 5: Auxiliary automata mediate between MPI and algorithm automata to yield a reliable FIFO channel [6]

The translated node can be thought of as an autonomous Java program which can run on a host. Each data type specified in the IOA is transformed during translation into a Java class file where all automaton's states are represented as system variables and transitions are transformed to Java methods. The interaction between other nodes and the mediators is done using Java procedure calls, firing the appropriated methods each time.

Since the IOA language is nondeterministic, a mechanism is needed so the translation can comply with the imperative nature of the Java language. This means that there should be a mechanism defining the order with which all actions of the algorithm will be fired and executed. For this to be accomplished, the IOA compiler uses *schedules* (in the same sense we have already seen them) specifying which action will be executed at any time at each node. Additionally, the *choose* clause allows picking random values within a range and used as execution parameters contributing to explicit nondeterminism. An example of choosing a random number between 0 and 3 would look as below [6]:

```
num:= choose n:Int where  $0 \leq n \wedge n < 3$ 
```

At this point, it is important to mention that even though IOA are input enabled since input actions are always enabled, the generated code is not. Instead the input is passed to the program during the run-time and only when that is required to happen, so this has to be considered when writing automata for compilation. For avoiding a node trying to read data that do not actually exist, the compiler is proposing the usage of buffers where the messages are appended to and the node is checking whether the buffer is not empty before proceeding and read. Of course, as a part of the nondeterminism, the programmer should define the starting values of the automaton's states by using the *initially* clause.

Finally, the IOA compiler was tested by implementing correctly several complex distributed algorithms such as the LCR Leader Election [20], the GHS algorithm [21], the Paxos algorithm [7] and many others.

2.7 The Consensus Problem and the Paxos Algorithm

The *consensus* problem [22] where a collection of processes must agree on a common value, is considered one of the most fundamental problems in distributed computing. In general, the problem of consensus refers to the case where n processes can propose a value, and at the end all of them should agree on the same value. Moreover, the resulting value cannot be different from those that were proposed by the processes. In other words, if set $\{S\}$ contains all the values proposed by the processes, the resulting value should exist in S . Finally, all non-faulty processes should decide on a value. What has been described so far corresponds to the conditions of *Agreement*, *Validity* and *Termination*. The first two have to do with *safety* conditions and should always be present for a consensus correctly to exist where the latter is a *liveness* condition and it is necessary only for performance issues.

The Paxos Algorithm is one of the most popular algorithms for solving consensus problem. Paxos was presented by Lamport in 1990 and published in 1998 [23]. What makes Paxos really important is that it can tolerate process crashes, message losses and timing failures. In addition, the algorithm assures direct communication between each process in the distributed system.

The Paxos algorithm is divided into six different phases [7] which are outlined below.

1. The leader starts a new ballot, that is a new voting, and informs 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 is last voted for.
3. Once the leader receives responses from a majority of votes, it chooses a value for the ballot that is based on the received values and announces that value to the others.
4. A process that learns about the 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 ballot's value once it receives messages from a majority of votes 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.

Paxos was one of the first algorithms that had been used to verify the ability of the IOA compiler for automated implementation using MPI as a communication channel. We have also used Paxos for our automated implementation using Java TCP Sockets. However we are not

going into details about exactly how the Paxos algorithm was implemented since it is beyond the scope of this Thesis. All information however can be found in [7] and [24].

2.8 Other Formal Methods

Input/Output automata is not the only formal method candidate for verification and analysis of distributed and parallel systems. Process Algebra [25], or just PA, provides us an alternative framework for modeling and analyzing such systems in a very concrete way.

The Process algebra family contains a lot of variations like Temporal Process Algebra (TPA) [26], and others. One of the most basic and fundamental PAs however is CCSv [2, 27] which is a value-passing calculus including conditional agents [2, 27], while CCS refers to Calculus of Communicating Systems. In order to define CCSv we firstly need a set of *constants*, a set of *functions*, and a set of *variables*. In addition we have to define a set of *channels* L which allows process communication.

As in IOA, PA actions are also divided as input, output and internal with α , $\tilde{\alpha}$, and τ being their representations. Internal actions arise when an input and an output action are performed in parallel in the communication channel and hence synchronization occurs. We can therefore say that input and output actions on the same channel are *complimentary* actions [28]. What has to be defined lastly when dealing with CCSv is a set of processes C . For each process P included in the set of C , the syntax of CCSv is as below:

$$P ::= 0 \mid \alpha.P \mid P1 + P2 \mid P1 \parallel P2 \mid P \setminus L \mid \text{cond} (e1 \blacktriangleright P1, \dots, en \blacktriangleright Pn) \mid C \langle \tilde{v} \rangle \text{ [28].}$$

Process 0 represents a process which is inactive. $\alpha.P$ means that process P can perform the action α and then behave as P . On the other hand $P1+P2$ represent the nondeterministic choice between these two processes whereas $P1 \parallel P2$ represent the parallel execution of them. The conditional process says that process P has the option to choose between those actions

included in the set and behave accordingly. Finally, $P \setminus L$ indicates that some actions will be restricted only for use in channel L and hence those components do not have direct interaction with the external environment of P . The greatest precedence has the $.$ operator, with the $+$ operator coming next.

As in the IOA model, PA also represents process in the model using the notion of transitions. A transition in CCS without considering internal actions is of the form $E \xrightarrow{\alpha} F$ meaning that process E is capable of performing action α and then behaving as process F . We therefore have the same notation as in IOA, (s, α, s') . On the other hand, if process E can perform action α but performing first some internal actions, we then denote that as $E \xRightarrow{\alpha} F$.

An attempt for comparing IOA and PA was presented in [29] by specifying and verifying the LCR algorithm [20] using both methods and evaluating the results concluding the following. First of all, both models are applicable for successfully model and verify the algorithm. Considering the correctness criterion of the algorithm (a common leader is elected) and the confluent behavior, the process-calculus seemed to be easier to apply. At the end, the researchers ask a newcomer to the two formalisms to evaluate them regarding the language they use. The result was that IOA are easier to understand as compared to PA.

The concurrency factory is an integrated toolset for specification, simulation, verification, and implementation of real-time concurrent systems such as communication protocol and process control systems [30]. Through a graphical user interface called VTView, a user can design and simulate concurrent systems using process algebra. Moreover, the tool uses a language called VPL, which can be translated through a compiler into networks of finite-state processes. Checkers and verification routines are also available in the tool supporting between others strong and weak bisimulation checkers. Finally, a graphical compiler translates VPL specifications into C++ executable code.

This tool however has been designed for sequential algorithm and to the best of our knowledge there does not exist other framework than IOA to provide automated implementation of *distributed* systems and algorithms. Hence, that is a huge advantage for the IOA model without of course ignoring or neglecting the abilities of Process Algebra.

Chapter 3

Enhancing Tempo with TCP/Java Sockets

In this chapter we present the basic primitive data types supported by Tempo. Then an explanation is provided about how the TCP communication protocol was embedded to the compiler and what Java classes have to be created. Finally, a review is given summarizing all problems were faced during this integration and how we managed to overcome them.

3.1 Tempo Language

In this section we present some of the primitive data types that are supported by Tempo and that are necessary for a programmer to know when writing a Tempo specification. Other user-defined data types are also supported by using "vocabularies" in the specification.

3.1.1 Booleans

This data type in Tempo can either be *true* or *false*. The following two tables list the notations and operators supported by this data type.

Tempo Symbol	Sample use	Meaning
	True	The logical value true
	False	The logical value false
\sim	$\neg p$	Negation (not)
\wedge	$p \wedge q$	Conjunction (and)
\vee	$p \vee q$	Disjunction (or)
\Rightarrow	$p \Rightarrow q$	Implication (implies)
\Leftrightarrow	$p \Leftrightarrow q$	Logical equivalences (if and only if)

Table 2: Boolean supported notations [31]

Symbol	Tempo Symbol	Sample use	Meaning
$=$		$x = y$	Equal to
\neq	$\sim =$	$x \neq y$	Not equal to
\forall	$\forall A$	$\forall n:\text{Nat} \neg (n < 0)$	For all
\exists	$\exists E$	$\exists i:\text{Int} (i < 0)$	There exists

Table 3: Boolean supported operators [31]

3.1.2 Natural Numbers

This data type contains all the non-negative integers 0,1,2.. and supports the following notations.

Symbol	Tempo Symbol	Sample use	Meaning
0,1,...		123	Natural Numbers
<i>succ</i>		<i>succ(x)</i>	Successor ($succ(x) = x + 1$)
<i>pred</i>		<i>pred(x)</i>	Predecessor ($pred(succ(x)) = x$)
+		x+y+z	Addition
-	-	x-y	Subtraction (undefined if $x < y$)
*		$x*(y**z)$	Multiplication, exponentiation
**		$x**y$	Exponentiation x^y
min, max		min(x,y)	Minimum, maximum
div, mod		mod(x,y)	Quotient, modulus
<, ≤		$x \leq y$	Less than (or equal to)
>, ≥		$x \geq y$	Greater than (or equal to)
=, ≠		$x = y$	Equal to, not equal to

Table 4: Natural Number supported operators [31]

3.1.3 Integers

This data type contains all integer numbers ranging from ..., -2, -1, 0 to 1, 2,As it easy to understand, Natural numbers are a subset of Integers and therefore all notations shown

above for the Natural numbers are also applicable for Integers. Furthermore, Integers are equipped with the following operations.

Symbol	Tempo symbol	Sample use	Meaning
-	-	-x	Additive inverse (unary minus)
<i>abs</i>		<i>abs(x)</i>	Absolute value

Table 5: Integer additional supported operators [31]

3.1.4 Characters

This data type consists of characters, letters, digits and all possible combinations between them. As above, we list all supported notations.

Symbol	Tempo Symbol	Sample use	Meaning
'A',..., 'Z'		'J'	Uppercase letters
'a',..., 'z'		'j'	Lowercase letters
'0',..., '9'		'7'	Digits
<, ≤, >, ≥		'A' < 'Z'	Alphabetic ordering

Table 6: Characters supported operators [31]

3.1.5 Extensions by nil

All elements that are contained into the Null[E] data type are equipped with an additional element, nil, that supports the following notations.

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

Table 7: Nil supported notations [31]

3.2 Java TCP Sockets Integration

Our work is based on [12] where an abstract channel specification and an algorithm implementing it using java sockets was introduced consisted of several automata. First of all, an automaton was specified modeling the behavior of a many-to-many asynchronous communication channel, called ABSCH.tioa. Then, an automaton called JVMCH, modeling the behavior of the Java interface to a communication channel using TCP was specified. Finally, two additional automata were used based on Tauber's approach [4] for establishing a mediation between the sending application, communication channel, and the destination application. Figure 5 illustrates the approached briefly described above.

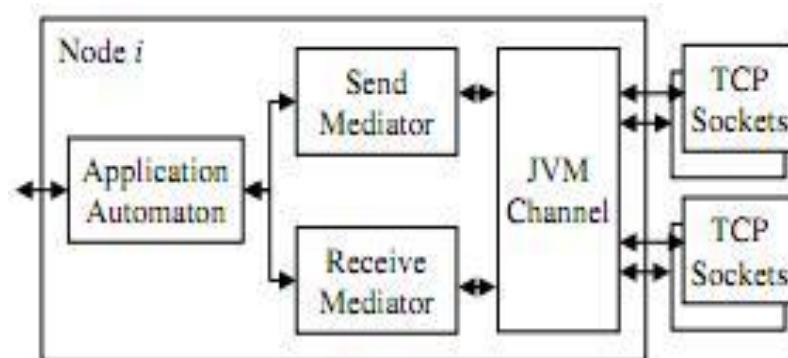


Figure 6: Node automata [12]

For integrating Java TCP Sockets to the toolkit several changes had to be done in the Tempo to Java project which can be found in the Tempo's SVN Repository [32]. This package, as it implied by its name, it is responsible for translating Timed I/O Automata into java executable code.

The very first files that needed to be changed were *BasicTranslator.java* and *CompositeTranslator.java* which are responsible for translating primitive and composite automata respectively. These two classes implement the algorithm automata code which will contain all import declarations, main method and of course schedules. In the following paragraphs all changes that were done during the integration are presented step by step.

Since we wanted to enhance Tempo to support both MPI and Java TCP Sockets we should define a mechanism for distinguishing which communication protocol will be used and generate the appropriate code at each case. In both cases however, we have to import the *Datatypes* package which contains all those *Datatypes* needed to establish either an MPI or Sockets communication. This import statement is specified as below:

```
_automaton.appendToStart(0, "import Datatypes.*;" + EOL);
```

meaning that `import Datatypes.*;` will be added in the head of the file and then will change a line, by using the *EOL* clause.

The communication type is specified before running the plug-in to translate the code in the Argument tabs of Run Configuration and can be accessed during translation by using the `_spec.getCommType()` method. When using Java TCP Sockets we decided to allow the following declarations: *TCP*, *JVM*, *UDP* all producing the same code. Figure 6 depicts how we handle the case when the algorithm will use Java Sockets for communication type and import `java.util.Vector`, `java.net`, and `java.io` packages.

```

if ( _spec.getCommType() != null && _spec.getCommType().toUpperCase().equals("MPI")) {
    _automaton.appendToStart(0, "import java.util.Vector;" + EOL + EOL);
    _automaton.appendToStart(0, "import mpi.*;" + EOL + EOL);
}
else if ( _spec.getCommType() != null && ( _spec.getCommType().toUpperCase().equals("TCP") ||
    _spec.getCommType().toUpperCase().equals("JVM") || _spec.getCommType().toUpperCase().equals("UDP"))) {

    _automaton.appendToStart(0, "import java.util.Vector;" + EOL + EOL);
    _automaton.appendToStart(0, "import java.net.*;" + EOL + EOL);
    _automaton.appendToStart(0, "import java.io.*;" + EOL + EOL);

}
} //end else if
else
    _automaton.appendToStart(0, EOL);

```

Figure 7: Java TCP Sockets import statements

Another distinction that has to be made when using either MPI or Java Sockets is the number of the input parameters that have to be passed to the main method of the algorithm. MPI needs three input parameters whereas Java Sockets needs seven. Firstly, we need to pass an IP address of the form xxx xxx xxx xxx (for example 192 168 10 4) which can be used both in the schedules and for creating sockets to that address. Next, we need to specify the port number to which the server socket will listen to and finally, a timeout period for the socket and the trajectory need to be defined.

If the main method is fed with correct number of parameters, it will then create an array of *java.lang.String* that will hold in those parameters and use them later on for instantiating all other components (sendMediator, recvMediator, driver, and so on). Otherwise, an appropriate message will be thrown to the user. The reason why *java.lang.String* is used instead of simply String is because Tempo's string is not exactly the same as Java String data type and hence we had to create a separate java file and make this distinction possible. Figure 7, depicts how the generated code that will responsible for reading all program's input arguments and store them into an array, when using TCP communication will look like.


```

else if (_spec.getCommType() != null &&
        (_spec.getCommType().toUpperCase().equals("TCP") ||
         _spec.getCommType().toUpperCase().equals("UDP") ||
         _spec.getCommType().toUpperCase().equals("JVM"))) {
    _automaton.appendToBody(2, "java.lang.String [] tcp_args = " +
                             "new java.lang.String[args.length];" + EOL);
    _automaton.appendToBody(2, "for (int i = 0; i < args.length; i++) {" + EOL);
    _automaton.appendToBody(3, "tcp_args[i] = new java.lang.String(args[i]);" + EOL);
    _automaton.appendToBody(2, "}" + EOL);
}

```

Figure 8: Java TCP Sockets Input parameters

Finally, code was added for telling the compiler which data types classes should be created depending again on the type of the communication channel.

```

if (_spec.getCommType().equals("TCP"))
    VocabTranslator.createTCPClasses(_log, _outputDir, _spec);

```

The next class that had to be modified after *BasicTranslator.java* and *CompositeTranslator.java* was the one just mentioned above, the *VocabTranslator.java*. This java class file provides the actual translation of all data types specified in TIOAs in the form of vocabularies and which are needed for the MPI or Java Socket model to work.

First of all, we created an ArrayList adding all data types supported by the new model. Those are *JVMError*, *JVMStream*, *JVMServerSocket*, *JVMSocket*, and *TCPNode*. For each one of them, we had to create a separate Java class file which would contain all methods and fields supported by each one of them.

The reason why we had to create our own data types and not just use *ServerSocket*, *Socket* and so on was because Tempo has certain assumptions about its supported data types and therefore we could not use directly those types. They are going to be referred to in the

following sections. In the following figure illustrates how the split between MPI's and JVM's data types was managed.

```
private static ArrayList<String> JVMTypes = new ArrayList<String>
    (Arrays.asList("JVMEError", "JVMSStream", "JVMServerSocket",
                  "JVMSocket", "TCPNodeVoc"));
private static ArrayList<String> MPITypes = new ArrayList<String>
    (Arrays.asList("mpi_status", "mpi_request", "mpi_status_voc",
                  "mpi_request_voc", "mpi_message_voc", "mpi_voc"));
```

Figure 9: Java TCP Sockets data types declaration

When the IOA compiler is calling *VocabTraslator.java* firstly checks if the data type going to be translated is included into the ArrayLists specified above. If that does not happen, it then informs the user that has been found a data type for which a separate placeholder class should be created. Otherwise it proceeds to the translation.

For this translation separate Java classes were created with each one corresponding to one of our custom data types, with all Java Classes being under the *com.veromodo.tempo.java.structure.comm* package. In the table that follows a mapping is provided showing which class file corresponds to which data type.

Data type	Class file responsible for the translation
JVMServerSocket	JVMServerSocketNode
JVMSocket	JVMSocketNode
JVMEError	JVMEErrorNode
JVMSStream	JVMSStreamNode
TCPNodeVoc	TCPNode

Table 8. Data types translation mapping

In the following subsections a reference is given for each one of the classes we created for TCP data types explaining what each method that is contained in them is responsible to do. Finally, the Java code is listed in Appendix A.

3.2.1 *JVMError*

The first custom data type that had to be created is the *JVMError*. This class consists of two constructors and two methods. The first constructor, *JVMError()* does not take any input parameters and it simply instantiates an object of type *JVMError*. On the other hand, *JVMError(java.lang.String.m)* takes one parameter and sets the message of the *JVMError* object to the one passed in as parameter.

Finally, *public java.lang.String value()*, and *public java.lang.String toString()* just return the message that has been set to the object.

We have created this class to handle cases when other functions, such as creating a new socket need to throw an exception. Because of the incompatibility issue of Tempo with exceptions, all methods return a new *JVMError* instead.

3.2.2 *JVMServerSocket*

This class extends the `java.net.ServerSocket` class imitating its behaviors. *JVMServerSocket* class is used by a node that will behave as a server and for all the reasons explained earlier it was not feasible to use the `ServerSocket` class directly.

This class consists of one constructor and three methods. *JVMServerSocket(int arg0*, is only responsible to create and return a new *JVMServerSocket* object listening to the port passed in as a parameter.

Null<JVMServerSocket> JVM_TCPServerSocketOpen(tuple_4<Nat, Nat, Nat, Nat> i, Nat port, Nat timeout) method, accepts three parameters and returns either a Null object or a new *JVMServerSocket*. The first parameter corresponds to the IP address of the node that will behave as a server. The second and third arguments correspond to the port that the server will listen to, the to the timeout period of the server socket. When this method is invoked, first of all we create a new *JVMServerSocket* bounded to a specific port. We then set the maximum period of time the object will exist before shutting down. If there are no errors during this process we return the *JVMServerSocket* object, otherwise we return an instance of a Null object.

The next method contained in *JMVServerSocket* class is the *Null<JVMSocket> JVM_TCPServerSocketAccept(JVMServerSocket sS)* and is called immediately after the *JVM_TCPServerSocketOpen*. This method creates a new *JVMSocket* object which initially is null. It then invokes the *JVMSocket accept()* method waiting for an incoming request by a client. When that request arrives, the *JVMSocket* object is initialized and returned.

Finally, *Null<JVMEError> JVM_TCPServerSocketClose(JVMServerSocket sS)* is responsible for closing the *JVMServerSocket* passed in as a parameter and return either Null if the closing will successful, or a *JVMEError* in any other case.

3.2.3 JVMSocket

This data type is primarily related to the node acting as a client. It contains two constructors and six methods. Starting from the constructors, *JVMSocket()* returns an instance of an unconnected *JVMSocket*. On the other hand, *JVMSocket(InetAddress addr, int port)* creates a *JVMSocket* and connects it to the specified address and port.

Null<JVMSocket> JVM_TCPSocketOpen(tuple_4<Nat, Nat, Nat, Nat> j, Nat port, Nat timeout) is the first address that a client-like node has to invoke. The first parameter is the IP address of the *JVMServer* that client will connect to. Port represents the port number that the *JVMSocket* will listen, and finally, timeout indicates the maximum period the object will exist. This method, creates an *InetAddress* by using the IP address extracted from *tuple_4* and then calls *JVMSocket(InetAddress addr, int port)* to create the object. If there are problems during these steps, a *JVMSocket* object is returned otherwise we return a new *Null* object. If there is need any more for having the socket, *Null<JVMSocket> JVM_TCPSocketClose(JVMSocket cS)* can be invoked for closing the socket and return a *Null* or a *JVMError* object.

We sometimes need to know if the *JVMSocket* object is bounded to a local address and if so, to also get the IP address of the machine that the socket is remotely connected to. For these reasons we use *Null < tuple_4 <Nat, Nat, Nat, Nat>> JVM_TCPSocketGetLocalIP(Null<JVMSocket>socket)* and *Null < tuple_4 <Nat, Nat, Nat, Nat>> JVM_TCPSocketGetRemoteIP(Null<JVMSocket> socket)*. Moreover, to check whether the *JVMSocket* is connected we created a boolean method, *Bool JVM_TCPSocketIsConnected(Null<JVMSocket> cS)*.

Finally, the most important interaction between a client and server machine, that is sending messages to each other, is done via the *Null<JVMSocket>*

`JVM_write_TCPSocket(JVM_Socket socket, Object msg)` and `Null<tuple_3<Object, tuple_4<Nat,Nat,Nat,Nat>,tuple_4<Nat,Nat,Nat,Nat>>>JVM_read_TCPSocket(Null<JVM_Socket>cS)` which invoke `JVMStream.JVM_write_TCPStream(socket, msg)` and `JVMStream.JVM_read_TCPStream(cS)` respectively.

3.2.4 JVMStream

This class file makes writing and reading from a `JVM_Socket` feasible. It contains only two methods, one handling writing to a socket and another for reading from a socket. The first one is the `Null<JVMEError> JVM_write_TCPStream(JVM_Socket cS, Object msg)` which takes as input the `JVM_Socket` that is going to be used for writing and the actual message to be sent. First of all, we create an `ObjectOutputStream` which we assign to the `JVM_Socket`. We then use `writeObject` and `flush` methods to send the message. If that is successful, we return `Null`, else we return a `JVMEError`.

On the other hand, `Null < tuple_3 < Object , tuple_4 < Nat, Nat, Nat, Nat >, tuple_4<Nat,Nat,Nat,Nat>>> JVM_read_TCPStream(Null<JVM_Socket> cS)`, creates as a first step a message object and then an `ObjectInputStream` assigned to the `JVM_Socket` passed as a parameter. It then waits there until a message is arrived. As soon as this happens, it reads the message using the `readObject` and returns the message. Again, as in previous methods, a `Null` is returned in those cases something goes wrong.

3.2.5 TCPNodeVoc

This class file that consists of three methods allows us to perform operations on `Node` types. The first method, `Bool GT(tuple_4<Nat,Nat,Nat,Nat> p0, tuple_4<Nat,Nat,Nat,Nat>`

$p1$) returns true if the first Node, $p0$, is greater than the node $p1$. *Bool* $EQ(tuple_4<Nat,Nat,Nat,Nat> p0, tuple_4<Nat,Nat,Nat,Nat> p1)$ returns true if the two nodes are equal, and finally, *Bool* $LT(tuple_4<Nat,Nat,Nat,Nat> p0, tuple_4<Nat,Nat,Nat,Nat> p1)$ returns true if $p0$ is less than $p1$.

3.3 Problems Faced and How They Were Solved

Enhancing the Tempo compiler to support Java TCP Sockets was not an easy task. We faced many difficulties and spent many hours trying to overcome these obstacles in order to accomplish our objectives successfully.

First of all, we faced significant difficulties configuring the Eclipse environment and started modifying the code. Even though we followed the instructions as shown in [32] we faced problems in defining the run configuration, export the correct projects from the SVN repositories and others.

Our first try was to configure the Eclipse in a Linux environment. We managed to create an SSH connection to the SVN repository and started checking out the code from it. While we were trying to become familiar with the Tempo compiler, the projects and classes it consists of, the Eclipse crashed and the whole workspace was no longer accessible. We uninstalled Eclipse and tried to reconfigure it again, but then for some reason we could not access the Subversion plug-in to check out the projects again. After working on it for about a week with no success, we decided to leave Ubuntu and migrate to a Windows 7 machine.

After migrating to a Windows 7 machine and configuring the Eclipse we then tried to embed Java TCP Sockets into the code. The first approach included additional if-else statements when the MPI appeared specifying that way how the compiler should behave when

the communication type will be TCP. The communication type is specified in advance in the Arguments tab of run configuration as shown in the following figure

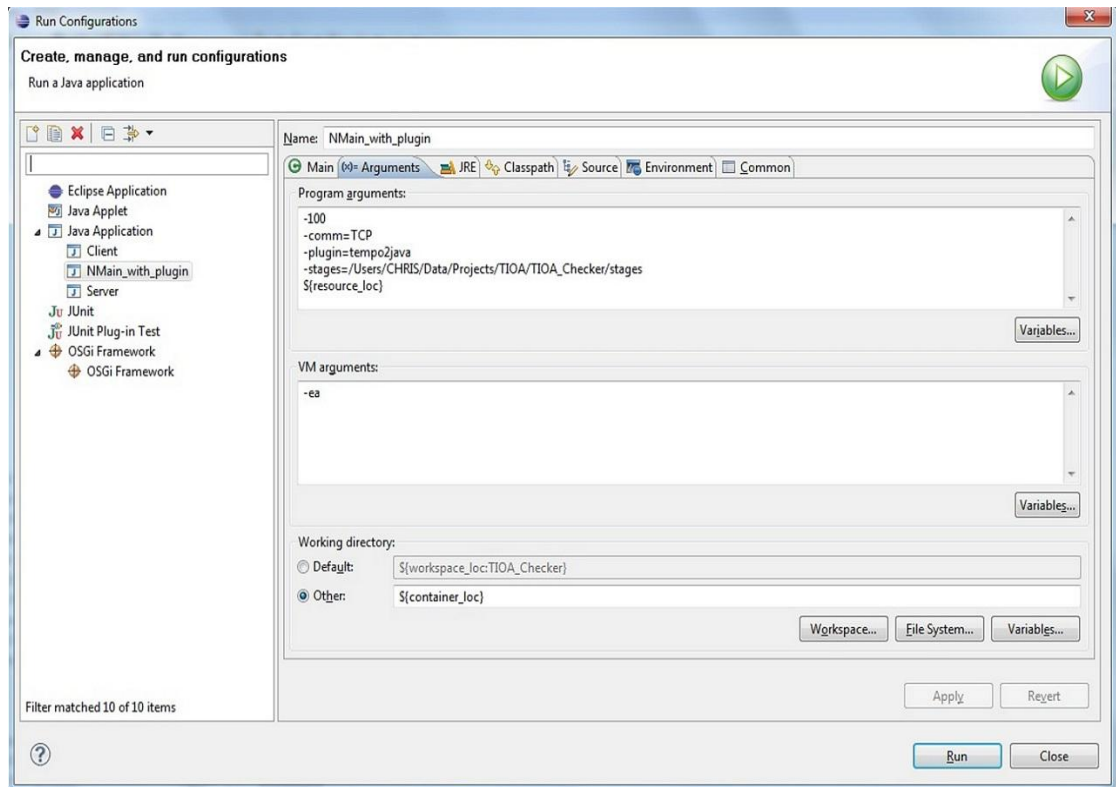


Figure 10: Specifying the communication type in Tempo

We therefore had to replace only those files where the MPI was shown and that seemed to be easy. First of all, *FunctionTranslator.java* which is responsible for translating all methods defined in *myvocabs.tioa* was modified. We then proceeded and changed *ExprTranslator.java* which is responsible for checking whether a function is passed the correct number of parameters and if not then break. Finally we changed *BasicTranslator.java* and *CompositeTranslator.java* which are responsible for translating primitive automata and composite automata respectively. Moreover, these classes are responsible for creating the *main* method and the *import* statements in the generated code.

After making these changes and running the model we realized that even though a code was generated there were errors in it regarding our custom data types. Tempo has certain

assumptions about the data types it supports and hence the idea of modifying *FunctionTranslator.java* and *ExprTranslator.java* directly to create our methods seemed to be problematic. In addition, another issue to be faced was that Tempo is also not compatible with "Exceptions".

For the above problems to be solved, it was decided to abandon the first approach and therefore delete almost all changes we had done up to that point and follow a different methodology. The new approach is the one was explained in the previous section, that is creating a separate java file for each custom data type and use these files for the translation.

After creating the methods and running the example we wrote, we had to test that everything was going according to the plan. We then realized that we were facing a problem when the receiving mediator was waiting to get a message from the sending mediator. The node was proceeding up to the line where it was waiting for input data but from that point and on it was like the socket connection was lost. After spending many hours in testing and trying to understand what was causing this issue, we realized that it had to do with the mechanism we used for Deep Copying the object since after the connection was dropped. The reason that was happening had to do with the object's serialization, since Sockets are not able to be serialized. Hence we had to modify this mechanism so it will return the object untouched and keep the connection. We faced many other minor problems, which are too technical to mention.

Chapter 4

Proof-of-concept Implementation

In this Chapter we present the two algorithms that have been used in order to demonstrate that our Tempo enhancement works. First, we describe a simple algorithm we implemented showing that a Socket connection and a message exchange can be established and then we explain how we managed to obtain an automated implementation of the Paxos algorithm.

4.1 A First Implementation of an Algorithm Using a TCP Channel

The first algorithm to be automatically compiled having Java TCP Sockets as the communication channel was JVMChanTest, a basic algorithm that was created for testing and was based on [12]. In the following paragraphs we describe what this algorithm was intended to do and the approach we followed towards this.

For our algorithm implementation several automata have been used. At first we have used an automaton modeling the algorithm which we have named JVMChanTest.tioa. Schedule was included in this automaton specifying when and how the actions will be fired. An illustration of the schedule used is given in the Figure 11.

```

schedule
states
  MIP :Node := [ n1, n2,n3,n4]; % IP from parameters
  SIP :Node := [127,0,0,1]; % [192,168,10, 4]; % server IP
  CIP :Node := [127,0,0,1]; % [192,168,10,11]; % client IP

  port:Nat := n5;
  timeout:Nat := n6;

  msrv :Null[Message] := nil();
  mcli :Null[Message] := nil();
do
  % -- message format [message, sender, receiver]
  mcli := embed(["HiFromClient", MIP, SIP]);
  msrv := embed(["HiFromServer", CIP, MIP]);

  %% server side
  if (MIP = SIP) then
    % -- bind
    fire output D.TCP_bind(MIP);
    fire output R.TCP_respBind(MIP);

    % -- listen and accept
    fire output R.TCP_accept;
    fire input R.TCP_respAccept;

    % -- read the network medium
    fire output R.TCP_rRead;
    fire input R.TCP_respRRead;
    %fire output R.RECEIVE(mcli,MIP,C
    fire output R.RECEIVE(mcli);

    if (mcli ~= nil()) then
      print val(mcli);
    fi
  fi

  %% client side -- order in which i and j are used as
  %% parameters is important.
  if (MIP = CIP) then
    % -- connect
    fire output D.TCP_senderOpen(MIP,SIP, port);

    fire output D.SEND(val(mcli),SIP,MIP);

```

Figure 11: Part of JVMChanTest.tioa's schedule

This algorithm is based on the Client-Server model [11] and uses two different types of nodes, a Sending node and a Receiving node. In our case, the role of the client was acting the sending node whereas the server was the receiving node. For the purpose of this example we

are going to refer to them as *i* and *j* respectively. The aim was to create a socket connection between them, send a message from *j* to *i* and display that message in the *i*'s side.

Furthermore, a driver automaton, *driver.tioa*, was used to define how all TCP methods will be used by the user. Two additional automata, *RecvMed* and *SendMed* used to represent the receiving and sending node. Finally, a vocabulary automaton, *myvocabs.tioa*, supporting all custom data types used and an automaton modeling all supported actions by the channel, *TCPChan.tioa* were also defined. Such custom data types include *JVMSSocket* and *JVMSServerSocket*.

Both *i* and *j* run the same generated code, but what each one of them was allowed to do is distinguished in the schedule section depending on the IP address of each node, as partially shown in Figure 11. Moreover, the way we defined the schedule ensures that both automata will proceed in parallel.

Starting from the receiving node, *i*, it firstly creates a *JVMSServerSocket* object by calling the *JVM_TCPServerSocketOpen(i, port, timeout)*. The returned *JVMSServerSocket* is bounded to the port passed in as a parameter and the maximum period of existence of the object is determined by the timeout argument. Moreover, along with the creating of *JVMSServerSocket*, the accept status of the node changes to *accepting*.

The next step *i*, performs is changing its accept status to *waiting* and then through *Input_TCP_respAccept()* that is declared in the *RecvMed*, invokes the *JVM_TCPServerSocketAccept(SSocket.val())* passing as a parameter the *JVMSServerSocket* object created in the previous step. This function call is used to create a *JVMSSocket* object which initially is null and then waits for an incoming connection to arrive.

On the other side, j 's, execution starts by invoking *Input_TCP_senderOpen*(*tuple_4*<Nat,Nat,Nat,Nat> *s*, *tuple_4*<Nat,Nat,Nat,Nat> *r*, *Nat port*) and then *JVM_TCPSocketOpen* with the same parameters as the first one. Those two methods return a *JVM_Socket* object that is connected to the IP address corresponding to the *tuple_4* of the receiving node, *i*. From that point and on both nodes proceed in parallel.

When *i* accepts the connection request from *j*, it appends its *JVM_Socket* to the receiving channel in order to can access it later on for reading and/or writing purposes. *Input_TCP_respRRead* and *JVM_read_TCPSocket* function calls lead the server in waiting for a message to arrive. Figure 12, shows a part of the automated translated Java code illustrating how all function calls are performed.

```

if( false && TJMath.EQ( TJMath.EQ(MIP, SIP) , new Bool( true )).value() ) {
  _D._VarJarOutput_TCP_bind.i = MIP;
  _D.Output_TCP_bind( MIP );
  MIP = _D._VarJarOutput_TCP_bind.i;
  _S.matchMaker( "Input_TCP_bind", MIP );
  _R.matchMaker( "Input_TCP_bind", MIP );
  _D.matchMaker( "Input_TCP_bind", MIP );
  _R._VarJarOutput_TCP_respBind.i = MIP;
  _R.Output_TCP_respBind( MIP );
  MIP = _R._VarJarOutput_TCP_respBind.i;
  _S.matchMaker( "Input_TCP_respBind", MIP );
  _R.matchMaker( "Input_TCP_respBind", MIP );
  _D.matchMaker( "Input_TCP_respBind", MIP );
  _R.Output_TCP_accept( );
  _S.matchMaker( "Input_TCP_accept" );
  _R.matchMaker( "Input_TCP_accept" );
  _D.matchMaker( "Input_TCP_accept" );
  _R.Input_TCP_respAccept( );
  _R.Output_TCP_rRead( );
  _S.matchMaker( "Input_TCP_rRead" );
  _R.matchMaker( "Input_TCP_rRead" );
  _D.matchMaker( "Input_TCP_rRead" );
}

```

Figure 12: Schedule's automated translated code

The sender on the other hand, appends the message to be sent into a buffer and then calls *JVM_write_TCPSocket*. At this point it is important to say that all calls to the JVM classes are done through interaction of the algorithm mediator with the driver, *RecvMed* and *SendMed*.

When i gets the message send, it then appends it on its own receiving buffer and finally prints this message from the buffer and terminates. J also continues and terminates. The *Trajectory* condition used in this example has to do with the timeout period which is passed to the algorithm automaton as a parameter.

All automata specified for this example are given in Appendix B.

4.2 An Implementation of Paxos Algorithm using TCP Sockets

The Paxos implementation is based on Roberto De Prisco's Msc Thesis in 1997 [33]. Paxos is presented here in a modular basis, divided into several components with each one handling a different aspect of the problem. Most importantly, *detector.tioa* is responsible for detecting process failure and recoveries, whereas, *bpleader.tioa* models the process which is responsible for running the algorithm. *bpagent.tioa* models the "agent's" behavior and finally *bpsuccess.tioa* announces to all nodes a reached decision. In the following paragraphs a detailed description is provided about how the model works as presented by De Prisco.

- To initiate a round, the leader sends a "Collect" message to all agents announcing that it wants to start a new round and at the same time asking for information about previous rounds in which agents may have been involved.
- An agent that receives a message sent in step 1 from the leader of the round, responds with a "Last" message giving its own information about rounds previously conducted. With this, the agent makes a kind of commitment for this particular round that may prevent it from accepting (in step 4) the value proposed in some other round. If the agent is already committed for a round with a bigger

number then it informs the leader of its commitment with an "OldRound" message.

- Once the leader has gathered information about previous rounds from a majority of agents, it decides, according to some rules, the value to propose for its round and sends to all agents a "Begin" message announcing the value and asking them to accept it. In order for the leader to be able to choose a value for the round it is necessary to provide initial values. If no initial value is provided the leader must wait for an initial value before proceeding with step 3. The set of processes from which the leader gathers information is called the *info-quorum* of the round.
- An agent that receives a message from the leader of the round sent in step 3, responds with an "Accept" message by accepting the value proposed in the current round, unless it is committed for a later round and thus must reject the value proposed in the current round. In the latter case, the agent sends an "OldRound" message to the leader indicating the round for which it is committed.
- If the leader gets "Accept" messages from a majority of agents, then the leader sends its own output value to the value proposed in the round. At this point the round is successful. The set of agents that accept the value proposed by the leader is called the *accepting-quorum*.

A separate Schedule for the leader is used which it is currently somehow complicated even though a lot of effort has been made in making it as simpler as possible. Currently, the schedule lacks in catching all the dynamic behavior that Paxos is capable of and this is considered to be a future work. In contrast, the agent's schedule is very simple and robust.

Other limitations based on how the schedule is currently implemented have to do with the absence of a reset procedure covering those cases where the leader fails. Anyhow, the schedule is functional enough to run correctly and produce results.

In order to run Paxos, an IP address has to be provided that will correspond to the server node. Moreover, all IP addresses of the other nodes have to be specified in advanced and this also something that could be improved in the future for better dynamicity. As a first step, the leader election is based on the IP addresses where the node having the highest IP address becomes the initial leader.

Before explaining how Paxos schedule has been implemented to support TCP communication, a brief review is presented about all other automata further to the TCP Paxos. *tioa* are needed for the algorithm to run.

First of all, *myvocabs.tioa* contains all data types needed for the TCP Channel to run starting from the IPv4 type which is a tuple of four Nat numbers and simply represents an IP address. IPv6 is also available by the model for those cases where the deployment settings support it. As next, we defined the *JVMError* which as the previous algorithm we described is of the form of *String*.

In Chapter 3, Section 3.2.5, we have mentioned that *TCPNode* allows us to perform additional operations on nodes such as the greater/less than and equality tests, and that is needed for our implementation since we want to have comparisons between the participating nodes. Moreover, it is essential know the current state of the Node, and that is determined by using the *NodeMode* type and defined as *live*, *stopped*, *begin*, *last*, *accept*, *success*, *oldround*, *collect*, *gatherlast*, *wait*, *bgincast*, *gatheraccept*, *decided*, *rnddone* and *ack*.

Round, Data, Message, and Mode types are also required for Paxos to run. Round is defined as a tuple of an Int and a Node, Data as a tuple of a NodeMode, two Round types and an Int. Message type is a tuple of Data, and two Node. Finally mode is an enumeration of done, working, leader and notleader.

Finally, we need JVMSocket, JVMServerSocket having all operators we refer to earlier, and a Channel. The latter uses all previous mentioned data types, and is defined having a MessageTuple, a Status, and a Channel type in addition to an operator checking if the channel is empty or not.

Next, an automaton for modeling the interaction between the algorithm and the TCP Channel has been created, and that is *TCP_ChanMed.tioa*. This automaton enables the creation and management of JVMServerSockets and JVMSockets. tcpChannel state variable contains all the established connections and recvBuffer the messages extracted from the network. Finally, TCPSendMed.tioa and TCPRecvMed.tioa model the interaction of the algorithm automata with automaton modeling the TCP protocol.

The first action that is fired when running the Paxos schedule is the *TCP_Bind(myIP)* which creates a new JVMServerSocket and changes the local status of each node from idle to connecting, indicating their willingness to accept connections. Moreover, the nodes initialize a local variable, localError, holding any possible errors. After that, *TCP_respBind(error,local)* is fired, changing the local status of the node from connecting to accepting, if the localError state variable is equal to null. These two actions are fired in the schedule as below:

```
fire output R.TCP_bind(myIP);
fire output C.TCP_respBind(error,myIP);
```

We then handle separately the server and the client node by firing the corresponding actions. If the node's IP address is the same as the one we defined for the server, the machine enters a loop and fires *TCP_accept* method,

```
fire output C.TCP_respAccept(error);
```

and that changes its status to waiting and forcing it to stay there until an incoming request will arrive.

On the other hand, if the IP address of a running node is not the same as the one that will behave as a server, *TCP_senderOpen(server, port)* action is fired,

```
fire output S.TCP_senderOpen( server, port );
```

trying to create a connection the node has its IP address is the same as the one passed in as a parameter and that will listen to the port specified.

After all participating nodes create a TCP connection, either as a JVMServer or JVMClient, they all run *InformAlive(n :Node)* that lays into *detector.tioa* [24] adding themselves into the alive and world nodes list. Then, the leader election algorithm runs as follows. For all nodes added into the world list in the previous step, internal *Check(world[y])* and then *SEND(ms)* are fired keeping a track which nodes have sent their message, when that happened and updating the list containing messages to be sent.

TCP_write(ms, myIP, world[y]) comes next,

```
fire output S.TCP_write( ms, myIP, world[y]);
```

adding the message to the channel and as a result the *TCP_ChanMed* writes it into the *JVMSocket* by invoking the *JVM_write_TCPsocket(val(tcpChannel[n].socket), val(m))* method. At the next step, the receiving node extract any messages exist in the channel using *TCP_read* which initiates read on all open connections. *TCP_respRead(m)* reads a message from the first reading socket and the message is added to the receiver's buffer where it can later on be extracted. Finally, the leader election process continues until a leader is elected and

announced. After the leader election, the algorithm proceeds as presented in [7] using TCP methods for writing.

All automata used for the TCP implementation of Paxos are presented in Appendix C. At this point it is important to say that all Automata used for the TCP Paxos implementation have been developed by Dr. Peter M. Musial based on the Java Sockets integration we managed to perform.

4.3 Automated translation procedure

For an algorithm to be automated translated to Java executable code, the following steps have to be performed:

1. The algorithm and the communication channels have to be expressed using the TIOA model.
2. The algorithm automaton, which has to be in a nodechannel form, has to be verified using the IOA Checker
3. A schedule has to be written for resolving the nondeterminism

So far, the composition was performed prior to compilation leading to unreadable and unmanageable code. In the current way of translation, the composition is performed during the execution time using the "*matchMaker*" method that is found in each of the automaton components and the corresponding invocation in the schedule of the top automaton.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Automated implementation of complex distributed algorithms specified in a formal language, such as IOA, is nowadays a very challenging and promising field. Distributed algorithms are emerging in a very fast rate and we can easily and without any doubts say that the future belongs to distributed computing and processing. Therefore, having such mechanisms not only for specifying and verifying complex distributed algorithms but also to automatically implement them is extremely useful.

Before this work, the Tempo toolkit and IOA Compiler gave us the ability to automatically implement such algorithms using MPI. Even though that was good enough for many years, it has certain limitations which had to be addressed. Limitations including the fact that the generated code was only able to run in a LAN and not on the Internet, and the fact all participating nodes should be defined in advance, not allowing dynamic entrance of a new node.

In this Thesis we have enhanced the Tempo toolkit to support not only MPI but Java TCP Sockets as well. This is extremely useful, since not only do we allow dynamic creation and tearing down of communication links, but also the algorithms can now run on WANS, MAN, or even on the Internet. Furthermore, by using Sockets the automated implementation of dynamic distributed systems/algorithms can be extended into planetary-scale networks as was done for RAMBO in [34].

We have firstly checked our implementation using a very basic algorithm we created that was consisted of two machines, one acting as a sender and the other one as a receiver. Our aim was to send a message from the sender to the receiver, and display that message in the receiver's side. As soon as this was done, the integration was finally tested on something much more complex, on Paxos algorithm, which also ran correctly. To the best of our knowledge this is the *first* work that generates a verifiable implementation of Paxos using Java TCP Sockets in an automated way.

5.2 Future work

In this Thesis we mainly concentrated in enabling Tempo to support Java TCP Sockets as a way of communication between the participating nodes in an algorithm. The schedules we have used focus on establishing a connection and sending and receiving successfully a message from one node to the other. Therefore, further testing is needed to be done. For example, it would be nice to test fully the closing procedures of the sockets. Moreover, we plan to make the TCP Channel model better in terms of errors reporting, and, of course, test the model on more algorithms for different fundamental problems.

In addition, further improvement is needed for eliminating all limitations mentioned earlier about Paxos automated implementation. This includes the fact that all participating nodes should be known in advance and the absence of a reset procedure covering those cases where the leader fails.

Furthermore, future work could include further enhancements to the Tempo compiler to include and support even more model interfaces, such as UDP, unicast, multicast, or both, database interfaces, quantified expressions and others. Finally, we should consider to create a GUI that will make the automated implementation process even more friendly.

Bibliography

- [1] Andrew S. Tanenbaum, Maarten van Steen, "Distributed Systems, Principles and Paradigms" Prentice Hall, 2002.
- [2] R. Milner, "A Calculus of Communicating Systems", Lecture Notes in Comput. Sci. 92, Springer, 1980.
- [3] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, Frits Vaandrager, "The Theory of Timed I/O Automata. *Synthesis Lectures of Distributed Computing Theory*", 2nd Edition, 2011.
- [4] Joshua A. Tauber, "Verifiable Compilation of I/O Automata without Global Synchronization", Phd Thesis, Massachusetts Institute of Technology, 2005.
- [5] Chryssis Georgiou, Panayiotis Mavrommatis, and Joshua A. Tauber, "Implementing Asynchronous Distributed Systems Using the IOA Toolkit", MIT CSAIL Technical Report MIT-LCS-TR-966, Cambridge, MA, 2004.
- [6] Chryssis Georgiou, Nancy Lynch, Panayiotis Mavrommatis, and Joshua A. Tauber, "Automated Implementation of Complex Distributed Algorithms Specified in the IOA Language", in International Journal on Software Tools for Technology Transfer (STTT), Volume 11, No. 2, pp. 153-171, Springer, April 2009.
- [7] Chryssis Georgiou, Procopis Hadjiprocopiou, and Peter Musial, "On the Automated Implementation of Timed-based Paxos Using the IOA Compiler", in Proc. of the 14th International Conference on Principles of Distributed Systems (OPODIS2010), pp. 235-252, Tozeur, Tunisia, 2010.
- [8] N Lynch, L. Michel, and A. Shavrtsman, "Tempo: A toolkit for The Timed Input/Output Automata Formalism", In Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks, and Systems (SIMUTools 2008), 2008.
- [9] About Tempo, <http://www.veromodo.com/code/tempo.html> - Last access on 5/5/2011.
- [10] The Message Passing Interface (MPI), Lawrence Livermore National Laboratory <https://computing.llnl.gov/tutorials/mpi/> - Last access on 17/5/2011.

- [11] Lesson: All About Sockets, <http://download.oracle.com/javase/tutorial/networking/sockets/> - Last access on 2/5/2011.
- [12] Chryssis Georgiou, Peter Musial, Alexander Shvartsman, and Elaine Sonderegger, "An Abstract Channel Specification and an Algorithm Implementing It Using Java Sockets", in the Proc. of the 7th IEEE International Symposium on Network Computing and Applications (NCA 2008), pp. 211-219, Cambridge, MA, 2008.
- [13] Nancy A. Lynch and Mark R. Tuttle, "An Introduction to Input/Output Automata" *CWI-Quarterly*, 2(3):219-246, 1989.
- [14] Communicating Sequential Processes (CSP), <http://www.usingcsp.com/> - Last access on 01/06/10.
- [15] Eclipse Rich Client Platform, <http://www.eclipse.org/home/categories/rcp.php> - Last access on 01/06/10.
- [16] Aamir Shafi, , "Parallel Programming with Java", National University of Sciences and Technology (NUST).
- [17] Kenneth J. Goldman. "Highly concurrent logically synchronous multicast". *Distributed Computing*, 6(4):189–207, 1991
- [18] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, and Ram Sethuraman. "The Programmers' Playground: I/O abstraction for user-configurable distributed applications". *IEEE Transactions on Software Engineering*,21(9):735–746, September 1995
- [19] Oleg Cheiner and Alex Shvartsman. Implementing an eventually-serializable data service as a distributed system building block. In *Networks in Distributed Computing*, volume 45 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 43–72. American Mathematical Society, 1999
- [20] G´erard Le Lann. "Distributed systems - towards a formal approach". In *Information Processing 77*, volume 7 of Proceedings of IFIP Congress, pages 155–160. North-Holland Publishing Co., 1977.
- [21] P. A. Humblet R. G. Gallager and P.M. Spira. "A distributed algorithm for minimum-weight spanning trees". In *ACM Transactions on Programming Languages and Systems*, volume 5(1), pages 66–77, January 1983
- [22] Michael J. Fischer. "The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)", *FCT* 1983: 127-140, 1983
- [23] L. Lamport, "The part-time parliament. *ACM Transactions on Computer Systems*", 16 (2):133-169, 1998.
- [24] N. Lynch and A. Shvartsman, "Paxos made even simpler", Manuscript, 2002.

- [25] J. A. Bergsrea, A. Ponse, and S. A. Smolka. "Handbook of Process Algebra". North-Holland 2001.
- [26] Xavier Nicollin and Joseph Sifakis. "An overview and synthesis on timed process algebras". North-Holland 2001.
- [27] C. Tofts, "Proof Methods and Pragmatics for Parallel Programming", PhD Thesis, Univ. of Edinburgh, 1990.
- [28] R. Milner, "Communication and Concurrency", Prentice-Hall, 1989.
- [29] Marina Gelastou, Chryssis Georgiou, and Anna Philippou, "On the Application of Formal Methods for Specifying and Verifying Distributed Protocols", in the Proc. of the 7th IEEE International Symposium on Network Computing and Applications (NCA 2008), pp. 195-204, Cambridge, MA, 2008
- [30] Rance Cleaveland, Philip M. Lewis, Scott A. Smolka and Oleg Sokolsky, "The Concurrency Factory Software Development Environment", In Proceedings of TACAS'1996. pp.391~395.
- [31] Nancy A. Lynch, Stephen J. Garland, Dilsun Kaynar, Laurent Michel and Alex Shvartsman, "The Tempo Language User Guide and Reference Manual", Massachusetts Institute of Technology, 2008.
- [32] The TIOA Eclipse Development Setup http://tioa.dyndns.org/wiki/index.php/TIOA_Eclipse_Development_Setup - Last access on 23/5/2011.
- [33] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch: Revisiting the PAXOS algorithm. Theor. Comput. Sci. 243(1-2): 35-91 (2000).
- [34] Chryssis Georgiou, Nicolas Hadjiprocopiou, and Peter Musial, Evaluating a Dependable Sharable Atomic Data Service on a Planetary-Scale Network, in the Proc. of the 9th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2009), pp. 580-592.

APPENDIX A

Java TCP Custom Classes

A1. JVMError Class

```
public class JVMError {  
  
    private java.lang.String _message = null;  
  
    public JVMError() { }  
    public JVMError(java.lang.String m) { this._message = m; }  
    public java.lang.String value() { return _message; }  
    public java.lang.String toString() { return _message; }  
  
}
```

A2. JVMServerSocket Class

```
import java.io.IOException;  
import java.net.ServerSocket;  
  
public class JVMServerSocket extends ServerSocket {  
  
    public JVMSocket accept() throws IOException {  
        JVMSocket s = new JVMSocket( );  
        if (!isClosed() && isBound()) {  
            implAccept(s);  
            s.setSoTimeout(this.getSoTimeout());  
        }  
        return s;  
    }  
  
    public JVMServerSocket(int arg0) throws IOException {  
        super(arg0);  
    }  
  
    public static Null<JVMServerSocket>  
    JVM_TCPServerSocketOpen(tuple_4<Nat, Nat, Nat, Nat> i, Nat port, Nat  
    timeout) {  
        JVMServerSocket server = null;  
        try{  
            server = new JVMServerSocket(port.value());  
            server.setSoTimeout(timeout.value());  
        }  
        catch (IOException e) { }  
    }  
}
```

```

        return new Null<JVMServerSocket>( server );
    }

    public static Null<JVMEError>
    JVM_TCPServerSocketClose(JVMServerSocket sS) {
        JVMEError error = null;
        try{
            sS.close();
        } catch (IOException e) { error = new JVMEError(
e.getMessage() ); }
        return new Null<JVMEError>( error );
    }

    public static Null<JVMSocket>
    JVM_TCPServerSocketAccept(JVMServerSocket sS) {
        JVMSocket client = new JVMSocket( );
        try {
            client = sS.accept();
        } catch (IOException e) { }
        return new Null<JVMSocket>( client );
    }
}

```

A3. JVMSocket Class

```

import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.SocketAddress;
import java.net.UnknownHostException;

public class JVMSocket extends Socket {

    public JVMSocket() { super(); }

    public JVMSocket(InetAddress addr, int port) throws IOException
    { super(addr, port); }

    public static Null<JVMSocket> JVM_TCPSocketOpen(tuple_4<Nat,
        Nat, Nat, Nat> j, Nat port, Nat timeout){
        int ipPart1 = j.f0().intValue();
        int ipPart2 = j.f1().intValue();
        int ipPart3 = j.f2().intValue();
        int ipPart4 = j.f3().intValue();
        JVMSocket cS = new JVMSocket();
        try {
            InetAddress addr = InetAddress.getByName(
                Integer.toString(ipPart1) + '.' +
                Integer.toString(ipPart2) + '.' +
                Integer.toString(ipPart3) + '.' +
                Integer.toString(ipPart4));
            cS = new JVMSocket(addr, port.intValue());
            if (!cS.isConnected()) {
                SocketAddress sockaddr = new
                InetSocketAddress(addr, port.intValue());
                cS.connect(sockaddr);
            } else
                cS.setSoTimeout(timeout.value());
        }
    }
}

```

```

    catch (UnknownHostException e) {
        System.out.println(e.getMessage()); }
    catch (IOException e) {
        System.out.println(e.getMessage()); }
    return new Null<JVMSocket>( cS );
}

public static Null<JVMEError> JVM_TCPSocketClose(JVMSocket cS) {
    JVMEError error = null;
    try { cS.close(); }
        catch (IOException e) {
            error = new JVMEError( e.getMessage() ); }
    return new Null<JVMEError>( error );
}

public static Null<tuple_4<Nat,Nat,Nat,Nat>>
    JVM_TCPSocketGetLocalIP( Null<JVMSocket> socket ) {
    if (socket == null || socket.val() == null)
        return new Null<tuple_4<Nat,Nat,Nat,Nat>>( );
    byte[] localAddress =
        socket.val().getLocalAddress().getAddress();
    return new Null<tuple_4<Nat,Nat,Nat,Nat>>(
        new tuple_4<Nat,Nat,Nat,Nat>(
            new Nat( localAddress[0] & 255 ), new Nat(
                localAddress[1] & 255 ),
            new Nat( localAddress[2] & 255 ), new Nat(
                localAddress[3] & 255 ));
}

public static Null<tuple_4<Nat,Nat,Nat,Nat>>
    JVM_TCPSocketGetRemoteIP( Null<JVMSocket> socket ) {
    if (socket == null || socket.val() == null)
        return new Null<tuple_4<Nat,Nat,Nat,Nat>>( );
    byte[] localAddress =
        socket.val().getRemoteSocketAddress() != null ?
        socket.val().getInetAddress().getAddress() : new byte[0];
    if (localAddress.length == 0)
        return new Null<tuple_4<Nat,Nat,Nat,Nat>>( );
    return new Null<tuple_4<Nat,Nat,Nat,Nat>>(
        new tuple_4<Nat,Nat,Nat,Nat>(
            new Nat( localAddress[0] & 255 ),
            new Nat( localAddress[1] & 255 ),
            new Nat( localAddress[2] & 255 ),
            new Nat( localAddress[3] & 255 ));
}

public static Null<tuple_3<Object, tuple_4<Nat, Nat, Nat, Nat>,
tuple_4<Nat, Nat, Nat, Nat>>> JVM_read_TCPSocket(Null<JVMSocket> cS)
{
    return JVMStream.JVM_read_TCPStream(cS);
}

public static Null<JVMEError> JVM_write_TCPSocket(JVMSocket
    socket, Object msg) {
    return JVMStream.JVM_write_TCPStream(socket, msg);
}

public static Bool JVM_TCPSocketIsConnected(Null<JVMSocket> cS)
{
    if (cS == null || cS.val() == null)
        return new Bool( false );
}

```

```

        return new Bool( cS.val().isConnected() );
    }
}

```

A4. JVMStream Class

```

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.SocketTimeoutException;

public class JVMStream {

    public static
    Null<tuple_3<Object,tuple_4<Nat,Nat,Nat,Nat>,tuple_4<Nat,Nat,Nat,Nat>,Nat>>> JVM_read_TCPStream(Null<JVMSocket> cS) {

        tuple_3<Object,tuple_4<Nat,Nat,Nat,Nat>,tuple_4<Nat,Nat,Nat,Nat>>> msg = null;
        if (cS != null && cS.val() != null) try {
            ObjectInputStream in = new
            ObjectInputStream(cS.val().getInputStream());
            msg =
            (tuple_3<Object,tuple_4<Nat,Nat,Nat,Nat>,tuple_4<Nat,Nat,Nat,Nat>>>
            in.readObject());
        }
        catch(SocketTimeoutException e) { return new
        Null<tuple_3<Object,tuple_4<Nat,Nat,Nat,Nat>,tuple_4<Nat,Nat,Nat,Nat>>>( ); }
        catch(ClassNotFoundException e) { return new
        Null<tuple_3<Object,tuple_4<Nat,Nat,Nat,Nat>,tuple_4<Nat,Nat,Nat,Nat>>>( ); }
        catch(IOException e){ return new
        Null<tuple_3<Object,tuple_4<Nat,Nat,Nat,Nat>,tuple_4<Nat,Nat,Nat,Nat>>>( ); }
        return new
        Null<tuple_3<Object,tuple_4<Nat,Nat,Nat,Nat>,tuple_4<Nat,Nat,Nat,Nat>>>( msg );
    }

    public static Null<JVMEError> JVM_write_TCPStream(JVMSocket cS,
                                                    Object msg) {
        JVMEError error = null;
        if (cS != null && msg != null) try {
            ObjectOutputStream out = new
            ObjectOutputStream(cS.getOutputStream());
            out.writeObject(msg);
            out.flush();
        } catch (IOException e) {
            error = new JVMEError( e.getMessage() ); }
        return new Null<JVMEError>( error );
    }
}

```

A5. TCPNodeVoc Class

```

import Utils.TJMath;

public class TCPNodeVoc {

    public static Bool GT(tuple_4<Nat,Nat,Nat,Nat> p0,
                          tuple_4<Nat,Nat,Nat,Nat> p1) {
        if ( TJMath.GT(p0.f0(), p1.f0()).value() )
            return new Bool( true );
        if ( TJMath.EQ(p0.f0(), p1.f0()).value() &&
              TJMath.GT(p0.f1(), p1.f1()).value() )
            return new Bool( true );
        if ( TJMath.EQ(p0.f0(), p1.f0()).value() &&
              TJMath.EQ(p0.f1(), p1.f1()).value() &&
              TJMath.GT(p0.f2(), p1.f2()).value() )
            return new Bool( true );
        if ( TJMath.EQ(p0.f0(), p1.f0()).value() &&
              TJMath.EQ(p0.f1(), p1.f1()).value() &&
              TJMath.EQ(p0.f2(), p1.f2()).value() &&
              TJMath.GT(p0.f3(), p1.f3()).value() )
            return new Bool( true );
        return new Bool( false );
    }

    public static Bool EQ(tuple_4<Nat,Nat,Nat,Nat> p0,
                          tuple_4<Nat,Nat,Nat,Nat> p1) {
        if ( TJMath.EQ(p0.f0(), p1.f0()).value() &&
              TJMath.EQ(p0.f1(), p1.f1()).value() &&
              TJMath.EQ(p0.f2(), p1.f2()).value() &&
              TJMath.EQ(p0.f3(), p1.f3()).value() )
            return new Bool( true );
        return new Bool( false );
    }

    public static Bool LT(tuple_4<Nat,Nat,Nat,Nat> p0,
                          tuple_4<Nat,Nat,Nat,Nat> p1) {
        if ( TJMath.LT(p0.f0(), p1.f0()).value() )
            return new U( true );
        if ( TJMath.EQ(p0.f0(), p1.f0()).value() &&
              TJMath.LT(p0.f1(), p1.f1()).value() )
            return new Bool( true );
        if ( TJMath.EQ(p0.f0(), p1.f0()).value() &&
              TJMath.EQ(p0.f1(), p1.f1()).value() &&
              TJMath.LT(p0.f2(), p1.f2()).value() )
            return new Bool( true );
        if ( TJMath.EQ(p0.f0(), p1.f0()).value() &&
              TJMath.EQ(p0.f1(), p1.f1()).value() &&
              TJMath.EQ(p0.f2(), p1.f2()).value() &&
              TJMath.LT(p0.f3(), p1.f3()).value() )
            return new Bool( true );
        return new Bool( false );
    }
}

```

APPENDIX B

JVMChanTest Algorithm

B1. driver.tioa

```

imports JVMSocket
imports JVMServerSocket
imports TCPObjectsVoc
imports TCPNodeVoc
imports ChannelVoc
imports algorithm_voc

automaton driver

signature

  % paired to the receive mediator
  input RECEIVE(m:Null[Message])
  input respBind(i:Node)

  % paired to the send mediator
  output SEND(m:Message, r,s:Node)
  output TCP_senderOpen(s,r:Node, port:Nat)
  output TCP_senderClose(s,r:Node)

  % paired to the receive mediator
  output TCP_bind(i:Node)
  output TCP_stopListening(j:Node)
  output TCP_rClose(i,j:Node)

states
  noop:Bool := true;

transitions

  input RECEIVE(m)
  eff
    noop:=true;

  input respBind(i)
  eff
    noop:=true;

  output SEND(m,r,s)
  pre
    noop;
  eff
    noop:=true;

  output TCP_senderOpen(s,r,port)
  pre
    noop;
  eff
    noop:=true;

  output TCP_senderClose(s,r)
  pre
    noop;

```

```

eff
  noop:=true;

output TCP_bind(i)
pre
  noop;
eff
  noop:=true;

output TCP_stopListening(j)
pre
  noop;
eff
  noop:=true;

output TCP_rClose(i,j)
pre
  noop;
eff
  noop:=true;

```

B2. JVMChanTest.tioa

```

include "myvocabs.tioa"
include "driver.tioa"
include "SendMed.tioa"
include "RecvMed.tioa"

imports JVMSocket
imports JVMServerSocket
imports TCPObjectsVoc
imports TCPNodeVoc
imports ChannelVoc
imports algorithm_voc

automaton JVMChanTest(n1:Nat, n2:Nat, n3:Nat, n4:Nat, n5:Nat, n6:Nat)
  components
    S:SendMed(n5,n6);
    R:RecvMed(n5,n6);
    D:driver;
  schedule
  states
    MIP :Node := [ n1, n2,n3,n4]; % IP from parameters
    SIP :Node := [192,168,10, 4]; % server IP
    CIP :Node := [192,168,10,11]; % client IP

    port:Nat := n5;
    timeout:Nat := n6;

    msrv :Null[Message] := nil();
    mcli :Null[Message] := nil();
  do
    % -- message format [message, sender, receiver]
    mcli := embed(["HiFromClient", MIP, SIP]);
    msrv := embed(["HiFromServer", CIP, MIP]);

    %% server side
    if (MIP = SIP) then
      % -- bind
      fire output D.TCP_bind(MIP);

```



```

        fire output R.TCP_respBind(MIP);

        % -- listen and accept
        fire output R.TCP_accept;
        fire input  R.TCP_respAccept;

        % -- read the network medium
        fire output R.TCP_rRead;
        fire input  R.TCP_respRRead;
        fire output R.RECEIVE(mcli);

        if (mcli ~= nil()) then
            print val(mcli);
        fi
    fi

    %% client side -- order in which i and j are used as
    %% parameters is important.
    if (MIP = CIP) then
        % -- connect
        fire output D.TCP_senderOpen(MIP,SIP, port);

        % -- send a messages
        fire output D.SEND(val(mcli),SIP,MIP);
        fire output S.TCP_write(mcli,SIP,MIP);

        % -- close
        fire output D.TCP_senderClose(MIP,SIP);

        follow S.v(SIP) duration timeout;
    fi
od

```

B3. myvocabs.tioa

```

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],
    JVMEError : String
  end

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

%%% .:algorithm vocabs:.
vocabulary algorithm_voc
  imports TCPObjectsVoc
  imports TCPNodeVoc
  types Data      : String,
        Message   : Tuple[data:Data, sender:Node, receiver:Node]
  end

```

```

vocabulary JVMSocket
  imports TCPObjectsVoc, TCPNodeVoc
  imports algorithm_voc
  types JVMSocket
  operators
    JVM_TCPsocketOpen      : Node, Node, Nat -> Null[JVMSocket],
    JVM_TCPsocketClose    : JVMSocket -> Null[JVMError],
    JVM_TCPsocketGetLocalIP : Null[JVMSocket] -> Null[Node],
    JVM_TCPsocketGetRemoteIP: Null[JVMSocket] -> Null[Node],
    JVM_read_TCPsocket    : JVMSocket -> Null[Message],
    JVM_write_TCPsocket   : JVMSocket, Message -> Null[JVMError]
end

vocabulary JVMServerSocket
  imports TCPObjectsVoc, JVMSocket, TCPNodeVoc
  types JVMServerSocket
  operators
    JVM_TCPserverSocketOpen : Node, Nat, Nat ->
                                     Null[JVMServerSocket],
    JVM_TCPserverSocketClose : JVMServerSocket -> Null[JVMError],
    JVM_TCPserverSocketAccept: JVMServerSocket -> Null[JVMSocket]
end

vocabulary ChannelVoc
  imports JVMServerSocket, JVMSocket, TCPObjectsVoc, TCPObjectsVoc,
  TCPNodeVoc
  types
    MessageTuple : Tuple [msg:Message, sender:Node, receiver:Node],
    Status        : Enumeration [closed, notAccepting, opening,
                                emptying, connecting, reading,
                                rClosing, sConnected, connected,
                                accepting, waiting, stopping, idle],
    Channel       : Tuple[i:Node, j:Node, socket:Null[JVMSocket],
                          status:Status, emptying:Bool,
error:Null[JVMError]]
  operators
    empty_channel : -> Channel
end

```

B4. RecvMed.tioa

```

imports JVMSocket
imports JVMServerSocket
imports TCPObjectsVoc
imports TCPNodeVoc
imports ChannelVoc
imports algorithm_voc

automaton RecvMed(port:Nat, timeout:Nat)
signature
  input TCP_respRRead
  input TCP_readRError
  input TCP_bind(i:Node)
  input TCP_respAccept
  input TCP_stopListening(i:Node)
  input TCP_rClose(i, j:Node)

  output RECEIVE(m:Null[Message])

```

```

output TCP_rRead
output TCP_respBind(i:Node)
output TCP_accept
output TCP_stopAccepting
output TCP_rCloseStream(i,j:Node)

internal TCP_senderClosing(i,j:Node)

states
  recvBuffer : Seq[MessageTuple] := {};
  recvChannel : Seq[Channel] := {};
  acceptStatus: Status := idle;
  SSocket      : Null[JVMServerSocket] := nil();

let
  getFirst( r , s , i ) : Node, Node, Nat -> Null[Message] =
    if (i > len(recvBuffer)) then nil():Null[Message]
    else if (recvBuffer[i].receiver = r /\
             recvBuffer[i].sender = s) then
      embed(recvBuffer[i].msg)
    else
      getFirst( r , s , i + 1);

transitions

%%
%% emulates the RECEIVE state
%%

%% Two overloaded receive methods that get any message from
%% the receive buffer
%%
output RECEIVE(m) where len(recvBuffer) = 0
pre
  m = nil();

output RECEIVE(m) where len(recvBuffer) ~= 0
pre
  m = embed(head(recvBuffer).msg);
eff
  recvBuffer := tail(recvBuffer);

%-----

%%
%% Bind emulates the BIND and LISTEN socket states
%%
input TCP_bind(i)
eff
  acceptStatus := connecting;

%%
output TCP_respBind(i)
pre
  acceptStatus = connecting;
eff
  SSocket := JVM_TCPServerSocketOpen(i, port, timeout);
  acceptStatus := accepting;

%-----

```

```

%%
%% emulates the ACCEPT state of the socket, however here
%% this is state is decoupled with creation of the stream
%%
output TCP_accept
pre
  acceptStatus = accepting;
eff
  acceptStatus := waiting;

%%
%% a response to the accept is return of a stream
%%
input TCP_respAccept
locals
  socket:Null[JVM_Socket] := nil();
eff
  if acceptStatus = waiting then
    socket := JVM_TCP_ServerSocketAccept(val(SSocket));
    if (socket ~= nil()) then
      recvChannel := recvChannel |-
        [val(JVM_TCP_SocketGetLocalIP(socket)),
         val(JVM_TCP_SocketGetRemoteIP(socket)), socket,
         connected, false, nil()];
      fi
      acceptStatus := accepting;
    fi
  fi

%%
output TCP_stopAccepting
locals
  error:Null[JVM_Error] := nil();
pre
  acceptStatus = stopping;
eff
  acceptStatus := idle;
  error := JVM_TCP_ServerSocketClose(val(SSocket));

%-----

%%
%% prior to entering the CLOSE_SOCKET state we empty
%% the local buffers.
%%
input TCP_stopListening(i)
eff
  if acceptStatus ~= idle then
    acceptStatus := stopping;
  fi

%%
%% emulates CLOSE_SOCKET state.
%% implementation closes the server socket
%%
input TCP_rClose(i,j)
locals
  tempRecvBuffer : Seq[MessageTuple] := {};
eff
  for y:Nat where y < len(recvBuffer) do
    if (recvBuffer[y].sender = i /\ recvBuffer[y].receiver = j)
    then

```

```

        tempRecvBuffer := tempRecvBuffer |- recvBuffer[y];
    fi
od
recvBuffer := {};
for y:Nat where y < len(tempRecvBuffer) do
    recvBuffer := recvBuffer |- tempRecvBuffer[y];
od
for y:Nat where y < len(recvChannel) do
    if (recvChannel[y].i = i /\ recvChannel[y].j = j)
    then
        recvChannel[y].status := rClosing;
    fi
od

%-----

%%
%% sockets provide access to streams, which then are can be read
%% this action emulates the read access to a stream
%%
output TCP_rRead
pre
    len(recvChannel) > 0;
eff
    for y:Nat where y < len(recvChannel) do
        if (recvChannel[y].socket ~= nil() /\
            recvChannel[y].status = connected)
        then
            recvChannel[y].status := reading;
        fi
    od

input TCP_respRRead
locals
    msg:Null[Message] := nil();
eff
    for y:Nat where y < len(recvChannel) do
        if (recvChannel[y].socket ~= nil() /\
            recvChannel[y].status = reading)
        then
            msg := JVM_read_TCPSocket (val(recvChannel[y].socket));
            if (msg ~= nil()) then
                recvBuffer := recvBuffer |- [val(msg),
                    recvChannel[y].i, recvChannel[y].j];
            fi
        fi
    od

input TCP_readRError
eff
    for y:Nat where y < len(recvChannel) do
        if (recvChannel[y].socket ~= nil() /\
            recvChannel[y].status ~= closed)
        then
            recvChannel[y].emptying := true;
        fi
    od

%-----

```

```

output TCP_rCloseStream(i,j)
  locals
    error:Null[JVMError] := nil();
  pre
    len(recvChannel) > 0;
  eff
    for y:Nat where y < len(recvChannel) do
      if (recvChannel[y].i = i /\
          recvChannel[y].j = j /\
          recvChannel[y].status = rClosing /\
          ~recvChannel[y].emptying = true)
        then
          recvChannel[y].status := closed;
        fi
      od
    od

%-----

internal TCP_senderClosing(i,j)
  locals
    noMessages:Bool := true;
  pre
    len(recvChannel) > 0;
  eff
    for y:Nat where y < len(recvChannel) do
      if recvChannel[y].status = emptying then
        for v:Nat where v < len(recvBuffer) do
          if (recvBuffer[v].sender ~= i) then
            noMessages := false;
          fi
        od
      fi
    od
    if noMessages then
      for y:Nat where y < len(recvChannel) do
        if (recvChannel[y].i = i)
          then
            recvChannel[y].status := closed;
          fi
        od
      fi
    fi
  fi

```

B5. SendMed.tioa

```

imports JVMSocket
imports JVMServerSocket
imports TCPObjectsVoc
imports TCPNodeVoc
imports ChannelVoc
imports algorithm_voc

automaton SendMed(port:Nat, timeout:Nat)

signature
  input SEND(m:Message, r,s:Node)

  input TCP_senderOpen(s,r:Node, port:Nat)
  input TCP_senderClose(s,r:Node)

  output TCP_write(m: Null[Message], r,s:Node)

```

```

states
  sendBuffer : Seq[MessageTuple] := {};
  sendChannel : Seq[Channel] := {};
  clocks      : Array[Node, AugmentedReal];
initially
  clocks = constant(timeout);

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

transitions

%%%
%%% SEND simply deposits a message to the channel
%%%
input SEND(m,r,s)
eff
  for y:Nat where y < len(sendChannel) do
    if (sendChannel[y].i = s /\
        sendChannel[y].j = r /\
        sendChannel[y].status ~= closed /\
        sendChannel[y].emptying ~= true)
    then
      sendBuffer := sendBuffer |- [m,r,s];
    fi
  od;

%-----

%%%
%%% Messages to be sent must be written to the channel (analogous
%%% to calling flush() on socket. TCP_write action either returns
%%% null if there are no more messages or the message that was
%%% just sent onto the network medium.
%%%
output TCP_write(m,r,s) where len(sendBuffer) = 0
pre
  m = nil();

output TCP_write(m,r,s) where len(sendBuffer) ~= 0
locals
  tempSendBuffer :Seq[MessageTuple] := {};
  error :Null[JVMError] := nil();
  msg :Null[MessageTuple] := nil();
pre
  sendBuffer ~= {};
  m = getMessage(r,s,0);
eff
  for y:Nat where y < len(sendChannel) do
    if (sendChannel[y].i = s /\ sendChannel[y].j = r)
    then
      for v:Nat where v < len(sendBuffer) do

```

```

    if (sendBuffer[v].sender = s /\ sendBuffer[v].receiver = r) then
      error :=
JVM_write_TCPsocket (val (sendChannel[y].socket), sendBuffer[v].msg);
      clocks[r] := 0;
      tempSendBuffer := tempSendBuffer |- sendBuffer[v];
      if (error ~= nil()) then
        sendChannel[y].error := error;
        print val(error);
      fi
    fi
  od
fi
od
sendBuffer := {};
for y:Nat where y < len(tempSendBuffer) do
  sendBuffer := sendBuffer |- tempSendBuffer[y];
od

%-----

%%
%% models set up of the client socket
%%
input TCP_senderOpen(s,r,port)
locals
  match :Bool := false;
  index :Nat := 0;
  socket:Null[JVMsocket] := nil();
  error :Null[JVMError] := nil();
eff
  for y:Nat where y < len(sendChannel) do
    if (sendChannel[y].i = s /\ sendChannel[y].j = r)
    then
      match := true;
      if (sendChannel[y].socket = nil())
      then
        socket := JVM_TCPsocketOpen(r,s,port);
        sendChannel[y].socket := socket;
      fi
    fi
  od
  if (~match) then
    socket := JVM_TCPsocketOpen(r,s,port);
    sendChannel := sendChannel |- [s, r, socket, opening, false,
nil()];
  fi

%-----

input TCP_senderClose(s,r)
locals
  error:Null[JVMError] := nil();
eff
  for y:Nat where y < len(sendChannel) do
    if (sendChannel[y].i = s /\ sendChannel[y].j = r)
    then
      sendChannel[y].emptying := true;
      error := JVM_TCPsocketClose (val (sendChannel[y].socket));
      if (error ~= nil()) then
        sendChannel[y].error := error;
        print val(error);

```



```

        fi
    fi
od

```

```

%%%
%%% Trajectory added to support timeout
%%%
trajectories
  trajdef v(n:Node)
    invariant len(sendBuffer) ~= 0;
    stop when clocks[n] >= timeout;
    evolve d(clocks[n]) = 1;

```

B6. TCPChan.tioa

```

include "myvocabs.tioa"

imports TCPObjects
imports myVocab

automaton JvmCh
signature
  input TCP_write(m:Message)
  input TCP_rRead
  input TCP_accept(j:Node)
  input TCP_stopAccepting(j:Node)
  input TCP_createStream(i,j:Node)
  input TCP_senderCloseStream (i,j:Node)
  input TCP_receiverCloseStream (i,j:Node)

  output TCP_respRead(m:Message)
  output TCP_writeError(m:Message)
  output TCP_readError
  output TCP_respAccept(i,j:Node)
  output TCP_respCreateStream(i,j:Node)
  output TCP_createStreamError (i,j:Node)

  internal TCP_senderClosingStream (i,j:Node)
states
  jvmBuffer    : Seq[MessageTuple] := {};
  writeErrors  : Seq[MessageTuple] := {};
  sReading     : Seq[Stream]       := {};
  sAccepting   : Seq[Node]         := {};
  jvmChannel   : Seq[Channel]      := {};

transitions

input TCP_write(m,s)
  eff
    for y:Nat where y < len(jvmChannel) do
      if (jvmChannel[y].status = connected \
        jvmChannel[y].status = sConnected)
      then
        jvmBuffer := jvmBuffer |- [m, , jvmChannel[y].j];
      else
        writeErrors := writeErrors |- [m, embed(s),
jvmChannel[y].j];

```

```

        fi;
    od;

input TCP_read(s)
    eff
        sReading := sReading |- s;

input TCP_accept(j)
    eff
        sAccepting := sAccepting |- j;

input TCP_stopAccepting(j)
    locals
        tempAccepting:Seq[Node] := {};
    eff
        for y:Nat where y < len(sAccepting) do
            if (j ~= sAccepting[y]) then
                tempAccepting := tempAccepting |- sAccepting[y];
            fi;
        od;
        sAccepting := {};
        for y:Nat where y < len(tempAccepting) do
            sAccepting := sAccepting |- tempAccepting[y];
        od;

input TCP_createStream(i,j)
    locals
        tempAccepting:Seq[Node] := {};
    eff
        if j \in sAccepting then
            jvmChannel := jvmChannel |- [i,j,nil(),connecting,false];
            for y:Nat where y < len(sAccepting) do
                if (j ~= sAccepting[y]) then
                    tempAccepting := tempAccepting |- sAccepting[y];
                fi;
            od;
            sAccepting := {};
            for y:Nat where y < len(tempAccepting) do
                sAccepting := sAccepting |- tempAccepting[y];
            od;
        else
            for y:Nat where y < len(jvmChannel) do
                if (jvmChannel[y].i = i /\
                    jvmChannel[y].j = j)
                then
                    jvmChannel[y].status := notAccepting;
                fi;
            od;
        fi;

input TCP_senderCloseStream(i,j)
    eff
        for y:Nat where y < len(jvmChannel) do
            if (jvmChannel[y].status = connected /\
                jvmChannel[y].status = sConnected)
            then
                jvmChannel[y].emptying := true;
            else
                jvmChannel[y].status := closed;
            fi;
        od;

```

```

input TCP_receiverCloseStream(i,j)
  locals
    tempJvmBuffer:Seq[MessageTuple] := {};
    tempBool:Bool := false;
    tempNatSeq:Seq[Nat] := {};
  eff
    for y:Nat where y < len(jvmChannel) do
      for z:Nat where y < len(jvmBuffer) do
        if (val(jvmChannel[y].s) = val(jvmBuffer[z].stream) /\
            jvmChannel[y].emptying)
          then
            jvmChannel[y].status := closed;
            tempBool := close_TCPStream(val(jvmChannel[y].s));
            tempNatSeq := tempNatSeq |- z;
          fi;
        od;

      od;

    od;
    for y:Nat where y < len(jvmBuffer) do
      if (y \notin tempNatSeq) then
        tempJvmBuffer := tempJvmBuffer |- jvmBuffer[y];
      fi;
    od;
    jvmBuffer := {};
    for y:Nat where y < len(tempJvmBuffer) do
      jvmBuffer := jvmBuffer |- tempJvmBuffer[y];
    od;

output TCP_respRead(m, s)
  locals
    tempJvmBuffer:Seq[MessageTuple] := {};
    tempReading:Seq[Stream] := {};
  pre
    len(jvmBuffer) ~= 0;
    s \in sReading;
  eff
    for y:Nat where y < len(jvmBuffer) do
      if (jvmBuffer[y].data ~= m /\
          val(jvmBuffer[y].stream) ~= s)
        then
          tempJvmBuffer := tempJvmBuffer |- jvmBuffer[y];
        fi;
      od;
    jvmBuffer := {};
    for y:Nat where y < len(tempJvmBuffer) do
      jvmBuffer := jvmBuffer |- tempJvmBuffer[y];
    od;

    for y:Nat where y < len(sReading) do
      if (tempReading[y] ~= s) then
        tempReading := tempReading |- sReading[y];
      fi;
    od;
    sReading := {};
    for y:Nat where y < len(tempReading) do
      sReading := sReading |- tempReading[y];
    od;

output TCP_writeError(m,s)

```

```

locals
  tempWriteErrors:Seq[MessageTuple] := {};
pre
  len(writeErrors) ~= 0;
eff
  for y:Nat where y < len(writeErrors) do
    if (val(writeErrors[y].stream) ~= s /\
        writeErrors[y].data ~= m)
    then
      tempWriteErrors := tempWriteErrors |- writeErrors[y];
    fi;
  od;
writeErrors := {};
for y:Nat where y < len(tempWriteErrors) do
  writeErrors := writeErrors |- tempWriteErrors[y];
od;

output TCP_readError(s)
  locals
    tempJvmChannel:Seq[Channel] := {};
  pre
    s \in sReading;
  eff
    for y:Nat where y < len(jvmChannel) do
      if (val(jvmChannel[y].s) = s) then
        if (jvmChannel[y].status ~= closed) then
          tempJvmChannel := tempJvmChannel |- jvmChannel[y];
        fi;
      else
        tempJvmChannel := tempJvmChannel |- jvmChannel[y];
      fi;
    od;
  jvmChannel := {};
  for y:Nat where y < len(tempJvmChannel) do
    jvmChannel := jvmChannel |- tempJvmChannel[y];
  od;

output TCP_respAccept(i,j)
  pre
    jvmChannel ~= {};
  eff
    for y:Nat where y < len(jvmChannel) do
      if (jvmChannel[y].i = i /\
          jvmChannel[y].j = j /\
          jvmChannel[y].status = sConnected) then
        jvmChannel[y].status := connected;
      fi;
    od;

output TCP_respCreateStream (i,j,s)
  pre
    jvmChannel ~= {};
  eff
    for y:Nat where y < len(jvmChannel) do
      if (jvmChannel[y].i = i /\
          jvmChannel[y].j = j /\
          jvmChannel[y].status = connecting /\
          val(jvmChannel[y].s) ~= s)
      then
        jvmChannel[y].status := sConnected;
        jvmChannel[y].s := embed(new_TCPStream);
      fi;
    od;

```

```

        fi;
    od;

output TCP_createStreamError(i,j)
pre
    jvmChannel ~= {};
eff
    for y:Nat where y < len(jvmChannel) do
        if (jvmChannel[y].i = i /\
            jvmChannel[y].j = j /\
            jvmChannel[y].status = notAccepting)
        then
            jvmChannel[y].status := closed;
        fi;
    od;

internal TCP_senderClosingStream(i, j)
pre
    len(jvmChannel) > 0;
eff
    for y:Nat where y < len(jvmChannel) do
        if (jvmChannel[y].i = i /\
            jvmChannel[y].j = j /\
            jvmChannel[y].status = closed /\
            jvmChannel[y].emptying)
        then
            jvmChannel[y].status := closed;
            jvmChannel[y].emptying := false;
        fi;
    od;

```

APPENDIX C

Paxos TCP Implementation

C1. TCPPaxos.tioa

```

include "myvocabs.tioa"

imports JVMSocket
imports JVMServerSocket
imports TCPObjectsVoc
imports TCPNodeVoc
imports ChannelVoc
imports paxos_voc

%%% .:TCP mediator automata:.
include "TCPRecvMed.tioa"
include "TCPSendMed.tioa"
include "TCP_ChanMed.tioa"

%%% .:Paxos automata
include "starteralg.tioa"
include "bpleader.tioa"
include "bpagent.tioa"
include "bpsuccess.tioa"

```

```

include "leaderelector.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(n1:Nat, n2:Nat, n3:Nat, n4:Nat, port:Nat,
               timeout:Nat)

  components
    A_starteralg :starteralg([n1,n2,n3,n4],5,timeout,510,500);
    A_detector   :detector([n1,n2,n3,n4],5,timeout,510,500);
    A_bpleader   :bpleader([n1,n2,n3,n4],5,timeout,510,500);
    A_bpagent    :bpagent([n1,n2,n3,n4],5,timeout,510,500);
    A_bpsuccess  :bpsuccess([n1,n2,n3,n4],5,timeout,510,500);
    A_leaderelector:leaderelector([n1,n2,n3,n4]);
    S            :SendMed(port,timeout);
    R            :RecvMed(port,timeout);
    C            :ChanMed(port,timeout);

  schedule

  states
    AtHome :Bool := true;
    myIP   :Node := [n1,n2,n3,n4]; % IP from parameters
    world  :Seq[Node];
    %
    D :AugmentedReal := 0;
    Z :AugmentedReal := 20;
    %
    dummy :Null[Message] := nil();
    ms    :Null[Message] := nil();
    mr    :Null[Message] := nil();
    %
    leaderIP :Node;
    decision :Null[Int] := nil();
    value:Int := 0;
    %
    exitloop :Bool := false;
    dowhile  :Bool := true;
    error    :Null[JVMError] := nil();

    server   :Node := [192,168,2,2];

    % -- especially when variable initialization involves automata
    % parameters it is best to use the initially block
    initially
      D = timeout /\ leaderIP = [n1,n2,n3,n4] /\ world = world |-
[n1,n2,n3,n4];

    % -- begins paxos schedule
    do

      % -- Everyone sets up their server socket.

      % -- bind
      fire output R.TCP_bind(myIP);
      fire output C.TCP_respBind(error,myIP);

```

```

% -- A dedicated server awaits connection requests.
if (myIP = server) then
  while( dowhile = true ) do
    % -- 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
  if (val(error) = "NoConnectionOnAccept") then dowhile := false; fi
  fi
  od
else
  % -- create connections to the server
  fire output S.TCP_senderOpen( server, port );
fi

if (AtHome = true)
then
  if ([192,168,2,2] \notin world) then
    world := world |- [192,168,2,2]; % seed the world
    fire output A_detector.InformAlive( [192,168,2,2]);
  fi
  if ([192,168,2,3] \notin world) then
    world := world |- [192,168,2,3]; % seed the world
    fire output A_detector.InformAlive( [192,168,2,3]);
  fi
else
  if ([136,145,181,12] \notin world) then
    world := world |- [136,145,181,12]; % seed the world
    fire output A_detector.InformAlive( [136,145,181,12]);
  fi
  if ([136,145,181,41] \notin world) then
    world := world |- [136,145,181,41]; % seed the world
    fire output A_detector.InformAlive( [136,145,181,41]);
  fi
fi

%%% Run the leader election protocol. %%%

% -- prep and send alive messages
fire internal A_detector.PrepareAliveMessages;
for y:Nat where y < len(world) do
fire internal A_detector.Check( world[y] );
  %%%fire output A_detector.InformStopped( world[y] );
  fire output A_detector.SEND( ms );
  fire output S.TCP_write( ms, myIP, world[y]);
  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();
  % -- extract messages from channel if there are any
  fire output R.TCP_read;
  dowhile := true;
  while( dowhile = true ) do

```

```

        fire output C.TCP_respRead( mr );
        fire output R.RECEIVE( mr );
        if (mr ~= nil()) then
fire output A_detector.InformAlive( val(mr).sender );
        if (val(mr).sender \notin world) then
            world := world |- val(mr).sender;
        fi
fire output A_detector.InformAlive( val(mr).sender );
        print val(mr);
        else
            dowhile := false;
        fi
    od;
    fire output A_detector.HasEnough( exitloop );
od
% -- locally, announce leader
fire output A_leaderselector.Leader( leaderIP );
print leaderIP;

%%% RUN PAXOS %%%

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;

        % -- prep collect messages
        fire internal A_bpleader.Collect;

        % -- send collect messages
        for y:Nat where y < len( world ) do
            fire output A_bpleader.SEND( ms );
            fire output S.TCP_write(ms, myIP, world[y]);
            if (ms ~= nil()) then print val(ms); fi
            ms := nil( );
        od
follow 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
            fire output C.TCP_respRead( mr );
            fire output R.RECEIVE( mr );
            if (mr ~= nil()) then
                print val(mr);
            else
                dowhile := false;
            fi
        od;
    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
            fire output A_bpleader.SEND( ms );
            fire output S.TCP_write(ms, myIP, world[y]);
            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();

        % -- extract messages from channel if there are any
        fire output R.TCP_read;
        dowhile := true;
        while( dowhile = true ) do
            fire output C.TCP_respRead( mr );
            fire output R.RECEIVE( mr );
            if (mr ~= nil()) then
                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
            fire output A_bpsuccess.SEND( ms );
            fire output S.TCP_write(ms, myIP, world[y]);
            if (ms ~= nil()) then print val(ms); fi
            ms := nil( );
        od;

        follow 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

```

```

        fire output C.TCP_respRead( mr );
        fire output R.RECEIVE( mr );
        if (mr ~= nil()) then
            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_bpagent.v duration \infty();

        % agents collect
        % -- extract messages from channel if there are any
        fire output R.TCP_read;
        dowhile := true;
        while( dowhile = true ) do
            fire output C.TCP_respRead( mr );
            fire output R.RECEIVE( mr );
            if (mr ~= nil()) then
                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_bpagent.LastAccept;
        fire internal A_bpagent.Accept;
        fire internal A_bpsuccess.GatherSuccess;

        % -- send response
        dowhile := true;
        while dowhile do
            fire output A_bpagent.SEND( ms );
        fire output S.TCP_write(ms, myIP, leaderIP);
            if (ms ~= nil()) then
                print val(ms);
                ms := nil();
            else
                dowhile := false;
            fi
        od;

        fire output A_bpsuccess.NextPhase( exitloop );
        od;

        fire internal A_bpsuccess.SendSuccess;
        for y:Nat where y < len( world ) do
            fire output A_bpsuccess.SEND( ms );
            fire output S.TCP_write(ms, myIP, world[y]);
        end
    end
end

```

```

        ms := nil( );
    od;

    fi

    fire output A_bpsuccess.Decide( decision );

    if (decision ~= nil()) then
        print val(decision);
    fi

    fire output R.TCP_read;
    dowhile := true;
    while( dowhile = true ) do
        fire output C.TCP_respRead( mr );
        fire output R.RECEIVE( mr );
        if (mr ~= nil()) then
            print val(mr);
        else
            dowhile := false;
        fi
    od;

od

```

C2. myvocabs.tioa

```

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],
    JVMEError : String
  end

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

%%% ..Paxos vocabs:.
vocabulary paxos_voc
  imports TCPObjectsVoc
  imports TCPNodeVoc
  types NodeMode : Enumeration [live, stopped, begin, last, accept,
    success, oldround,
                                collect, gatherlast,
    wait, beginicast, gatheraccept,
                                decided, rnddone, ack],
    Round      : Tuple [C:Int, O:Node],
    Data       : Tuple [M:NodeMode, R:Round, RP:Round, V:Int],
    Message    : Tuple[data:Data, sender:Node, receiver:Node],
    Mode       : Enumeration[done,working,leader,notleader]
  end

```

```

vocabulary JVMSocket
  imports TCPObjectsVoc, TCPNodeVoc
  imports paxos_voc
  types JVMSocket
  operators
    JVM_TCPSocketOpen      : Node, Nat, Nat -> Null[JVMSocket],
    JVM_TCPSocketClose     : JVMSocket -> Null[JVMError],
    JVM_TCPSocketGetLocalIP : Null[JVMSocket] -> Null[Node],
    JVM_TCPSocketGetRemoteIP : Null[JVMSocket] -> Null[Node],
    JVM_read_TCPSocket     : Null[JVMSocket] -> Null[Message],
    JVM_write_TCPSocket    : JVMSocket, Message -> Null[JVMError],
    JVM_TCPSocketIsConnected : Null[JVMSocket] -> Bool
end

vocabulary JVMServerSocket
  imports TCPObjectsVoc, JVMSocket, TCPNodeVoc
  types JVMServerSocket
  operators
    JVM_TCPServerSocketOpen : Node, Nat, Nat ->
Null[JVMServerSocket],
    JVM_TCPServerSocketClose : JVMServerSocket -> Null[JVMError],
    JVM_TCPServerSocketAccept : 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 JVMServerSocket, JVMSocket, TCPObjectsVoc, TCPObjectsVoc,
TCPNodeVoc
  types
    MessageTuple : Tuple [msg:Message, sender:Node, receiver:Node],
    Status       : Enumeration [closed, notAccepting, opening,
emptying,
                                connecting, reading, rClosing, sConnected,
connected,
                                accepting, waiting, stopping, idle],
    Channel      : Tuple[node :Node, socket:Null[JVMSocket],
status:Status, emptying:Bool,
error:Null[JVMError]]
  operators
    empty_channel : -> Channel
end

```

C3. TCP_ChanMed.tioa

```

automaton ChanMed(port :Nat, timeout :Nat)
signature
  % -- actions paired with the RecvMed
  input  TCP_read
  output TCP_respRead(m :Null[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_rCloseStream(remote :Node)

% -- actions paired with the SendMed
input TCP_senderOpen(remote :Node, port :Nat)
input TCP_senderClose(remote :Node)
input TCP_write(m: Null[Message], s,r :Node)

% -- internal actions
internal TCP_senderClosing(remote :Node)

% -- universal - only reports partial error information
%   per each established connection
output TCP_getError(e :Null[JVMError], remote :Node)

states
% -- server socket
SSocket      : Null[JVMServerSocket] := nil();
% -- current status of the server socket
acceptStatus : Status := idle;
% -- error (if any) from the last operation on server socket
SError       : Null[JVMError] := nil();
% -- error (if any) on the last accept attempt
AError       : Null[JVMError] := nil();
% -- a list of all established connections
tcpChannel   : Seq[Channel] := {};
% -- a buffer for network extracted messages
recvBuffer   : Seq[Message] := {};

let
% -- searches all established connections and returns an
%   error (if any) by the last operation on that connection
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);

transitions

%%%%%%%%%%%%% READ %%%%%%%%%%%%%%
% -- goes through connected sockets and sets their
%   status to reading
input TCP_read
locals
  msg : Null[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_TCPSocket(tcpChannel[n].socket);
      if (msg = nil()) then
        tcpChannel[n].error := embed("TimeoutOnRead");
      else
        recvBuffer := recvBuffer |- val(msg);
      fi
    fi
  fi

```

```

    od

% -- reads a message from the first reading socket
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

%%%%%%%%%%%%% BIND to SERVER SOCKET %%%%%%%%%%%%%%

% -- bind emulates the BIND and LISTEN socket states
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;
eff
  if (SSocket ~= nil()) then
    acceptStatus := accepting;
  fi

%%%%%%%%%%%%% ACCEPT %%%%%%%%%%%%%%

% -- emulates the ACCEPT state of the socket, however here
%   this is state is decoupled with creation of the stream
input TCP_accept
locals
  socket:Null[JVM_Socket] := nil();
  found :Bool := false;
eff
  if (acceptStatus = accepting) then
    acceptStatus := waiting;
    socket := JVM_TCPServerSocketAccept(val(SSocket));
    if (socket ~= nil()) then
      for n:Nat where n < len(tcpChannel) /\ ~found do

```

```

        if ( tcpChannel[n].node =
val (JVM_TCPSocketGetRemoteIP(socket)) )
        then
            found := true;
            if (tcpChannel[n].status ~= connected /\
                ~JVM_TCPSocketIsConnected( tcpChannel[n].socket
))
            then
                tcpChannel[n].socket := socket;
                tcpChannel[n].status := connected;
                tcpChannel[n].emptying := false;
            fi
        fi
    od
    if (found = false /\ JVM_TCPSocketIsConnected( socket ))
    then
        tcpChannel := tcpChannel |-
[val (JVM_TCPSocketGetRemoteIP(socket)), socket, connected, false,
nil()];
    else
        AError := embed("NoConnectionOnAccept");
    fi
    else
        AError := embed("NoConnectionOnAccept");
    fi
    fi

% -- a response to the accept is return of a stream
output TCP_respAccept(error)
pre
    error = AError;
eff
    if acceptStatus = waiting then
        acceptStatus := accepting;
        AError := nil();
    fi

% --
input TCP_stopAccepting
locals
    error:Null[JVMError] := nil();
eff
    if (acceptStatus = stopping) then
        acceptStatus := idle;
        error := JVM_TCPServerSocketClose( val(SSocket) );
        if (error ~= nil())
        then
            print error;
        fi
    fi

%%%%%%%%%%%%% CLOSING %%%%%%%%%%%%%%

% -- prior to entering the CLOSE_SOCKET state we empty
% the local buffers.
input TCP_stopListening(remote)
eff
    if acceptStatus ~= idle then
        acceptStatus := stopping;
    fi

```

```

% -- emulates CLOSE_SOCKET state.
input TCP_rClose(remote)
locals
  tempRecvBuffer : Seq[MessageTuple] := {};
eff
  for y:Nat where y < len(tcpChannel) do
    if (tcpChannel[y].node = remote)
    then
      tcpChannel[y].status := rClosing;
    fi
  od

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

%%%%%%%%%%%%%% ERROR %%%%%%%%%%%%%%%

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

%%%
%%% ACTIONS PAIRED WITH THE SEND MEDIATOR
%%%

% -- given a message with sender and receiver as s and r
respectively
%   a message is written to the appropriate channel

```



```

input TCP_write(m,s,r)
locals
  error :Null[JVMError] := nil();
  found :Bool := false;
eff
  for n:Nat where (n < len(tcpChannel)) /\ (found = false) do
    if (tcpChannel[n].socket = nil())
    then
      tcpChannel[n].status := closed;
    fi
    if (tcpChannel[n].socket ~= nil() /\
        tcpChannel[n].status = connected /\
        tcpChannel[n].node = r)
    then
      found := true;
      error := JVM_write_TCPSocket(val(tcpChannel[n].socket),
val(m));
      if (error ~= nil()) then
        tcpChannel[n].error := error;
        print val(error);
      fi
    fi
  od

% -- results in a connection being established (if possible)
%   with the remove with specified port.
input TCP_senderOpen(remote,port)
locals
  match :Bool := false;
  index :Nat := 0;
  socket:Null[JVMSocket] := nil();
  error :Null[JVMError] := nil();
eff
  for n:Nat where n < len(tcpChannel) do
    if (tcpChannel[n].node = remote)
    then
      match := true;
      if (tcpChannel[n].socket = nil() /\
          tcpChannel[n].status = closed)
      then
        tcpChannel[n].socket :=
JVM_TCPSocketOpen(remote,port,timeout);
      fi
    fi
  od
  if (match = 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

% -- results in closing of the connection between s and r
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

```

C4. TCPRecvMed.tioa

```

automaton RecvMed(port:Nat,timeout:Nat)
signature
  % -- delivers a message to the algorithm automaton
  output RECEIVE(m:Null[Message])

  % -- initiates read on all open connections
  output TCP_read
  % -- returns messages from all open connections (if any)
  input  TCP_respRead(m :Null[Message])

  % -- binds system to the server socket
  output TCP_bind(local :Node)
  input  TCP_respBind(error :Null[JVMError], local :Node)

  % -- accept connection methods
  output TCP_accept
  input  TCP_respAccept(error: Null[JVMError])
  output TCP_stopAccepting

  % -- extracts the last error for a given connection
  input  TCP_getError(e :Null[JVMError], remote :Node)

  % -- close and clean up
  output TCP_stopListening(remote :Node)
  output TCP_rCloseStream(remote :Node)
  output TCP_rClose(remote:Node)

states
  recvBuffer  : Seq[Message] := { };
  recvErrors  : Map[Node,Null[JVMError]];
  remoteStatus: Map[Node,Status];
  localStatus : Status := idle;
  localError  : Null[JVMError] := nil();
  noop       : Bool := true;

transitions

%%%%%%%%%%%%% DELIVER %%%%%%%%%%%%%%%

% -- delivers message to the algorithm automata
output RECEIVE(m) where len(recvBuffer) = 0
pre
  m = nil();

output RECEIVE(m) where len(recvBuffer) ~= 0

```

```

pre
  m = embed(head(recvBuffer));
eff
  recvBuffer := tail(recvBuffer);

%%%%%%%%%%%%% EXTRACT %%%%%%%%%%%%%%

% -- send request to read a message from a remote node
output TCP_read
pre
  localStatus ~= idle;
%eff
  %% @FIXME: this should be acceptable by the FE, but it is not!
  %recvErrors := {};

% -- if there was a message then add it to the message buffer
input TCP_respRead(m)
eff
  if (m ~= nil())
  then
    recvBuffer := recvBuffer |- val(m);
  fi

%%%%%%%%%%%%% BIND %%%%%%%%%%%%%%

% -- bind emulates the BIND and LISTEN socket states
output TCP_bind(local)
pre
  localStatus = idle;
eff
  localStatus := connecting;
  localError := nil();

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

%%%%%%%%%%%%% ACCEPT %%%%%%%%%%%%%%

% -- emulates the ACCEPT state of the socket, however here
% this state is decoupled with creation of the stream
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
    true;
eff
    noop := true;

% -- prior to entering the CLOSE_SOCKET state we empty the local
buffers.
output TCP_stopListening(remote)
pre
    true;
eff
    noop := true;

% -- emulates CLOSE_SOCKET state. implementation closes the server
socket
output TCP_rClose(remote)
pre
    true;
eff
    noop := true;

% --
output TCP_rCloseStream(remote)
pre
    true;
eff
    noop := true;

% -- get an error message associated with the remote connection
input TCP_getError(e, remote)
eff
    if (e ~= nil())
    then
        recvErrors := update(recvErrors, remote, e);
        print val(e);
    fi

```

C5. TCPRecvMed.tioa

```

automaton SendMed(port:Nat, timeout:Nat)

signature
% -- deposits a message from the algorithm automata
input SEND(m:Null[Message])

% The following actions are paired with the ChanMed

% -- creates a connection with the remote node
output TCP_senderOpen(remote:Node, port:Nat)
% -- closes a connection with the remote node
output TCP_senderClose(remote:Node)
% -- forces a message out to the network
output TCP_write(m: Null[Message], s,r:Node)

states
    sendBuffer : Seq[MessageTuple] := { };

```

```

clocks      : Array[Node, AugmentedReal];

initially
  clocks = constant(\infty);

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

transitions

  % -- deposits a message to the send buffer
  input SEND(m)
  eff
    if (m ~= nil()) then
      sendBuffer := sendBuffer |-
[val(m), val(m).sender, val(m).receiver];
    fi

  % -- results in opening of a connection with the remote node
  output TCP_senderOpen(remote, port)
  pre
    true;

  % -- results in closing of a connection with the remote node
  output TCP_senderClose(remote)
  pre
    true;

  % -- TCP_write a message to the channel.  this results in
  %   the TCP_ChanMed to write it into the TCP socket.
  output TCP_write(m, s, r) where len(sendBuffer) = 0
  pre
    m = nil();

  output TCP_write(m, s, r) where len(sendBuffer) ~= 0
  locals
    tempSendBuffer : Seq[MessageTuple] := { };
    msg : Null[Message] := nil();
  pre
    m = getMessage(s, r, 0);
  eff
    msg := getMessage(s, r, 0);
    if (msg ~= nil()) then
      for n:Nat where n < len(sendBuffer) do
        if (sendBuffer[n].msg = val(msg)) then
          clocks[r] := 0;
        else
          tempSendBuffer := tempSendBuffer |- sendBuffer[n];
        fi
      od
      sendBuffer := tempSendBuffer;
    fi
  fi

```

```
%-----  
  
%%%  
%%% Trajectory modeling the delay needed for a message to be  
%%% delivered to the remote node.  
%%%  
trajectories  
  trajdef v(n:Node)  
    invariant len(sendBuffer) ~= 0;  
    stop when clocks[n] >= timeout;  
    evolve d(clocks[n]) = 1;
```