On the Automated Implementation of Time-based Paxos Using the IOA Compiler *

Chryssis Georgiou† Procopis Hadjiprocopiou‡ Peter M. Musial§

Abstract

Paxos is a well known algorithm for achieving consensus in distributed environments with uncertain processing and communication timing. Implementations of its variants have been successfully used in the industry (eg., *Chubby* by Google, *Autopilot cluster management* in Bing by Microsoft, and many others). This paper addresses the challenge of the manual coding of complex distributed algorithms, such as Paxos, where this is an error prone process. Our approach in ensuring correct implementation is to use a verified automated translator to compile a source specification that has been proven to be itself correct. We use specification of the Paxos algorithm in the *General Timed Automata* (GTA) model, an extension of I/O Automata, as input to an augmented compiler for the *Input/Output Automata notation* (a.k.a., the IOA compiler) in order to generate executable Java code. The resulting code is interfaced with MPI for communication needs. We have extended the IOA compiler to support a version of the GTA model, which uses time-passage actions such as $\nu(t)$, to model the passage of time by t time units. A time-based version of Paxos is used to demonstrate the capabilities of our extension. In this paper we describe the process to be followed in order to compile time-based Paxos, or similar algorithms. The utility of our approach is supported by an experimental evaluation of our Paxos implementation on a collection of workstations. To the best of our knowledge, our case study constitutes the first example of a time-dependent distributed algorithm that has been specified, verified and implemented in an automated way, using a common formal methodology.

1 Introduction

Reasoning about the behavior of complex distributed systems and algorithms is a challenging task. Over the years, several formal methodologies for specifying distributed systems have been proposed and associated techniques and tools have been developed for verifying such systems (e.g., [3, 17, 20, 27, 22, 28]). However, the benefits of using formal methods has not reached its full potential due to the remaining challenge of implementing such systems; usually the programmer has to manually map the functionality of the abstract specification to detailed programs in order to be executed on target distributed platforms. This raises the question whether the correctness of the abstract specification is maintained during the coding process. To this respect, some tools have been developed in an attempt to provide automated simulation or implementation of formally specified code (e.g., [2, 6, 7, 1, 24]). To the best of our knowledge, the IOA Toolkit [1] is the only system to date that combines a language with formally specified semantics (IOA language and checker), automated proof assistants (IOAtoLR theorem prover), simulator (IOA simulator) *and* compiler (IOAtoJava code generator). A number of *asynchronous* algorithms, specified and proved correct using the IOA framework, have been successfully implemented in an automated way using the IOA code generator (see [14, 30, 15]); the generator translates the IOA specification of a given algorithm to Java code which then can be executed on a network of workstations, where communication is established using MPI [10]. However, before our work, the IOA code generator did not support timing issues.

Existing distributed systems can be viewed as *partially synchronous systems* in the sense that some bounds on processes computation time and messages delays can be estimated and be assumed, but cannot be guaranteed to hold at all times; that is, these bounds might be violated, leading to *timing failures*. Moreover, implementations of algorithms and programs on such systems usually make use of timeouts in order to render some progress of the computation (that is, to provide some liveness guarantees) and to detect component failures. Being able to specify, prove correct and

*This work is supported in part by research funds from the University of Cyprus and the University of Puerto Rico Rio Piedras.

†Department of Computer Science, University of Cyprus, CY-1678 Nicosia, Cyprus. Email: chryssis@cs.ucy.ac.cy

‡Department of Computer Science, University of Cyprus, CY-1678 Nicosia, Cyprus. Email: cs02cp2@cs.ucy.ac.cy

§Department of Computer Science, University of Puerto Rico Rio Piedras, USA. Email: peter.musial@uprrp.edu

automatically implement such algorithms on a real distributed system using a common formal methodology is the focus of this work.

We have extended the IOAtoJava code generator (or simply IOA compiler) to handle actions modeling passage of time. More specifically, we have extended the IOA compiler (including the IOA syntax checker and IOA composer) to support a version of the General Timed Automata (GTA) model, a timed I/O Automaton model introduced by Lynch and Vaandrager [26]. To demonstrate the functionality of this extension, we used a timed specification of Paxos algorithm [21] as an input to the augmented compiler. The GTA model provides a systematic way of describing the timing behaviors of partially synchronous distributed systems subject to timing failures. The model (and variations of it) can be used for the study of the performance and fault-tolerance analysis (i.e., the liveness) of practical distributed systems under stabilization conditions (see for example the work in [8]). Lynch and Shvartsman [25] produced a GTA-based specification of a timed version of the Paxos algorithm, they proved its correctness (safety) and performed a latency analysis conditioned on certain timing and failure assumptions. The proof presented in [25] was checked to be correct using the interactive IOAtoLR theorem prover by Win and Immorlica in [18] (see also [31]). The specification we used to produce an automated implementation of Paxos using our extended version of the IOA compiler was based on the one in [25].

The rest of the paper is organized as follows. Section 2 overviews the I/O Automata and GTA models as well as the IOA notation and compiler. Also the Paxos algorithm is discussed. In Section 3 we present an in-depth analysis of the procedure for compiling and executing Paxos. Experimental results obtained by implementing Paxos on a network of workstations are presented in Section 4. We conclude in Section 5.

2 Background

In this section we provide the necessary background required in the remainder sections.

2.1 I/O Automata and the GTA Models

The I/O Automata framework was introduced by Lynch and Tuttle in [27]. A detailed description of this model can be obtained there and in [23, Chapter 8]. An I/O Automaton is a labeled state machine in which a set of transitions connects the actions with the states. It entails a set of states (not necessarily finite) with a nonempty subset of start states, a transition relation, and a set of *actions*. These actions are classified as *input*, *output* and *internal*. The utilization of input and output actions enables the communication of an automaton with its environment. Input actions are controlled by the external environment, whereas internal and output transitions are controlled by the automaton. Actions are given in a precondition-effect style. An action is said to be *enabled* if its preconditions are satisfied. Input actions are always enabled. A transition (also called a step) is given in the form (s,π,s') where s, s' are states and π an action. I/O Automata support the operation of (parallel) *composition* where automata can be combined to form a larger, multifunctional automaton representing a complicated distributed system. The I/O Automata model is *nondeterministic* since in any given state any number of actions may be enabled and there are no restrictions on when an enabled action should be performed.

The GTA model of Lynch and Vaandrager [26] (see also [23, Chapter 23]) is a variant of the I/O Automata model that enables the modeling of timing restrictions. These restrictions can be encoded directly into the states and transitions of the automaton. In addition to input, output and internal actions, a GTA uses *time-passage* actions to model the passage of time. In particular, an action $\nu(t)$ of type time-passage specifies the passage of time by t time-units, $t \in \Re^+$. Like internal and output actions, time-passage actions are also controlled by the automaton. Unlike I/O automata, GTAs do not have *tasks* components. Hence, a GTA is composed of four components: (*i*) the signature which contains the input, output, internal and time-passage actions, (*ii*) a set of states, (*iii*) a set of initial states, and (*iv*) the state-transition relation (steps). The GTA model supports the composition of automata similarly to the I/O automaton model. Particularly, a composition of (compatible) GTA automata yields a GTA automaton.

For the purposes of this work, we consider a *free version* of GTA [25] which is similar to the concept of Clock GTA as introduced by De Prisco [8]. In particular, if an automaton A is a GTA, then the free version of A (denoted by $free(A)$ in [25]) is a variant of A that behaves like A, except that it relaxes time constraints by allowing any amount of time to pass in situations where A specifies that a particular amount of time should pass. This enables our extended version of the IOA compiler to handle situations in which the exact time constraints are not met by the program (e.g., due to unexpected processing and communication delays).

2.2 The IOA Language and the IOA Compiler

The IOA notation is a language used to describe I/O Automata, and can be used both as a formal specification language and a programming language [11]. States are described by the means of the values of variables and transitions in precondition-effect style, instead of state-action-state triples. Preconditions and parameters of the transition must hold whenever this action is executed. The IOA language supports axiomatic and operational descriptions of programing implementations. The language inherits the nondeterministic nature of the I/O Automata model. The IOA notation is supported by the IOA Toolkit [1] via a sequence of tools, such as the checker, the simulator, the theorem prover, and also the compiler. The compiler translates IOA code into Java code.

It was proven that a restricted set of source IOA specifications [30] can be compiled to executable Java code while preserving the *safety* properties of the source specification. To name few such restrictions, specifications must be presented in a node-channel form (discussed next), and specifications must be input delay insensitive. As noted in [14] and [30], a challenging problem (which remains open) is to enable the code generator to also provide some kind of *liveness* guarantees.

Let us now turn our attention to Paxos. To be suitable for compilation, the Paxos specification must be in the node-channel form. Meaning, the algorithm will have two components: First, modeling algorithm code being executed on each network location (or *algorithm automaton*), such as ballot preparation, voting, and reaching the consensus decision. Second, modeling communication channels (or *channel automata*) between different network locations. During the specification phase such channels will be abstract, but with specific safety properties (ex., lossy, reliable, secure, etc.). For the moment let us assume that a node-channel representation of Paxos exists.

Unfolded below is a high level description of the procedure required for the compilation and execution of Paxos. A detailed, algorithm-independent, step-by-step description of the compilation procedure can be found in [14]. We start with the syntactically correct IOA specification of Paxos (described in detail in the next section) in the node-channel form, which can be verified using the IOA *checker*.

Next step is to replace the abstract communication channel with a specific implementation. In our case communication is implemented using the Message Passing Interface (MPI) [10], which is supported by the IOA compiler. The MPI channel is modeled as a *channel automaton* that is a composition of *SendMediator* and *ReceiveMediator* automata. These automata provide the linking to the MPI native libraries and an appearance of interfacing with the abstract channel. All communication between nodes in Paxos is modeled as point-to-point connections. Note that the use of MPI with the Paxos specification does not affect the safety properties of the specification. Preserving the liveness properties, as mentioned above, remains an open challenge. However, our experiments do suggest that under the scenarios considered, the use of MPI does not fault executions of Paxos.

Before the specification is fed to the compiler, additional annotations must be given to resolve nondeterminism. The nondeterminism, inherent from the IOA model, is resolved by requiring the programmer to write a *schedule*. A schedule is a function of the state of the local node that picks the next action to execute at the node. That is, the schedule function selects the next enabled transition as well as the values of its parameters and operates the effects of that transition. In format, a schedule is written at the IOA level in an auxiliary non-determinism resolution language (NDR) consisting of imperative programming constructs similar to those used in IOA effects clauses. Therefore, we developed a (non-trivial) schedule appropriate for Paxos which is contained in Figure 10.

The following steps are independent of input specification. The *composite node automaton* is described as the composition of the algorithm automaton with the channel mediator automata. A *composer* expands this composition into a new, equivalent IOA program in primitive where each piece of the automaton is explicitly instantiated. The resulting *automaton* is annotated with the schedule that describes sequence of computations per each node. The automaton along with its schedule is the final input program to the compiler. The *composite node automaton* augmented with a schedule is now ready for compilation. All the nodes in the system differ in parameterization and input. A common information can be provided to the nodes through the automaton parameters just before the execution of the system. The rank of each node *MPIrank*, described as a unique non-negative integer, is provided by MPI. Another operator supported by MPI is the *MPIsize* which records the number of nodes in the system. The compiler translates each scheduled node automaton into its own Java program suitable to run on the target host.

2.3 The Paxos Algorithm

Reaching *consensus* is a fundamental problem in distributed systems. The consensus problem addresses the situation in which there is a set of n processes; each process can propose a value, but in order for the system to reach a

consensus state, every process must decide on the same value. In particular three conditions must hold: (a) *Agreement*, all (correct) processes agree on the same value. (b) *Validity*, the agreed value was among the ones proposed by the processes. (c) *Termination*, eventually each (correct) process decides. The first two conditions are *safety* conditions, that is, they must hold at all times. The third one is a *liveness* condition and it can only be met under certain constraints (e.g., it is well known that consensus cannot be solved in a purely asynchronous systems in the presence of a single process crash failure [9]). Distributed consensus has been extensively studied under various system and failure models, see e.g., [23, 4].

Paxos is an algorithm designed to solve the consensus problem. It was presented by Lamport in 1990 and was published in 1998 [21]. A considerable advantage of this popular algorithm is that it tolerates processes crashes (and recoveries), message loss, duplication and reordering as well as timing failures. Paxos is guaranteed to work safely (that is, it satisfies agreement and validity) regardless of process, channel and timing failures. When the distributed system stabilizes (that is, there are no failures and a majority of the processes are not crashed, for a long period of time), termination is also achieved [8].

Description of Paxos. In brief, Paxos works as follows: a leader starts ballots, tries to associate a value to each ballot, and tries to collect enough approval for each ballot to use the value of that ballot as the decision value. The leader bases its choice of a value to associate with a ballot on the information returned by a quorum of processes[1]. Once the value is associated with the ballot, the leader tries to collect approval from a quorum of processes: if it succeeds, the ballot's value becomes the final consensus decision value. In general, several leaders may operate at the same time and may interfere with each other's work. However, under a stable state only one leader operates and ensures that a ballot completes. We now outline the main phases of Paxos.

(1) The leader starts a new ballot and informs the others about it.

(2) A process that learns about the new ballot abstains from any earlier ballot for which it has not voted for. In response, a process replies to the leader with the value of the ballot for which it last voted for.

(3) Once the leader receives responses from a quorum, it chooses a value for the ballot that is based on the received values and announces that value to others.

(4) A process that learns about a new value may vote for the ballot, if it has not already abstained. If the process votes, then it informs the leader and others about its vote.

(5) The leader decides on the ballot's value once it receives messages from a quorum with a vote for that value. In case that the leader has failed, a separate leader election service is used to elect a new one. Timeouts are used to determine which processes are operational, and among these, the one with the highest id is elected as the leader. After the election, the new leader starts a new ballot.

(6) Timeouts are also used for the leader to decide when it should start new ballots (that is, there is a limit on how long it takes for a given ballot to be accepted by a quorum of processes).

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

Specification and Correctness of Paxos. A manuscript by Lynch and Shvartsman [25] provides a formal presentation of the Paxos algorithm. The presentation includes a General Timed Automata specification of the algorithm, a correctness proof (safety) and a performance analysis. The correctness proof, which ensures the agreement and validity properties, was done by hand and it is based on a mapping to an abstract state machine representing a non-distributed version of the algorithm. The performance analysis proves latency bounds, conditioned on certain timing and failure assumptions.

In [18, 31] using a time-free version of the Paxos specification of [25] (essentially the last two timing-dependent phases were not considered), and using the IOA2LSL translation tool of the IOA toolkit, the safety of Paxos was mechanically checked. More precisely, it has been shown that every possible externally observable outcome of the Paxos algorithm is also an externally outcome of a general consensus specification. That is, a forward simulation relation from the Paxos automaton to the consensus automaton was defined. Furthermore, the automata and forward

[1]Quorums are sets of processes such that each quorum has a pairwise intersection with any other quorum. Majorities are special cases of quorums.

simulation conjecture were translated into a readable form by the Larch Prover [12] using an automated translation by the IOA2LSL Tool of the IOA toolkit.

It is worth mentioning that Musial [29] has also translated a version of the Paxos specification of [25] to Java code. The communication medium used was Java Sockets with TCP (instead of MPI) but the translation was done in a manual manner (as opposed to the automated translation offered by the IOA Toolkit). It is also worth mentioning that work is underway in enabling the IOA compiler to also use Java Sockets and TCP [16].

In [5] a deconstruction of (untimed) Paxos into two main abstractions, register and leader, is presented. The eventual register abstraction encapsulates the safety properties of Paxos whereas the eventual leader election abstraction encapsulates its liveness. The IOA Paxos specification presented in this paper (see next section) makes a similar deconstruction of (timed) Paxos: safety is encapsulated via a Paxos Process automaton and liveness via a Ballot Trigger automaton (which includes a leader election module).

3 Implementation of the Paxos Algorithm

3.1 Extending the IOA Compiler

In order to implement Paxos, we had to extend the IOA toolkit to support timing issues. In particular, we had to enable the checker, composer and compiler to support the free variant of the GTA model (discussed in Section 2.1). Recall that GTA, besides the action types input, output and internal of IOA, also requires a fourth action type, that of time-passage (that specifies the passage of time). Introducing this new action type was a non-trivial task which involved making several changes and adjustments to various parts of the checker, composer and compiler code.

In addition, for the successful implementation of the time-based Paxos we implemented a set of operators and data types. Each IOA data type is implemented by a hand-coded Java class. A library of such classes for the standard IOA data types is included in the compiler. Each IOA data type (e.g., `Set[]`) and operator (e.g., `Set[]`$\rightarrow$ `Nat`) is matched with its Java implementation class using a data type registry [30], which we extended in this work. Examples of operators that we have developed and included in the compiler to support the implementation of Paxos are (their usage is shown in later sections): *maxElement, maxBallot, getprocid, getseqno, setBallot, allessdead, ifProposed,existVal, valProposed, notnil, internalDecideOp, timePsg, ifmajv* and *votedBallot*. The Java code for some of the operators and data types are depicted in the Appendix.

Recall that in [30] it was shown that the IOA compiler preserves the safety properties of the source IOA code (the specification of the algorithm to be implemented). As the safety properties are not affected by timing issues, it follows that the Java code generated by our extended version of the compiler for Paxos preserves the safety properties of the source GTA specification. As already mentioned, preserving some liveness guarantees in an automated manner is an open research question [14, 30].

Although in this work we have focused on Paxos, we believe that our extended version of the IOA compiler (including checker and composer) can be used for the automated implementation of other timing-dependent distributed algorithms where their computational progress relies on timeouts, and which adhere to the aforementioned restrictions imposed by the IOA compiler.

3.2 Procedure

The compilation steps of the time-based Paxos specification are as outlined in Section 2.2, where instead of using the IOA compiler we used our developed extended version (that supports the free variant of the GTA model).

3.2.1 Paxos Specification.

Our Paxos specification is based on the one given in [25], but it had to be expressed in the IOA notation suitable for compilation. In addition we had to develop several auxiliary operators and data structures. The specification includes two automata: the *PaxosProcess* and *BallotTrigger*. The former implements the first four main phases of Paxos as outlined in Section 2.3 whilst the two last (timing-dependent) phases are implemented by the latter. Note that for simplicity of presentation we used majorities instead of quorums. We present the specification of each automaton along with the new operators and data structures we have developed. Each automaton specification was syntactically checked using our updated version of the IOA checker.

```
Signature
Input:
 init (const MPIrank:Int, vInit:Int)
 fail (const MPIrank:Int)
 newBallot(const MPIrank:Int)
 RECEIVE(m:Message,const MPIrank:Int,u:Int)
Output:
 decide(const MPIrank: Int, vDecide:Int)
 SEND(m:Message ,const MPIrank:Int ,u:Int)
 assignVal(const MPIrank:Int,
       bAssignVal:Ballot, vAssignVal:Int)
 makeBallot(const MPIrank: Int,bMakeBallot:Ballot)
States:
mode:ModeType := idle
proposed:Array[Int,Set[Int]]:=constant({})
failed:Bool := false
ballots:Set[Ballot]:={}
val:Array[Int,Array[Ballot,Null[Int]]] :=
    constant(constant(nil))
voted:Array[Int,Array[Int,Set[Ballot]]] :=
     constant(constant({}))
abstained:Array[Int,Array[Int,Set[Ballot]]] :=
        constant(constant({}))
doMakeBallot:Array[Int,Bool]:=constant(false)
succeeded:Array[Int,Set[Ballot]]:=constant({})
done:Array[Int,Bool] := constant(false)
neighbours:Set[Int] := {}
tempnghbrs:Array[Ballot,Set[Int]] := constant({})
rcvBallots:Set[Ballot] := {}
sendVote:Bool := false
readyAssign:Bool := false
ballotsucceeded:Ballot := setBallot(-1,-1)
queueOut:Map[Link,Seq[Message]]
queueIn:Map[Link,Seq[Message]]
lnks:Set[Link] := {}
```

```
Internal:
 abstain (const MPIrank:Int, BAbstain:Set[Ballot])
 vote (const MPIrank:Int, bVote:Ballot)
 internalDecide(const MPIrank:Int,
       bInternDecide:Ballot)
 valueDecision(const MPIrank:Int, You:Int,
       LatestVal:Int, ballot:Ballot)
 gossip(const MPIrank:Int)
TimePassage:
 v(T)


seqNo:Int := 0
lastProposedBallot:Ballot := setBallot(-1,-1)
lastvotedvalue:Int := -1
lastValue:Array[Ballot,Set[Last]]
leader:Int := -1
assignvalue:Int := -1
tempLast:Last
tempVal:Int
tempballot:Ballot
tempbalDecide:Ballot
nodes:Set[Int]
countVote:Int := 0
balvalsucc:Int := -1
Clock:Real := 0
nextGossipTime:Real := 0
period:Real
T:Real
mProposed:Int := -1
mBallots:Ballot := setBallot(-1,-1)
mVal:Int := -1
mVoted:Ballot := setBallot(-1,-1)
mAbstained:Set[Ballot] := {}
```

Figure 1: *PaxosProcess(i): Signature and State variables*

PaxosProcess Automaton. Figure 1 shows the signature and the state variables of the *PaxosProcess(i)* automaton. The analysis of the new data types and operators follows in this section. Figure 2 shows the transitions of actions *init*, *newBallot* and *makeBallot*.

```
input init(i,vInit)
eff
 if ¬failed then
  if(mode=idle) then
   mode := active;
   Clock := clock;
   proposed[i] := proposed[i] ∪ {vInit};
   for k:Int in nodes-{i} do
    queueOut[[i,k]] := queueOut[[i,k]] ⊢
       sProposed([PROPOSED,[i,k],vInit]);
   od;
   mProposed:=vInit;
 fi; fi;

input newBallot(i)
eff
 if ¬failed then
  if mode= active then
   doMakeBallot[i] := true;
 fi; fi;
```

```
output makeBallot(i,MakeBallot)
pre ¬failed;
 mode=active;
 doMakeBallot[i];
 (getprocid(maxBallot(ballots))<
    getprocid(MakeBallot))∨
 ((getseqno(maxBallot(ballots))<
    getseqno(MakeBallot)));
 getprocid(MakeBallot) = i;
eff
 seqNo := seqNo + 1;
 ballots := insert(MakeBallot, ballots);
 lastProposedBallot := MakeBallot;
 doMakeBallot[i] := false;
 for k:Int in nodes-{i} do
   queueOut[[i,k]] :=  queueOut[[i,k]] ⊢
     sBallot([BALLOT,[i,k],MakeBallot]);
 od;
 rcvBallots := {};
 rcvBallots := insert(MakeBallot, rcvBallots);
 mBallots:=MakeBallot;
```

Figure 2: *PaxosProcess(i): Transitions of actions init, newBallot, makeBallot*

The *init* action proposes and records the submitted value. It also changes the mode to active and sends the value *vProposed* to the other processes. The *newBallot* input action notifies the *PaxosProcess(i)* to originate a new ballot. The *makeBallot* action is triggered once a request for a new ballot has arrived. In this action a new sequence number that is bigger than any previously known sequence number is selected, and then the leader sends the new ballot b. The new ballot identifier is a two field record of the sequence number and the identifier of the new ballot's originator. At this point no value is associated with the ballot. The *maxBallot* operator that is imported in the *makeBallot* action, identifies and returns the largest ballot that has been witnessed so far. In case a process has crashed, the *fail* action is executed (variable $failed$ is set to true). It is important to highlight that only a leader process can start a new ballot.

```
internal abstain(i,BalAbstain)
pre
  mode=active;
  ¬failed;
  getseqno(maxBallot(BalAbstain))<
  getseqno(maxBallot(ballots))∨
  getprocid(maxBallot(BalAbstain))<
  getprocid(maxBallot(ballots));
  (voted[i][i] ∪ abstained[i][i]) ∩ BalAbstain={};
eff
  abstained[i][i] := abstained[i][i] ∪ BalAbstain;
  for k:Int in nodes-{i} do
    queueOut[[i,k]] :=  queueOut[[i,k]] ⊢
    sAbstain([ABSTAIN,[i,k],BalAbstain,
    getprocid(maxBallot(BalAbstain))]);
  od;
  for j:Ballot in BalAbstain do
    rcvBallots := delete(j,rcvBallots);
  od;
  mAbstained := BalAbstain;

internal valueDecision(i, u,latestVal,ballot)
pre mode≠idle;
  head(queueIn[[i,u]])=sLatestValue([LATESTVAL,
  [u,i],latestVal,ballot]);
eff
 queueIn[[i,u]] := tail(queueIn[[i,u]]);
 tempnghbrs[ballot]:=insert(u,tempnghbrs[ballot]);
 if ¬(ballot ∈ abstained[i][i]) then
  if ((size(tempnghbrs[ballot]))<
    (div(size(neighbours),2))) then
   if (lastValue(lastValue[ballot],latestVal))then
    for k:Last in lastValue[ballot] do
     if getvalue(k)=latestVal then
      lastValue[ballot] := delete(k,
        lastValue[ballot]);
      tempLast := setLast(getnodeNum(k)+1,
        getvalue(k));
      lastValue[ballot] := insert(tempLast,
      lastValue[ballot]);
     fi
     od;
    else
     tempLast:=setLast(1,latestVal);
     lastValue[ballot]:= insert(tempLast,
       lastValue[ballot]);
    fi
  else
   tempVal:=0;
   for k:Last in lastValue[ballot] do
    if getnodeNum(k)>tempVal then
     tempVal := getnodeNum(k);
     assignvalue := getvalue(k);
    fi
   od;
   if assignvalue=(-1) then
    assignvalue := chooseRandom(proposed[i]);
   fi
   readyAssign := true;
 fi; fi;

output assignVal (i,balAssignVal,valAssignVal)
pre
 ¬failed;
 mode=active;
 readyAssign;
 balAssignVal ∈ ballots;
 getprocid(balAssignVal)=i;
 val[i][balAssignVal]=nil;
 ifProposed(proposed,valAssignVal);
 (allessdead(ballots,balAssignVal,abstained[i],nodes)∨
 existval(val,valAssignVal,abstained[i],ballots,nodes))
eff
 val[i][balAssignVal]:=embed(valAssignVal);
 for k:Int in nodes-{i} do
  queueOut[[i,k]] := queueOut[[i,k]] ⊢
    sValue([VALUE,[i,k],balAssignVal,valAssignVal]);
 od;
 readyAssign:=false;
 mVal:=valAssignVal;
```

Figure 3: *PaxosProcess(i): Transitions of actions abstain, valueDecision and assignVal*

Figure 3 contains the transitions of actions *abstain*, *valueDecision* and *assignVal*. The *PaxosProcess(i)* automaton uses the *abstain* action to abstain from all the ballots of a set B. This is allowed when the known identifier of a ballot is larger than any other ballot in B, and provided that it has not already voted for any of the ballots of the set B in an earlier state. After the initiation of a ballot process, a value for the ballot has to be chosen. The internal action *ValueDecision* is used to choose a value for the ballot b. The specified transition is being executed only by the leader. All processes have to send the value of the latest ballot that they have voted for (if voted) to the leader. When the leader receives the values from a majority of the processes it chooses a the value for ballot b. The leader ignores all values equal to -1 (indicating that the sender has not voted for any ballot yet). The prevailed value will be assigned to ballot b.

PaxosProcess(i) uses the internal action *assignValue* to assign the value v to ballot b. The possibility to assign a value v to a ballot is based on an important consistency check with smaller ballots. Specifically, *PaxosProcess(i)* checks whether b is a known ballot and that i is the originator of ballot b. So far, no value has yet been assigned to b, as far as i knows. But since i is the process that originally started ballot b, i is the one that has the ability to assign the value v to b. Value v must be known to be the initial value of a process. Besides, all smaller ballots either must have the value v, or are known as "dead". The specified transition uses the operators *ifProposed*, *allessdead*, *existval* and *dead*. The *ifProposed* operator examines whether value v is one of the values that had been proposed by processes. The *allessdead* operator checks if all the ballots that are smaller than b are dead. Also, *existval* checks if v has been assigned to all the smaller non-dead ballots. Once the value has been assigned to the ballot, the leader notifies the other processes about the new value.

Figure 4 depicts the transitions of actions *vote*, *internalDecide* and *decide*. For the system to reach a consensus state, processes have to accept the value of the ballot by voting the ballot. *PaxossProcess(i)* may vote for a ballot b if it is known that a value has been assigned to b, and if i has not yet abstained from b. The responsibility of action *vote(i, b)* is for process i to vote for ballot b and to inform the environment about its participation, by sending a Vote

```
internal vote(i,balVote)
pre
  mode=active;
  ¬failed;
  valproposed( ballots,balVote);
  notnil(val,balVote);
  ¬(balVote ∈ abstained[i][i]);
  ¬(balVote ∈ voted[i][i]);
eff
   voted[i][i]:=voted[i][i] ∪ {balVote};
   for k:Int in nodes-{i} do
    queueOut[[i,k]] :=  queueOut[[i,k]] ⊢ sVote([
    VOTE,[i,k],balVote,getprocid(balVote)]);
    countVote:=countVote+1;
   od;
   sendVote:=false;
   rcvBallots:=delete(balVote,rcvBallots);
   lastvotedvalue:=val[getprocid(balVote)]
      [balVote].val;
   mVoted:=balVote
```

```
internal internalDecide(i,balInternDecide)
pre
  ¬failed;
  mode=active;
  internalDecideOp(nodes,balInternDecide,voted[i])
eff
   succeeded[i]:= succeeded[i] ∪ {balInternDecide};
   if(val[getprocid(balInternDecide)]
      [balInternDecide]≠nil)then
   balvalsucc:=val[getprocid(balInternDecide)]
      [balInternDecide].val fi;
   ballotsucceeded:=balInternDecide;

output decide(i, valDecide)
pre
  ¬failed;
  ¬done[i];
  mode = active;
  ballotsucceeded ∈ succeeded[i];
  embed(valDecide) = val[getprocid(ballotsucceeded)]
     [ballotsucceeded];
eff
  done[i] := true;
```

Figure 4: *PaxosProcess(i): Transitions of actions vote, internalDecide and decide*

message. This action consists of the operators *valproposed* and *notnil*; *valproposed* checks whether ballot b has been proposed by a process, whereas the *notnil* operator examines if a value has been given to b.

Once it is known that a majority of processes have approved the ballot b with value v, *PaxosProcess(i)* may decide that the system has reached consensus by executing the internal action *internalDecide(i,b)*. This action, using the *internalDecideOp* operator, checks whether a majority of processes have accepted b.

Finally, *PaxosProcess(i)* announces the decision to the external environment with the *decide(i)* action. The SEND and RECEIVE actions are used to propagate information among processes reaching consensus. The information includes the proposed and sets of ballots, and the value, voted and abstained maps. Figure 6 presents the transitions of actions SEND and RECEIVE.

For the best manipulation of messages, we created two queue-type data structures *queueOut* and *queueIn*, in which we record the out and in bound messages respectively. In action *SEND(m,i,u)*, process i sends the message that is at the top of queueOut to receiver u. Once the message is sent, it is removed from the queue. A process can decide and terminate when it sends all the voted messages that exist in *queueOut*.

In *RECEIVE(m,i,u)*, the received messages are stored in the queue named queueIn for further utilization. Paxos restricts the communications among processes so as only important information to be sent, thus sending periodically gossip messages at interval of $period$. This message restriction is achieved through the *v(T)* and *gossip(i)* actions of the *PaxosProcess(i)* automaton described in Figure 5.

```
timePassage v(T)
pre ¬failed;
    isEmptyQue(queueOut);
eff Clock := Clock + T;

internal gossip(i)
pre ¬failed;
    Clock ≥ nextGossipTime;
eff
  for k:Int in nodes-{i} do
     queueOut[[i,k]] := queueOut[[i,k]] ⊢
       sState([[i,k],mProposed,mBallots,
       mVal,mVoted,mAbstained]);
  od;
  nextGossipTime := nextGossipTime + period;
```

Figure 5: *PaxosProcess(i): v(T) and gossip.*

In particular, the *v(T)* action models the passage of time. The *Clock* variable (initialized to zero) is increased by T units, T being a predefined quantity and specifies the (worst-case) time needed for all the abovementioned transitions to take place; as we explain later, both T and $period$ are system-dependent and therefore these parameters must be computed based on timing properties of the target deployment platform.

BallotTrigger Automaton. The *BallotTrigger* automaton is the one to specify how a new leader is elected and when a leader generates a new ballot. That is, this automaton is the one to specify the main timing issues of time-based Paxos. The *BallotTrigger(i)* signature and state variables are presented in Figure 7. Figure 8 presents the transitions of *BallotTrigger(i)*.

The *BallotTrigger(i)* automaton handles the event of the ballot voting timeout as follows. If a ballot voting does not complete within a predefined time interval, it is terminated by having the leader initiate a new ballot voting. (Assume

```
output SEND (m,i,u)
pre ¬failed;
    mode= active;
    queueOut[[i, u]] ≠ ({});
    m =head(queueOut[[i,u]]);
eff
 if tag(m)=sVote then
    countVote:=countVote-1
 fi;

if m =head(queueOut[[i,u]]) then
 queueOut[[i,u]] := tail(queueOut[[i,u]]);
fi


input RECEIVE(m,i,u)
eff
if ¬failed then
   if mode= active ∧ (tag(m)=sAbstain ∨tag(m)=sValue
   ∨tag(m)=sProposed ∨tag(m)=sBallot∨tag(m)=sVote
   ∨tag(m)=sState∨tag(m)=sLatestValue) then
      queueIn[[i,u]] := queueIn[[i,u]] ⊢ m;


      if (queueIn[[i, u]]≠ {})∧
       tag(head(queueIn[[i,u]])) = sProposed ∧
       (head(queueIn[[i,
       u]])).sProposed.valueProposed ≠ -1 then
         proposed[u]:= insert((head(queueIn[[i,
         u]])).sProposed.valueProposed,proposed[u]);
         queueIn[[i,u]] := tail(queueIn[[i,u]]);
      fi;

      if (queueIn[[i,u]]≠{})∧
        tag(head(queueIn[[i,u]]))= sBallot∧
        getprocid((head(queueIn[[i,
        u]])).sBallot.ballotSend) ≠(-1) then
          ballots:= insert((head(queueIn[[i,
            u]])).sBallot.ballotSend, ballots);

   if (getseqno((head(queueIn[[i,
      u]])).sBallot.ballotSend)≥seqNo ) then
         seqNo:=getseqno((head(queueIn[[i,
           u]])).sBallot.ballotSend);
     fi;
     rcvBallots:= insert((head(queueIn[[i,
       u]])).sBallot.ballotSend,rcvBallots);
     queueOut[[i,u]]:= queueOut[[i,u]]⊢
       sLatestValue([LATESTVAL,[i,u],
       lastvotedvalue,
       (head(queueIn[[i, u]])).sBallot.ballotSend]);

 queueIn[[i,u]] := tail(queueIn[[i,u]]);
  fi;

   if (queueIn[[i,u]] ≠ {})∧
     tag(head(queueIn[[i, u]])) = sValue ∧
     (head(queueIn[[i,u]])).sValue.value ≠(-1) then
       val[u][(head(queueIn[[i,u]])).sValue.ballot]:=
         embed((head(queueIn[[i,u]])).sValue.value);
       val[i][(head(queueIn[[i,u]])).sValue.ballot]:=
         embed((head(queueIn[[i,u]])).sValue.value);
       queueIn[[i,u]] := tail(queueIn[[i,u]]);
   fi;

    if (queueIn[[i, u]] ≠ {})∧
     tag(head(queueIn[[i, u]])) = sVote ∧
     getprocid((head(queueIn[[i,
       u]])).sVote.ballot)≠(-1)then
      voted[i][u]:=voted[i][u] ∪
        (head(queueIn[[i, u]])).sVote.ballot};
      queueIn[[i,u]] := tail(queueIn[[i,u]]);
    fi;

    if (queueIn[[i, u]] ≠ {})∧
       tag(head(queueIn[[i, u]])) = sAbstain ∧
       ¬((head(queueIn[[i,u]])).sAbstain.BAbstain={})then
       abstained[i][u]:=abstained[i][u] ∪
         (head(queueIn[[i, u]])).sAbstain.BAbstain;
       queueIn[[i,u]] := tail(queueIn[[i,u]]);
    fi;

    if ¬(queueIn[[i, u]] = {})∧
     tag(head(queueIn[[i, u]])) = sState then

      if (head(queueIn[[i,u]])).sState.proposed≠(-1)then
        proposed[u]:= insert((head(queueIn[[i,
          u]])).sState.proposed,proposed[u]);
   fi;

      if getprocid((head(queueIn[[i,
        u]])).sState.ballots) ≠(-1) then
         ballots:= insert((head(queueIn[[i,
           u]])).sState.ballots, ballots);
         if ( getseqno((head(queueIn[[i,
           u]])).sState.ballots)≥seqNo ) then
            seqNo:=getseqno((head(queueIn[[i,
              u]])).sState.ballots);
         fi;
         rcvBallots:= insert((head(queueIn[[i,
           u]])).sState.ballots,rcvBallots);
         queueOut[[i,u]] :=  queueOut[[i,u]]⊢
         sLatestValue([LATESTVAL,[i,u],lastvotedvalue,
         (head(queueIn[[i, u]])).sState.ballots]);
      fi;

   if (head(queueIn[[i,u]])).sState.val ≠(-1) then
    val[u][(head(queueIn[[i,u]])).sState.ballots]:=
      embed((head(queueIn[[i,u]])).sState.val);
    val[i][(head(queueIn[[i,u]])).sState.ballots]:=
      embed((head(queueIn[[i,u]])).sState.val);
    fi;

    if getprocid((head(queueIn[[i,
      u]])).sState.voted) ≠ (-1) then
      voted[i][u]:=voted[i][u] ∪
      {(head(queueIn[[i, u]])).sState.voted}
     fi;

    if ¬((head(queueIn[[i,
      u]])).sState.abstained= {}) then
     abstained[i][u]:=abstained[i][u]
       ∪ (head(queueIn[[i, u]])).sState.abstained;
    fi;
        queueIn[[i,u]] := tail(queueIn[[i,u]]);
     fi;


  fi;
fi;
```

Figure 6: PaxosProcess(i): Transitions of actions SEND and RECEIVE

```
Signature
 Input:
  init(const MPIrank: Int,vInit: Int)
  fail(const MPIrank: Int)
  decide(const MPIrank: Int,vDecide:Int)
  assignVal(const MPIrank: Int,
       bAssignVal:Ballot,vAssignVal:Int)
  RECEIVE(m:Message, const MPIrank: Int, u:Int)
 Output:
  newBallot(const MPIrank: Int)
  sendAlive(const MPIrank:Int,u:Int)
  SEND(m:Message,const MPIrank: Int,u:Int)
 Internal:
  nodeTimeout(const MPIrank:Int,u:Int)
 TimePassage:
   v(T)

States
  mode:Mode := idle
  suspected:Set[Int] := {}
  timeout:Array[Int,Real]
  nextBallotTime:Real := -1
  nextSendTime:Array[Int,Real]
  leader:Int := -1
  Clock:Real := 0
  failed: Bool := false
  delay:Real
  period:Real
  T:Real
  nodes:Set[Int]
  done:Bool := false
  queueOut:Map[Link, Seq[Message]]
  queueIn:Map[Link, Seq[Message]]
```

Figure 7: *: BallotTrigger(i): Signature and State variables*

```
Transitions

input RECEIVE(m,i,u)
eff
 if ¬failed then
  if mode= active ∧ tag(m)=sAlive then
   queueIn[[i,u]] := queueIn[[i,u]] ⊢ m;
   if head(queueIn[[i,u]]) = sAlive([ALIVE,[u,i]]) then
    queueIn[[i,u]] := tail(queueIn[[i,u]]);
    timeout[u]:= Clock+delay;
    if u ∈ suspected then
     nextSendTime[u] := Clock;
     suspected := suspected - {u};
     if u>leader then
      leader:=u;
    fi; fi;
    if u≠leader then
     nextBallotTime := -1;
 fi; fi; fi; fi;

input init(i,v)
eff
 if ¬failed then
  if(mode=idle) then
   mode := active;
   Clock := clock;
   leader := maxElement(nodes);
   for k:Int in (nodes - {i}) do
    nextSendTime[k] := Clock;
    timeout[k] := Clock+delay;
   od;
   if i=leader then
    nextBallotTime := Clock;
 fi; fi; fi;

internal nodeTimeout(i,u)
pre ¬failed;
  mode=active;
  Clock ≥ timeout[u]∧ ¬(timeout[u]=(-1));
eff
 suspected:=suspected ∪ {u};
 timeout[u]:=-1;
 nextSendTime[u]:=-1;
 if leader=u then
  leader := maxElement((nodes-suspected));
 fi
 if i=leader ∧ i<u ∧ ¬done then
  nextBallotTime := Clock;
 fi

input assignVal(i,b, v)
eff
 if ¬failed then
  if mode=active then
   nextBallotTime:=Clock+delay;
 fi; fi;

input decide(i,v)
eff if ¬failed then
       if mode=active then
          done:=true;
          nextBallotTime:=-1;
       fi
    fi;

output newBallot(i)
pre ¬failed;
  mode=active;
  Clock ≥nextBallotTime∧ ¬(nextBallotTime=(-1));
  ¬done;
eff nextBallotTime:=Clock+delay;

timePassage v(T)
pre ¬failed;
    (Clock+T)≤(nextBallotTime)∨ nextBallotTime=(-1);
    timePsg(Clock,T, timeout);
    timePsg(Clock,T,nextSendTime);

eff Clock:=Clock+T;

input fail(i)
eff mode:=failed;


output sendAlive(i,u)
pre ¬failed;
  mode= active;
  ¬ (u ∈ suspected);
  nextSendTime[u] ≤ (Clock +T);
eff
 queueOut[[i,u]]:=queueOut[[i,u]] ⊢ sAlive([ALIVE,[i,u]]);
 nextSendTime[u]:= Clock + delay;

output SEND (m,i,u)
pre ¬failed;
    mode= active;
    queueOut[[i, u]] ≠ ({});
    m =head(queueOut[[i,u]]);
eff
if m =head(queueOut[[i,u]]) then
 queueOut[[i,u]] := tail(queueOut[[i,u]]);
fi
```

Figure 8: *BallotTrigger(i): Transitions*

that i is the current leader.) Particularly, the leader measures the time starting from the execution of action *newBallot(i)* and checks whether the *decide(*,i)* action is executed within the predefined time period. If the execution has not been completed and i is still the leader, then the *newBallot(i)* action is triggered for the initiation of a new ballot voting. The next *BallotTime* variable determines the time when the leader should create a new ballot, whilst the *nextSendTime* defines the time that the acknowledgment message will be sent.

Another responsibility of the *BallotTrigger* automaton is to execute a failure detection mechanism in order for a new leader to be elected, when the current one seems to have crashed. In particular, the automaton implements process crash detection by having the processes interchanging "alive" messages at regular time intervals. When a process i does not receive the alive message of process u within a predetermined time interval, then i inserts u into a set of "suspected" processes (this is implemented by the *nodeTimeout(i,u)* action). The *sendAlive(i,u)* action allows process i to send an alive message to process u after the passage of time and when u is not a suspected process. The receipt of alive messages is implemented using the *recvAlive(i,u)* action. So, when process i receives a message from process u, the timeout variable ($Clock + delay$) is renewed for process u. The variable $delay$ is system-dependent and hence, as with T, its value was computed based on empirical performance measurements of our deployment platform (more details are provided in Section 4). Due to the fact that the system is partially synchronized (and hence, it exhibits timing failures) it is possible that i might not receive u's alive message within the predetermined period and place u in the suspected set, although u is in fact still operational. However, when i receives the delayed message, it removes u from the suspected set. When the leader is included in the set of suspected processes of some process, a new leader election operation is triggered.

The *BallotTrigger(i)* automaton contains the input actions *init* and *decide* for the processes to reach consensus. As an effect of the *init* action, the automaton's state toggles from idle to active, and the current timing value is assigned to the Clock value of the automaton. Initially, each process is assigned as a leader. However, when process i receives an alive message from process u that has greater id, then i grants its leadership to u. In the end, after correct processes exchange alive messages, the leader is the one with the highest id. The input action *decide(i,u)* is activated when consensus is achieved. Consequently, the *decide* variable is toggled to true, and the value -1 (coding infinity) is assigned to *nextBallotTime*. It is important to mention that the action *decide(i,u)* of the *PaxosProcess(i)* automaton activates the corresponding action of the *BallotTrigger* automaton when the two automata are composed (the two automata have been specified in such a way that are *composition compatible* [23]).

The passage of time is specified via the *v(T)* action. The *Clock* variable is increased by T units, T being a predefined quantity and specifies the (worst-case) time needed for all the above mentioned transitions to take place. Finally, the action *fail(i)* specifies the crash of process i (the process state changes from active to failed, and hence no further actions can be triggered from i).

3.2.2 Obtaining the PaxosNode Automaton and Resolving Nondeterminism.

As mentioned in Section 2.2 after the description of the system into IOA language the programmer must combine the algorithm automaton with auxiliary, channel automata. The developed automaton named PaxosNodeCom (Figure 9), composes the algorithm automata (PaxosProcess and BallotTrigger) with the mediator automata responsible for the establishment of the communication (via MPI) among processes. The *SendMediator* automaton consists of the actions *Isend*, *resp_Isend* and *resp_test*, while the *ReceiveMediator* consists of the actions *Iprobe*, *resp_Iprobe*, *receive* and *resp_receive* (which specify standard MPI constructs). More on these mediator automata can be found in [14, 30]. The PaxosNodeCom automaton is fed to the composer which generates the PaxosNode automaton (it includes all states and transitions of the composed automata).

```
automaton PaxosNode(MPIrank:Int,MPIsize:Int)
  components
  P: PaxosProcess(MPIrank,MPIsize);
  B:BallotTrigger(MPIrank,MPIsize);
  RM[j: Int]: ReceiveMediator(Message, Int,
                                MPIrank,j);
  SM[j: Int]: SendMediator(Message, Int,
                MPIrank, j)
```

Figure 9: *PaxosNodeCom: Composition Automaton*

After the composition, and before compilation, we included a schedule, presented in Figure 10, to resolve non-determinism. The schedule consists of the operators *ifmajv* and *votedBallot*. The first checks whether a majority of processes have approved a proposed ballot, whereas the latter operator returns the approved ballot.

Finally, the scheduled *PaxosNode* automaton (which includes the schedule) was fed to our updated version of the IOA compiler (which can handle time-passage action types and includes the developed operators and data structures) and we obtained the Paxos.java file which was then compiled into a *class* file (a JVM executable).

```
schedule
states

links:Set[Link],
lnk:Link,
newBallot:Ballot,
tmpBallots:Set[Ballot]:={},
tempVal:Int:=-1,
temprcvBallot:Ballot,
setTemp:Set[Ballot]:={},
flag:Int:=0

do
 fire input init(MPIrank,valInit);
 while(P.done[MPIrank]≠true ∧ P.failed=false ∧
   ¬(B.failed) ∧ B.mode=active) do
  links:=P.lnks;
  while(¬isEmpty(links) ∧ P.done[MPIrank]≠true ∧
    P.failed=false) do
   lnk := chooseRandom(links);
   links := delete(lnk, links);
   if ¬P.failed ∧  P.mode=active ∧ B.mode=active ∧
     B.Clock≥B.nextBallotTime ∧ ¬(B.nextBallotTime=(-1))
     ∧ ¬B.done then
    fire output newBallot(MPIrank);
    if P.doMakeBallot[MPIrank] then
     newBallot:=setBallot(P.seqNo+1, MPIrank);
     fire output makeBallot(MPIrank, newBallot);
     setTemp:=((P.ballots-{maxBallot(P.ballots)}) -
      P.abstained[MPIrank][MPIrank]) -
      P.voted[MPIrank][MPIrank];
     if setTemp ≠ {} then
      fire internal abstain(lnk.i,setTemp);
   fi; fi; fi;
   setTemp := ((P.ballots-{maxBallot(P.ballots)}) -
     P.abstained[MPIrank][MPIrank]) -
     P.voted[MPIrank][MPIrank];
   if(setTemp≠{}) then
    fire internal abstain(lnk.i,setTemp); fi;
   if flag=0 then
    if P.queueOut[lnk]≠{} then
     fire output SEND(head(P.queueOut[lnk]),MPIrank,lnk.u);
       elseif B.queueOut[lnk]≠{} then
     fire output SEND(head(B.queueOut[lnk]),MPIrank,lnk.u);
     flag:=1;
    fi
   else if flag=1 then
    if B.queueOut[lnk]≠{} then
     fire output SEND(head(B.queueOut[lnk]),MPIrank,lnk.u);
    elseif P.queueOut[lnk] ≠ {} then
     fire output SEND(head(P.queueOut[lnk]),MPIrank,lnk.u);
     flag:=0;
   fi; fi;
   if SM[lnk.u].status=idle ∧ SM[lnk.u].toSend≠{} then
    fire output Isend(head(SM[lnk.u].toSend),MPIrank,lnk.u);
   fi
   if SM[lnk.u].status=idle ∧ SM[lnk.u].handles≠{} then
    fire output test(head(SM[lnk.u].handles),MPIrank,lnk.u);
   fi
   if RM[lnk.u].status=idle ∧ RM[lnk.u].ready=false then
    fire output Iprobe(MPIrank, lnk.u); fi;
   if RM[lnk.u].status=idle ∧ RM[lnk.u].ready=true then
    fire output receive(MPIrank, lnk.u); fi;
   if RM[lnk.u].toRecv ≠ {} then
    fire output RECEIVE(head(RM[lnk.u].toRecv), MPIrank,
      lnk.u);
   fi
   if P.queueIn[[lnk.i,lnk.u]]≠{} ∧ P.mode=active
    ∧ B.mode=active
    ∧ tag(head(P.queueIn[[lnk.i,lnk.u]]))=sLatestValue
   then
    fire internal valueDecision(lnk.i,lnk.u,(head(
     P.queueIn[[lnk.i,lnk.u]])).sLatestValue.latestvalue,
     (head(P.queueIn[[lnk.i,lnk.u]])).sLatestValue.ballot);
   fi
   if P.mode=active ∧ B.mode=active ∧ P.readyAssign=true
    ∧ P.val[MPIrank][P.lastProposedBallot]=nil
    ∧ ifProposed(P.proposed,P.assignvalue)
    ∧ (allessdead(P.ballots,P.lastProposedBallot,
    P.abstained[MPIrank],P.quorum) ∨ existval(P.val,
    P.assignvalue,P.abstained[MPIrank],P.ballots,P.quorum))
   then
    tempVal:=P.assignvalue;
    fire output assignVal(MPIrank,P.lastProposedBallot,
      tempVal);
   fi
   if P.rcvBallots≠{} then
    temprcvBallot:=chooseRandom(P.rcvBallots);
    if notnil(P.val,temprcvBallot) ∧ B.mode=active ∧
     P.mode=active ∧ ¬(P.failed) ∧ valproposed(
     P.ballots,temprcvBallot) ∧ ¬(temprcvBallot ∈
     (P.abstained[MPIrank])[MPIrank]) ∧ ¬(temprcvBallot
     ∈(P.voted[MPIrank])[MPIrank]) then
    fire internal vote(MPIrank, temprcvBallot);
   fi; fi;
   if ifquorumv(P.voted[MPIrank],P.quorum) ∧
     P.mode=active∧B.mode=active∧P.balvalsucc=(-1) then
    fire internal internalDecide(MPIrank,
     votedBallot(P.voted[MPIrank], P.quorum));
   fi
   if P.ballotsucceeded ∈ P.succeeded[MPIrank]∧
    P.balvalsucc≠(-1) ∧ P.countVote=0 ∧ P.mode=active
    ∧ B.mode=active ∧ embed(P.balvalsucc)=
    P.val[MPIrank][P.ballotsucceeded] then
    fire output decide(MPIrank, P.balvalsucc);
   fi
   if B.mode= active ∧ P.mode=active
    ∧ ¬lnk.u ∈ B.suspected
    ∧ B.nextSendTime[lnk.u]≤B.Clock then
    fire output sendAlive(lnk.i,lnk.u);
   fi
   if B.mode=active ∧ P.mode=active
    ∧ B.Clock≥B.timeout[lnk.u]
    ∧ ¬(B.timeout[lnk.u]=(-1))  then
    fire internal nodeTimeout(lnk.i,lnk.u);
   fi
   if (¬P.failed ∧ isEmptyQue(P.queueOut))
    ∨ (¬B.failed ∧ (B.Clock+B.T≤B.nextBallotTime
    ∨ B.nextBallotTime=-1)
    ∧ timePsg(B.Clock,B.T,B.timeout)
    ∧ timePsg(B.Clock,B.T,B.nextSendTime))
    ∧ P.mode=active ∧ B.mode=active then
    fire timePassage v(P.T);
   fi
   if (¬(P.failed) ∧ P.Clock ≥ P.nextGossipTime) then
    fire internal gossip(MPIrank);
   fi
od; od; od;
```

Figure 10: *Paxos Schedule*

4 Experimentation

To demonstrate the functionality of the augmented compiler, we have run the derived Java code (which implements Paxos) on a network of workstations and obtained some experimental data.

Platform and Preparation. Our experimentation platform consists of a cluster of 17 local machines. Each machine is powered by an Intel Pentium V 1.5 GHz CPU and is running Linux (Fedora Core v5 OS).

As aforementioned in the previous section, the time-related parameters T, $delay$, and $period$ used abstractly in the specification are system-dependent. In particular, T is the (worst-case) time needed for a node to perform a certain sequence of actions (as specified in the previous section). This time may vary on different platforms. Parameter $delay$ includes the (worst-case) time for a message round-trip, local computation and other Java-related delays. Clearly, this depends on the implementation platform. Parameter $period$ may be thought as programmer-defined, in the sense that it is up to the programmer to decide on how often the nodes should gossip. Of course, this decision also depends on the deployment platform, as the period should depend on the message round-trip time, the network topology, and the system load.

Therefore, in order to identify sensible values for these parameters for our deployment platform, we performed some initial experiments with simple executions of the code, taking into consideration the system's ping times and the performance analysis presented in [25]. From this preprocessing phase the following values (in *msecs*) were finally chosen: $T = 22,\ delay = 8822$, and $period = 24$.

Scenarios and Results. It is noteworthy that Paxos is capable of dealing with small transient failures which are concealed by the use of majority voting. MPI is not fault tolerant and when nodes fail the system can suffer a failure as a whole – due to resource depletion. Removing this limitation is subject of future work where the MPI mediator automata can be replaced with Java mediator automata (as proposed in [16]) that allow more dynamic behaviors.

Despite the above note, a practical evaluation of our automatically generated code is still meaningful: first, it demonstrates that indeed the generated code is executable, second that the resulting program behaves as expected, and finally that we obtain a reasonable performance. To this end we present three scenarios.

The first scenario aims to identify the average execution time and number of sent messages for achieving consensus on a single ballot voting, while the number of participants increases gradually from 2 to 17. The objective of the second scenario is to test the resilience of Paxos to message loss. Since MPI is not fault-tolerant, we introduce code on the sender side that randomly chooses messages to be dropped before the MPI send primitive is invoked. Scenarios 2A and 2B drop 10% and 20% of messages respectively. The third scenario seeks to measure the performance of our Paxos implementation in the presence of leader crashes. In particular, in this scenario we simulate the leader crash (by setting the status of the leader as failed in the schedule block) and hence we force the algorithm to initiate the leader election and new ballot mechanisms.

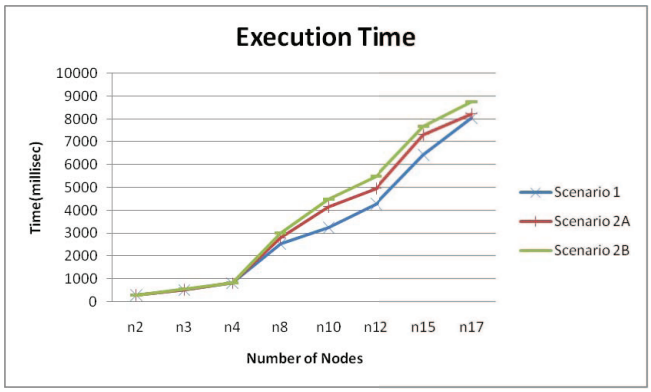
Each scenario was run 10 times and thus each plot point depicts the average of the runs. Figures 11(a) to 11(d) illustrate the average execution time and sent messages respectively for each scenario. The first scenario is used as baseline against the other two scenarios.

The experimental data in Figures 11(a) and 11(b) demonstrate, as expected, that Paxos is able to cope well with message omission. The difference in message count in Figure 11(b) is negligible between the scenarios, which is to be expected, since non-leader node message loss is amortized by the use of majority voting, whereas, leader message loss contributes only to a few additional messages; it does, however, contribute to timeouts and hence the increased operation latency as depicted in Figure 11(a).

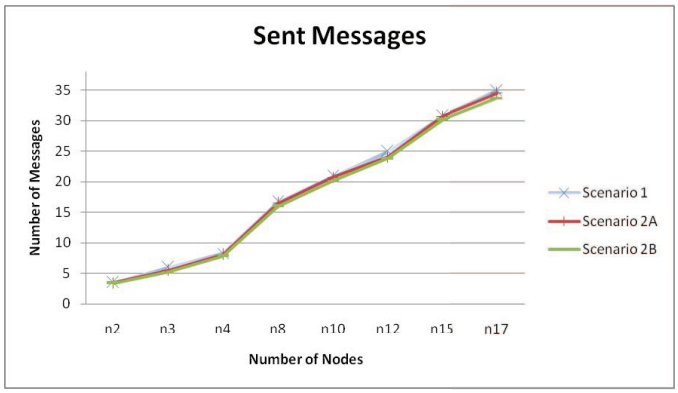
The timing in Figure 11(c) predictably favors the case when the leader is stable. It is important to point out that the performance in scenarios 1 and 3 is parallel where the difference reflects the timeout until a new leader election is triggered. We also observe a linear decay in performance as the number of nodes increases, which is to be expected. However, we do not expect this behavior to last indefinitely, especially when the network becomes saturated.

5 Conclusions

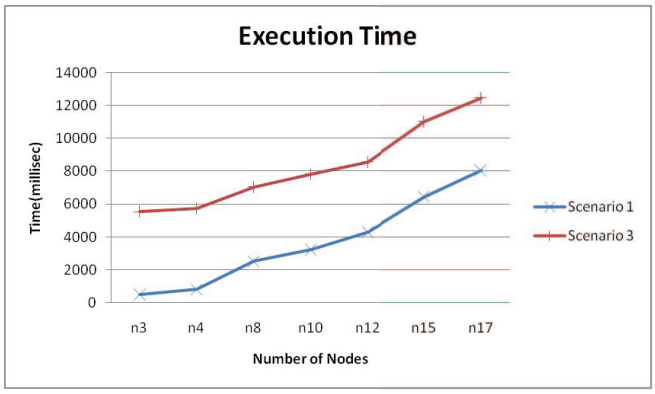
In this paper we have described our experience in specifying, compiling and running a time-based version of the popular Paxos consensus algorithm. In particular, by using a GTA specification of Paxos (which was proved to be correct in [25] and machine-checked in [18, 31]) and by extending the IOA checker, composer and compiler (of the IOA Toolkit) in supporting a variant of the GTA framework, we have managed to develop an automated implementation



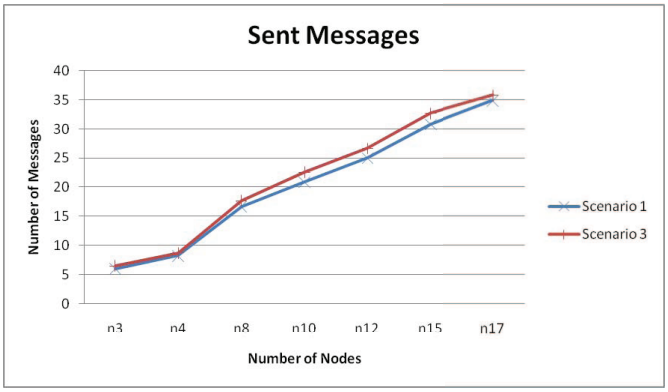
(a)



(b)



(c)



(d)

Figure 11: *Experimental results: (a) Avg. execution time for scenarios 1, 2A, & 2B, (b) Avg. number of sent messages for scenarios 1, 2A, & 2B, (c) Avg. execution time for scenarios 1 & 3, and (d) Ave. of sent messages for scenarios 1 & 3.*

of time-based Paxos. To the best of our knowledge, our work constitutes the first example of a time-dependent complex distributed algorithm that has been specified, verified and implemented in an automated way, using a common formal methodology (IOA in our case).

Several future research directions emanate from our work. First, it would be interesting to assess the efficiency of the automated implementation produced by the compiler. One way is to compare our implementation of Paxos with the one of Musial [29] which was done in a manual way. However, at this point such a comparison would not be fair, as the implementation of Musial uses Java Sockets and TCP, and not MPI for communication. This brings us to a second future objective. Currently the compiler is limited to static participation and use in LANs due to the use of MPI. The compiler design is general enough to enable the use of other communication paradigms. In [16] an alternative communication paradigm is suggested (Java Sockets with TCP) that enables the automated implementation of algorithms that have dynamic participation (nodes may join and leave the computation at any time). Ongoing work is attempting to incorporate this alternative paradigm into (our version of) the IOA compiler.

The TIOA framework (an extension of the IOA framework) models distributed systems with timing constraints as collections of interacting state machines, called Timed Input/Output Automata (an extension of Input/Output Automata) [19]. This framework can be considered more general than GTA, since a state in TIOA not only can be changed by discrete transitions but also by trajectories. A trajectory is a (continuous or discontinuous) function that describes the evolution of the state variables over intervals of time. Therefore, it seems that TIOA can be used to specify a wider family of time-based algorithms (and not just the ones that their computational progress depends on timeouts – like Paxos). A TIOA toolkit is underway [24] which currently includes a TIOA checker, a theorem prover and a TIOA simulator with limited functionality. A very challenging research direction is to develop a TIOA code generator. Our work can be considered an important step towards that direction.

Acknowledgments. We would like to thank Panayiotis Mavrommatis for several helpful discussions.

References

[1] IOA language and toolset. URL: `http://theory.lcs.mit.edu/tds/ioa/`.

[2] INMOS Ltd: occam Programming Manual, 1984.

[3] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2nd edition, 2004.

[5] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47-67, 2003.

[6] R. Cleaveland, J. N. Gada, P. M. Lewis, S. A. Smolka, O. Sokolsky, and S. Zhang. The Concurrency Factory: Practical tools for specification, simulation, verification and implementation of concurrent systems. In *DIMACS Workshop*, pages 75–89, 1994.

[7] R. Cleaveland, J. Parrow, and B. U. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM TOPLAS*, 15(1), 1993.

[8] R. De Prisco. Revisiting the Paxos Algorithm. Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1997. Also as TR: MIT-LCS-TR-717.

[9] M. Fisher, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, 1985.

[10] M. P. I. Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[11] S. Garland, N. Lynch, J. Tauber, and M. Vaziri. IOA user guide and reference manual. Technical Report MIT/LCS/TR-961, July 2004. URL: `http://theory.lcs.mit.edu/tds/ioa/manual.ps`.

[12] S. J. Garland and J. V. Guttag. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, 1991.

[13] Ch. Georgiou, P. Hadjiprocopiou, and P.M. Musial. On the automated implementation of time-based Paxos using the IOA compiler. Technical Report, 2010. URL: `http://www.cs.ucy.ac.cy/~chryssis/pubs/tpaxos.pdf`.

[14] Ch. Georgiou, N. Lynch, P. Mavrommatis, and J. A. Tauber. Automated implementation of complex distributed algorithms specified in the IOA language. *Journal of Software Tools for Technology Transfer*, 11(2):153–171, 2009.

[15] Ch. Georgiou, P. Mavrommatis, and J. A. Tauber. Implementing asynchronous distributed systems using the IOA toolkit. Technical Report MIT/LCS/TR-966, 2004.

[16] Ch. Georgiou, P. M. Musial, A. A. Shvartsman, and E. L. Sonderegger. An abstract channel specification and an algorithm implementing it using Java sockets. In *Proceedings of the 7th IEEE International Symposium on Network Computing and Applications (NCA 2008)*, pages 211–219, 2008.

[17] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, UK, 1985.

[18] N. Immorlica and T. Win. A Case Study: Proving Paxos with the IOA Toolkit. *Manuscript*, 2002.

[19] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata* (Synthesis Lectures in Computer Science). Morgan & Claypool Publishers, 2006.

[20] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[21] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[22] K. Larsen and P. Pettersson. Uppaal in a nutshell. *Journal of Software Tools for Technology Transfer*, 1(1/2):134–152, 1997.

[23] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[24] N. Lynch, L. Michel, and A. Shvartsman. 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.

[25] N. Lynch and A. Shvartsman. Paxos made even simpler (and formal). *Manuscript*, 2002.

[26] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, 1996.

[27] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[28] R. Milner. *Communication and Concurrency*. Prentice-Hall International, UK, 1989.

[29] P. M. Musial. *From High Level Specification to Executable Code: Specification, Refinement, and Implementation of a Survivable and Consistent Data Service for Dynamic Networks*. PhD thesis, Dept. of Computer Science and Engineering, University of Connecticut, 2007.

[30] J. A. Tauber. *Verifiable Compilation of I/O Automata without Global Synchronization*. PhD thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., 2005.

[31] T. N. Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, Dept. of Electrical Engineering and Computer Science, M.I.T., 2003.

Appendix

Java Code for Some Operators and Data Structures

The Java code for the ifProposed and ifmajv operators.

```
public static BoolSort ifProposed(ArraySort proposed, IntSort value){
BoolSort b= BoolSort.False();
Enumeration allkeys= proposed.enumIndices();
while(allkeys.hasMoreElements()){
    IndexSeq key=(IndexSeq)allkeys.nextElement();
    SetSort set=(SetSort)proposed.elementAt(key);
    Iterator i = set.getSet().iterator();

    while(i.hasNext()) {
     IntSort val=(IntSort)i.next();
    if(val.value==value.value){
    return BoolSort.True();
 }
   }
}
return b;
 }

static public BoolSort ifmajv(ArraySort votedBallots,SetSort nodes){
    int majority=0;
    majority=(nodes.size() / 2) + 1 ;
    MsetSort allVotedBallots =MsetSort.empty();
    Enumeration allkeys1= votedBallots.enumIndices();

    while(allkeys1.hasMoreElements()){
           IndexSeq key=(IndexSeq)allkeys1.nextElement();
           SetSort set= (SetSort)votedBallots.elementAt(key);
           if (!SetSort.isEmpty(set).booleanValue()){
               Iterator i = set.getSet().iterator();
                while(i.hasNext()) {
                    BallotSort tempInt=(BallotSort)i.next();

                    allVotedBallots = allVotedBallots.insert(tempInt);
                }
           }
    }
  if (!MsetSort.isEmpty(allVotedBallots).booleanValue()){
    Iterator j= allVotedBallots.map.keySet().iterator();
    while(j.hasNext()) {
      ADT element = (ADT)j.next();
      if (allVotedBallots.count(element) >=majority)
        return BoolSort.True();
    }
   }

  return BoolSort.False();

}
```

A part from the BallotSort data type Java code:

```
package ioa.runtime.adt;
import ioa.simulator.Entity;
import ioa.util.logger.IOACategory;
import ioa.util.sexp.*;
import java.lang.Integer;
import java.lang.Math;
import java.util.Iterator;
import java.math.BigInteger;
import java.util.HashSet;
import java.util.Set;
```

```
public class BallotSort extends ComparableADT
     implements java.io.Serializable, MPINode {

  private static IOACategory cat =
    IOACategory.getInstance (BallotSort.class.getName());
  protected int seqno;
  protected int procid;
  protected int Bid;
  //Constructors

  protected BallotSort( int seqno,int procid) {
this.seqno = seqno;
                        this.procid=procid;
                        if(seqno!=-1 && procid!=-1)
                           this.Bid=ioa.runtime.adt.Check.concatenate(
                           IntSort.lit(seqno),IntSort.lit(procid)).intValue();
                        else this.Bid=-1;
  }
  protected BallotSort() { }


  /**
   * Return the seqno and procid of <code>this</code>.
   */
  public static IntSort getseqno(BallotSort b){
    return new IntSort(b.seqno);}

  int seqno(){return seqno;}

  public static IntSort getprocid(BallotSort b) {
     return new IntSort(b.procid);}

  int procid(){return procid;}

  public int Bid(){return Bid;}

  public static BallotSort setBallot(IntSort seqno, IntSort procid ){
   return new BallotSort (seqno.value,procid.value);

  }

public static Object maxBallot(SetSort set) {

    if (set.size() == 0)
      return new BallotSort(-1,-1);

    Iterator i = set.getSet().iterator();
    BallotSort maxElement = (BallotSort)i.next();
    while(i.hasNext()) {
      BallotSort element = (BallotSort)i.next();
       if (element.seqno() > maxElement.seqno()) {
maxElement = element;

      }
      else if (element.seqno() == maxElement.seqno()){
         if(element.procid()>maxElement.procid())
              maxElement = element;
      }
    }
    return maxElement;
  }
```