A survey of formal verification of Paxos and a case study with an algebraic specification language

By Thanisorn Apasuthirat

A project paper submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Kazuhiro Ogata

September, 2014

A survey of formal verification of Paxos and a case study with an algebraic specification language

By Thanisorn Apasuthirat (1210201)

A project paper submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Kazuhiro Ogata

and approved by
Professor Kazuhiro Ogata
Professor Kunihiko Hiraishi
Associate Professor Toshiaki Aoki

August, 2014 (Submitted)

Abstract

Software sometimes has some bugs or errors since made by human. Some systems may run correctly even it contains some errors, while some systems can cause the damage to life even containing a tiny error. Therefore, we need to check that system does not contain any fault by doing the software verification. Software verification is an vital part for checking the correctness of system. It consists of two techniques for doing verification; model checking and theorem proving. Model checking involves automatically exploring the set of reachable states of model to ensure that some formulas needed to check holds. On the other hand, theorem proving uses some theories to prove, and the theorem to be proved need to be formulated as formulas involving some mathematics. The theorem proving technique needs the guidance of human taking the form of lemmas while the model checking can be automatically done. However, model checking can cause the state explosion problem since it searches in the state space of the complex system.

This research aims to verify an algorithm for solving consensus problem in distributed system. The algorithm for solving consensus is called consensus algorithm. The consensus we focused is Paxos algorithm which is a family of consensus algorithms. We conduct the Paxos model and specify it on both CafeOBJ language (by OTS) and maude language. Then we verify Paxos that enjoys agreement property, which is a property of consensus algorithms, by using proof scores in OTS/CafeOBJ and CITP in maude which considered as a theorem proving technique.

Futhermore, we survey the related formal verification of an similar consensus algorithm with Paxos (called LastVoting algorithm). They proposed the way to reduce the verification problem to a small model checking problem by involving single phases of algorithm configuration. They used some notions of round-based model to model asynchronous consensus algorithm and reduced the model checking problem of some properties such as agreement and termination to the satisfiability problem for a formula in some logic. They used a Yices (Satisfiability Module Theories) to check the satisfiable of the formula. In their experimental result, they only successfully verified the number of processes up to around 10 processes. Difference from our approach that use theorem proving, we do not need to bound any number of processes and it can be proved infinite number of processes.

Contents

Acknowledgements

I would like to express my special appreciation and thanks to my advisor Professor Kazuhiro Ogata, you have been a tremendous mentor for me. I would like to thank you for encouraging my research and for allowing me to grow as a researcher even I have made some mistakes. Your advice on both research as well as on my career have been priceless. I would also like to thank my committee members, Professor Kunihiko Hiraishi, and Associate Professor Toshiaki Aoki for serving as my committee members even at hardship. I also want to thank you for giving the precious comments and suggestions, thanks to you. I would especially like to thank Zhang Min for his many advices on my research. The quality of this research was significantly improved because of him.

A special thanks to my family for their support and encouragement throughout my study and throughout my life. I would also like to thank all of my seniors and friends who support and give advantageous advices to strive towards my goal.

Chapter 1

Introduction

Software verification is an essential part for checking the correctness of system. Many people use software verification to check many systems. This research intends to survey the existing paper of formal verification of a *LastVoting* algorithm which is one of algorithms to solve consensus problem in distributed system and similar to *Paxos* algorithm; the well-known algorithm which has been used in the real world, and conducts a model to verify the Paxos algorithm.

To understand the broad view of this research this chapter starts with the brief explanation of software verification, consensus problem, and Paxos algorithm. Then our purposed approach to survey and verify the Paxos is shortly explained. Finally, we provide a structure of this research.

1.1 Overview

1.1.1 Software Verification

Formal verification of hardware and software system has gained popularity in industry since the advent of famous "Pentium bug" in 1994, which caused Intel to recall faulty chips and take a loss of $475 million [CMMP95]. Since this event formal verification of hardware and software systems has been commonplace using mostly model checkers but also using theorem provers. The benefits reaped in the hardware sector has led the software sector to consider whether similar benefits could be achieved in the context program correctness. In the context of verifying program correctness, the correctness problem of software is formally defined. Verifying the correctness of a program involves formulating a property to be verified using suitable logic such as first order logic or temporal logic [BBF+10].

When assessing the correctness of the program, two distinct approaches using properties are used - *pre/post condition* and *invariant assertion*. Pre/post condition approaches formulate the correctness problem as the relationship between a formula that is assumed to hold at the beginning of the program execution, and a formula that should hold at the end of program execution. Approaches based on an invariant assertion define correctness of a program as an invariant formula, which must be verified to hold throughout the

program execution. Invariants can be specified by the user, denoted a *specification*, or can be automatically inferred from the program code. Proofs of correctness are typically achieved through the derivation of a theorem. However, software verification can also be achieved without mathematical proofs. A popular approach to formal verification, called *model checking*, is increasingly being used to verify software.

Software verification through model checking

The model checking problem involves the construction of an abstract model $\mathcal{M}$, in the form of variations on finite state automata, and the construction of specification formulas ϕ, in the form of variations on temporal logic [BBF+10]. The model checking verification problem involves establishing that the model semantically entails the specification $\mathcal{M} \models \phi$.

The verification algorithm used in the model checking involves exploring the set of reachable states of the model to ensure that the formula ϕ holds. If ϕ is an invariant assertion, the model checking approach explores the entire state space to ensure that the formula holds in all states. In order to guarantee termination, such as approach requires that the set of reachable states be finite. Furthermore, verification by model checking has gained popularity in industry because the verification procedure can be fully automated and counter examples are automatically generated if the property being verified does not hold. Since model checkers rely on exhaustive state space enumeration to establish whether a property holds or does not hold, it can put immediate limits on the state space problem that can be explored. This problem, known as the state explosion problem, is an often cited drawback of verification by model checking [CGJ+01].

Software verification through theorem proving

Theorem provers used to prove program properties are based on variations of Hoare logic [Hoa69]. It describes a calculus to reason about program correctness in term of pre and post conditions. Hoare's approach to proving correctness introduced the concept of a "Hoare triple", which is a formula in the form $\{\phi_{PRE}\}P\{\phi_{POST}\}$. This formula can be read as "if property ϕ_{PRE} holds before program P starts, ϕ_{POST} holds after the execution of P". The program P can refer to an entire program or a single function call, depending on the unit that is being verified. In Hoare's calculus, axioms and rules of inference are used to derive ϕ_{POST} based on ϕ_{PRE} and P.

A key difference between the theorem approach and the model checking approach to software verification is that theorem provers do not need to exhaustively visit the program's state space to verify properties. Consequently, a theorem prover approach can reason about infinite state spaces and state spaces involving complex datatypes and recursion. This can be achieved because a theorem prover reasons about constraints on states, not instances of states. Theorem provers search for proofs in the syntactic domain, which is typically much smaller than the semantic domain searched by model checkers. Although theorem prover support fully automated analysis in restricted cases, some inductive structures must perform by doing some mathematical induction (e.g. trees, lists, or stacks). Nevertheless, this tradeoff is acceptable in certain instances since this type of

analysis cannot be performed by model checkers, but is still important to the verification effort.

1.1.2 Consensus

Distributed system is a collection of computers connected through a network and working together as one large computer. Since, there are many computers, and some of them may crash which may cause the whole system temporarily stop. Some systems do not require high level of fault-tolerance while others, fault-tolerance is between life and death such as medical and aviation applications. The consensus problem is a key aspect of fault tolerance: ensuring that a system of processors makes the correct decision even if one or more processors or links has failed.

Considering the airplane or spacecraft, if there is a controller to control the direction turning left or right inside the airplane or spacecraft, and it does not respond when the pilot controls to turn left or right, so the airplane or spacecraft could crash into something or fall. The solution is to add several controllers and makes them deciding on a single value even one or some controllers are saying the opposite, or not respond.

Therefore, consensus is the problem of getting all processes or nodes to agree on the same decision. Each process is assumed to have a proposed value at the beginning and is required to eventually decide on a value by some processes.

1.1.3 Paxos

Since, the fault-tolerance can be achieved through replication in distributed system. A common approach is to use a consensus algorithm to ensure that all replicas are mutually consistent. Paxos is a flexible and fault tolerant protocol for solving the consensus problem. Paxos can be used to solve the atomic problem in distributed transactions, or to order client requests sent to a replicated state machine (RSM). An RSM provides fault tolerance and high availability, by implementing a service as a deterministic state machine and replicating it in different machines. Furthermore, Paxos is used in production systems such as Chubby and ZooKeeper [CGR07, HKJR10] among many others. The detail about Paxos algorithm is explained later in Chapter 2.

1.2 Proposed Approach

In this research we do a survey of formal verification in [TS11]. In, [TS11], authors proposed a semi-automatic verification approach for asynchronous consensus algorithms based on model checking techniques but we only focus on LastVoting consensus algorithm which is similar with Paxos. However, the state space of doing model checking is huge, often infinite, thus making model checking infeasible. Their approach is to reduce the verification problem to small model checking problems that involve only single phases of algorithm execution. Since a phase consists of finite number of rounds, it can be effectively

solve these problem by using satisfy ability solving. Although the state space is bounded, they can only model checked several consensus algorithms up to around 10 processes.

Besides surveying, we conduct a model of Paxos and do some specification and verification used theorem proving technique for verication Paxos in OTS/CafeOBJ called *proof scores* and CITP in maude to check that Paxos enjoys some desired properties.

1.3 Roadmap

This research is structured as the following. Chapter 2 provides the technical details of consensus, Paxos, CafeOBJ and maude. Chapter 3 explains the existing verification of consensus algorithm by using the satisfiability solving. Chapter 4 describes a case study of Paxos by constructing a model, doing specification and verification in both CafeOBJ with proof scores and Maude with CITP, and ends up with conclusion and future works in Chapter 5

Chapter 2

Technical Background

2.1 Consensus

Consensus is the problem of getting process to agree on the same decision whether some faults occur. Consensus is central to the construction of fault-tolerant distributed systems. For example, atomic broadcast, which is at the core of state machine replication, can be implemented as a sequence of consensus instances [CT96]. Other services, such as view synchrony and membership, can also be constructed using consensus [GS01]. Because of its importance, many researchers have devoted to developing new algorithms for this problem.

The consensus problem should satisfies the following properties:

- Validity: Any decision value is the proposed value of some process
- Agreement: No two different values are decided
- Termination : All processes eventually decide

2.2 Paxos Algorithm

Paxos is a family of very intriguing fault-tolerant distributed consensus algorithms. Paxos was proposed by Lamport in his seminal paper [Lam98] and later gave a simplified description in [Lam01a]. Paxos can be used to solve the atomic commit problem in distributed transactions, or to order client requests sent to a replicated state machine. The Paxos algorithm can solve the consensus problem in an asynchronous model with the realistic assumptions that (1) process can operate at arbitrary speed, (2) process may fail by stopping and may restart (the information can be remembered) and (3) process cannot tell a lie. Besides process, messages sending in network can (1) take arbitrarily long to be delivered, (2) be duplicated and lost, but (3) cannot be corrupted.

The Paxos algorithm for solving consensus is used to implement a fault-tolerant system, and it must satisfies the safety requirement of agreement and validity even if some

processes may crash to fail. Progress is granted as long as a subset of processes is alive and communicating normally.

In this research, we will focus on Basic Paxos which is the most basic of the Paxos family, and it only tolerates crash-stop failures. However, it can be also modified to survive byzantine failures [Lam01b, Lam02].

2.2.1 Agent

A distributed application that uses Paxos has different processes which are interested in receiving values, submitting them, or both. It is described by three roles to perform by three classes of agents: *proposers*, *acceptors*, and *learners*. A single process may act as more than one agents.

Proposer

The proposer is responsible for proposing values submitted by the clients until those are delivered. Proposers rely on an external leader election service, which should nominate a coordinator or leader among them. Proposers which are not the current leader can be idle; only leader can do the task.

The leader proposer sends client values through the broadcast to the set of acceptors. For each client value submitted, it chooses the *unique number* (N_u) and bind the value to it. The leader is the only one who connects the client when consensus is reached and must deliver the consensus value to client.

Acceptor

The task of acceptor is relatively simple: it waits for messages from proposers and answers to them or sends messages to all learners. For each instance, the acceptor keeps a state record consisting of $< N_p, N_a, V_a >$, where N_p and N_a is an type of integer related to the highest-number proposal that was accepted from proposer, V_a is a value from the leader proposer.

Learner

Learner is responsible for listening to acceptors decisions, finding the consensus value, and broadcasting the consensus value to all agents.

Whenever the learner realises that a *majority* of acceptors has been reached for an instance, it must decide a value from values which is received from acceptors. All learners must decide a value and that value is called a *consensus value*.

2.2.2 Messages

Since each agent has to communicate by sending messages to other agents, so that knowing the context of messages is important. In Basic Paxos, there are four important messages;

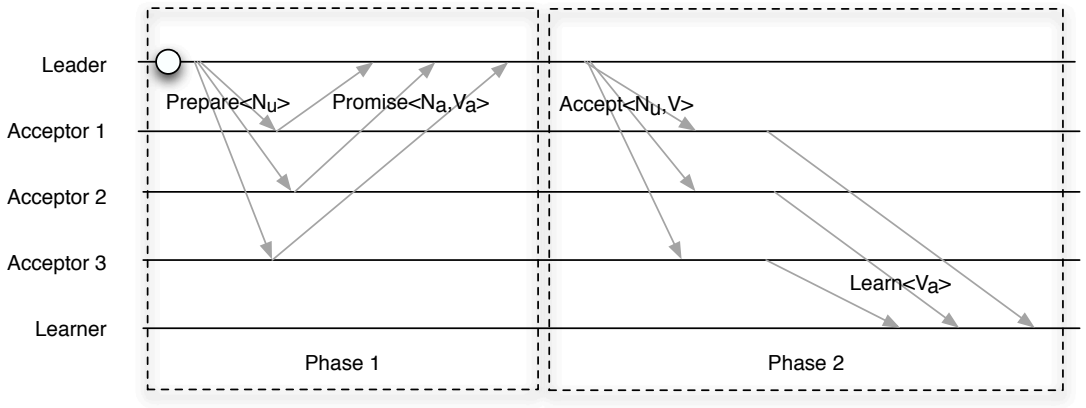

Figure 2.1: The phase of Basic Paxos algorithm

prepare, promise, accept, and *learn* using to communicate between agents. We ignore the broadcast message sending a consensus value from learner to all agents.

Prepare Message

Prepare message consists of a unique number $Prepare(N_u)$, where N_u is the proposal number which is unique for each proposer. The prepare message sends from the leader proposer to all of acceptors.

Promise Message

Promise message consists of two values for each acceptor $Promise(N_a, V_a)$, where N_a is the highest-number proposal that acceptor has accepted, and V_a is the value that acceptor has accepted. The promise message sends from each acceptor to the leader proposer.

Accept Message

Accept message consists of two values $Accept(N_u, V)$, where V is the value from client which a leader proposer proposes. The accept message sends from the leader proposer to all of acceptors.

Learn Message

Learn message consists of a value $Learn(V_a)$ sending from each acceptor to learner. The learn message is a responsible to tell all learners which value has been decided from the majority of acceptors.

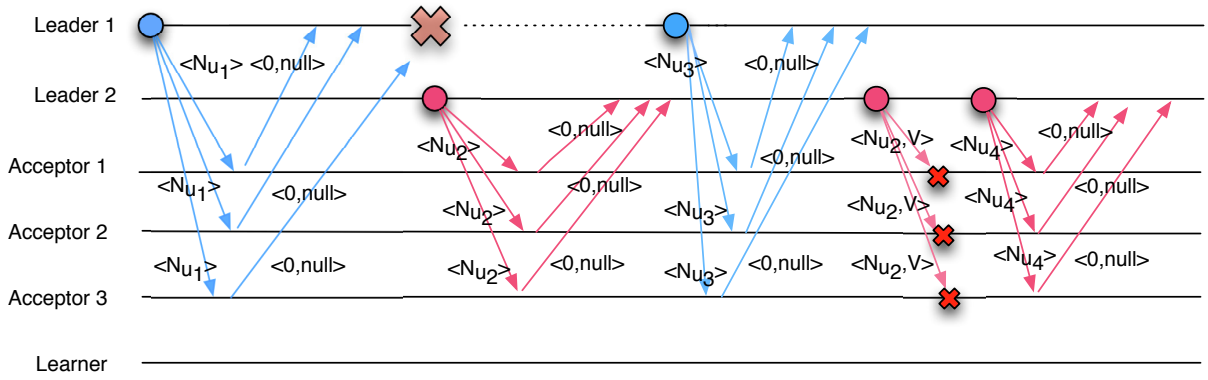

Figure 2.2: The phase of Basic Paxos algorithm with a leader crash

2.2.3 Phases

The algorithm of Basic Paxos consists of two phases as figure 2.1:

Phase 1

Leader proposer selects a unique number(N_u) and sends $Prepare(N_u)$ to the set of acceptors by putting into network. When the acceptor receives a $Prepare(N_u)$, it compares with the promise number(N_p). If the received number is larger than or equal promise number, the acceptor accepts and returns a $Promise(N_a, V_a)$ to leader to promise that acceptor will ignore all future message which unique number less than N_p.

Phase 2

Leader waits for a majority of $Promise(N_a, V_a)$ coming from a set of acceptors. When the majority of promise messages are reached, the leader sends $Accept(N_u, V)$ which a value V, among the promise messages with the highest accepted numberN_a and selects the value V_a from a pair of $< N_a, V_a >$, or a value from client, to the set of acceptors. When the acceptor receives $Accept(N_u, V)$, it compares between N_u and N_p again. If the condition (N_u not less than N_p) is satisfied the acceptor updates the promise number(N_p) and accepted number(N_a) to be equal to N_u, and accepted value(V_a) to be equals to V. Then acceptor sends $Learn(V_a)$ to all learners. When the learner receives the value, it decides which value has been accepted by a majority of acceptors. After that the learner broadcasts that value to all processes. So, every process has the same one output value.

However, a leader may crash and the system must select a new leader as in figure 2.2. *Leader 1* crashes after receiving two promise messages of *Acceptor 1 and 2*, so the *Leader 1* cannot receive the promise message of *Acceptor 3*. Then the system selects a new leader(*Leader 2*) and begins the Paxos algorithm with *phase 1*. The *Leader 2* sends the prepare message to all acceptors with its unique number (assume that $N_{u_2} > N_{u_1}$). When

the acceptor receives the prepare message, it replies the promise message. Before *Leader 2* broadcasting the accept message, *Leader 1* recovers and continues sending the prepare message with a new unique number, which greater than the previous unique number ($N_{u3} > N_{u2}$), and receiving the promise messages from acceptors. Once acceptor receives the prepare message with greater the promise number ($N_p = N_{u2}$) that has been accepted, then it updates the promise number($N_p := N_{u3}$) and replies the promise message back. After that *Leader 2* begins *phase 2* and broadcasts the accept message with value V to all acceptors, however acceptors cannot receive the accept messages. They have already promised not to receive any messages which lower than their previous accepted number. So they discard the accept messages and do not reply anything. When leader waits for several times and messages have not come yet, it sends prepare message with new unique number($N_{u4} > N_{u3}$) again. At this moment, there are two leaders in the system, and they can compete to send their new unique numbers to all acceptors and acceptors finally do not receive any accept message to decide the value. So, Basic Paxos only guarantee safety property but not guarantee liveness property which is termination unless there is a single leader in the system.

2.3 CafeOBJ

CafeOBJ [DF98] is an executable algebraic specification language, implementing equational logic by rewriting logic. Equations are treated as left to right rewrite rules. It can also be used as a powerful interactive theorem prover with the proof scores method.

A module in CafeOBJ encapsulates definitions of a sort or a set of sorts. A module,which declared with keywords `mod{...}`, consists of three parts as follows:

(1) importation of modules (e.g.`pr(`M`)`)
where M is a previously defined module.

(2) signature which consists of sorts, subsorts and operators belonging to the sorts to be specified (`[`s`]`, `[` s `<` s' `]`, and `op` f `:` $s_1 \cdots s_n$ `->` s `.`)
where s and s' are sorts, `[` s `<` s' `]` means that s is a sub sort of s', f is defined as an operation.

(3) axioms for giving semantic which consists of variables and equations (`vars` v v' `:` s, `eq` t `=` t', `ceq` t `=` t' `if` *cond* .)
where v and v' are variables of sort s, t and t' are defined as a term.

For example, we declare a functional module of natural number in CafeOBJ as follows:

```
mod! NAT{
  -- sorts
  [Zero NzNat < Nat]
  -- operators
  op 0 : -> Zero
```

```
  op s_ : Nat -> NzNat
  op _+_ : Nat Nat -> Nat [assoc comm]
  -- variables
  vars X Y : Nat
  -- equations
  eq 0 + Y = Y .
  eq s(X) + Y = s(X + Y) .
}
```

where `0` is a constant for zero in natural number, `s_` means the successor of the input sort `Nat`, `+` is an operator for addition in natural number, `X` and `Y` are variables of sort `Nat` . Two equations are axioms that define the operator `+` in CafeOBJ.

Moreover, CafeOBJ provides built-in modules, and one of the most important module is `BOOL` in which propositional logic is specified. `BOOL` is automatically imported by almost every module unless otherwise stated. In `BOOL`, the sort is `Bool` which consists of constants `true` and `false`, and operators denoting some basic logical connectives. Among the operators are `not_`, `_and_`, `_or_`, `_xor_`, `_implies_` and `_iff_` denoting negation($\neg$), conjuction($\wedge$), disjunction($\vee$), exclusive disjunction(xor), implication($\Rightarrow$) and logical equivalence($\Leftrightarrow$), respectively.

2.3.1 Observational Transition System (OTS)

In OTS, abstract data types are used to formalise values such as natural numbers, Boolean value, and strings in software systems. System's states are characterised by the values that are returned by a special class of functions called *observers*, unlike traditional state transition systems where states are represented as sets of variables. Transitions between states are also specified by functions called *transitions* to differ them from ordinary functions. We define the definition of OTS , Reachable states, and invariant as the same in [OF06].

We suppose that all abstract data types have been predefined for the values used in a system and denote them by D_*. Let Υ denote a universal state space.

Definition 1 (OTSs). *An OTS $\mathcal{S}$ is a $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that*

- *$\mathcal{O}$: A finite set of observers . Each observer is a function $o_{x_1:D_{o1},\ldots,x_m:D_{om}} : \Upsilon \rightarrow D_o$ is an indexed function that has m indexes $x_1, \ldots, x_m$ whose types are $D_{o1}, \ldots, D_{om}$. The equivalence relation ($v_1 =_{\mathcal{S}} v_2$) betweenn two states $v_1, v_2 \in \Upsilon$ is defined as $\forall o_{x_1,\ldots,x_m} : \mathcal{O}.\ (o_{x_1,\ldots,x_m}(v_1) = o_{x_1,\ldots,x_m}(v_2))$, where $\forall o_{x_1,\ldots,x_m} : \mathcal{O}$ is the abbreviation of $\forall o_{x_1,\ldots,x_m} : \mathcal{O}.\ \forall x_1 : D_{o1} \ldots \forall x_m : D_{om}$.*

- *$\mathcal{I}$: The set of initial states such that $\mathcal{I} \subseteq \Upsilon$.*

- *$\mathcal{T}$: A finite set of transitions. Each transition $t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \rightarrow \Upsilon$ is an indexed function that has n indexes $y_1, \ldots, y_n$ whose types are $D_{t1}, \ldots, D_{tn}$ provided that $t_{y_1,\ldots,y_n}(v_1) =_{\mathcal{S}} t_{y_1,\ldots,y_n}(v_2)$ for each $[v] \in \Upsilon/ =_{\mathcal{S}}$, each $v_1, v_2 \in [v]$ and each $y_k : D_{tk}$ for $k = 1, \ldots, n$. $t_{y_1,\ldots,y_n}(v)$ is called the successor state of v wrt $\mathcal{S}$. Each transition*

$t_{y_1,\ldots,y_n}$ has the condition c-$t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \rightarrow Bool$, which is called the effective condition of the transition. If c-$t_{y_1,\ldots,y_n}(v)$ does not hold, then $t_{y_1,\ldots,y_n}(v) =_{\mathcal{S}} v$.

OTSs can be specified in CafeOBJ as equational specifications. Each equation defined for initial states is in the form of:

`eq` $o(v_0, x_1, \ldots, x_m) = T[x_1, \ldots, x_m]$.

Keyword `eq` is used to declare an equation in CafeOBJ. The above equation is defined for an observer in the form of $o_{x_1:D_{o1},\ldots,x_m:D_{om}} : \Upsilon \rightarrow D_o$, where $v_0, x_j (j = 1, \ldots, m)$ are variables of Υ and D_{oj} respectively. T is a term of D_o, representing the value observed by o with arguments $x_1, \ldots, x_m$ in all initial states.

Each equation defined for an observer $o_{x_1:D_{o1},\ldots,x_m:D_{om}} : \Upsilon \rightarrow D_o$ and a transition $t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \rightarrow \Upsilon$ is in the following form:

`ceq` $o(t(v, y_1, \ldots, y_n), x_1, \ldots, x_m) = T[v, y_1, \ldots, y_n, x_1, \ldots, x_m]$ `if` c-$t(v, y_1, \ldots, y_n)$.

Keyword `ceq` is used to declare a conditional equation. The equation specifies all the values observed by o in the state $t(v, y_1, \ldots, y_n)$, where $y_i (i = 1, \ldots, n)$ is a variable of D_{ti}. The condition part is the effective condition of t, which says if the effective condition holds, the values observed by o in the state $t(v, y_1, \ldots, y_n)$ are equal to those represented by the term T. If the effective condition does not hold, the state $t(v, y_1, \ldots, y_n)$ is equal to v, which is formalised by the following equation:

`ceq` $t(v, y_1, \ldots, y_n) = v$ `if not` c-$t(v, y_1, \ldots, y_n)$.

Definition 2 (Reachable states). *Given an OTS $\mathcal{S}$, reachable states wrt $\mathcal{S}$ are inductively defined:*

- *Each $v_{init} \in \mathcal{I}$ is reachable wrt $\mathcal{S}$.*
- *For each $t_{y_1,\ldots,y_n} \in \mathcal{T}$ and each $y_k : D_{tk}$ for $k = 1, \ldots, n, t_{x_1,\ldots,x_n}(v)$ is reachable wrt $\mathcal{S}$ if $v \in \Upsilon$ is reachable wrt $\mathcal{S}$.*

Let $\mathcal{R}_{\mathcal{S}}$ be the set of all reachable states wrt $\mathcal{S}$.

Predicates whose types are $\Upsilon \rightarrow$ `Bool` are called state *predicates*. All properties considered are *invariants*.

Definition 3 (Invariants). *Any state predicate $p : \Upsilon \rightarrow$ `Bool` is called invariant wrt $\mathcal{S}$, i.e. $\forall v : \mathcal{R}_{\mathcal{S}}.p(v)$.*

We suppose that each state predicate p considered has the form $\forall z_1 : D_{p1} \ldots \forall z_a : D_{pa}.P(v, z_1, \ldots, z_a)$, where $v, z_1, \ldots, z_a$ are all variables in p and $P(v, z_1, \ldots, z_a)$ does not contain any quantifiers.

2.3.2 Verification in OTS/CafeOBJ Method

Generally, there are two ways of verifying systems' properties in the OTS/CafeOBJ method. One is by searching(or model checking), another is by theorem proving.

Verification by Searching (Model Checking)

Searching is a technique for verifying properties in CafeOBJ. By searching, CafeOBJ traverses the states that are reachable from a given initial state, and check which states satisfy a specific condition. To check the condition for example safety property, it can be checked by the negation of the property. Then the counterexamples will be shown if there exists an execution path from an initial state to a state where the property does not hold, which is considered as failure of the property.

Searching in CafeOBJ is an effective way to find counterexamples, and particularly useful when the size of system's states is reasonably small. A more efficient searching functionality is implemented in Maude (a sibling language of CafeOBJ). Besides searching, model checking facilities is implemented in Maude which are more efficient to find counterexamples of invariant properties and even liveness properties.

Verification by Theorem Proving

Another technique is a verifying by theorem proving. The basic idea of verification is to construct proof score in CafeOBJ for an invariant property by using CafeOBJ as a proof assistant. Proof scores are instructions that can be executed in CafeOBJ. As we mentioned before that to do verification by theorem proving, human guidance is needed. In CafeOBJ, the user create proof plan in which proof should be performed. Then CafeOBJ evaluate proof scores based on the proof plan. A desired property is proved if the proof scores are successfully completed.

The strategy of constructing proof scores is by structural induction on system states and case analysis. In the based case, we check whether the property being proved holds in the initial states defined in an OTS. If it holds, we continue to deal with the induction case. Otherwise, the proof fails. In the induction case, we make the induction hypothesis for a state, for example s, and check whether it holds for all possible successor states of s. If it is true, the proof is finished, otherwise it fails. During proving, we may need some lemmas which are necessary to prove the main property, and if such that lemmas are used, we also need to prove these lemmas.

2.4 Maude

Maude [CDE+11] is a language and tool which focuses on simplicity, expressiveness, and performance. It is an algebraic specification, originated from OBJ family. The Maude specification formalism is based on first-order equational and rewriting logic specification techniques. Data types are dened by algebraic equational specifications in *membership equational logic*, which contains *order-sorted* equational logic as a sublogic.

In Maude, a *functional* module is declared with keywords `fmod` ... `endfm` and contains a set of declarations consisting of:

- importations of previously defined modules (e.g. `protecting`, `including`)
- declarations of sorts (`sort` s `.` or `sorts` s s' `.`)

- subsort declarations (`subsort` s `<` s' `.`)
- declarations of function symbols (`op` f : $s_1 \ldots s_n$ `->` s `.`)
- declarations of variables (`vars` v v' : s `.`)
- unconditional equations (`eq` $t = t'$ `.`), and
- conditional equations (`ceq` $t = t'$ `if` *cond* `.`)

For example, we declare a functional module of natural number as follows:

```
fmod NAT is
    protecting BOOL .
    sorts Zero NzNat Nat .
    subsort Zero NzNat < Nat .
    op 0 : -> Zero [ctor] .
    op s_ : Nat -> NzNat [ctor] .
endfm
```

where `0` is a constant for zero in natural number, and `s_` means the successor of the input value `Nat`. For instances, `s 0` means the successor of `0`, and it equals to "1" in natural number.

Besides functional module, maude has a *functional theories* declared with the keywords `fth` ... `endfth`. It can also do the same thing which functional module do such as declaring sorts, operators, and variables, and can import other theories or modules. Theories have a *loose* semantics, in the sense that any algebra satisfying the equations and membership axioms in the theory is an acceptable model.

However, there is a full maude which is the extension of maude. Full maude's syntax is similar with maude but some syntax are different for example, the parenthesis to cover the functional module or functional theories `(fmod` ... `endfm)` or `(fth` ... `endfth)`.

2.4.1 Constructor-Based Inductive Theorem Prover (CITP)

CITP [GZCA13] is a tool (currently implemented in Maude) for proving inductive properties of software systems specified with constructor based logics. CITP is equipped with a default proof strategy for the automated verification of OTS. The proof strategy can be created by user or the basic tactics can be applied.

A goal $\mathtt{SP} \vdash E$ consists of a specification `SP` and set of formulas E. The proof rules

$$\frac{\mathtt{SP}_1 \vdash E_1 \ldots \mathtt{SP}_n \vdash E_n}{\mathtt{SP} \vdash E}$$

of the specification can be regarded, upside down, as basic tactics for decomposing problems. By applying a tactic to a goal $\mathtt{SP} \vdash E$, we obtain the set of goals $\{\mathtt{SP}_1 \vdash E_1 \ldots \mathtt{SP}_n \vdash E_n\}$ if some preconditions are satisfied. The syntax to put a goal is as the follows:

`(goal` *ModuleName* `|-` *EquationSet/RuleSet/MemAxSet* `.)`

where *ModuleName* is the name of a Maude program representing a specification, *EquationSet*, *RuleSet* and *MemAxSet* are the equations, rewriting rules and memberships, respectively. After entering the goal, user only needs to give commands to discharge the goal. The basic tactics commands of CITP consist of

- Simultaneous Induction (`SI`): applies induction to a goal `SP` $\vdash$ `E` consisting of a specification `SP` and a set of formulas `E`. The *induction variables* are specified by the command `(set ind on` $VarSet$ `.)`. Each variable should be given with syntax consisting of an identifier (`X:Sort`). This tactic can be applied by giving the command `(apply SI.)`.

- Case Analysis (`CA`): adds conditions to the specification of a goal from conditional equations. Conditional equations marked with a string starting with `"CA-"` are used for case analysis. This tactic can be applied by giving the command `(apply CA .)`.

- Theorem of Constants (`TC`): instantiates variables appearing in the formula of the goal by fresh constants. The constants are automatically generated, and sort information of these constants are added to the specification. This tactic can be applied by giving the command `(apply TC .)`.

- Implication (`IP`): adds the condition of a quantifier-free sentence of a goal to the specification of the goal, as assumption. This tactic can be applied by giving the command `(apply IP .)`.

- Reduction (`RD`): is applied automatically by the system. Any goal can be reduced to the normal form if there are some equations or rewrite rules related to that goal. This tactic can be applied by giving the command `(apply RD .)`.

We can use the command `(auto .)` to discharge a goal automatically by applying the order of commands.

Besides these commands, user can add lemmas to help discharging the goal. It is considered as non-executable equations. It can enter into the CITP as follows:

`(init` *Lemma* `by` *Substitution* `.)`

Lemma is the label of a non-executable equation or rule that is initialized according to *Substitution* which is of the form $\mathtt{V}_1$ `<-` $\mathtt{T}_1$ `;` $\cdots$ `;` $\mathtt{V}_n$ `<-` $\mathtt{T}_n$, where $\mathtt{V}_i$ are variables and $\mathtt{T}_i$ are ground terms.

Example: `(init lemma-inv by X <- x ; Y <- y ; Z <- z .)`
The axioms labeled by `lemma-inv` is initialised by substituting the constants `x`, `y`, and `z` for `X`, `Y`, and `Z`, respectively.

In the case that the assumptions conflict or it cannot reduce to any normal form, we can make critical pair by two equations to make it reducible. The command to make critical pair is `(cp` *equation1* `. ><` *equation2* `.)`. Then use the command `(equation .)` to add the equation of critical pairs to the assumptions.

Chapter 3

A Survey of Verification of Consensus Algorithm

As we mentioned before that there are several algorithms to solve consensus problem. In [TS11], authors proposed a semi-automatic verification approach for asynchronous consensus algorithms by reducing the verification problem to small model checking problems that involve only single phases of algorithm execution. The authors did experiment with many consensus algorithms but we only focus on the *LastVoting* algorithm, which is similar with Paxos algorithm, with verification of agreement property. To better understanding of how they verify the LastVoting algorithm, we firstly explain their notation and a model that easily to understand for the verification part.

3.1 Round-Based Model

The consensus algorithm expressed in the asynchronous system augmented with failure detectors can be translated into a round-based consensus algorithm. We follows the round-based model as in [TS11].

In round-based model, the computation consists of rounds of message exchange. In each round r, each process p sends a message according to a sending function S_p^r to every process. At the end of round r, computes a new state according to a state transition function T_p^r. The state transition function takes as input the set of messages received in round r (a message sent in round r can only be received in round r) and the current process state.

Also they used the notation introduced by the Heard-Of (HO) model [CBS09]. If Π is the set of processes, $HO(p, r) \subseteq \Pi$ denotes the set of processes from which p receives a message in round $r : HO(p, r)$ is the "heard of" set of p in round r. If $q \notin HO(p, r)$ while q sent a message to p in round r, then p does not receive any message from q in round r. This can be due to the asynchrony of communication or process, or to a process or link failure.

The round-based model can naturally be extended to accommodate coordinator-based algorithms, by letting a communication predicate deal with not only HO sets but also

with coordinators.

A process is usually coordinator for a sequence of rounds, and this sequence of rounds is called a *phase*. We denote by k the number of rounds that compose a single phase. Let $Coord(p, \phi) \in \Pi$ denote the coordinator of process p in phase ϕ, and assume that p knows its coordinator $Coord(p, \phi)$ in phase ϕ and that the coordinator does not change during that phase. The domain is the collection of $HO(p, r)$ and $Coord(p, \phi)$, for all $p \in \Pi, r > 0, \phi > 0$. The sending function and the state transition function are now represented as $S_p^r(s_p, Coord(p, \phi))$ and $T_p^r(Msg, s_p, Coord(p, \phi))$, where ϕ is the phase that round r belongs to.

Since the correctness of an algorithm can be considered as safety or liveness. The safety is that in every phase no bad things happen, while the latter means that a single good phase is required to satisfy termination.

3.2 The LastVoting Algorithm

They presented the LastVoting algorithm (figure C.1 in Appendix C) that is used as a running example throughout the paper [CBS09]. LastVoting can be view as an HO model-version of Paxos. It is also close to the $\Diamond\mathcal{S}$ consensus algorithm by Chandra and Toueg [CT96].

In LastVoting a phase consists of four rounds. In the first round (round $4\phi - 3$), coordinators collect the current estimate x_p and the timestamp ts_p from processes. If a coordinator obtains these values from a majority of processes, then it picks up the estimate that is associated with the greatest timestamp and set $vote_p$ equal to that estimate. In the second round (round $4\phi - 2$) the coordinator broadcasts $vote_p$ to all processes. If a process p receives this value, then it updates timestamp ts_p to the current phase number ϕ and then votes for that value by replying ack to the coordinator in the third round (round $4\phi - 1$). If the coordinator obtains a majority of votes, then it again broadcasts the value of $vote_p$ in the fourth round (round4ϕ). If a process receives this value, then it decides on this value.

For the LastVoting algorithm, agreement can never be violated no matter how bad the HO set is; that is the algorithm is always safe, even in completely asynchronous runs. We only focus on the agreement property which is considered as safety, so we ignore other properties.

3.3 Verification of Agreement

This section is to verify the agreement property in LastVoting algorithm. It consists of two levels; phase-level analysis, which shows that agreement verification can be accomplished by examining only single phases of algorithm execution, and model checking of single phases describes how model checking can be used to analyse the single phases at the round level.

3.3.1 Phase level analysis

The agreement holds if, whenever all correct processes decide in a phase, which the decided values are the same (e.g. v). Formally, agreement holds if:

$$\begin{aligned} \forall c \in \mathit{Reachable} : \forall \langle c, d, c' \rangle \in R : \\ d = \emptyset \vee \exists v : (d = \{v\} \wedge c' \text{ is } v\text{-valent}) \end{aligned} \tag{3.1}$$

where $\langle c, d, c' \rangle$ corresponds to any phase that can occur such that: (1) c is the configuration at the beginning of the phase number ϕ of c (denoted by $\phi(c)$), (2) d is the set of all values decided in the phase, and (3) c' is the configuration at the beginning of the next phase (phase $\phi(c')$). *Reachable* is the set of all configurations that can occur in a run R, $d = \emptyset$ means that no decision is made in the phase, and $\exists v : (d = \{v\} \wedge c' \text{ is } v\text{-valent})$ means that a single value v is decided in that phase and the next phase starts with a v-valent configuration. The v-valent means that if the configuration decides the value v in some phase, the next phase must not decide the different value. For example, when the algorithm runs the sequence of run R can be infinite $c_1 d_1 c_2 d_2 \cdots$ such that c_1 is the set of configurations that can occur at the beginning of phase 1. If the value v decides in some phase (e.g. phase 5): $c_1 d_1 c_1 d_2 c_3 d_3 c_4 d_4 c_5 d_5$ and $d_5 = v$, then the next phase and others $c_6 d_6 c_7 d_7 \cdots$ the value of $d_6, d_7, \ldots$ must be equal to v. However, it is impractical to directly check this formula, because obtaining *Reachable* is as hard as examining all runs. So, the authors made an over-approximation of the set of reachable state, and it is usually referred to as an *invariant*.

The authors used a technique to prove the invariant called k-induction [MRS]; a generalization of induction. The knowledge about k-induction can see from [Wah13]. Since, they used the variation of k-induction to help verifying the invariant. So, the single phase that they considered is an inductive step for verifying the property. In the agreement verification, it is assumed that invariant Inv and a predicate $U(v)$ is specified where Inv is a set of configuration that is an invariant, $\mathit{Reachable} \subseteq Inv$, and $U(v)$ is a subset of v-valent reachable configurations.

Given Inv and $U(v)$, the agreement is modified that agreement holds if

$$\begin{aligned} \forall c \in Inv : \forall \langle c, d, c' \rangle \in R : \\ d = \emptyset \vee \exists v : (d = \{v\} \wedge c' \in U(v)) \end{aligned} \tag{3.2}$$

3.3.2 Model checking of single phases

To model check the LastVoting, it can be used to determine if Formula 3.2 holds or not. Model checking is the process of exploring a state transition system to determine whether or not a given property holds. Since in this problem involves only single phases, it only need to consider k consecutive state transitions of the LastVoting algorithm, where k is the number of rounds per phase. To model check the agreement property in model checker, the formula is changed as follows:

$$d = \emptyset \vee \exists v : (d = \{v\} \wedge c^{k+1} \in U(v)) \tag{3.3}$$

where $d = \left(\cup_{p\in\Pi,1\leq i\leq k}\{dv_p^i\}\right) \setminus \{?\}$.

The c^i is the configuration at the beginning of the i-th round of the phase. The dv_p^i is the value decided by each process p in the i-th round. If a process p does not decide in the round, then $dv_p^i =?$.

If the value of all processes have not been decided yet or have been decided the same value. These show that agreement holds for this algorithm. This model checking problem only concerns exactly k consecutive transitions. Because of this, *bounded model checking* [CBRZ01] can be most effectively used to solve it. The idea of bounded model checking is to reduce the model checking problem to the *satisfiability problem* for a formula in some logic.

In order to check 3.3 with this model checking technique, they constructed the formulas consisting of:

- X represents the behaviour of all one-phase executions.
- INV represents the invariant that $c^1 \in Inv$.
- Agr represents a formula 3.3 holds.

where

X is composed as $X \triangleq Dom \wedge T^1 \wedge T^2 \wedge \cdots \wedge T^k$ where
(1) Dom is a domain of one-phase executions. In the LastVoting algorithm one-phase execution has four rounds by including the HO model as its implementation. Since the logic of formula only allows integer and boolean variables. So, they mapped the possible decided value Val to $[1 \ldots \infty]$ and ? to 0. Consequently, the domain of LastVoting algorithm is considered that, for example, the decided value must be either greater or equal 0, the coordinator is between the process from 1 to n if there are n processes in the system.
(2) T^i is a mathematical representation of i-th round of the algorithm. So, the value i is four since the LastVoting algorithm has four rounds. In each round, it can be constructed some mathematical formulas by considering sending and receiving messages.

INV specifies that $c^1 \in Inv$. In the LastVoting algorithm c^1 is considered that for all process p, $commit_p$ and $ready_p$ are $false$ and timestamp ts_p is less than a phase number ϕ at the first round.

Agr specifies that 3.3 holds. It can be explained that the majority of processes send ack to the coordinator to agree on the same value and received by coordinator, then the value has been decided. On the other hand, the coordinator does not receive the majority of acks, then the value has not been decided yet.

From those mathematical formulas, the agreement verification can be checked by the satisfiability of:

$$X \wedge INV \wedge \neg Agr \tag{3.4}$$

This formula can only be satisfied by a value assignment corresponding to a one-phase execution that (1) starts from Inv and (2) for which 3.3 does not hold. Therefore, every

one-phase execution that starts from Inv meets 3.3 if and only if formula 3.4 is unsatisfiable.

Finally, they did some experiments with Yices [DdM06] satisfiability solver. The result shows that this method can verified the agreement property of LastVoting algorithm with around maximum 10 processes in the system. Difference from our approach that use theorem proving technique that does not need to bound any number of system and can be verified the infinite number of processes which is described in the Chapter 4.

Chapter 4

A Paxos Case Study

After we surveying of Paxos algorithm, we decided to conduct the behaviour of Paxos by modelling it which be described in the section *Paxos Model*. Then we formalized the Paxos model in CafeOBJ by using OTS, and maude. After that we verified an agreement property, which is one of safety properties in the consensus distributed system, by using theorem proving technique (proof score in OTS/CafeOBJ and CITP in maude).

4.1 Paxos Model

This section described our model of Paxos by representing in a state diagram. The state diagram can be divided into three parts by the role of agent in Paxos algorithm; proposer, acceptor, and learner.

4.1.1 Model of Proposer

We formalized the model of proposer as a transition system $M_p = \langle Q_p, q_{init-p}, \delta_p \rangle$ where

(a) Q_p is a finite set of state M_p for each $q \in Q_p$, where

$$q = \langle p\text{-}l_p, N_{u-p}, V_{c-p}, list_p, nw \rangle$$

The meaning of each element in q are

(i) p-l_p: a label indicating state of proposer p

(ii) N_{u-p}: a unique proposal number of proposer p

(iii) V_{c-p}: a value from client of proposer p

(iv) $list_p$: a list of proposer p to receive promise messages

(v) nw: a network in the system

(b) q_{init-p} is the initial state of M_p where the initial value of each element is:

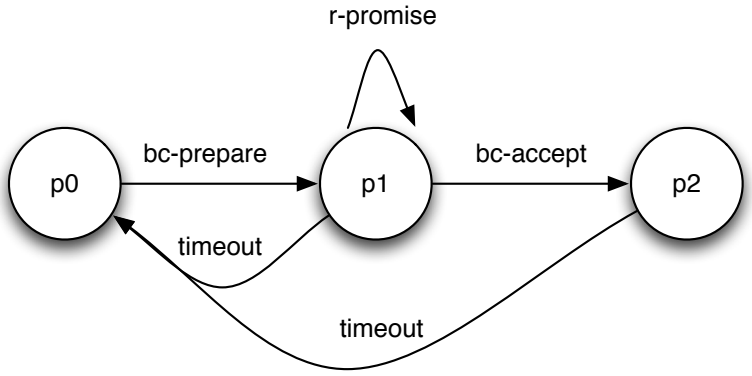

Figure 4.1: State transition system of proposer

(i) $p\text{-}l_p$:= $p0$

(ii) N_{u-p}:= unique number (e.g. process id)

(iii) V_{c-p}:= v, where v is the value which come from client

(iv) $list_p$:= empty list

(v) nw := empty

(c) δ_p is transition rules which δ_p can be represented in Figure 4.1 where $\delta_p = (q, q')$ if

$$q = \langle p\text{-}l_p, N_{u-p}, V_{c-p}, list_p, nw \rangle$$
$$q' = \langle p\text{-}l'_p, N'_{u-p}, V'_{c-p}, list'_p, nw' \rangle$$

The transition rules of δ_p are defined as *action*(left) and *transition rule*(right) divided by the symbol **":"**.

(i) **bc-prepare :** $p\text{-}l_p = p0 \wedge p\text{-}l'_p := p1 \wedge nw' := Prepare < N_{u-p} > \cup\, nw$

(ii) **r-promise :** $p\text{-}l_p = p1 \wedge p\text{-}l'_p := p1 \wedge Promise < N_{a-ac}, V_{a-ac} > \in nw \wedge list'_p := update(N_{a-ac}, V_{a-ac}, list_p)$,
where N_{a-ac} is a number of proposal that has been accepted by acceptor ac, and V_{a-ac} is a accepted value of acceptor ac

(iii) **bc-accept :** $p\text{-}l_p = p1 \wedge p\text{-}l'_p := p2 \wedge |list_p| \geq \lceil \frac{\#Acceptor}{2} \rceil \wedge nw' := Accept < N_{u-p}, V >$,
where $|list_p|$ is a length of $list_p$, and $\#Acceptor$ is the number of acceptors in the system, and V may be V_{c-p} or V_{a-ac} from promise messages

(iv) **timeout :** $(p\text{-}l_p = p1 \vee p\text{-}l_p = p2) \wedge p\text{-}l'_p := p0 \wedge list\text{-}ld'_p :=$ empty list $\wedge N'_{u-p} := N_{u-p} + \#process$,
where $\#process$ is the total number of processes in the system

The state transition system of Paxos in Figure 4.1 shows behaviour of leader proposer in the system. Beginning with state $p0$, the leader proposer puts the prepare message into the network with its unique number to all of acceptors. Then it goes to the state $p1$ waiting the promise messages. Each acceptor receives the prepare message depending on a condition, and replies the promise message back with the accepted number (N_a) and accepted value (V_a), which initial vale are 0 and $null$, respectively. The leader proposer receives the promise message by updating the content of message into its list. Since the majority of promise messages arrive, the leader proposer can decide the value depending on the content of promise messages. It selects the value with the highest proposal number among the promise message, but if the value is $null$, it chooses the value from client (V_c). After that it broadcasts the accept message to all acceptors again, go to state $p2$, and waits for consensus value from learner.

Timeout can occur since the assumption of network that message can be loss. The leader proposer may not receive majority of promise messages in $p1$ or accept message is loss when the leader is in $p2$. When timeout occurs, the leader must go to state $p0$ and starts sending a prepare message with new unique number again.

4.1.2 Model of Acceptor

The model of acceptor was focused as the a transition system $M_a = \langle Q_a, q_{init-a}, \delta_a \rangle$ where

(a) Q_a is a finite set of state M_a for each $q \in Q_a$, where

$$q = \langle a\text{-}l_{ac}, N_{p-ac}, N_{a-ac}, V_{a-ac}, nw \rangle$$

The meaning of each element in q are

(i) $a\text{-}l_{ac}$: a label indicating state of acceptor ac

(ii) N_{p-ac}: a promise number of acceptor ac

(iii) N_{a-ac}: an accepted number of acceptor ac

(iv) V_{a-ac}: an accepted value of acceptor ac

(v) nw: a network in the system

(b) q_{init-a} is the initial state of M_a where the initial value of each element is:

(i) $a\text{-}l_{ac} := a0$

(ii) $N_{p-ac} := 0$

(iii) $N_{a-ac} := 0$

(iv) $V_{a-ac} := null$

(v) $nw :=$ empty

(c) δ_a is transition rules which δ_a can be represented in Figure 4.2 where $\delta_a = (q, q')$ if

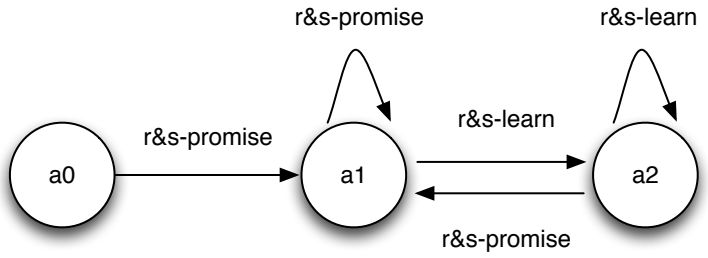

Figure 4.2: State transition system of acceptor

$$q = \langle a\text{-}l_{ac}, N_{p-ac}, N_{a-ac}, V_{a-ac}, nw \rangle$$
$$q' = \langle a\text{-}l'_{ac}, N'_{p-ac}, N'_{a-ac}, V'_{a-ac}, nw' \rangle$$

The transition rules of δ_a are defined as *action* and *transition rule*.

(i) **r&s-promise :** $(a\text{-}l_{ac} = a0 \vee a\text{-}l_{ac} = a1 \vee a\text{-}l_{ac} = a2) \wedge Prepare < N_{u-p} > \in nw \wedge N_{u-p} \geq N_{p-ac} \wedge a\text{-}l'_{ac} := a1 \wedge N'_{p-ac} := N_{u_p} \wedge nw' := Promise < N_{a-ac}, V_{a-ac} > \cup nw$

(ii) **r&s−learn :** $(a\text{-}l_{ac} = a1 \vee a\text{-}l_{ac} = a2) \wedge Accept < N_{u-p}, V > \in nw \wedge N_{u-p} \geq N_{p-ac} \wedge a\text{-}l'_{ac} := a2 \wedge N'_{p-ac} := N_{u_p} \wedge N'_{a-ac} := N_{u_p} \wedge V'_{a-ac} := V \wedge nw' := Learn < V_{a-ac} > \cup nw$

Figure 4.2 shows how acceptors do when there are some prepare and accept messages in the network by representing in the state transition system. By initial state the all acceptors are in state $a0$ and wait for a leader proposer sending the prepare message. Each acceptor receives the message and check the content whether its promise number(N_{p-ac}) less than the unique number of proposerN_{u-p}. If so, it updates its promise number to be equal to the unique number($N_{p-ac} := N_{u-p}$), and sends a promise message containing accepted number(N_{a-ac}) and accepted value(V_{a-ac}) back to the leader proposer. Otherwise, it rejects the message and does not reply anything.

Since there may be some situations that multiple leaders are in the system. In this case, once each acceptor receives a prepare message and returns a promise message back, this acceptor can accept other prepare messages from other leaders which have their unique number greater than or equal to its promise number. That is why in a state $a1$, it has a transition **r&s-promise** which is point out and go back to itself.

After sending a promise message, acceptor waits for accept message. When it comes, acceptor checks with the same condition when it received the prepare message. If the condition is satisfied, it updates their local values; consisting of promise number(N_{p-ac}), accepted number(N_{a-ac}), and accepted value(V_{a-ac}). These values is change to be equal to N_{u-p}, N_{u-p}, and V respectively. Then each acceptor sends $Learn < V_{a-ac} >$ to all learners, and change a state to $a2$.

4.1.3 Model of Learner

The important part of Paxos is learner because they receive values from acceptors and decide a value. This value must be a single value which is the same in all learners. Learner has a responsible for broadcasting this value to all processes but we ignore this action. So, the model of leaner can be represented as a transition system $M_l = \langle Q_l, q_{init-l}, \delta_l \rangle$ where

(a) Q_l is a finite set of state M_l for each $q \in Q_l$, where

$$q = \langle l\text{-}l_l, list_l, V_{l-l}, nw \rangle$$

The meaning of each element in q are

(i) $l\text{-}l_l$: a label indicating state of learner l

(ii) $list_l$: a list of learner l to receive learn messages

(iii) V_{l-l}: a consensus value of leaner l

(iv) nw: a network in the system

(b) q_{init-l} is the initial state of M_l where the initial value of each element is:

(i) $l\text{-}l_l := l0$

(ii) $list_l := emptylist$

(iii) $V_{l-l} := null$

(iv) $nw := empty$

(c) δ_l is transition rules which δ_l can be represented in Figure 4.3 where $\delta_l = (q, q')$ if

$$q = \langle l\text{-}l_l, list_l, V_{l-l}, nw \rangle$$
$$q' = \langle l\text{-}l'_l, list'_l, V'_{l-l}, nw' \rangle$$

The transition rules of δ_l are defined as *action* and *transition rule*.

(i) **r-learn :** $l\text{-}l_l = l0 \wedge Learn < N_{a-ac}, V_{a-ac} > \in nw \wedge V_{a-ac} \neq null \wedge l\text{-}l'_l := l0 \wedge list'_l := update(V_{a-ac}, list_l)$

(ii) **decide :** $l\text{-}l_l = l0 \wedge |list_l| \geq \lceil \frac{\#Acceptor}{2} \rceil \wedge decideV(list_l) \neq null \wedge l\text{-}l'_l := l1 \wedge V'_l := decideV(list_l)$,
where $decideV$ is a function to return a majority of values in the $list_l$

The transition system in Figure 4.3 shows that learner only receives a learn message from all acceptors, and update its list($list_l$). If the majority of acceptors decided the same value, then they send the same value by learn messages in the network. So, the majority of element of list of learner will have the same value, and learner can decide a value based on majority of received value from acceptors. The value, which sending from acceptors, must not be *null*, and must be some proposed value from some leader proposers.

The model of Paxos can be represented as a transition system by combining the model of proposer, acceptor, and learner $M_{paxos} = M_p \wedge M_a \wedge M_l$ where nw is a network for the Paxos system.

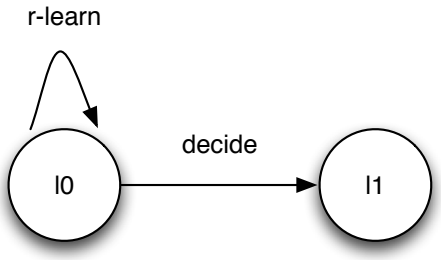

Figure 4.3: State transition system of learner

4.2 Specification of OTSs in CafeOBJ

This section describes how we formalized OTSs of a Paxos system in OTS/CafeOBJ. Υ is denoted by a sort, says `Sys`. Each $o \in \mathcal{O}$ is denoted by an operator called an *observation operator* declared as follows:

`op o : Sys` $\mathtt{D}_{o1}$ ... $\mathtt{D}_{om}$ `-> D`$_o$

where each $\mathtt{D}_*$ is a sort corresponding to D_*.

An arbitrary initial state $\mathcal{I}$ is demoted by an operator declared as follows:

```
op init : -> Sys {constr}
```

For each $o \in \mathcal{O}$, the following equation is declared:

`eq o(init,`$\mathtt{X}_1,\ldots,\mathtt{X}_m$`) =` $\mathtt{T}^{\mathtt{o}}_{\mathtt{X}_1,\ldots,\mathtt{X}_m}$ `.`

where each $\mathtt{X}_*$ is a CafeOBJ variable of sort $\mathtt{D}_*$ and $\mathtt{T}^{\mathtt{o}}_{\mathtt{X}_1,\ldots,\mathtt{X}_m}$ is a term denoting the value returned by o, together with any other parameters, in an arbitrary initial state.

Each $t \in \mathcal{T}$ is denoted by an operator called a *transition operator* declared as follows:

`op t : Sys` $\mathtt{D}_{t1}$ ... $\mathtt{D}_{tn}$ `-> Sys {constr}`

For each `o` and `t`, a conditional equation is declared:

`ceq o(t(S,`$\mathtt{Y}_1,\ldots,\mathtt{Y}_n$`),`$\mathtt{X}_1,\ldots,\mathtt{X}_m$`) =` $\mathtt{o\text{-}t}_{\mathtt{S},\mathtt{Y}_1,\ldots,\mathtt{Y}_n,\mathtt{X}_1,\ldots,\mathtt{X}_m}$ `if c-t(S,`$\mathtt{Y}_1,\ldots,\mathtt{Y}_n$`) .`

where `c-t(S,...)` corresponds to c-$t(v,\ldots)$, and $\mathtt{o\text{-}t}_S\mathtt{,\ldots}$ is a term whose sort is the same as the sort of `o` and does not use any transition operators. The equation says how t changes the value observed by o if the effective condition holds. If $\mathtt{o\text{-}t}_S\mathtt{,\ldots}$ is always equal to `o(S,`$\mathtt{X}_1\mathtt{,\ldots,}\mathtt{X}_m$`)`, the condition may be omitted.

For each `t`, one more conditional equation is declared:

`ceq t(S,`$\mathtt{Y}_1,\ldots,\mathtt{Y}_n$`) = S if not c-t(S,`$\mathtt{Y}_1,\ldots,\mathtt{Y}_n$`) .`

which says that t changes nothing if the effective condition does not hold.

As indicated by `constr`, `init` and each `t` are constructors of `Sys`, which corresponds to $\mathcal{R}_{\mathcal{S}}$.

4.2.1 Paxos Observers

Paxos is formalized as an OTS $\mathcal{S}_{Paxos}$. $\mathcal{S}_{Paxos}$ uses 12 observers based on each role of agents. The corresponding observation operators are declared as follows:

```
op p-l : Sys Proposer -> Label
op a-l : Sys Acceptor -> Label
op l-l : Sys Learner -> Label
op nw : Sys -> Network
op unique : Sys Proposer -> Num
op vClient : Sys Proposer -> Val
op list-Ld : Sys Proposer -> TriList
op n-p : Sys Acceptor -> Num
op n-a : Sys Acceptor -> Num
op v-a : Sys Acceptor -> Val
op list-Ln : Sys Learner -> PairList
op v-d : Sys Learner -> Val
```

where module `Label` is a label of state consisting of *p0, p1, p2, a0, a1, a2, l0 and l1.* The module `Network` is a channel for sending and receiving messages. The module `Num` and `Val` are number and value which has been describe in the Paxos algorithm. The module `TriList` and `PairList` are list where element in the list are triple and pair, respectively. For more information and better understanding of module used in Paxos can be found in Appendix A.

Given a state s : `Sys`, proposer ID p : `Proposer`, acceptor ID a : `acceptor`, and learner ID l : `Learner`.

- `p-l`(s, p), `a-l`(s, a) and `l-l`(s, l) denote the label at which process p, a and l are in the state s.
- `nw`(s) denotes the shared network in the state s.
- `unique`(s, p) denotes the unique number of proposer p in the state s.
- `vClient`(s, p) denotes the client value of proposer p in the state s.
- `list-Ld`(s, p) denotes the list of triple $< Ac : Acceptor, N : Num, V : Val >$ of proposer p in the state s.
- `n-p`(s, a) denotes the promise number of acceptor a in the state s.
- `n-a`(s, a) denotes the accepted number of acceptor a in the state s.
- `v-a`(s, a) denotes the accepted value of acceptor a in the state s.
- `list-Ln`(s, l) denotes the list of pair $< Ac : Acceptor, V : Val >$ of learner l in the state s.

- `v-d`(s, l) denotes the consensus value of learner l in the state s.

In the rest of this section, we declared the CafeOBJ variables as follows:

```
var S : Sys
vars Pp Pp1 : Proposer
vars Ac Ac1 : Acceptor
vars Ln Ln1 : Learner
var N : Num
var V : Val
```

And we have the following equations for `init`, the constant denoting an arbitrary initial state:

```
op init : -> Sys

eq p-l(init,Pp) = p0 .
eq a-l(init,Ac) = a0 .
eq l-l(init,Ln) = l0 .
eq nw(init) = noMsg .
eq unique(init,Pp) = PpID .
eq vClient(init,Pp) = vc .
eq list-Ld(init,Pp) = tnil .
eq n-p(init,Ac:Acceptor) = 0 .
eq n-a(init,Ac:Acceptor) = 0 .
eq v-a(init,Ac:Acceptor) = null .
eq list-Ln(init,Ln:Learner) = pnil .
eq v-d(init,Ln:Learner) = null .
```

where `noMsg` is a constant denotes the empty channel in module the `NETWORK` (see in Appendix A), `PpID` is a constant denotes the unique ID number of proposer, `vc` is a constant denotes the client value, `tnil` and `pnil` is a constant denotes the empty list of module `TRILIST` and `PAIRLIST`, respectively.

4.2.2 Paxos Transitions

$\mathcal{S}_{Paxos}$ uses eight transitions. The corresponding transition operators are declared as follows:

```
op bc-prepare : Sys Proposer -> Sys {constr}
op r-promise : Sys Proposer Acceptor Num Val -> Sys {constr}
op bc-accept : Sys Proposer -> Sys {constr}
op timeout : Sys Proposer -> Sys {constr}
op r&s-promise : Sys Acceptor Proposer Num -> Sys {constr}
op r&s-learn : Sys Proposer Acceptor Num Val -> Sys {constr}
op r-learn : Sys Acceptor Learner Num Val -> Sys {constr}
op decide : Sys Learner -> Sys {constr}
```

Given a state s : `Sys`, proposer ID p : `Proposer`, acceptor ID a : `acceptor`, learner ID l : `Learner`, number n : `Num`, and value v : `Val`.

- `bc-prepare`(s, p) denotes the successor state of s when p executes the statement at label p0 in s
- `r-promise`(s, p, a, n, v) denotes the successor state of s when p executes the statement of receiving $Promise < n, v >$ which sending from a at label p1 in s
- `bc-accept`(s, p) denotes the successor state of s when p executes the statement at label p1 in s
- `timeout`(s, p) denotes the successor state of s when p executes the statement at label p1 or p2 in s
- `r&s-promise`(s, a, p, n) denotes the successor state of s when a executes the statement of receiving $Prepare < n >$ from p at label a0, a1 or a2 in s
- `r&s-learn`(s, p, a, n, v) denotes the successor state of s when a executes the statement of receiving $Accept < n, v >$ from p at label a1 or a2 in s
- `r-learn`(s, a, l, n, v) denotes the successor state of s when l executes the statement of receiving $Learn < v >$ from a at label l0 in s
- `decide`(s, l) denotes the successor state of s when l executes the statement at label l0 in s

The set of equations for `bc-prepare` is as follows:

```
ceq p-l(bc-prepare(S,Pp1),Pp) = (if Pp1 = Pp then p1 else pc(S,Pp) fi)
    if c-bc-prepare(S,Pp1) .
eq a-l(bc-prepare(S,Pp),Ac) = pc(S,Ac) .
eq l-l(bc-prepare(S,Pp),Ln) = pc(S,Ln) .
ceq nw(bc-prepare(S,Pp)) = prepare-m(Pp,unique(S,Pp)) nw(S) if c-bc-prepare(S,Pp) .
eq unique(bc-prepare(S,Pp1),Pp) = unique(S,Pp) .
eq vClient(bc-prepare(S,Pp1),Pp) = vClient(S,Pp) .
eq list-Ld(bc-prepare(S,Pp1),Pp) = list-Ld(S,Pp) .
eq n-p(bc-prepare(S,Pp),Ac) = n-p(S,Ac) .
eq n-a(bc-prepare(S,Pp),Ac) = n-a(S,Ac) .
eq v-a(bc-prepare(S,Pp),Ac) = v-a(S,Ac) .
eq list-Ln(bc-prepare(S,Pp),Ln) = list-Ln(S,Ln) .
eq v-d(bc-prepare(S,Pp),Ln) = v-d(S,Ln) .
ceq bc-prepare(S,Pp) = S if not c-bc-prepare(S,Pp) .
```

where the operator `c-bc-prepare` is declared and defined as follows:

```
op c-bc-prepare : Sys Proposer -> Bool
eq c-bc-prepare(S,Pp) = (p-l(S,Pp) = p0) .
```

The transition `bc-prepare` has a responsible for putting a prepare message `prepare-m` into the network when the condition `c-bc-prepare` (proposer `Pp` is in a state at label p0) is satisfied.

There are four messages in the system which has been describe in the module `MESSAGE` as follows:

```
mod! MESSAGE {
  pr(PROPOSER)
  pr(ACCEPTOR)
  pr(LEARNER)
  pr(NUM)
  pr(VALUE)
  [Msg]
  -- operators
  op prepare-m : Proposer Num -> Msg {constr}
  op promise-m : Acceptor Proposer Num Val -> Msg {constr}
  op accept-m : Proposer Num Val -> Msg {constr}
  op learn-m : Acceptor Num Val -> Msg {constr}
  op _=_ : Msg Msg -> Bool {comm}
  -- variables
  vars N N1 : Num
  vars V V1 : Val
  vars Pp Pp1 : Proposer
  vars Ac Ac1 : Acceptor
  vars Ln Ln1 : Learner
  -- equations
  eq (prepare-m(Pp,N) = prepare-m(Pp1,N1)) = ((Pp = Pp1) and (N = N1)) .
  eq (promise-m(Ac,Pp,N,V) = promise-m(Ac1,Pp1,N1,V1)) = ((Ac = Ac1)
     and (Pp = Pp1) and (N = N1) and (V = V1)) .
  eq (accept-m(Pp,N,V) = accept-m(Pp1,N1,V1)) = ((Pp = Pp1)
     and (N = N1) and (V = V1)) .
  eq (learn-m(Ac,N,V) = learn-m(Ac1,N1,V1)) = ((Ac = Ac1)
     and (N = N1) and (V = V1)) .

  eq (prepare-m(Pp,N) = promise-m(Ac,Pp1,N1,V)) = false .
  eq (prepare-m(Pp,N) = accept-m(Pp1,N1,V)) = false .
  eq (prepare-m(Pp,N) = learn-m(Ac,N1,V)) = false .
  eq (promise-m(Ac,Pp,N,V) = accept-m(Pp1,N1,V1)) = false .
  eq (promise-m(Ac,Pp,N,V) = learn-m(Ac1,N1,V1)) = false .
  eq (accept-m(Pp,N,V) = learn-m(Ac,N1,V1)) = false .
}
```

The module MESSAGE consists of four messages; prepare-m, promise-m, accept and learn-m. Two messages can be the same message if the name and content of message are the same. For example, prepare-m is not the same as promise-m. The message between prepare-m(p1,n1) and prepare-m(p2,n2) are the same message if proposer p1 and p2 are the same proposer and same unique number (n1 = n2). Otherwise, the messages are different.

The set of equations for r-promise is as follows:

```
ceq p-l(r-promise(S,Pp1,Ac,N,V),Pp) = (if Pp1 = Pp then p1 else p-l(S,Pp) fi)
    if c-r-promise(S,Pp1,Ac,N,V) .
eq a-l(r-promise(S,Pp,Ac1,N,V),Ac) =  a-l(S,Ac) .
eq l-l(r-promise(S,Pp,Ac,N,V),Ln) =  l-l(S,Ln) .
eq nw(r-promise(S,Pp,Ac,N,V)) = nw(S) .
eq unique(r-promise(S,Pp1,Ac,N,V),Pp) = unique(S,Pp) .
eq vClient(r-promise(S,Pp1,Ac,N,V),Pp) = vClient(S,Pp) .
ceq list-Ld(r-promise(S,Pp1,Ac,N,V),Pp) = (if Pp1 = Pp then updateTri(< Ac , N , V >,
    list-Ld(S,Pp)) else list-Ld(S,Pp) fi) if c-r-promise(S,Pp1,Ac,N,V) .
eq n-p(r-promise(S,Pp,Ac1,N,V),Ac) = n-p(S,Ac) .
eq n-a(r-promise(S,Pp,Ac1,N,V),Ac) = n-a(S,Ac) .
eq v-a(r-promise(S,Pp,Ac1,N,V),Ac) = v-a(S,Ac) .
eq list-Ln(r-promise(S,Pp,Ac,N,V),Ln) = list-Ln(S,Ln) .
eq v-d(r-promise(S,Pp,Ac,N,V),Ln) = v-d(S,Ln) .
ceq r-promise(S,Pp,Ac,N,V) = S if not c-r-promise(S,Pp,Ac,N,V) .
```

where the operator c-r-promise is declared and defined as follows:

```
op c-r-promise : Sys Proposer Acceptor Num Val -> Bool
eq c-r-promise(S,Pp,Ac,N,V) = ((p-l(S,Pp) = p1) and (member(promise-m(Ac,Pp,N,V),
   nw(S)))) .
```

In the transition r-promise, the leader proposer updates its list-Ld with the content of promise message receiving from network by using a function updateTri. However, the leader must check a condition c-r-promise whether the condition (proposer Pp is in a state at label p1 and promise-m is in nw) is satisfied.

The function updateTri is in the module TRILIST and it was defined as follows:

```
mod! TRILIST {
  pr(LIST(X <= TRIV2TRI-ARY)
     * {sort List -> TriList,
         op nil -> tnil})
  -- operator
  op updateTri : Tri TriList -> TriList
  -- variables
  vars Ac Ac1 : Acceptor
```

```
  vars N N1 : Num
  vars V V1 : Val
  var TL : TriList
  -- equations
  eq updateTri(< Ac , N , V >,tnil) = < Ac , N , V > | tnil .
  ceq updateTri(< Ac , N , V >,< Ac , N1 , V1 > | TL)
      = < Ac , N , V > | TL if N >= N1 .
  ceq updateTri(< Ac , N , V >,< Ac , N1 , V1 > | TL)
      = < Ac , N1 , V1 > | TL if N < N1 .
  ceq updateTri(< Ac , N , V >,< Ac1 , N1 , V1 > | TL)
      = < Ac1 , N1 , V1 > | updateTri(< Ac , N , V >,TL) if not(Ac = Ac1) .
}
```

This function is updating triple, which consists of `<Ac,N,V>` ; Acceptor, Num, Val, into the list of triple `TL`. If `TL` is `tnil`, it updates by adding the element into the list, otherwise checks the first element of triple. If the message comes from the same acceptor, it checks the number `N` and `N1` again. If `N` is less than `N1`, the updating is ignored. Otherwise, it replaces the value `N1` and `V1` with `N` and `V`. Another case that message comes from different acceptors, it checks again with other remaining element in the list.

The function `member` is in the module `NETWORK` and it was defined as follows:

```
mod! NETWORK {
  pr(MULTISET(X <= TRIV2MESSAGE)
     * {sort MSet -> Network,
         op empty -> noMsg} )
  -- operator
  op member : Msg Network -> Bool
  -- variables
  var NW : Network
  vars M M1 : Msg
  -- equations
  eq member(M,noMsg) = false .
  eq member(M,M NW) = true .
  ceq member(M,M1 NW) = member(M,NW) if not(M = M1) .
}
```

The function `member` just checks whether message is in the network `NW`. If the message is in `NW`, it returns `true`, otherwise returns `false`.

The set of equations for `bc-accept` is as follows:

```
ceq p-l(bc-accept(S,Pp1),Pp) = (if Pp1 = Pp then p2 else p-l(S,Pp) fi)
    if c-bc-accept(S,Pp1) .
eq a-l(bc-accept(S,Pp),Ac) =  a-l(S,Ac) .
eq l-l(bc-accept(S,Pp),Ln) =  l-l(S,Ln) .
```

```
ceq nw(bc-accept(S,Pp)) = accept-m(Pp,unique(S,Pp),findV(list-Ld(S,Pp))) nw(S)
    if (c-bc-accept(S,Pp) and not(findV(list-Ld(S,Pp)) = null)) .
ceq nw(bc-accept(S,Pp)) = accept-m(Pp,unique(S,Pp),vClient(S,Pp)) nw(S)
    if (c-bc-accept(S,Pp) and (findV(list-Ld(S,Pp)) = null)) .
eq unique(bc-accept(S,Pp1),Pp) = unique(S,Pp) .
eq vClient(bc-accept(S,Pp1),Pp) = vClient(S,Pp) .
ceq list-Ld(bc-accept(S,Pp1),Pp) = (if Pp1 = Pp then tnil else list-Ld(S,Pp) fi)
    if c-bc-accept(S,Pp1) .
eq n-p(bc-accept(S,Pp),Ac) = n-p(S,Ac) .
eq n-a(bc-accept(S,Pp),Ac) = n-a(S,Ac) .
eq v-a(bc-accept(S,Pp),Ac) = v-a(S,Ac) .
eq list-Ln(bc-accept(S,Pp),Ln) = list-Ln(S,Ln) .
eq v-d(bc-accept(S,Pp),Ln) = v-d(S,Ln) .
ceq bc-accept(S,Pp) = S if not c-bc-accept(S,Pp) .
```

where the operator `c-bc-accept` is declared and defined as follows:

```
op c-bc-accept : Sys Proposer -> Bool
eq c-bc-accept(S,Pp) = ((p-l(S,Pp) = p1)
   and ((len(list-Ld(S,Pp)) * 2) >= (nAcceptor + 1))) .
```

The transition `bc-accept` selects a value whether the value among the promise messages in `list-Ld` or the value from client `vClient` depending on the value returned by function `findV`. If the returned value of `findV` is *null*, broadcasts the `vClient` to all acceptors with unique number. Otherwise, selects the value with the highest accepted number among the promise messages and sends it with unique number by accept message in the network. After broadcasting the accept message, `list-Ld` changes to be empty, and the state is changed to label p2.

It can do the `bc-accept` when the condition `c-bc-accept` (proposer `Pp` is in a state at label p1 and received the majority of promise messages from acceptors; it can be defined by the length of `list-Ld`) is satisfied.

To select the value, it depends on the function `findV` in the module `TRILIST` which is described as follows:

```
mod! TRILIST {
  pr(LIST(X <= TRIV2TRI-ARY)
     * {sort List -> TriList,
         op nil -> tnil})
  -- operator
  op findV : TriList -> Val
  -- variables
  vars Ac Ac1 : Acceptor
  vars N N1 : Num
  vars V V1 : Val
```

```
  var TL : TriList
  -- equations
  eq findV(tnil) = null .
  eq findV(< Ac , N , V > | tnil) = V .
  ceq findV(< Ac , N , V > | < Ac1 , N1 , V1 > | TL)
      = findV(< Ac , N , V > | TL) if N >= N1 .
  ceq findV(< Ac , N , V > | < Ac1 , N1 , V1 > | TL)
      = findV(< Ac1 , N1 , V1 > | TL) if N < N1 .
}
```

The function `findV` returns the value with the highest number from the list. Firstly, if the list is empty return the value *null*. If it is not empty, finds the element in triple with the highest number `N` or `N1`, and keeps this triple. It checks until there is only one element triple in the list and returns only the value `V` from the triple.

The set of equations for `timeout` is as follows:

```
ceq p-l(timeout(S,Pp1),Pp) = (if Pp1 = Pp then p0 else p-l(S,Pp) fi)
    if c-timeout(S,Pp1) .
eq a-l(timeout(S,Pp),Ac) = a-l(S,Ac) .
eq l-l(timeout(S,Pp),Ln) = l-l(S,Ln) .
eq nw(timeout(S,Pp)) = nw(S) .
ceq unique(timeout(S,Pp1),Pp) = (if Pp1 = Pp then (unique(S,Pp) + nProcess)
    else unique(S,Pp) fi) if c-timeout(S,Pp1) .
eq vClient(timeout(S,Pp1),Pp) = vClient(S,Pp) .
ceq list-Ld(timeout(S,Pp1),Pp) = (if Pp1 = Pp then tnil else list-Ld(S,Pp) fi)
    if c-timeout(S,Pp1) .
eq n-p(timeout(S,Pp),Ac) = n-p(S,Ac) .
eq n-a(timeout(S,Pp),Ac) = n-a(S,Ac) .
eq v-a(timeout(S,Pp),Ac) = v-a(S,Ac) .
eq list-Ln(timeout(S,Pp),Ln) = list-Ln(S,Ln) .
eq v-d(timeout(S,Pp),Ln) = v-d(S,Ln) .
ceq timeout(S,Pp) = S if not c-timeout(S,Pp) .
```

where the operator `c-timeout` is declared and defined as follows:

```
op c-timeout : Sys Proposer -> Bool
eq c-timeout(S,Pp) = ((p-l(S,Pp) = p1) or (p-l(S,Pp) = p2)) .
```

Since, the assumption that message can be lost. The leader proposer may not receive the majority of promise messages, and cause the leader waits forever. So, the leader must have a time to count when it sends the prepare message to all acceptors. If it does not receive a majority of messages in a period of time in a state $p1$ or $p2$, timeout occurs. When the transition `timeout` occurs, everything must be reset. The unique number `unique` must be increasing but still is unique. There are several ways to increment the

unique number, one way is increment by the number of processes. The list `list-Ld` must also be reset to be an empty list, and a state changes to p0 to start sending a prepare message.

The set of equations for `r&s-promise` is as follows:

```
eq p-l(r&s-promise(S,Ac,Pp1,N),Pp) = p-l(S,Pp) .
ceq a-l(r&s-promise(S,Ac1,Pp,N),Ac) = (if Ac1 = Ac then a1 else a-l(S,Ac) fi)
    if c-r&s-promise(S,Pp,Ac1,N) .
eq l-l(r&s-promise(S,Ac,Pp,N),Ln) = l-l(S,Ln) .
ceq nw(r&s-promise(S,Ac,Pp,N)) = promise-m(Ac,Pp,n-a(S,Ac),v-a(S,Ac)) nw(S)
    if (c-r&s-promise(S,Pp,Ac,N) and N >= n-p(S,Ac)) .
ceq nw(r&s-promise(S,Ac,Pp,N)) = nw(S) if (c-r&s-promise(S,Pp,Ac,N)
    and N < n-p(S,Ac)) .
eq unique(r&s-promise(S,Ac,Pp1,N),Pp) = unique(S,Pp) .
eq vClient(r&s-promise(S,Ac,Pp1,N),Pp) = vClient(S,Pp) .
eq list-Ld(r&s-promise(S,Ac,Pp1,N),Pp) = list-Ld(S,Pp) .
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = (if Ac1 = Ac then N else n-p(S,Ac) fi)
    if (c-r&s-promise(S,Pp,Ac1,N) and N >= n-p(S,Ac1)) .
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = n-p(S,Ac) if (c-r&s-promise(S,Pp,Ac1,N)
    and N < n-p(S,Ac1)) .
eq n-a(r&s-promise(S,Ac1,Pp,N),Ac) = n-a(S,Ac) .
eq v-a(r&s-promise(S,Ac1,Pp,N),Ac) = v-a(S,Ac) .
eq list-Ln(r&s-promise(S,Ac,Pp,N),Ln) = list-Ln(S,Ln) .
eq v-d(r&s-promise(S,Ac,Pp,N),Ln) = v-d(S,Ln) .
ceq r&s-promise(S,Ac,Pp,N) = S if not c-r&s-promise(S,Pp,Ac,N) .
```

where the operator `c-r&s-promise` is declared and defined as follows:

```
op c-r&s-promise : Sys Proposer Acceptor Num -> Bool
eq c-r&s-promise(S,Pp,Ac,N) = (((a-l(S,Ac) = a0) or (a-l(S,Ac) = a1)
   or (a-l(S,Ac) = a2)) and member(prepare-m(Pp,N),nw(S))) .
```

The transition `r&s-promise` checks whether the prepare message is in the network, and then changes state to $a1$. To put the promise message into the network, it must check the number `N`, which came from the prepare message. If `N` is less than the promise number `n-p`, it does not reply and update anything. Otherwise, it updates the local number `n-p` to be equal to `N`, and sends the promise message back with value `n-a` and `v-a` for promising not accepting the messages with lower number than `n-p`.

The set of equations for `r&s-learn` is as follows:

```
eq p-l(r&s-learn(S,Pp1,Ac,N,V),Pp) = p-l(S,Pp) .
ceq a-l(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then a2 else a-l(S,Ac) fi)
    if(c-r&s-learn(S,Pp,Ac1,N,V)) .
eq l-l(r&s-learn(S,Pp,Ac,N,V),Ln) = l-l(S,Ln) .
```

```
ceq nw(r&s-learn(S,Pp,Ac,N,V)) = learn-m(Ac,N,V) nw(S)
    if (c-r&s-learn(S,Pp,Ac,N,V) and N >= n-p(S,Ac)) .
ceq nw(r&s-learn(S,Pp,Ac,N,V)) = nw(S) if (c-r&s-learn(S,Pp,Ac,N,V)
    and N < n-p(S,Ac)) .
eq unique(r&s-learn(S,Pp1,Ac,N,V),Pp) = unique(S,Pp) .
eq vClient(r&s-learn(S,Pp1,Ac,N,V),Pp) = vClient(S,Pp) .
eq list-Ld(r&s-learn(S,Pp1,Ac,N,V),Pp) = list-Ld(S,Pp) .
ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then N else n-p(S,Ac) fi)
    if (c-r&s-learn(S,Pp,Ac1,N,V) and N >= n-p(S,Ac1)) .
ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-p(S,Ac) if (c-r&s-learn(S,Pp,Ac1,N,V)
    and N < n-p(S,Ac1)) .
ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then N else n-a(S,Ac) fi)
    if (c-r&s-learn(S,Pp,Ac1,N,V) and N >= n-p(S,Ac1)) .
ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-a(S,Ac) if (c-r&s-learn(S,Pp,Ac1,N,V)
    and N < n-p(S,Ac1)) .
ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then V else v-a(S,Ac) fi)
    if (c-r&s-learn(S,Pp,Ac1,N,V) and N >= n-p(S,Ac1) and not(V = v-a(S,Ac))) .
ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then V else v-a(S,Ac) fi)
    if (c-r&s-learn(S,Pp,Ac1,N,V) and N >= n-p(S,Ac1) and V = v-a(S,Ac)) .
ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = v-a(S,Ac) if (c-r&s-learn(S,Pp,Ac1,N,V)
    and N < n-p(S,Ac1)) .
eq list-Ln(r&s-learn(S,Pp,Ac,N,V),Ln) = list-Ln(S,Ln) .
eq v-d(r&s-learn(S,Pp,Ac,N,V),Ln) = v-d(S,Ln) .
ceq r&s-learn(S,Pp,Ac,N,V) = S if not c-r&s-learn(S,Pp,Ac,N,V) .
```

where the operator `c-r&s-learn` is declared and defined as follows:

```
op c-r&s-learn : Sys Proposer Acceptor Num Val -> Bool
eq c-r&s-learn(S,Pp,Ac,N,V) =  (((a-l(S,Ac) = a1) or (a-l(S,Ac) = a2))
   and member(accept-m(Pp,N,V),nw(S))) .
```

The transition `r&s-learn` checks whether accept message is in the network, then change a state to $a2$. Before putting the learn message into the network. A condition which the number `N` from `accept-m(Pp,N,V)` must greater or equal to the promise number `n-p` should be satisfied. If the condition is satisfied, then the acceptor updates its promise number (`n-p`) and accepted number (`n-a`) to be equal to `N`, and accepted value (`v-a`) to be equal to `V` when the value `V` is not the same as `v-a`, and also put the learn message (`learn-m(Ac,N,V)`) into the network . Otherwise, it does not change anything.

The set of equations for `r-learn` is as follows:

```
eq p-l(r-learn(S,Ac,Ln,N,V),Pp) = p-l(S,Pp) .
eq a-l(r-learn(S,Ac1,Ln,N,V),Ac) = a-l(S,Ac) .
ceq l-l(r-learn(S,Ac,Ln1,N,V),Ln) = (if Ln1 = Ln then l0 else l-l(S,Ln) fi)
    if c-r-learn(S,Ac,Ln1,N,V) .
```

```
eq nw(r-learn(S,Ac,Ln,N,V)) = nw(S) .
eq unique(r-learn(S,Ac,Ln,N,V),Pp) = unique(S,Pp) .
eq vClient(r-learn(S,Ac,Ln,N,V),Pp) = vClient(S,Pp) .
eq list-Ld(r-learn(S,Ac,Ln,N,V),Pp) = list-Ld(S,Pp) .
eq n-p(r-learn(S,Ac1,Ln,N,V),Ac) = n-p(S,Ac) .
eq n-a(r-learn(S,Ac1,Ln,N,V),Ac) = n-a(S,Ac) .
eq v-a(r-learn(S,Ac1,Ln,N,V),Ac) = v-a(S,Ac) .
ceq list-Ln(r-learn(S,Ac,Ln1,N,V),Ln) = (if (Ln1 = Ln) then
    updateTri(< Ac , N , V >,list-Ln(S,Ln1)) else list-Ln(S,Ln) fi)
        if c-r-learn(S,Ac,Ln1,N,V) and not(V = null) .
eq v-d(r-learn(S,Ac,Ln1,N,V),Ln) = v-d(S,Ln) .
ceq r-learn(S,Ac,Ln,N,V) = S if not c-r-learn(S,Ac,Ln,N,V) .
```

where the operator `c-r-learn` is declared and defined as follows:

```
op c-r-learn : Sys Acceptor Learner Num Val -> Bool
eq c-r-learn(S,Ac,Ln,N,V) = ((l-l(S,Ln) = l0)
   and member(learn-m(Ac,N,V),nw(S))) .
```

The transition `r-learn` only updates the list of learner `list-Ln` by using the function `updateTri`. It can be updated the list when the network `nw` has a learn message `learn-m(Ac,N,V)` and when the value `V` is not equal to *null*.

The set of equations for `decide` is as follows:

```
eq p-l(decide(S,Ln),Pp) = p-l(S,Pp) .
eq a-l(decide(S,Ln),Ac) = a-l(S,Ac) .
ceq l-l(decide(S,Ln1),Ln) = (if Ln1 = Ln then l1 else l-l(S,Ln) fi)
    if c-decide(S,Ln1) .
eq nw(decide(S,Ln)) = nw(S) .
eq unique(decide(S,Ln),Pp) = unique(S,Pp) .
eq vClient(decide(S,Ln),Pp) = vClient(S,Pp) .
eq list-Ld(decide(S,Ln),Pp) = list-Ld(S,Pp) .
eq n-p(decide(S,Ln),Ac) = n-p(S,Ac) .
eq n-a(decide(S,Ln),Ac) = n-a(S,Ac) .
eq v-a(decide(S,Ln),Ac) = v-a(S,Ac) .
eq list-Ln(decide(S,Ln1),Ln) = list-Ln(S,Ln) .
ceq v-d(decide(S,Ln1),Ln) = (if (Ln1 = Ln) then decideV(list-Ln(S,Ln1))
    else v-d(S,Ln) fi) if c-decide(S,Ln1) .
ceq decide(S,Ln) = S if not c-decide(S,Ln) .
```

where the operator `c-decide` is declared and defined as follows:

```
op c-decide : Sys Learner -> Bool
eq c-decide(S,Ln) = ((l-l(S,Ln) = l0)
   and ((majN(ctoPL(list-Ln(S,Ln))) * 2) >= (nAcceptor + 1))
   and not(decideV(list-Ln(S,Ln)) = null)) .
```

The transition `decide` is very important part of the Paxos system because it has a responsible to decide the same value for all learners. Each learner decides a value when the majority of acceptors agree on the same value and the decided value is not *null*. That can be represented by a function `((majN(ctoPL(list-Ln(S,Ln))) * 2) >= (nAcceptor + 1))` and `not(decideV(list-Ln(S,Ln)) = null)`. If the condition is satisfied, learner decides a value. Both functions `majN` and `decideV` are in a module `PAIRLIST`.

The module `PAIRLIST` was defined as follows:

```
mod! PAIRLIST {
  pr(LIST(X <= TRIV2PAIR)
     * {sort List -> PairList,
        op nil -> pnil})
  pr(TRILIST)
  -- observers
  op decideV : TriList -> Val
  op ctoPL : TriList -> PairList
  op incV : Val PairList -> PairList
  op majV : PairList -> Val
  op majN : PairList -> Nat
  -- variables
  vars Ac Ac1 : Acceptor
  vars V V1 : Val
  vars Nu : Num
  vars N N1 : Nat
  var PL : PairList
  vars TL : TriList
  -- equations
  eq decideV(TL) = majV(ctoPL(TL)) .

  eq ctoPL(tnil) = pnil .
  eq ctoPL(< Ac , Nu , V > | TL) = incV(V,ctoPL(TL)) .

  eq incV(V,pnil) = < V , 1 > | pnil .
  eq incV(V,< V , N > | PL) = < V , N + 1 > | PL .
  ceq incV(V,< V1 , N > | PL) = < V1 , N > | incV(V,PL) if not(V = V1) .

  eq majV(pnil) = null .
  eq majV(< V , N > | pnil) = V .
  ceq majV(< V , N > | < V1 , N1 > | PL) = majV(< V , N > | PL)
      if (N > N1) .
  ceq majV(< V , N > | < V1 , N1 > | PL) = majV(< V1 , N1 > | PL)
      if (N <= N1) .

  eq majN(pnil) = 0 .
```

```
eq majN(< V , N > | pnil) = N .
ceq majN(< V , N > | < V1 , N1 > | PL) = majN(< V , N > | PL)
    if (N > N1) .
ceq majN(< V , N > | < V1 , N1 > | PL) = majN(< V1 , N1 > | PL)
    if (N <= N1) .
```

Function `ctoPL` do changing the `TriList` to `PairList` by only keeping the value `V` from the element `<Ac,N,V>` in the list of triple, and counts the number of occurrences of value `V` in the list by the auxiliary function `incV`. The result of `ctoPL` is the list of pair where the element of list consists of two values $< V, N >$; V for value, and N for number of occurrences of V. For example, if there are five acceptors and three of them decide the same value $V1$, others decides value $V2$, and all acceptors sent learn message to a learner. The result of function `ctoPL` by input is the list of learner is `<V1,3> | <V2,2> | pnil`

The function `majV` and `majN` are similar. Both of them take the list of pair (`PairList`) as input, do the same algorithm; finding the maximum occurrences of value V, but return different result. The function `majV` returns the value V , however `majN` returns the number N of occurrences of V. So, the function `decideV` returns the value with the number of occurrences of value V is the highest in list of pair.

One of properties that Paxos should enjoy is the agreement property, which can be expressed as an invariant wrt $\mathcal{S}_{Paxos}$. The state predicate concerned is denoted by the operator declared and defined as follows:

```
op inv1 : Sys Learner Learner -> Bool
eq inv1(S,Ln,Ln1) = (not(v-d(S,Ln) = null) and not(v-d(S,Ln1) = null)
                     implies v-d(S,Ln) = v-d(S,Ln1)) .
```

To verify that Paxos enjoys the agreement property, all we have to do is to prove that $(\forall l, l1 : \texttt{Learner})$ $\texttt{inv1}(s, l, l1)$ is an invariant wrt $\mathcal{S}_{Paxos}$.

We suppose that $\mathcal{S}_{Paxos}$ is specified as a module `PAXOS`, and operators denoting state predicate such as `inv1` are declared in a module `PRED-PAXOS` that imports `PAXOS` (see in Appendix A).

4.3 Verification

4.3.1 Verification by Proof Scores

In the OTS/CafeOBJ method, invariants are proved by writing proof scores in CafeOBJ and executing them with its processor. We use the proof of $(\forall s : \texttt{Sys})(\forall l, l1 : \texttt{Learner})$ $\texttt{inv1}(s, l, l1)$ as a case study to describe proof scores in the OTS/CafeOBJ method.

We do a proof score on Paxos specification (see in Appendix B). We prove the invariant $\texttt{inv1}(s, l, l1)$ by structural induction on s. To do the prove by induction, we need to split into two cases; base case and induction case.

Base case, a state s takes place by an initial state `init`, and we do a proof score as follows:

```
--> base case
open INV1
  -- arbitrary objects
  ops l l1 : -> Learner .
  red inv1(init,l,l1) .
close
```

Since, the initial value `v-d` for both l and $l1$ are *null*. CafeOBJ returns *true* for the initial case.

Induction case, we assume the *induction hypothesis* that an invariant (`inv1`$(s, l, l1)$) is true in the state s, then we do a proof score for a successor state of each transition; `bc-prepare`, `r-promise`, `bc-accept`, `timeout`, `r&s-prepare`, `r&s-learn`, `r-learn`, and `decide`. All transitions except for a transition *decide*, CafeOBJ easily returns *true* because the predicate *inv*1 is related to `v-d` of learner which will have a value after transition *decide*. Some examples of proof score of the transition *bc-prepare* and *r&s-promise* are shown as below:

```
--> bc-prepare
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
   op pp : -> Proposer .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(bc-prepare(s,pp),l,l1) .
close

--> r&s-promise
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
   op pp : -> Proposer .
   op ac : -> Acceptor .
   op n : -> Num .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(r&s-promise(s,ac,pp,n),l,l1) .
close
```

where `inv1(bc-prepare(s,pp),l,l1)` and `inv1(r&s-promise(s,ac,pp,n),l,l1)` denote the formula to prove in the induction case, and `inv1(s,l,l1)` denotes instances of the induction hypothesis used.

While the transition *decide* has to do a case analysis by dividing into seven sub-cases as follows:

```
1. c-decide(s,l2) = true, l = l2, l1 = l2
2. c-decide(s,l2) = true, l = l2, (l1 = l2) = false, v-d(s,l1) = null
3. c-decide(s,l2) = true, l = l2, (l1 = l2) = false,
   (v-d(s,l1) = null) = false
4. c-decide(s,l2) = true, (l = l2) = false, l1 = l2, v-d(s,l) = null
5. c-decide(s,l2) = true, (l = l2) = false, l1 = l2,
   (v-d(s,l) = null) = false
6. c-decide(s,l2) = true, (l = l2) = false, (l1 = l2) = false
7. c-decide(s,l2) = false
```

where `s`, `l`, `l1` and `l2` are constants of `Sys`, `Learner`, `Learner` and `Learner`, respectively. `s` is used to denote an arbitrary state, and `l`, `l1` and `l2` for arbitrary learners. Sub-cases 3 and 5 need to use other state predicates as assumptions. Other state predicates are denoted by the operators declared and defined as follows:

```
op inv2 : Sys Learner Learner -> Bool
eq inv2(S,L,L1) = ((((majN(ctoPL(list-Ln(S,L))) * 2) >= (nAcceptor + 1))
   and ((majN(ctoPL(list-Ln(S,L1))) * 2) >= (nAcceptor + 1)))
      implies majV(ctoPL(list-Ln(S,L))) = majV(ctoPL(list-Ln(S,L1)))) .

op inv3 : Sys Learner -> Bool
eq inv3(S,L) = (not(v-d(S,L) = null)
   implies ((majN(ctoPL(list-Ln(S,L))) * 2) >= (nAcceptor + 1))) .

op inv4 : Sys Learner -> Bool
eq inv4(S,L) = (not(v-d(S,L) = null)
   implies majV(ctoPL(list-Ln(S,L))) = v-d(S,L)) .
```

For example, proving the sub-cases 3, we apply these three state predicates as follows:

```
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
-- variables
   var S : Sys
   vars L L1 : Learner
-- assumptions
   eq l-l(s,l2) = l0 .
   eq ((majN(ctoPL(list-Ln(s,l2))) * 2) >= (nAcceptor + 1)) = true .
   eq (majV(ctoPL(list-Ln(s,l2))) = null) = false .
```

```
   eq l = l2 .
   eq (l1 = l2) = false .
   eq (v-d(s,l1) = null) = false .
   -- inv2
   ceq majV(ctoPL(list-Ln(s,l2))) = majV(ctoPL(list-Ln(s,l1)))
       if (((majN(ctoPL(list-Ln(s,l2))) * 2) >= (nAcceptor + 1))
       and ((majN(ctoPL(list-Ln(s,l1))) * 2) >= (nAcceptor + 1))) .
   -- inv3
   ceq ((majN(ctoPL(list-Ln(s,l1))) * 2) >= (nAcceptor + 1)) = true
       if not(v-d(s,l1) = null) .
   -- inv4
   ceq majV(ctoPL(list-Ln(s,l1))) = v-d(s,l1) if not(v-d(s,l1) = null) .
-- |=
-- check if the predicate is true.
   red inv1(decide(s,l2),l,l1) .
close
```

by replacing variables `S`, `L` and `L1` with the constants `s`, `l2` and `l1` in `inv2(S,L,L1)`, respectively, and also replacing variables `S` and `L` with constants `s` and `l1` in `inv3(S,L)` and `inv4(S,L)`, respectively. The result of reduction `inv1(decide(s,l2),l,l1)` is *true*.

Proving of `inv1(S,L,L1)` is not successful since proving `inv1(S,L,L1)` needs three more invariants (`inv2`, `inv3`, and `inv4`). So, we need to continue proving these three invariants by doing proof scores. If we successfully prove them, in that case we can say that the `inv1(S,L,L1)` (agreement property) holds for Paxos.

4.3.2 Verification by CITP

Besides proof scores in CafeOBJ, we also do proving of agreement property of Paxos in CITP (see in Appendix E). To do that we specified the Paxos in maude (see in Apeendix D) which is similar to implementing in OTS/CafeOBJ.

To do a proof in CITP, firstly we set an invariant of agreement as a goal.

```
(goal GOAL |- ceq (v-d(S:Sys,L:Learner) ~ v-d(S:Sys,L1:Learner))
     = true if (v-d(S:Sys,L:Learner) ~ null) = false
     /\ (v-d(S:Sys,L1:Learner) ~ null) = false ; )
```

To discharge the goal, we need other invariants as we used in proof scores which defined in the module as follows:

```
(fth GOAL is
  inc PAXOS .
  var S : Sys .
  vars L L1 : Learner .
  ceq [lemma1] : majV(ctoPL(list-Ln(S,L))) = majV(ctoPL(list-Ln(S,L1)))
```

```
      if ((majN(ctoPL(list-Ln(S,L))) * 2) >= (nAcceptor + 1)) = true
      /\ ((majN(ctoPL(list-Ln(S,L1))) * 2) >= (nAcceptor + 1)) = true [nonexec] .
  ceq [lemma2] : ((majN(ctoPL(list-Ln(S,L))) * 2) >= (nAcceptor + 1)) = true
    if (v-d(S,L) ~ null) = false [nonexec] .
  ceq [lemma3] : majV(ctoPL(list-Ln(S,L))) = v-d(S,L)
    if (v-d(S,L) ~ null) = false [nonexec] .
endfth)
```

where `inv2`, `inv3` and `inv4` are considered in CITP as `lemma1`, `lemma2` and `lemma3`, respectively.

```
~~~~~~~~~~~~~~~~~~~~~~~~      Generated GOALS      ~~~~~~~~~~~~~~~~~~~~~~~~

============================ GOAL 1-1-2-17-1 ============================
< Module GOAL is concealed
  ...
  op L#3 : -> Learner .
  op L1#4 : -> Learner .
  op x#1 : -> Sys .
  op z#2 : -> Learner .
  ...

  eq L#3 = z#2  . --> added by prover!
  eq len(list-Ln(x#1,z#2))* 2 >= nAcceptor + 1 = true  . --> added by prover!
  eq majN(ctoPL(list-Ln(x#1,z#2)))* 2 >= nAcceptor + 1 = true  . --> added by prover!
  eq L1#4 ~ z#2 = false  . --> added by prover!
  eq null ~ majV(ctoPL(list-Ln(x#1,z#2))) = false  . --> added by prover!
  eq null ~ v-d(x#1,L1#4) = false  . --> added by prover!
  eq l-l(x#1,z#2) = l0  . --> added by prover!
  ceq v-d(x#1,L:Learner)~ v-d(x#1,L1:Learner) = true
      if null ~ v-d(x#1,L:Learner)= false /\ null ~ v-d(x#1,L1:Learner)= false  . --> added by
    prover!
  ceq majN(ctoPL(list-Ln(S:Sys,L:Learner)))* 2 >= nAcceptor + 1
    = true
    if v-d(S:Sys,L:Learner)~ null = false [label lemma2 nonexec] .

  ceq majV(ctoPL(list-Ln(S:Sys,L:Learner)))
    = majV(ctoPL(list-Ln(S:Sys,L1:Learner)))
    if majN(ctoPL(list-Ln(S:Sys,L:Learner)))* 2 >= nAcceptor + 1 = true /\ majN(ctoPL(list-Ln(
    S:Sys,L1:Learner)))* 2 >= nAcceptor + 1 = true [label lemma1 nonexec] .

  ceq majV(ctoPL(list-Ln(S:Sys,L:Learner)))
    = v-d(S:Sys,L:Learner)
    if v-d(S:Sys,L:Learner)~ null = false [label lemma3 nonexec] .


  End of the module,

  eq majV(ctoPL(list-Ln(x#1,z#2)))~ v-d(x#1,L1#4) = true [metadata "added"]--> to be proved!
>
unproved
```

Figure 4.4: Sub-goal of transition decide

We use induction technique to prove by setting induction on `S (set ind on S:Sys .)`. Then use command to do the induction `(apply SI .)`. CITP returns the nine sub-goals since there are nine transition in Paxos including an initial state.

As the same in proof scores in OTS/CafeOBJ, we have to use `lemma1`, `lemma2` and `lemma3` to discharge the sub-goal of transition `decide` in Paxos. For example, in the Figure 4.4

We apply these three lemmas to this sub-goal by entering the constant to each variable as follows:

```
(init lemma1 by (S:Sys <- x#1 ;) (L:Learner <- z#2 ;) (L1:Learner <- L1#4 ;) .)
(init lemma2 by (S:Sys <- x#1 ;) (L:Learner <- L1#4 ;) .)
(init lemma3 by (S:Sys <- x#1 ;) (L:Learner <- L1#4 ;) .)
```

To apply `lemma1` to discharge a sub-goal, we use the command `init lemma1` to add `lemma1` as assumption by replacing all variables of `S`, `L` and `L1` with the constant `x#1`, `z#2` and `L1#4`, respectively.

To add `lemma2` and `lemma3` as assumptions, we do the same as `lemma1`. Then we apply the reduction `(apply RD .)` to prove this sub-goal. The CITP returns that this sub-goal is proved.

Except the transition `decide`, others can easily prove by using the basic commands in CITP.

4.4 Result of Verification

To do verification, we sometimes need to look back and check the specification whether it is correct or not. After re-checking on specification many times, we incidentally found a counterexample for our Paxos model. The counterexample shows the situation that there are two leader proposers in the system and they compete to send their prepare message. Since the asynchronous system, messages can take arbitrary long to be delivered. So there is a situation that two learners can decide different values.

The counterexample begins with *Leader 1* sends $Prepare < 1 >$ to all acceptors (figure 4.5). Each acceptor checks the condition between N_p and $N_{u_{Leader1}} = 1$. However, the initial value of N_p and N_a are 0, and V_a is $null$, so that each acceptor replies the $Promise < 0, null >$ to *Leader 1* but these promise messages take arbitrary long and have not been received by *Leader 1*. Also timeout of *Leader 1* occurs and starts a new round.

Since Paxos can have multiple leaders, and in figure 4.6 a new leader proposer (*Leader 2*) is active and sends $Prepare < 2 >$ to all acceptors. Since acceptors have not received any accept message yet, they also reply $Promise < 0, null >$ to *Leader 2*. After considering the value among promise messages, *Leader 2* sends the value from its client $V2$ to all acceptors. Acceptors update their local values and sends $Learn < V2 >$ to all learners. In this case, only *Learner 1* receives the learn messages and decide a consensus value $V2$. Because of message delay, *Learner 2* have not received the learn messages from all acceptors.

Figure 4.7 shows that after timeout, *Leader 1* selects new unique number (3) and sends it to all acceptors again. At this time, all acceptors have already accepted the accept message from *Leader 2*, so they reply $Promise < 2, V2 >$ to *Leader 1* but *Leader 1* have not received these promise messages yet.

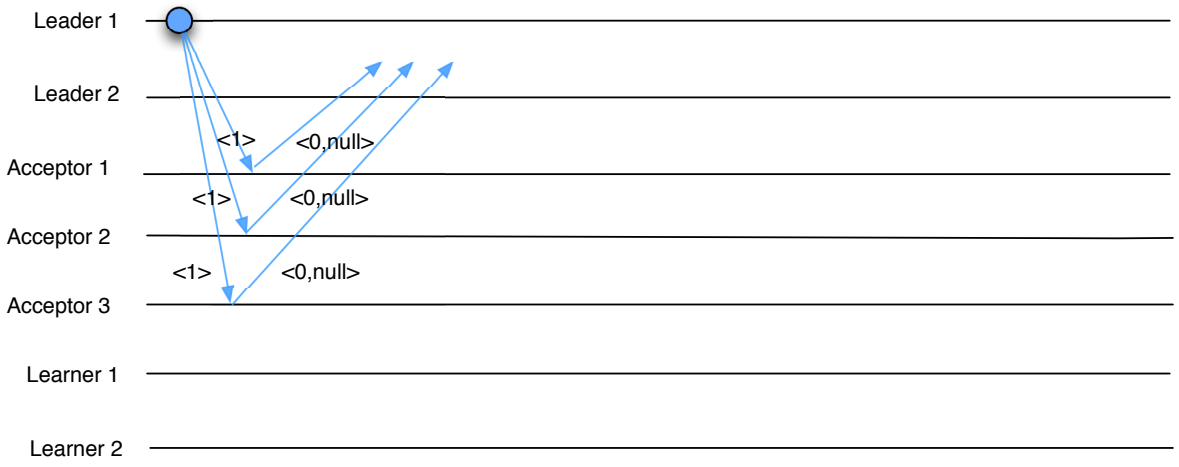

Figure 4.5: Conterexample of agreement property1

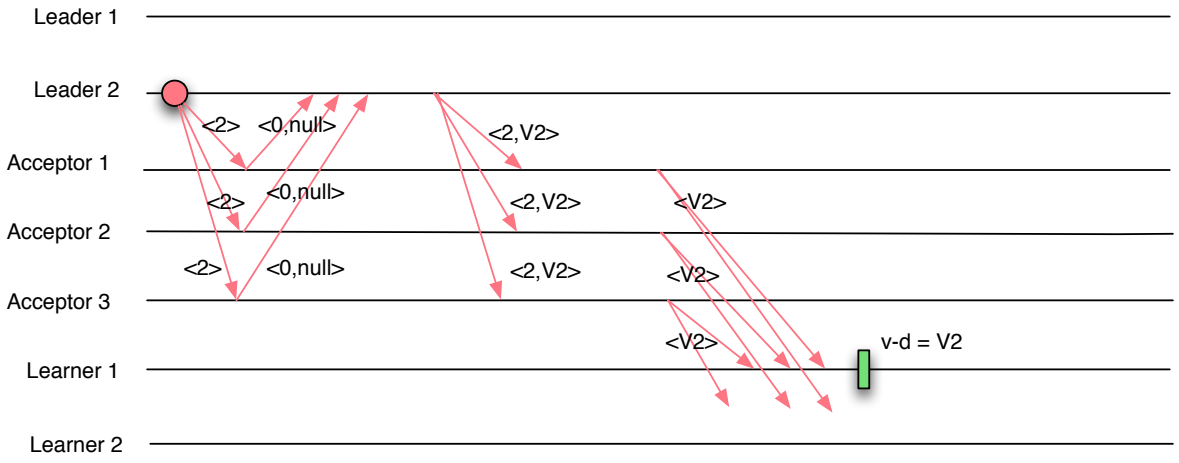

Figure 4.6: Conterexample of agreement property2

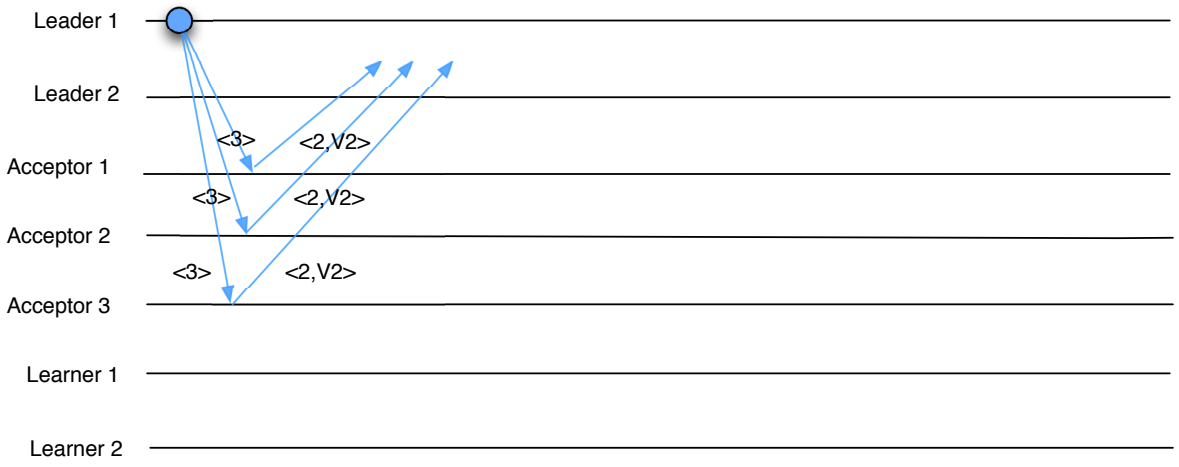

Figure 4.7: Conterexample of agreement property3

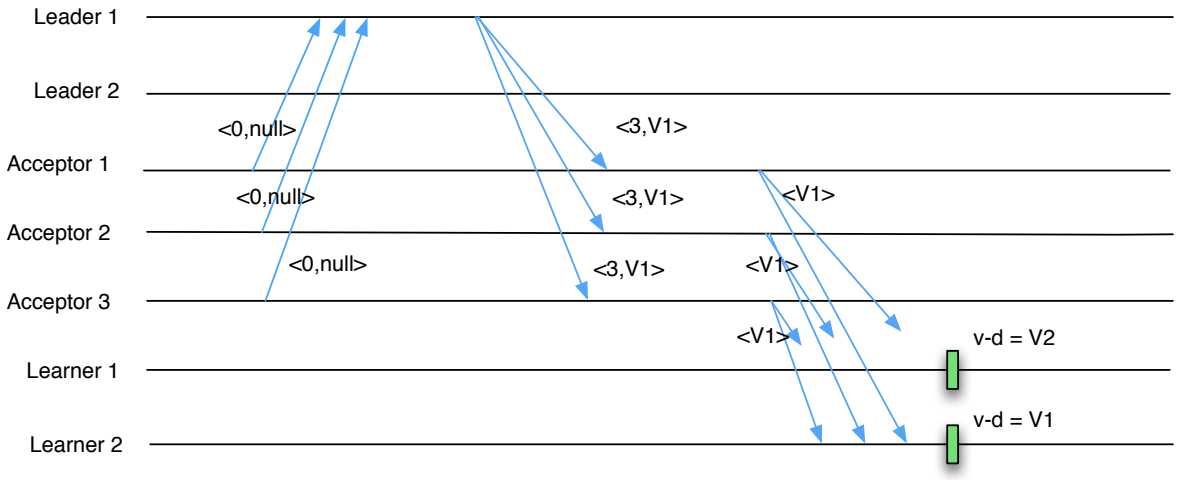

Figure 4.8: Conterexample of agreement property4

Since Paxos is asynchronous model, message from previous round can be received in these round. In figure 4.8, *Leader 1* receives $Promise < 0, null >$ from previous round, then it selects the value from its client $V1$ because all values in the promise messages are *null*. It sends $Accept < 3, V1 >$ to all acceptors, and acceptors receive and update their local value since $N_u = 3$ greater than $N_p = 2$. Acceptors broadcast the new decided value $V1$ to all learners. At this time, only *Learner 2* receives and decides the consensus value $V1$ which the value $V1$ can be different from $V2$ in *Learner 1*. Therefore, the agreement property that all learners decide the same value is violated.

The problem is that the content of promise message do not have the round number value to tell which round of the message sending by acceptors. So, we have to re-check with the original Paxos algorithm in [Lam01a].

The Paxos algorithm in [Lam01a] did not describe that the promise message should have one more parameter of N_p in both main part of Paxos algorithm and proof. It only describes mostly the end of paper that "*The Paxos consensus algorithm is precisely the one described above, where requests and responses are sent as ordinary messages. (Response messages are tagged with the corresponding proposal number to prevent confusion.)*". Moreover, we found another paper [MM13] describes the pseudocode (Figure 4.9) of Paxos algorithm that promise message consists of three parameters (*rnd*, *vrnd*, and *vval* which are N_p, N_a, and V_a, respectively).

Therefore, we modified our model that promise message consists of three values $Promise < N_p, N_a, V_a >$, and continue to do the verification.

Algorithm 1 Paxos — Proposer p

1: **Constants:**
2: A, n, and f. {A is the set of acceptors. $n = |A|$ and $f = \lfloor (n-1)/2 \rfloor$.}
3: **Init:**
4: $crnd \leftarrow -1$ {Current round number}
5: **on** ⟨PROPOSE, val⟩
6: $crnd \leftarrow$ pickNextRound($crnd$)
7: $cval \leftarrow val$
8: $P \leftarrow \emptyset$
9: **send** ⟨PREPARE, $crnd$⟩ **to** A
10: **on** ⟨PROMISE, $rnd, vrnd, vval$⟩ with $rnd = crnd$ **from** acceptor a
11: $P \leftarrow P \cup (vrnd, vval)$
12: **on event** $|P| \geq n - f$
13: $j = \max\{vrnd : (vrnd, vval) \in P\}$
14: **if** $j \geq 0$ **then**
15: $V = \{vval : (j, vval) \in P\}$
16: $cval \leftarrow$ pick(V) {Pick proposed value $vval$ with largest $vrnd$}
17: **send** ⟨ACCEPT, $crnd, cval$⟩ **to** A

Figure 4.9: Pseudocode of Paxos proposer

Chapter 5

Conclusion and Future work

5.1 Conclusion

Proof score and CITP are some techniques doing the theorem prover. The verification will be correct when we have the correct specification. However, specification is made by human which easily cause some bugs or errors, and when doing a proof, the proof may not be correct. So, in this section we describe the problem of conducting Paxos. Then conclude our research and plan for continue proving in future work section.

5.1.1 Research Problem

The specification may not correct since made by human, and may cause some errors in the verification part. Once the specification is made, human sometimes does not check it carefully, and it is probably that some part of specification are not correct. They may do not know that their specification are wrong if thier verification part do not say any errors, and it seems that lemmas (from doing verification) are true. One way of checking the correctness of these lemmas is using another tool for automatically checking. In our case, we finished proving the agreement property by getting two lemmas after doing a proof score in OTS/CafeOBJ, and these two lemmas seems correct. However, we did double check in CITP and know that just two lemmas do not enough to verify agreement since our specification of OTS/CafeOBJ is not correct. Therefore, we start doing proof again after fixing some mistakes in the specification, and we currently get three lemmas to prove the agreement. We also have to prove these three lemmas. However, when we re-checked the specification, we found a counterexample that the agreement does not hold for our model. However, in the mostly final part of the paper of Paxos in [Lam01a] described only a sentence covered with parenthesis that "*Response messages are tagged with the corresponding proposal number to prevent confusion.*". It can be referred that promise message should include N_p when it replies to leader proposer. So our model, which consists of two parameters, should have one more to correctly prove the Paxos enjoys the agreement property.

5.1.2 Research Conclusion

In this research, we did surveying of formal verification of LastVoting algorithm (similar with Paxos algorithm) in the paper [TS11]. The authors in that paper proposed a verification approach for asynchronous consensus algorithms by using the model checking technique on single phases of the consensus algorithms. We only focused on the LastVoting algorithm (see in Appendix C) consisting of four rounds in each phase. They used the notion of round-based model and ho-model [CBS09] that only processes can receive message within that round, otherwise the processes ignore the messages come from other rounds. They also used some notations to create the formula of agreement property in LastVoting algorithm and only concern the one-phase executions of the algorithm to reduce the model checking problem to the satisfiability problem. They used Yices [DdM06] (*Satisfiability Modulo Theories* solver) to check the satisfiability of agreement formula. Since verification by model checking must traverse in to a state space which can cause the state exploded. However, their techniques can use bounded model checking which bound some values, and the value that they bounded can only checked the agreement and termination properties up to around 10 processes.

Besides doing a survey, we conducted the Paxos model in CafeOBJ in OTS style and maude, and did some proof scores and CITP to check whether or not the agreement property holds for Paxos. We use the notion of over approximation to model the Paxos. For example, we did not implement the message loss in the network. In our specification message loss can be act as the process did not see the message in the network which can be implemented by using the notion of multi-set of network.

We considered agreement property as a safety property which can be refer to an invariant property. We constructed the state predicate of agreement and prove it by proof score of OTS/CafeOBJ. However, to prove this predicate, we need three lemmas which also can be considered as other state predicate of invariants. The agreement cannot be proved unless we successfully proved other three state predicates. Moreover, we used another tool (CITP) for double checking our proving.

Now we currently have five lemmas for proving agreement property. Since doing the verification, we sometimes look back and check the correctness of specification. After re-checking the specification, we found a counterexample that the agreement property does not hold based on our specification. However, our model is slightly different from the original Paxos in [Lam01a]. Since this paper did not describe in more details of the content of messages especially the promise message in both the main part of Paxos algorithm and some proofs, it implicitly described in the mostly final part of the paper with a sentence covered with parenthesis. Therefore, we need to modified our specification and do the verification again.

5.2 Future works

We will revise the [Lam01a] paper again, then modified the specification and do the verification to prove the agreement and conjecture lemmas to help verification. Some

lemmas are not easy to get it without better understanding of the algorithm and some mathematics, while some lemmas can easily get during the proof. Therefore, finding lemmas may take several days or weeks to found it.

After successfully doing proof of agreement, we continue to prove another safety property (validity); any consensus value comes from any proposed value. We do the prove by proof score in OTS/CafeOBJ and CITP in maude for checking, and do some survey of verification of other researches of consensus algorithm.

Appendix A

Paxos Specification in OTS/CafeOBJ

```
mod! MULTISET(X :: TRIV) {
  [Elt.X < MSet]
  op empty : -> MSet {constr}
  op __ : MSet MSet -> MSet {constr comm assoc id: empty}
}

mod! VALUE {
  pr(NAT)
  [ValConst < Val]
  op null : -> ValConst {constr}
  op _=_ : Val Val -> Bool {comm}
  vars N N1 : Nat
  vars V V1 : Val
  eq (V = V) = true .
}

mod! NUM {
  pr(NAT * {sort Nat -> Num})
  op _=_ : Num Num -> Bool {comm}
  vars N N1 : Num
  eq (N = N) = true .
  eq (N = N1) = (N <= N1 and N >= N1) .
}

mod! LIST(X :: TRIV) {
  pr(NAT)
```

```
  [List]
  op nil : -> List {constr}
  op _|_ : Elt.X List -> List {constr}
  op _@_ : List List -> List
  op len : List -> Nat
  var E : Elt.X
  vars L L1 L2 : List

  -- _@_
  eq nil @ L2 = L2 .
  eq (E | L1) @ L2 = E | (L1 @ L2) .
  -- len
  eq len(nil) = 0 .
  eq len(E | L) = 1 + len(L) .
}

mod! PROPOSER {
  pr(NAT)
  pr(NUM)
  pr(VALUE)
  [Proposer]
  op PpID : -> Nat {constr}
  op vc : -> Val {constr}
  op _=_ : Proposer Proposer -> Bool {comm}
  vars N N1 : Nat
  vars P P1 : Proposer
  eq (P = P) = true .
 }

mod! ACCEPTOR {
  pr(NAT)
  [Acceptor]
  op _=_ : Acceptor Acceptor -> Bool {comm}
  vars N N1 : Nat
  vars A A1 : Acceptor
  eq (A = A) = true .
}
```

```
mod! LEARNER {
  pr(NAT)
  [Learner]
  op _=_ : Learner Learner -> Bool {comm}
  vars N N1 : Nat
  vars L L1 : Learner
  eq (L = L) = true .
}

mod! MESSAGE {
  pr(PROPOSER)
  pr(ACCEPTOR)
  pr(LEARNER)
  pr(NUM)
  pr(VALUE)
  [Msg]
  -- operators
  op prepare-m : Proposer Num -> Msg {constr}
  op promise-m : Acceptor Proposer Num Val -> Msg {constr}
  op accept-m : Proposer Num Val -> Msg {constr}
  op learn-m : Acceptor Num Val -> Msg {constr}
  op _=_ : Msg Msg -> Bool {comm}
  -- variables
  vars N N1 : Num
  vars V V1 : Val
  vars Pp Pp1 : Proposer
  vars Ac Ac1 : Acceptor
  vars Ln Ln1 : Learner
  -- equations
  eq (prepare-m(Pp,N) = prepare-m(Pp1,N1)) = ((Pp = Pp1) and (N =
     N1)) .
  eq (promise-m(Ac,Pp,N,V) = promise-m(Ac1,Pp1,N1,V1))
     = ((Ac = Ac1) and (Pp = Pp1) and (N = N1) and (V = V1)) .
  eq (accept-m(Pp,N,V) = accept-m(Pp1,N1,V1))
     = ((Pp = Pp1) and (N = N1) and (V = V1)) .
  eq (learn-m(Ac,N,V) = learn-m(Ac1,N1,V1))
```

```
      = ((Ac = Ac1) and (N = N1) and (V = V1)) .

  eq (prepare-m(Pp,N) = promise-m(Ac,Pp1,N1,V)) = false .
  eq (prepare-m(Pp,N) = accept-m(Pp1,N1,V)) = false .
  eq (prepare-m(Pp,N) = learn-m(Ac,N1,V)) = false .
  eq (promise-m(Ac,Pp,N,V) = accept-m(Pp1,N1,V1)) = false .
  eq (promise-m(Ac,Pp,N,V) = learn-m(Ac1,N1,V1)) = false .
  eq (accept-m(Pp,N,V) = learn-m(Ac,N1,V1)) = false .
}

mod! TRI-ARY {
  pr(ACCEPTOR)
  pr(NUM)
  pr(VALUE)
  [Tri]
  op <_,_,_> : Acceptor Num Val -> Tri {constr}
}

view TRIV2TRI-ARY from TRIV to TRI-ARY {
  sort Elt -> Tri }

mod! TRILIST {
  pr(LIST(X <= TRIV2TRI-ARY) * {sort List -> TriList, op nil ->
     tnil})

  op updateTri : Tri TriList -> TriList
  op findV : TriList -> Val

  vars Ac Ac1 : Acceptor
  vars N N1 : Num
  vars V V1 V2 : Val
  vars TL TL1 : TriList
  -- updateTri
  eq updateTri(< Ac , N , V >,tnil) = < Ac , N , V > | tnil .
  ceq updateTri(< Ac , N , V >,< Ac , N1 , V1 > | TL)
      = < Ac , N , V > | TL if N >= N1 .
  ceq updateTri(< Ac , N , V >,< Ac , N1 , V1 > | TL)
```

```
      = < Ac , N1 , V1 > | TL if N < N1 .
  ceq updateTri(< Ac , N , V >,< Ac1 , N1 , V1 > | TL)
      = < Ac1 , N1 , V1 > | updateTri(< Ac , N , V >,TL) if not(Ac
          = Ac1) .

  -- findV
  eq findV(tnil) = null .
  eq findV(< Ac , N , V > | tnil) = V .
  ceq findV(< Ac , N , V > | < Ac1 , N1 , V1 > | TL)
      = findV(< Ac , N , V > | TL) if N >= N1 .
  ceq findV(< Ac , N , V > | < Ac1 , N1 , V1 > | TL)
      = findV(< Ac1 , N1 , V1 > | TL) if N < N1 .

}

mod! PAIR {
  pr(NAT)
  pr(VALUE)
  [Pair]
  op <_,_> : Val Nat -> Pair {constr}
}

view TRIV2PAIR from TRIV to PAIR {
  sort Elt -> Pair }

mod! PAIRLIST {
  -- pr(LIST(X <= TRIV2PAIR) * {sort List -> PairList, op nil ->
     pnil})
  pr(LIST(X <= TRIV2PAIR) * {sort List -> PairList, op nil -> pnil
     })
  pr(TRILIST)

  op decideV : TriList -> Val
  op ctoPL : TriList -> PairList
  op incV : Val PairList -> PairList
  op majV : PairList -> Val
  op majN : PairList -> Nat
```

```
  vars Ac Ac1 : Acceptor
  vars V V1 : Val
  vars Nu : Num
  vars N N1 : Nat
  var PL : PairList
  vars TL TL1 : TriList

   eq decideV(TL) = majV(ctoPL(TL)) .

   eq ctoPL(tnil) = pnil .
   eq ctoPL(< Ac , Nu , V > | TL) = incV(V,ctoPL(TL)) .

   eq incV(V,pnil) = < V , 1 > | pnil .
   eq incV(V,< V , N > | PL) = < V , N + 1 > | PL .
   ceq incV(V,< V1 , N > | PL) = < V1 , N > | incV(V,PL) if not(V
      = V1) .

   eq majV(pnil) = null .
   eq majV(< V , N > | pnil) = V .
   ceq majV(< V , N > | < V1 , N1 > | PL) = majV(< V , N > | PL)
      if (N > N1) .
   ceq majV(< V , N > | < V1 , N1 > | PL) = majV(< V1 , N1 > | PL)
       if (N <= N1) .

   eq majN(pnil) = 0 .
   eq majN(< V , N > | pnil) = N .
   ceq majN(< V , N > | < V1 , N1 > | PL) = majN(< V , N > | PL)
      if (N > N1) .
   ceq majN(< V , N > | < V1 , N1 > | PL) = majN(< V1 , N1 > | PL)
       if (N <= N1) .
}

view TRIV2MESSAGE from TRIV to MESSAGE {
  sort Elt -> Msg }

mod! NETWORK {
```

```
  pr(MULTISET(X <= TRIV2MESSAGE)
     * {sort MSet -> Network,
          op empty -> noMsg} )
  -- operator
  op member : Msg Network -> Bool
  -- variables
  var NW : Network
  vars M M1 : Msg
  -- equations
  eq member(M,noMsg) = false .
  eq member(M,M NW) = true .
  ceq member(M,M1 NW) = member(M,NW) if not(M = M1) .
}

mod! LABEL {
  [LabelConst < Label]
  ops p0 p1 p2 a0 a1 a2 l0 l1 : -> LabelConst
  pred (_=_) : Label Label {comm} .
  var L : Label
  eq (L = L) = true .
  vars Lc1 Lc2 : LabelConst
  eq (Lc1 = Lc2) = (Lc1 == Lc2) .
}

mod* PAXOS {
  pr(NETWORK)
  pr(LABEL)
  pr(TRILIST)
  pr(PAIRLIST)
  [Sys]
  -- any initial state
  op init : -> Sys
  -- constant value
  op nAcceptor : -> Nat
  op nProcess : -> Num
  -- observations
  op p-l : Sys Proposer -> Label
```

```
op a-l : Sys Acceptor -> Label
op l-l : Sys Learner -> Label
op nw : Sys -> Network
op unique : Sys Proposer -> Num
op vClient : Sys Proposer -> Val
op list-Ld : Sys Proposer -> TriList
op n-p : Sys Acceptor -> Num
op n-a : Sys Acceptor -> Num
op v-a : Sys Acceptor -> Val
op list-Ln : Sys Learner -> TriList
op v-d : Sys Learner -> Val

-- actions
op bc-prepare : Sys Proposer -> Sys {constr}
op r-promise : Sys Proposer Acceptor Num Val -> Sys {constr}
op bc-accept : Sys Proposer -> Sys {constr}
op timeout : Sys Proposer -> Sys {constr}
op r&s-promise : Sys Acceptor Proposer Num -> Sys {constr}
op r&s-learn : Sys Proposer Acceptor Num Val -> Sys {constr}
op r-learn : Sys Acceptor Learner Num Val -> Sys {constr}
op decide : Sys Learner -> Sys {constr}

-- initial
eq p-l(init,Pp:Proposer) = p0 .
eq a-l(init,Ac:Acceptor) = a0 .
eq l-l(init,Ln:Learner) = l0 .
eq nw(init) = noMsg .
eq unique(init,Pp:Proposer) = PpID .
eq vClient(init,Pp:Proposer) = vc .
eq list-Ld(init,Pp:Proposer) = tnil .
eq n-p(init,Ac:Acceptor) = 0 .
eq n-a(init,Ac:Acceptor) = 0 .
eq v-a(init,Ac:Acceptor) = null .
eq list-Ln(init,Ln:Learner) = tnil .
eq v-d(init,Ln:Learner) = null .

-- variables
```

```
  var S : Sys
  vars Pp Pp1 : Proposer
  vars Ac Ac1 : Acceptor
  vars Ln Ln1 : Learner
  var N : Num
  var V : Val

-- Proposer
  -- bc-prepare
  op c-bc-prepare : Sys Proposer -> Bool
  eq c-bc-prepare(S,Pp) = (p-l(S,Pp) = p0) .

  ceq p-l(bc-prepare(S,Pp1),Pp) = (if Pp1 = Pp then p1 else p-l(S,
     Pp) fi)
      if c-bc-prepare(S,Pp1) .
  eq a-l(bc-prepare(S,Pp),Ac) = a-l(S,Ac) .
  eq l-l(bc-prepare(S,Pp),Ln) = l-l(S,Ln) .
  ceq nw(bc-prepare(S,Pp)) = prepare-m(Pp,unique(S,Pp)) nw(S)
      if c-bc-prepare(S,Pp) .
  eq unique(bc-prepare(S,Pp1),Pp) = unique(S,Pp) .
  eq vClient(bc-prepare(S,Pp1),Pp) = vClient(S,Pp) .
  eq list-Ld(bc-prepare(S,Pp1),Pp) = list-Ld(S,Pp) .
  eq n-p(bc-prepare(S,Pp),Ac) = n-p(S,Ac) .
  eq n-a(bc-prepare(S,Pp),Ac) = n-a(S,Ac) .
  eq v-a(bc-prepare(S,Pp),Ac) = v-a(S,Ac) .
  eq list-Ln(bc-prepare(S,Pp),Ln) = list-Ln(S,Ln) .
  eq v-d(bc-prepare(S,Pp),Ln) = v-d(S,Ln) .
  ceq bc-prepare(S,Pp) = S if not c-bc-prepare(S,Pp) .

  -- r-promise
  op c-r-promise : Sys Proposer Acceptor Num Val -> Bool
  eq c-r-promise(S,Pp,Ac,N,V) = ((p-l(S,Pp) = p1)
     and (member(promise-m(Ac,Pp,N,V),nw(S)))) .

  ceq p-l(r-promise(S,Pp1,Ac,N,V),Pp) = (if Pp1 = Pp then p1 else
     p-l(S,Pp) fi)
      if c-r-promise(S,Pp1,Ac,N,V) .
```

```
eq a-l(r-promise(S,Pp,Ac1,N,V),Ac) =  a-l(S,Ac) .
eq l-l(r-promise(S,Pp,Ac,N,V),Ln) =  l-l(S,Ln) .
eq nw(r-promise(S,Pp,Ac,N,V)) = nw(S) .
eq unique(r-promise(S,Pp1,Ac,N,V),Pp) = unique(S,Pp) .
eq vClient(r-promise(S,Pp1,Ac,N,V),Pp) = vClient(S,Pp) .
ceq list-Ld(r-promise(S,Pp1,Ac,N,V),Pp)
    = (if Pp1 = Pp then updateTri(< Ac , N , V >,list-Ld(S,Pp))
        else list-Ld(S,Pp) fi) if c-r-promise(S,Pp1,Ac,N,V) .
eq n-p(r-promise(S,Pp,Ac1,N,V),Ac) = n-p(S,Ac) .
eq n-a(r-promise(S,Pp,Ac1,N,V),Ac) = n-a(S,Ac) .
eq v-a(r-promise(S,Pp,Ac1,N,V),Ac) = v-a(S,Ac) .
eq list-Ln(r-promise(S,Pp,Ac,N,V),Ln) = list-Ln(S,Ln) .
eq v-d(r-promise(S,Pp,Ac,N,V),Ln) = v-d(S,Ln) .
ceq r-promise(S,Pp,Ac,N,V) = S if not c-r-promise(S,Pp,Ac,N,V) .

-- bc-accept
op c-bc-accept : Sys Proposer -> Bool
eq c-bc-accept(S,Pp) = ((p-l(S,Pp) = p1)
   and ((len(list-Ld(S,Pp)) * 2) >= (nAcceptor + 1))) .

ceq p-l(bc-accept(S,Pp1),Pp) = (if Pp1 = Pp then p2 else p-l(S,
   Pp) fi)
    if c-bc-accept(S,Pp1) .
eq a-l(bc-accept(S,Pp),Ac) =  a-l(S,Ac) .
eq l-l(bc-accept(S,Pp),Ln) =  l-l(S,Ln) .
ceq nw(bc-accept(S,Pp)) = accept-m(Pp,unique(S,Pp),findV(list-Ld
   (S,Pp))) nw(S)
    if (c-bc-accept(S,Pp) and not(findV(list-Ld(S,Pp)) = null))
       .
ceq nw(bc-accept(S,Pp)) = accept-m(Pp,unique(S,Pp),vClient(S,Pp)
   ) nw(S)
    if (c-bc-accept(S,Pp) and (findV(list-Ld(S,Pp)) = null)) .
eq unique(bc-accept(S,Pp1),Pp) = unique(S,Pp) .
eq vClient(bc-accept(S,Pp1),Pp) = vClient(S,Pp) .
ceq list-Ld(bc-accept(S,Pp1),Pp) = (if Pp1 = Pp then tnil else
   list-Ld(S,Pp) fi)
    if c-bc-accept(S,Pp1) .
```

```
  eq n-p(bc-accept(S,Pp),Ac) = n-p(S,Ac) .
  eq n-a(bc-accept(S,Pp),Ac) = n-a(S,Ac) .
  eq v-a(bc-accept(S,Pp),Ac) = v-a(S,Ac) .
  eq list-Ln(bc-accept(S,Pp),Ln) = list-Ln(S,Ln) .
  eq v-d(bc-accept(S,Pp),Ln) = v-d(S,Ln) .
  ceq bc-accept(S,Pp) = S if not c-bc-accept(S,Pp) .

  -- timeout
  op c-timeout : Sys Proposer -> Bool
  eq c-timeout(S,Pp) = ((p-l(S,Pp) = p1) or (p-l(S,Pp) = p2)) .

  ceq p-l(timeout(S,Pp1),Pp) = (if Pp1 = Pp then p0 else p-l(S,Pp)
      fi)
       if c-timeout(S,Pp1) .
  eq a-l(timeout(S,Pp),Ac) = a-l(S,Ac) .
  eq l-l(timeout(S,Pp),Ln) = l-l(S,Ln) .
  eq nw(timeout(S,Pp)) = nw(S) .
  ceq unique(timeout(S,Pp1),Pp) = (if Pp1 = Pp then (unique(S,Pp)
     + nProcess)
      else unique(S,Pp) fi) if c-timeout(S,Pp1) .
  eq vClient(timeout(S,Pp1),Pp) = vClient(S,Pp) .
  ceq list-Ld(timeout(S,Pp1),Pp) = (if Pp1 = Pp then tnil else
     list-Ld(S,Pp) fi)
      if c-timeout(S,Pp1) .
  eq n-p(timeout(S,Pp),Ac) = n-p(S,Ac) .
  eq n-a(timeout(S,Pp),Ac) = n-a(S,Ac) .
  eq v-a(timeout(S,Pp),Ac) = v-a(S,Ac) .
  eq list-Ln(timeout(S,Pp),Ln) = list-Ln(S,Ln) .
  eq v-d(timeout(S,Pp),Ln) = v-d(S,Ln) .
  ceq timeout(S,Pp) = S if not c-timeout(S,Pp) .

-- Acceptor
  -- r&s-promise
  op c-r&s-promise : Sys Proposer Acceptor Num -> Bool
  eq c-r&s-promise(S,Pp,Ac,N) = (((a-l(S,Ac) = a0) or (a-l(S,Ac) =
      a1)
     or (a-l(S,Ac) = a2)) and member(prepare-m(Pp,N),nw(S))) .
```

```
eq p-l(r&s-promise(S,Ac,Pp1,N),Pp) = p-l(S,Pp) .
ceq a-l(r&s-promise(S,Ac1,Pp,N),Ac) = (if Ac1 = Ac then a1 else
   a-l(S,Ac) fi)
    if c-r&s-promise(S,Pp,Ac1,N) .
eq l-l(r&s-promise(S,Ac,Pp,N),Ln) = l-l(S,Ln) .
ceq nw(r&s-promise(S,Ac,Pp,N)) = promise-m(Ac,Pp,n-a(S,Ac),v-a(S
   ,Ac)) nw(S)
    if (c-r&s-promise(S,Pp,Ac,N) and N >= n-p(S,Ac)) .
ceq nw(r&s-promise(S,Ac,Pp,N)) = nw(S)
    if (c-r&s-promise(S,Pp,Ac,N) and N < n-p(S,Ac)) .
eq unique(r&s-promise(S,Ac,Pp1,N),Pp) = unique(S,Pp) .
eq vClient(r&s-promise(S,Ac,Pp1,N),Pp) = vClient(S,Pp) .
eq list-Ld(r&s-promise(S,Ac,Pp1,N),Pp) = list-Ld(S,Pp) .
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = (if Ac1 = Ac then N else n
   -p(S,Ac) fi)
    if (c-r&s-promise(S,Pp,Ac1,N) and N >= n-p(S,Ac1)) .
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = n-p(S,Ac)
    if (c-r&s-promise(S,Pp,Ac1,N) and N < n-p(S,Ac1)) .
eq n-a(r&s-promise(S,Ac1,Pp,N),Ac) = n-a(S,Ac) .
eq v-a(r&s-promise(S,Ac1,Pp,N),Ac) = v-a(S,Ac) .
eq list-Ln(r&s-promise(S,Ac,Pp,N),Ln) = list-Ln(S,Ln) .
eq v-d(r&s-promise(S,Ac,Pp,N),Ln) = v-d(S,Ln) .
ceq r&s-promise(S,Ac,Pp,N) = S if not c-r&s-promise(S,Pp,Ac,N) .

-- r&s-learn
op c-r&s-learn : Sys Proposer Acceptor Num Val -> Bool
eq c-r&s-learn(S,Pp,Ac,N,V) =  (((a-l(S,Ac) = a1) or (a-l(S,Ac)
   = a2))
   and member(accept-m(Pp,N,V),nw(S))) .

eq p-l(r&s-learn(S,Pp1,Ac,N,V),Pp) = p-l(S,Pp) .
ceq a-l(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then a2 else
   a-l(S,Ac) fi)
    if(c-r&s-learn(S,Pp,Ac1,N,V)) .
eq l-l(r&s-learn(S,Pp,Ac,N,V),Ln) = l-l(S,Ln) .
ceq nw(r&s-learn(S,Pp,Ac,N,V)) = learn-m(Ac,N,V) nw(S)
```

```
      if (c-r&s-learn(S,Pp,Ac,N,V) and N >= n-p(S,Ac)) .
  ceq nw(r&s-learn(S,Pp,Ac,N,V)) = nw(S)
      if (c-r&s-learn(S,Pp,Ac,N,V) and N < n-p(S,Ac)) .
  eq unique(r&s-learn(S,Pp1,Ac,N,V),Pp) = unique(S,Pp) .
  eq vClient(r&s-learn(S,Pp1,Ac,N,V),Pp) = vClient(S,Pp) .
  eq list-Ld(r&s-learn(S,Pp1,Ac,N,V),Pp) = list-Ld(S,Pp) .
  ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then N else n
     -p(S,Ac) fi)
      if (c-r&s-learn(S,Pp,Ac1,N,V) and N >= n-p(S,Ac1)) .
  ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-p(S,Ac)
      if (c-r&s-learn(S,Pp,Ac1,N,V) and N < n-p(S,Ac1)) .
  ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then N else n
     -a(S,Ac) fi)
      if (c-r&s-learn(S,Pp,Ac1,N,V) and N >= n-p(S,Ac1)) .
  ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-a(S,Ac)
      if (c-r&s-learn(S,Pp,Ac1,N,V) and N < n-p(S,Ac1)) .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then V else v
     -a(S,Ac) fi)
      if (c-r&s-learn(S,Pp,Ac1,N,V) and N >= n-p(S,Ac1) and not(V
         = v-a(S,Ac))) .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = (if Ac1 = Ac then V else v
     -a(S,Ac) fi)
      if (c-r&s-learn(S,Pp,Ac1,N,V) and N >= n-p(S,Ac1) and V = v-
         a(S,Ac)) .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = v-a(S,Ac)
      if (c-r&s-learn(S,Pp,Ac1,N,V) and N < n-p(S,Ac1)) .
  eq list-Ln(r&s-learn(S,Pp,Ac,N,V),Ln) = list-Ln(S,Ln) .
  eq v-d(r&s-learn(S,Pp,Ac,N,V),Ln) = v-d(S,Ln) .
  ceq r&s-learn(S,Pp,Ac,N,V) = S if not c-r&s-learn(S,Pp,Ac,N,V) .

-- Learner
  -- c-r-learn
  op c-r-learn : Sys Acceptor Learner Num Val -> Bool
  eq c-r-learn(S,Ac,Ln,N,V) = ((l-l(S,Ln) = l0)
     and member(learn-m(Ac,N,V),nw(S))) .

  eq p-l(r-learn(S,Ac,Ln,N,V),Pp) = p-l(S,Pp) .
```

```
eq a-l(r-learn(S,Ac1,Ln,N,V),Ac) = a-l(S,Ac) .
ceq l-l(r-learn(S,Ac,Ln1,N,V),Ln) = (if Ln1 = Ln then l0 else l-
   l(S,Ln) fi)
    if c-r-learn(S,Ac,Ln1,N,V) .
eq nw(r-learn(S,Ac,Ln,N,V)) = nw(S) .
eq unique(r-learn(S,Ac,Ln,N,V),Pp) = unique(S,Pp) .
eq vClient(r-learn(S,Ac,Ln,N,V),Pp) = vClient(S,Pp) .
eq list-Ld(r-learn(S,Ac,Ln,N,V),Pp) = list-Ld(S,Pp) .
eq n-p(r-learn(S,Ac1,Ln,N,V),Ac) = n-p(S,Ac) .
eq n-a(r-learn(S,Ac1,Ln,N,V),Ac) = n-a(S,Ac) .
eq v-a(r-learn(S,Ac1,Ln,N,V),Ac) = v-a(S,Ac) .
ceq list-Ln(r-learn(S,Ac,Ln1,N,V),Ln) = (if (Ln1 = Ln) then
    updateTri(< Ac , N , V >,list-Ln(S,Ln1)) else list-Ln(S,Ln)
       fi)
    if c-r-learn(S,Ac,Ln1,N,V) and not(V = null) .
eq v-d(r-learn(S,Ac,Ln1,N,V),Ln) = v-d(S,Ln) .
ceq r-learn(S,Ac,Ln,N,V) = S if not c-r-learn(S,Ac,Ln,N,V) .

-- c-decide
op c-decide : Sys Learner -> Bool
eq c-decide(S,Ln) = ((l-l(S,Ln) = l0)
   and ((majN(ctoPL(list-Ln(S,Ln))) * 2) >= (nAcceptor + 1))
   and not(decideV(list-Ln(S,Ln)) = null)) .

eq p-l(decide(S,Ln),Pp) = p-l(S,Pp) .
eq a-l(decide(S,Ln),Ac) = a-l(S,Ac) .
ceq l-l(decide(S,Ln1),Ln) = (if Ln1 = Ln then l1 else l-l(S,Ln)
   fi)
   if c-decide(S,Ln1) .
eq nw(decide(S,Ln)) = nw(S) .
eq unique(decide(S,Ln),Pp) = unique(S,Pp) .
eq vClient(decide(S,Ln),Pp) = vClient(S,Pp) .
eq list-Ld(decide(S,Ln),Pp) = list-Ld(S,Pp) .
eq n-p(decide(S,Ln),Ac) = n-p(S,Ac) .
eq n-a(decide(S,Ln),Ac) = n-a(S,Ac) .
eq v-a(decide(S,Ln),Ac) = v-a(S,Ac) .
eq list-Ln(decide(S,Ln1),Ln) = list-Ln(S,Ln) .
```

```
  ceq v-d(decide(S,Ln1),Ln) = (if Ln1 = Ln
      then decideV(list-Ln(S,Ln1)) else v-d(S,Ln) fi)
      if c-decide(S,Ln1) .
  ceq decide(S,Ln) = S if not c-decide(S,Ln) .
}

mod INV1 {
   pr(PAXOS)
-- declare a predicate to verify to be an invariant
   pred inv1 : Sys Learner Learner
-- CafeOBJ variables
   var S : Sys .
   vars L L1 : Learner .
-- define inv1
   eq inv1(S,L,L1) = ((not(v-d(S,L) = null) and not(v-d(S,L1) =
      null))
      implies v-d(S,L) = v-d(S,L1)) .
}
```

Appendix B

Paxos Verication by Proof Score

```
--> Induction on S
--> base case
open INV1
  -- arbitrary objects
  ops l l1 : -> Learner .
  red inv1(init,l,l1) .
close

--> induction case
--> bc-prepare
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
   op pp : -> Proposer .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(bc-prepare(s,pp),l,l1) .
close
--> r-promise
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
   op pp : -> Proposer .
   op ac : -> Acceptor .
```

```
   op n : -> Num .
   op v : -> Val .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(r-promise(s,pp,ac,n,v),l,l1) .
close
--> bc-accept
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
   op pp : -> Proposer .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(bc-accept(s,pp),l,l1) .
close
--> timeout
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
   op pp : -> Proposer .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(timeout(s,pp),l,l1) .
close

--> r&s-promise
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
   op pp : -> Proposer .
   op ac : -> Acceptor .
   op n : -> Num .
-- |=
-- check if the predicate is true.
```

```
   red inv1(s,l,l1) implies inv1(r&s-promise(s,ac,pp,n),l,l1) .
close
--> r&s-learn
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
   op pp : -> Proposer .
   op ac : -> Acceptor .
   op n : -> Num .
   op v : -> Val .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(r&s-learn(s,pp,ac,n,v),l,l1) .
close
--> r-learn
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
   op ac : -> Acceptor .
   op n : -> Num .
   op v : -> Val .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(r-learn(s,ac,l2,n,v),l,l1) .
close
--> decide, c-decide, l=l2, l1=l2
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
-- assumptions
   eq l-l(s,l2) = l0 .
   eq ((majN(ctoPL(list-Ln(s,l2))) * 2) >= (nAcceptor + 1)) = true
      .
   eq (majV(ctoPL(list-Ln(s,l2))) = null) = false .
```

```
   eq l = l2 .
   eq l1 = l2 .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(decide(s,l2),l,l1) .
close
--> decide, c-decide, l=l2, ~l1=l2, v-d(s,l1) = null
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
-- assumptions
   eq l-l(s,l2) = l0 .
   eq ((majN(ctoPL(list-Ln(s,l2))) * 2) >= (nAcceptor + 1)) = true
       .
   eq (majV(ctoPL(list-Ln(s,l2))) = null) = false .
   eq l = l2 .
   eq (l1 = l2) = false .
   eq v-d(s,l1) = null .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(decide(s,l2),l,l1) .
close
--> decide, c-decide, l=l2, ~l1=l2, ~v-d(s,l1)=null,lemmas
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
-- variables
   var S : Sys
   vars L L1 : Learner
-- assumptions
   eq l-l(s,l2) = l0 .
   eq ((majN(ctoPL(list-Ln(s,l2))) * 2) >= (nAcceptor + 1)) = true
       .
   eq (majV(ctoPL(list-Ln(s,l2))) = null) = false .
   eq l = l2 .
```

```
   eq (l1 = l2) = false .
   eq (v-d(s,l1) = null) = false .
   ceq majV(ctoPL(list-Ln(s,l2))) = majV(ctoPL(list-Ln(s,l1))) if
      (((majN(ctoPL(list-Ln(s,l2))) * 2) >= (nAcceptor + 1)) and
      ((majN(ctoPL(list-Ln(s,l1))) * 2) >= (nAcceptor + 1))) .
   ceq ((majN(ctoPL(list-Ln(s,l1))) * 2) >= (nAcceptor + 1)) =
      true if not(v-d(s,l1) = null) .
   ceq majV(ctoPL(list-Ln(s,l1))) = v-d(s,l1) if not(v-d(s,l1) =
      null) .
-- |=
-- check if the predicate is true.
   red inv1(decide(s,l2),l,l1) .
close

--> decide, c-decide, ~l=l2, l1=l2, v-d(s,l)= null
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
-- assumptions
   eq l-l(s,l2) = l0 .
   eq ((majN(ctoPL(list-Ln(s,l2))) * 2) >= (nAcceptor + 1)) = true
       .
   eq (majV(ctoPL(list-Ln(s,l2))) = null) = false .
   eq (l = l2) = false .
   eq l1 = l2 .
   eq v-d(s,l) = null .
-- |=
-- check if the predicate is true.
   red inv1(decide(s,l2),l,l1) .
close

--> decide, c-decide, ~l=l2, l1=l2, ~v-d(s,l)=null, lemmas
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l ln1 l2 : -> Learner .
```

```
-- assumptions
   eq l-l(s,l2) = l0 .
   eq ((majN(ctoPL(list-Ln(s,l2))) * 2) >= (nAcceptor + 1)) = true
       .
   eq (majV(ctoPL(list-Ln(s,l2))) = null) = false .
   eq (l = l2) = false .
   eq ln1 = l2 .
   eq (v-d(s,l) = null) = false .
   ceq majV(ctoPL(list-Ln(S,l2))) = majV(ctoPL(list-Ln(S,l))) if
      (((majN(ctoPL(list-Ln(S,l))) * 2) >= (nAcceptor + 1)) and ((
      majN(ctoPL(list-Ln(S,l2))) * 2) >= (nAcceptor + 1))) .
   ceq ((majN(ctoPL(list-Ln(s,l))) * 2) >= (nAcceptor + 1)) = true
       if not(v-d(s,l) = null) .
   ceq majV(ctoPL(list-Ln(s,l))) = v-d(s,l) if not(v-d(s,l) = null
      ) .
-- |=
-- check if the predicate is true.
   red inv1(decide(s,l2),l,ln1) .
close
--> decide, c-decide, ~l=l2, ~l1=l2
open INV1
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
-- assumptions
   eq l-l(s,l2) = l0 .
   eq ((majN(ctoPL(list-Ln(s,l2))) * 2) >= (nAcceptor + 1)) = true
       .
   eq (majV(ctoPL(list-Ln(s,l2))) = null) = false .
   eq (l = l2) = false .
   eq (l1 = l2) = false .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(decide(s,l2),l,l1) .
close
--> decide, ~c-decide
open INV1
```

```
-- arbitrary objects
   op s : -> Sys .
   ops l l1 l2 : -> Learner .
-- assumptions
   eq c-decide(s,l2) = false .
-- |=
-- check if the predicate is true.
   red inv1(s,l,l1) implies inv1(decide(s,l2),l,l1) .
close
```

Appendix C

LastVoting Algorithm

```
1: Initialization:
2:     x_p ∈ Val, initially v_p
                                   {Estimate for the decision. v_p is the proposed value of p.}
3:     vote_p ∈ Val ∪ {?}, initially ?
                                   {Val is the set of values that may be proposed.}
4:     commit_p a Boolean, initially false
5:     ready_p a Boolean, initially false
6:     ts_p ∈ ℕ, initially 0
                                   {Timestamp. ℕ is the set of non-negative integers.}

7: Round r = 4φ − 3:
8:     S_p^r :
9:       send ⟨x_p , ts_p⟩ to Coord(p, φ)

10:    T_p^r :
11:      if p = Coord(p, φ) and number of ⟨ν , θ⟩ received > n/2 then
12:        let θ̄ be the largest θ from ⟨ν , θ⟩ received
13:        vote_p := one ν such that ⟨ν , θ̄⟩ is received
14:        commit_p := true

15: Round r = 4φ − 2:
16:    S_p^r :
17:      if p = Coord(p, φ) and commit_p then
18:        send ⟨vote_p⟩ to all processes

19:    T_p^r :
20:      if received ⟨v⟩ from Coord(p, φ) then
21:        x_p := v ; ts_p := φ

22: Round r = 4φ − 1:
23:    S_p^r :
24:      if ts_p = φ then
25:        send ⟨ack⟩ to Coord(p, φ)

26:    T_p^r :
27:      if p = Coord(p, φ) and number of ⟨ack⟩ received > n/2 then
28:        ready_p := true

29: Round r = 4φ:
30:    S_p^r :
31:      if p = Coord(p, φ) and ready_p then
32:        send ⟨vote_p⟩ to all processes

33:    T_p^r :
34:      if received ⟨v⟩ from Coord(p, φ) then
35:        DECIDE(v)
36:      if p = Coord(p, φ) then
37:        ready_p := false
38:        commit_p := false
```

Figure C.1: The LastVoting algorithm

Appendix D

Paxos Specification in Maude

```
(fmod MULTISET{X :: TRIV} is
  sort MSet .
  subsorts X$Elt < MSet .
--- constructors
  op empty : -> MSet [ctor] .
  op __ : MSet MSet -> MSet [ctor assoc comm id: empty] .
endfm)

(fth VALUE is
  inc BOOL .
  sort Val .
--- constructors
  op null : -> Val [ctor] .
  op _~_ : Val Val -> Bool [comm] .
  vars V V1 : Val .
  eq V ~ V = true .
  ceq V = V1 if V ~ V1 [nonexec] .
endfth)

(fth NUM is
  inc NAT .
  inc BOOL .
  sort Num .
  subsorts Nat < Num .
--- constructors
  op _~_ : Num Num -> Bool [comm] .
```

```
  vars N N1 : Num .
  eq N ~ N = true .
  ceq N = N1 if N ~ N1 [nonexec] .
endfth)

(fmod LIST{X :: TRIV} is
  inc NAT .
  sort List .
  op nil : -> List [ctor] .
  op _|_ : X$Elt List -> List [ctor] .
  op _@_ : List List -> List .
  op len : List -> Nat .
  var E : X$Elt .
  vars L L1 L2 : List .
--- _@_
  eq nil @ L2 = L2 .
  eq (E | L1) @ L2 = E | (L1 @ L2) .
--- len
  eq len(nil) = 0 .
  eq len(E | L) = 1 + len(L) .
endfm)

(fth PROPOSER is
  inc NUM .
  inc VALUE .
  sort Proposer .
--- constructors
  op PpID : -> Num .
  op vc : -> Val .
  op _~_ : Proposer Proposer -> Bool [comm] .
  vars P P1 : Proposer .
  eq P ~ P = true .
  ceq P = P1 if P ~ P1 [nonexec] .
endfth)

(fth ACCEPTOR is
  inc BOOL .
```

```
  sort Acceptor .
  op _~_ : Acceptor Acceptor -> Bool [comm] .
  vars A A1 : Acceptor .
  eq A ~ A = true .
  ceq A = A1 if A ~ A1 [nonexec] .
endfth)

(fth LEARNER is
  inc BOOL .
  sort Learner .
  op _~_ : Learner Learner -> Bool [comm] .
  vars L L1 : Learner .
  eq L ~ L = true .
  ceq L = L1 if L ~ L1 [nonexec] .
endfth)

(fmod MESSAGE is
  inc PROPOSER .
  inc ACCEPTOR .
  inc LEARNER .
  inc NUM .
  inc VALUE .
  sort Msg .

  op prepare-m : Proposer Num -> Msg [ctor] .
  op promise-m : Acceptor Proposer Num Val -> Msg [ctor] .
  op accept-m : Proposer Num Val -> Msg [ctor] .
  op learn-m : Acceptor Num Val -> Msg [ctor] .
  op _~_ : Msg Msg -> Bool [comm] .

  vars N N1 : Num .
  vars V V1 : Val .
  vars Pp Pp1 : Proposer .
  vars Ac Ac1 : Acceptor .
  vars Ln Ln1 : Learner .

  eq (prepare-m(Pp,N) ~ prepare-m(Pp1,N1)) = ((Pp ~ Pp1) and (N ~
```

```
    N1)) .
  eq (promise-m(Ac,Pp,N,V) ~ promise-m(Ac1,Pp1,N1,V1)) = ((Ac ~
    Ac1) and (Pp ~ Pp1) and (N ~ N1) and (V ~ V1)) .
  eq (accept-m(Pp,N,V) ~ accept-m(Pp1,N1,V1)) = ((Pp ~ Pp1) and (N
     ~ N1) and (V ~ V1)) .
  eq (learn-m(Ac,N,V) ~ learn-m(Ac1,N1,V1)) = ((Ac ~ Ac1) and (N ~
     N1) and (V ~ V1)) .

  eq (prepare-m(Pp,N) ~ promise-m(Ac,Pp1,N1,V)) = false .
  eq (prepare-m(Pp,N) ~ accept-m(Pp1,N1,V)) = false .
  eq (prepare-m(Pp,N) ~ learn-m(Ac,N1,V)) = false .
  eq (promise-m(Ac,Pp,N,V) ~ accept-m(Pp1,N1,V1)) = false .
  eq (promise-m(Ac,Pp,N,V) ~ learn-m(Ac1,N1,V1)) = false .
  eq (accept-m(Pp,N,V) ~ learn-m(Ac,N1,V1)) = false .
endfm)

(fmod TRI-ARY is
  inc ACCEPTOR .
  inc NUM .
  inc VALUE .
  sort Tri .
  op <_;_;_> : Acceptor Num Val -> Tri [ctor] .
endfm)

(view TRIV2TRI-ARY from TRIV to TRI-ARY is
 sort Elt to Tri .
endv)

(fth TRILIST is
  inc LIST{TRIV2TRI-ARY} * (sort List to TriList, op nil to tnil)
    .
  op updateTri : Tri TriList -> TriList .
  op findV : TriList -> Val .

  vars Ac Ac1 : Acceptor .
  vars N N1 : Num .
  vars V V1 V2 : Val .
```

```
  vars TL TL1 : TriList .

--- updateTri
  eq updateTri(< Ac ; N ; V >,tnil) = < Ac ; N ; V > | tnil .
  ceq updateTri(< Ac ; N ; V >,< Ac ; N1 ; V1 > | TL) = < Ac ; N ;
      V > | TL if N >= N1 .
  ceq updateTri(< Ac ; N ; V >,< Ac ; N1 ; V1 > | TL) = < Ac ; N1
     ; V1 > | TL if N < N1 .
  ceq updateTri(< Ac ; N ; V >,< Ac1 ; N1 ; V1 > | TL) = < Ac1 ;
     N1 ; V1 > | updateTri(< Ac ; N ; V >,TL) if (Ac ~ Ac1) =
     false .

  --- findV
  eq findV(tnil) = null .
  eq findV(< Ac ; N ; V > | tnil) = V .
  ceq findV(< Ac ; N ; V > | < Ac1 ; N1 ; V1 > | TL) = findV(< Ac
     ; N ; V > | TL) if N >= N1 .
  ceq findV(< Ac ; N ; V > | < Ac1 ; N1 ; V1 > | TL) = findV(< Ac1
      ; N1 ; V1 > | TL) if N < N1 .
endfth)

(fmod PAIR is
  inc NAT .
  inc VALUE .
  sort Pair .
  op <_;_> : Val Nat -> Pair [ctor] .
endfm)

(view TRIV2PAIR from TRIV to PAIR is
 sort Elt to Pair .
endv)

(fth PAIRLIST is
  inc LIST{TRIV2PAIR} * (sort List to PairList, op nil to pnil) .
  inc(TRILIST) .

  op decideV : TriList -> Val .
```

```
  op ctoPL : TriList -> PairList .
  op incV : Val PairList -> PairList .
  op majV : PairList -> Val .
  op majN : PairList -> Nat .

  vars Ac Ac1 : Acceptor .
  vars V V1 : Val .
  vars Nu : Num .
  vars N N1 : Nat .
  var PL : PairList .
  vars TL TL1 : TriList .

   eq decideV(TL) = majV(ctoPL(TL)) .

   eq ctoPL(tnil) = pnil .
   eq ctoPL(< Ac ; Nu ; V > | TL) = incV(V,ctoPL(TL)) .

   eq incV(V,pnil) = < V ; 1 > | pnil .
   eq incV(V,< V ; N > | PL) = < V ; N + 1 > | PL .
   ceq incV(V,< V1 ; N > | PL) = < V1 ; N > | incV(V,PL) if (V ~
      V1) = false .

   eq majV(pnil) = null .
   eq majV(< V ; N > | pnil) = V .
   ceq majV(< V ; N > | < V1 ; N1 > | PL) = majV(< V ; N > | PL)
      if (N > N1) .
   ceq majV(< V ; N > | < V1 ; N1 > | PL) = majV(< V1 ; N1 > | PL)
       if (N <= N1) .

   eq majN(pnil) = 0 .
   eq majN(< V ; N > | pnil) = N .
   ceq majN(< V ; N > | < V1 ; N1 > | PL) = majN(< V ; N > | PL)
      if (N > N1) .
   ceq majN(< V ; N > | < V1 ; N1 > | PL) = majN(< V1 ; N1 > | PL)
       if (N <= N1) .
endfth)
```

```
(view TRIV2MESSAGE from TRIV to MESSAGE is
 sort Elt to Msg .
endv)

(fth NETWORK is
  inc MULTISET{TRIV2MESSAGE} * (sort MSet to Network, op empty to
     noMsg) .
  op member : Msg Network -> Bool .

  var NW : Network .
  vars M M1 : Msg .

  eq member(M,noMsg) = false .
  eq member(M,M NW) = true .
  ceq member(M,M1 NW) = member(M,NW) if not(M ~ M1) .
endfth)

(fmod LABEL is
 sort Label .
 ops p0 p1 p2 a0 a1 a2 l0 l1 : -> Label [ctor].
 op _~_ : Label Label -> Bool [comm].
 var  L : Label .
 eq L ~ L = true .
 eq (p0 ~ p1) = false .
 eq (p0 ~ p2) = false .
 eq (p0 ~ a0) = false .
 eq (p0 ~ a1) = false .
 eq (p0 ~ a2) = false .
 eq (p0 ~ l0) = false .
 eq (p0 ~ l1) = false .
 eq (p1 ~ p2) = false .
 eq (p1 ~ a0) = false .
 eq (p1 ~ a1) = false .
 eq (p1 ~ a2) = false .
 eq (p1 ~ l0) = false .
 eq (p1 ~ l1) = false .
 eq (p2 ~ a0) = false .
```

```
eq (p2 ~ a1) = false .
eq (p2 ~ a2) = false .
eq (p2 ~ l0) = false .
eq (p2 ~ l1) = false .
eq (a0 ~ a1) = false .
eq (a0 ~ a2) = false .
eq (a0 ~ l0) = false .
eq (a0 ~ l1) = false .
eq (a1 ~ a2) = false .
eq (a1 ~ l0) = false .
eq (a1 ~ l1) = false .
eq (a2 ~ l0) = false .
eq (a2 ~ l1) = false .
eq (l0 ~ l1) = false .

ceq true = false if p0 = p1 .
ceq true = false if p0 = p2 .
ceq true = false if p0 = a0 .
ceq true = false if p0 = a1 .
ceq true = false if p0 = a2 .
ceq true = false if p0 = l0 .
ceq true = false if p0 = l1 .
ceq true = false if p1 = p2 .
ceq true = false if p1 = a0 .
ceq true = false if p1 = a1 .
ceq true = false if p1 = a2 .
ceq true = false if p1 = l0 .
ceq true = false if p1 = l1 .
ceq true = false if p2 = a0 .
ceq true = false if p2 = a1 .
ceq true = false if p2 = a2 .
ceq true = false if p2 = l0 .
ceq true = false if p2 = l1 .
ceq true = false if a0 = a1 .
ceq true = false if a0 = a2 .
ceq true = false if a0 = l0 .
ceq true = false if a0 = l1 .
```

```
 ceq true = false if a1 = a2 .
 ceq true = false if a1 = l0 .
 ceq true = false if a1 = l1 .
 ceq true = false if a2 = l0 .
 ceq true = false if a2 = l1 .
 ceq true = false if l0 = l1 .
endfm)

(fth PAXOS is
  inc LABEL .
  inc TRILIST .
  inc PAIRLIST .
  inc NETWORK .
  sort Sys .

--- initial
  op init : -> Sys [ctor] .
--- constant values
  op nAcceptor : -> Nat .
  op nProcess : -> Num .
--- transitions
  op bc-prepare : Sys Proposer -> Sys [ctor] .
  op r-promise : Sys Proposer Acceptor Num Val -> Sys [ctor] .
  op bc-accept : Sys Proposer -> Sys [ctor] .
  op timeout : Sys Proposer -> Sys [ctor] .
  op r&s-promise : Sys Acceptor Proposer Num -> Sys [ctor] .
  op r&s-learn : Sys Proposer Acceptor Num Val -> Sys [ctor] .
  op r-learn : Sys Acceptor Num Val Learner -> Sys [ctor] .
  op decide : Sys Learner -> Sys [ctor] .
--- observers
  op p-l : Sys Proposer -> Label .
  op a-l : Sys Acceptor -> Label .
  op l-l : Sys Learner -> Label .
  op nw : Sys -> Network .
  op unique : Sys Proposer -> Num .
  op vUser : Sys Proposer -> Val .
  op list-Ld : Sys Proposer -> TriList .
```

```
  op n-p : Sys Acceptor -> Num .
  op n-a : Sys Acceptor -> Num .
  op v-a : Sys Acceptor -> Val .
  op list-Ln : Sys Learner -> TriList .
  op v-d : Sys Learner -> Val .
--- variables
  var S : Sys .
  vars Pp Pp1 : Proposer .
  vars Ac Ac1 : Acceptor .
  vars Ln Ln1 : Learner .
  var N : Num .
  var V : Val .

--- initial
  eq p-l(init,Pp) = p0 .
  eq a-l(init,Ac) = a0 .
  eq l-l(init,Ln) = l0 .
  eq nw(init) = noMsg .
  eq unique(init,Pp) = PpID .
  eq vUser(init,Pp) = vc .
  eq list-Ld(init,Pp) = tnil .
  eq n-p(init,Ac) = 0 .
  eq n-a(init,Ac) = 0 .
  eq v-a(init,Ac) = null .
  eq list-Ln(init,Ln) = tnil .
  eq v-d(init,Ln) = null .

--- Proposer
--- bc-prepare
  ceq p-l(bc-prepare(S,Pp1),Pp) = p1 if p-l(S,Pp1) = p0 /\ Pp =
     Pp1 [metadata "CA-prepare-pl1"] .
  ceq p-l(bc-prepare(S,Pp1),Pp) = p-l(S,Pp) if Pp ~ Pp1 = false [
     metadata "CA-prepare-pl2"] .
  ceq p-l(bc-prepare(S,Pp1),Pp) = p-l(S,Pp) if p-l(S,Pp1) ~ p0 =
     false [metadata "CA-prepare-pl3"] .
  eq a-l(bc-prepare(S,Pp),Ac) = a-l(S,Ac) .
  eq l-l(bc-prepare(S,Pp),Ln) = l-l(S,Ln) .
```

```
  ceq nw(bc-prepare(S,Pp)) = prepare-m(Pp,unique(S,Pp)) nw(S) if p
     -l(S,Pp) = p0 [metadata "CA-prepare-nw1"] .
  ceq nw(bc-prepare(S,Pp)) = nw(S) if p-l(S,Pp) ~ p0 = false [
     metadata "CA-prepare-nw2"] .
  eq unique(bc-prepare(S,Pp1),Pp) = unique(S,Pp) .
  eq vUser(bc-prepare(S,Pp1),Pp) = vUser(S,Pp) .
  eq list-Ld(bc-prepare(S,Pp1),Pp) = list-Ld(S,Pp) .
  eq n-p(bc-prepare(S,Pp),Ac) = n-p(S,Ac) .
  eq n-a(bc-prepare(S,Pp),Ac) = n-a(S,Ac) .
  eq v-a(bc-prepare(S,Pp),Ac) = v-a(S,Ac) .
  eq list-Ln(bc-prepare(S,Pp),Ln) = list-Ln(S,Ln) .
  eq v-d(bc-prepare(S,Pp),Ln) = v-d(S,Ln) .

--- r-promise
  ceq p-l(r-promise(S,Pp1,Ac,N,V),Pp) = p1 if p-l(S,Pp1) = p1 /\
     member(promise-m(Ac,Pp,N,V),nw(S)) /\ Pp = Pp1 [metadata "CA-
     promise-pl1"] .
  ceq p-l(r-promise(S,Pp1,Ac,N,V),Pp) = p-l(S,Pp) if Pp ~ Pp1 =
     false [metadata "CA-promise-pl2"] .
  ceq p-l(r-promise(S,Pp1,Ac,N,V),Pp) = p-l(S,Pp) if not(member(
     promise-m(Ac,Pp,N,V),nw(S))) [metadata "CA-promise-pl3"] .
  ceq p-l(r-promise(S,Pp1,Ac,N,V),Pp) = p-l(S,Pp) if p-l(S,Pp1) ~
     p1 = false [metadata "CA-promise-pl4"] .
  eq a-l(r-promise(S,Pp1,Ac,N,V),Ac) = a-l(S,Ac) .
  eq l-l(r-promise(S,Pp1,Ac,N,V),Ln) = l-l(S,Ln) .
  eq nw(r-promise(S,Pp1,Ac,N,V)) = nw(S) .
  eq unique(r-promise(S,Pp1,Ac,N,V),Pp) = unique(S,Pp) .
  eq vUser(r-promise(S,Pp1,Ac,N,V),Pp) = vUser(S,Pp) .
  ceq list-Ld(r-promise(S,Pp1,Ac,N,V),Pp) = updateTri(< Ac ; N ; V
      >,list-Ld(S,Pp1)) if p-l(S,Pp1) = p1 /\ member(promise-m(Ac,
     Pp,N,V),nw(S)) /\ Pp = Pp1 [metadata "CA-promise-list-Ld1"] .
  ceq list-Ld(r-promise(S,Pp1,Ac,N,V),Pp) = list-Ld(S,Pp) if Pp ~
     Pp1 = false [metadata "CA-promise-list-Ld2"] .
  ceq list-Ld(r-promise(S,Pp1,Ac,N,V),Pp) = list-Ld(S,Pp) if not(
     member(promise-m(Ac,Pp,N,V),nw(S))) [metadata "CA-promise-
     list-Ld3"] .
  ceq list-Ld(r-promise(S,Pp1,Ac,N,V),Pp) = list-Ld(S,Pp) if p-l(S
```

```
    ,Pp1) ~ p1 = false  [metadata "CA-promise-list-Ld4"] .
  eq n-p(r-promise(S,Pp1,Ac,N,V),Ac) = n-p(S,Ac) .
  eq n-a(r-promise(S,Pp1,Ac,N,V),Ac) = n-a(S,Ac) .
  eq v-a(r-promise(S,Pp1,Ac,N,V),Ac) = v-a(S,Ac) .
  eq list-Ln(r-promise(S,Pp1,Ac,N,V),Ln) = list-Ln(S,Ln) .
  eq v-d(r-promise(S,Pp1,Ac,N,V),Ln) = v-d(S,Ln) .

--- bc-accept
  ceq p-l(bc-accept(S,Pp1),Pp) = p2 if p-l(S,Pp1) = p1 /\ ((len(
    list-Ld(S,Pp1)) * 2) >= (nAcceptor + 1)) /\ Pp = Pp1 [
    metadata "CA-accept-pl1"] .
  ceq p-l(bc-accept(S,Pp1),Pp) = p-l(S,Pp) if Pp ~ Pp1 = false [
    metadata "CA-accept-pl2"] .
  ceq p-l(bc-accept(S,Pp1),Pp) = p-l(S,Pp) if not((len(list-Ld(S,
    Pp1)) * 2) >= (nAcceptor + 1)) [metadata "CA-accept-pl3"] .
  ceq p-l(bc-accept(S,Pp1),Pp) = p-l(S,Pp) if p-l(S,Pp1) ~ p1 =
    false  [metadata "CA-accept-pl4"] .
  eq a-l(bc-accept(S,Pp1),Ac) = a-l(S,Ac) .
  eq l-l(bc-accept(S,Pp1),Ln) = l-l(S,Ln) .
  ceq nw(bc-accept(S,Pp)) = accept-m(Pp,unique(S,Pp),findV(list-Ld
    (S,Pp))) nw(S) if p-l(S,Pp) = p1 /\ ((len(list-Ld(S,Pp)) * 2)
     >= (nAcceptor + 1)) /\ (findV(list-Ld(S,Pp)) ~ null) = false
     [metadata "CA-accept-nw1"] .
  ceq nw(bc-accept(S,Pp)) = accept-m(Pp,unique(S,Pp),vUser(S,Pp))
    nw(S) if p-l(S,Pp) = p1 /\ ((len(list-Ld(S,Pp)) * 2) >= (
    nAcceptor + 1)) /\ (findV(list-Ld(S,Pp)) = null) [metadata "
    CA-accept-nw2"] .
  ceq nw(bc-accept(S,Pp)) = nw(S) if not((len(list-Ld(S,Pp)) * 2)
    >= (nAcceptor + 1)) [metadata "CA-accept-nw3"] .
  ceq nw(bc-accept(S,Pp)) = nw(S) if p-l(S,Pp) ~ p1 = false [
    metadata "CA-accept-nw4"] .
  eq unique(bc-accept(S,Pp1),Pp) = unique(S,Pp) .
  eq vUser(bc-accept(S,Pp1),Pp) = vUser(S,Pp) .
  eq list-Ld(bc-accept(S,Pp1),Pp) = list-Ld(S,Pp) .
  eq n-p(bc-accept(S,Pp),Ac) = n-p(S,Ac) .
  eq n-a(bc-accept(S,Pp),Ac) = n-a(S,Ac) .
  eq v-a(bc-accept(S,Pp),Ac) = v-a(S,Ac) .
```

```
  eq list-Ln(bc-accept(S,Pp),Ln) = list-Ln(S,Ln) .
  eq v-d(bc-accept(S,Pp),Ln) = v-d(S,Ln) .

--- timeout
  ceq p-l(timeout(S,Pp1),Pp) = p0 if (p-l(S,Pp1) = p1) /\ Pp = Pp1
      [metadata "CA-timeout-pl1"] .
  ceq p-l(timeout(S,Pp1),Pp) = p0 if (p-l(S,Pp1) = p2) /\ Pp = Pp1
      [metadata "CA-timeout-pl2"] .
  ceq p-l(timeout(S,Pp1),Pp) = p-l(S,Pp) if (Pp ~ Pp1) = false [
     metadata "CA-timeout-pl3"] .
  ceq p-l(timeout(S,Pp1),Pp) = p-l(S,Pp) if ((p-l(S,Pp1) ~ p1) =
     false) /\ ((p-l(S,Pp1) ~ p2) = false)  [metadata "CA-timeout-
     pl4"] .
  eq a-l(timeout(S,Pp),Ac) = a-l(S,Ac) .
  eq l-l(timeout(S,Pp),Ln) = l-l(S,Ln) .
  eq nw(timeout(S,Pp)) = nw(S) .
  ceq unique(timeout(S,Pp1),Pp) = (unique(S,Pp) + nProcess) if (p-
     l(S,Pp1) = p1) /\ Pp = Pp1 [metadata "CA-timeout-unique1"] .
  ceq unique(timeout(S,Pp1),Pp) = (unique(S,Pp) + nProcess) if (p-
     l(S,Pp1) = p2) /\ Pp = Pp1 [metadata "CA-timeout-unique2"] .
  ceq unique(timeout(S,Pp1),Pp) = unique(S,Pp) if (Pp ~ Pp1) =
     false [metadata "CA-timeout-unique3"] .
  ceq unique(timeout(S,Pp1),Pp) = unique(S,Pp) if ((p-l(S,Pp1) ~
     p1) = false) /\ ((p-l(S,Pp1) ~ p2) = false) [metadata "CA-
     timeout-unique4"] .
  eq vUser(timeout(S,Pp1),Pp) = vUser(S,Pp) .
  ceq list-Ld(timeout(S,Pp1),Pp) = tnil if (p-l(S,Pp1) = p1) /\ Pp
      = Pp1 [metadata "CA-timeout-list-Ld1"] .
  ceq list-Ld(timeout(S,Pp1),Pp) = tnil if (p-l(S,Pp1) = p2) /\ Pp
      = Pp1 [metadata "CA-timeout-list-Ld2"] .
  ceq list-Ld(timeout(S,Pp1),Pp) = list-Ld(S,Pp) if (Pp ~ Pp1) =
     false [metadata "CA-timeout-list-Ld3"] .
  ceq list-Ld(timeout(S,Pp1),Pp) = list-Ld(S,Pp) if ((p-l(S,Pp1) ~
      p1) = false) /\ ((p-l(S,Pp1) ~ p2) = false) [metadata "CA-
     timeout-unique4"] .
  eq n-p(timeout(S,Pp),Ac) = n-p(S,Ac) .
  eq n-a(timeout(S,Pp),Ac) = n-a(S,Ac) .
```

```
  eq v-a(timeout(S,Pp),Ac) = v-a(S,Ac) .
  eq list-Ln(timeout(S,Pp),Ln) = list-Ln(S,Ln) .
  eq v-d(timeout(S,Pp),Ln) = v-d(S,Ln) .

--- Acceptor
  --- r&s-promise
  eq p-l(r&s-promise(S,Ac,Pp1,N),Pp) = p-l(S,Pp) .
  ceq a-l(r&s-promise(S,Ac1,Pp,N),Ac) = a1 if a-l(S,Ac1) = a0 /\
     member(prepare-m(Pp,N),nw(S)) /\ Ac1 = Ac [metadata "CA-
     rspromise-al1"] .
  ceq a-l(r&s-promise(S,Ac1,Pp,N),Ac) = a1 if a-l(S,Ac1) = a1 /\
     member(prepare-m(Pp,N),nw(S)) /\ Ac1 = Ac [metadata "CA-
     rspromise-al2"] .
  ceq a-l(r&s-promise(S,Ac1,Pp,N),Ac) = a1 if a-l(S,Ac1) = a2 /\
     member(prepare-m(Pp,N),nw(S)) /\ Ac1 = Ac [metadata "CA-
     rspromise-al3"] .
  ceq a-l(r&s-promise(S,Ac1,Pp,N),Ac) = a-l(S,Ac) if (Ac1 ~ Ac) =
     false [metadata "CA-rspromise-al4"] .
  ceq a-l(r&s-promise(S,Ac1,Pp,N),Ac) = a-l(S,Ac) if not(member(
     prepare-m(Pp,N),nw(S))) [metadata "CA-rspromise-al5"] .
  ceq a-l(r&s-promise(S,Ac1,Pp,N),Ac) = a-l(S,Ac) if ((a-l(S,Ac1)
     ~ a0) = false /\ (a-l(S,Ac1) ~ a1) = false /\ (a-l(S,Ac1) ~
     a2) = false) [metadata "CA-rspromise-al6"] .
  eq l-l(r&s-promise(S,Ac,Pp,N),Ln) = l-l(S,Ln) .
  ceq nw(r&s-promise(S,Ac,Pp,N)) = promise-m(Ac,Pp,n-a(S,Ac),v-a(S
     ,Ac)) nw(S) if a-l(S,Ac) = a0 /\ member(prepare-m(Pp,N),nw(S)
     ) /\ N >= n-p(S,Ac) [metadata "CA-rspromise-nw1"] .
  ceq nw(r&s-promise(S,Ac,Pp,N)) = promise-m(Ac,Pp,n-a(S,Ac),v-a(S
     ,Ac)) nw(S) if a-l(S,Ac) = a1 /\ member(prepare-m(Pp,N),nw(S)
     ) /\ N >= n-p(S,Ac) [metadata "CA-rspromise-nw2"] .
  ceq nw(r&s-promise(S,Ac,Pp,N)) = promise-m(Ac,Pp,n-a(S,Ac),v-a(S
     ,Ac)) nw(S) if a-l(S,Ac) = a2 /\ member(prepare-m(Pp,N),nw(S)
     ) /\ N >= n-p(S,Ac) [metadata "CA-rspromise-nw3"] .
  ceq nw(r&s-promise(S,Ac,Pp,N)) = nw(S) if a-l(S,Ac) = a0 /\
     member(prepare-m(Pp,N),nw(S)) /\ N < n-p(S,Ac) [metadata "CA-
     rspromise-nw4"] .
  ceq nw(r&s-promise(S,Ac,Pp,N)) = nw(S) if a-l(S,Ac) = a1 /\
```

```
   member(prepare-m(Pp,N),nw(S)) /\ N < n-p(S,Ac) [metadata "CA-
   rspromise-nw5"] .
ceq nw(r&s-promise(S,Ac,Pp,N)) = nw(S) if a-l(S,Ac) = a2 /\
   member(prepare-m(Pp,N),nw(S)) /\ N < n-p(S,Ac) [metadata "CA-
   rspromise-nw6"] .
ceq nw(r&s-promise(S,Ac,Pp,N)) = nw(S) if not(member(prepare-m(
   Pp,N),nw(S))) [metadata "CA-rspromise-nw7"] .
ceq nw(r&s-promise(S,Ac,Pp,N)) = nw(S) if ((a-l(S,Ac) ~ a0) =
   false /\ (a-l(S,Ac) ~ a1) = false /\ (a-l(S,Ac) ~ a2) = false
   )  [metadata "CA-rspromise-nw8"] .
eq unique(r&s-promise(S,Ac,Pp1,N),Pp) = unique(S,Pp) .
eq vUser(r&s-promise(S,Ac,Pp1,N),Pp) = vUser(S,Pp) .
eq list-Ld(r&s-promise(S,Ac,Pp1,N),Pp) = list-Ld(S,Pp) .
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = N if a-l(S,Ac1) = a0 /\
   member(prepare-m(Pp,N),nw(S)) /\ (N >= n-p(S,Ac1)) /\ Ac1 =
   Ac [metadata "CA-rspromise-np1"].
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = N if a-l(S,Ac1) = a1 /\
   member(prepare-m(Pp,N),nw(S)) /\ (N >= n-p(S,Ac1)) /\ Ac1 =
   Ac [metadata "CA-rspromise-np2"].
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = N if a-l(S,Ac1) = a2 /\
   member(prepare-m(Pp,N),nw(S)) /\ (N >= n-p(S,Ac1)) /\ Ac1 =
   Ac [metadata "CA-rspromise-np3"].
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = n-p(S,Ac) if a-l(S,Ac1) =
   a0 /\ member(prepare-m(Pp,N),nw(S)) /\ (N < n-p(S,Ac1)) /\
   Ac1 = Ac [metadata "CA-rspromise-np4"].
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = n-p(S,Ac) if a-l(S,Ac1) =
   a1 /\ member(prepare-m(Pp,N),nw(S)) /\ (N < n-p(S,Ac1)) /\
   Ac1 = Ac [metadata "CA-rspromise-np5"].
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = n-p(S,Ac) if a-l(S,Ac1) =
   a2 /\ member(prepare-m(Pp,N),nw(S)) /\ (N < n-p(S,Ac1)) /\
   Ac1 = Ac [metadata "CA-rspromise-np6"].
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = n-p(S,Ac) if (Ac1 ~ Ac) =
   false [metadata "CA-rspromise-np7"].
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = n-p(S,Ac) if not(member(
   prepare-m(Pp,N),nw(S))) [metadata "CA-rspromise-np8"].
ceq n-p(r&s-promise(S,Ac1,Pp,N),Ac) = N if ((a-l(S,Ac1) ~ a0) =
   false /\ (a-l(S,Ac1) ~ a1) = false /\ (a-l(S,Ac1) ~ a2) =
```

```
   false) [metadata "CA-rspromise-np9"].
eq n-a(r&s-promise(S,Ac1,Pp,N),Ac) = n-a(S,Ac) .
eq v-a(r&s-promise(S,Ac1,Pp,N),Ac) = v-a(S,Ac) .
eq list-Ln(r&s-promise(S,Ac,Pp,N),Ln) = list-Ln(S,Ln) .
eq v-d(r&s-promise(S,Ac,Pp,N),Ln) = v-d(S,Ln) .

--- r&s-learn
eq p-l(r&s-learn(S,Pp1,Ac,N,V),Pp) = p-l(S,Pp) .
ceq a-l(r&s-learn(S,Pp,Ac1,N,V),Ac) = a2 if a-l(S,Ac1) = a1 /\
   member(accept-m(Pp,N,V),nw(S)) /\ Ac1 = Ac [metadata "CA-
   rslearn-al1"] .
ceq a-l(r&s-learn(S,Pp,Ac1,N,V),Ac) = a2 if a-l(S,Ac1) = a2 /\
   member(accept-m(Pp,N,V),nw(S)) /\ Ac1 = Ac [metadata "CA-
   rslearn-al2"] .
ceq a-l(r&s-learn(S,Pp,Ac1,N,V),Ac) = a-l(S,Ac) if (Ac1 ~ Ac) =
   false [metadata "CA-rslearn-al3"] .
ceq a-l(r&s-learn(S,Pp,Ac1,N,V),Ac) = a-l(S,Ac) if not(member(
   accept-m(Pp,N,V),nw(S))) [metadata "CA-rslearn-al4"] .
ceq a-l(r&s-learn(S,Pp,Ac1,N,V),Ac) = a-l(S,Ac) if ((a-l(S,Ac1)
   ~ a1) = false /\ (a-l(S,Ac1) ~ a2) = false) [metadata "CA-
   rslearn-al5"] .
eq l-l(r&s-learn(S,Pp,Ac,N,V),Ln) = l-l(S,Ln) .
ceq nw(r&s-learn(S,Pp,Ac,N,V)) = learn-m(Ac,N,V) nw(S) if (a-l(S
   ,Ac) = a1) /\ member(accept-m(Pp,N,V),nw(S)) /\ (N >= n-p(S,
   Ac)) [metadata "CA-rslearn-nw1"] .
ceq nw(r&s-learn(S,Pp,Ac,N,V)) = learn-m(Ac,N,V) nw(S) if (a-l(S
   ,Ac) = a2) /\ member(accept-m(Pp,N,V),nw(S)) /\ (N >= n-p(S,
   Ac)) [metadata "CA-rslearn-nw2"] .
ceq nw(r&s-learn(S,Pp,Ac,N,V)) = nw(S) if (a-l(S,Ac) = a1) /\
   member(accept-m(Pp,N,V),nw(S)) /\ (N < n-p(S,Ac)) [metadata "
   CA-rslearn-nw3"] .
ceq nw(r&s-learn(S,Pp,Ac,N,V)) = nw(S) if (a-l(S,Ac) = a2) /\
   member(accept-m(Pp,N,V),nw(S)) /\ (N < n-p(S,Ac)) [metadata "
   CA-rslearn-nw4"] .
ceq nw(r&s-learn(S,Pp,Ac,N,V)) = nw(S) if not(member(accept-m(Pp
   ,N,V),nw(S))) [metadata "CA-rslearn-nw5"] .
ceq nw(r&s-learn(S,Pp,Ac,N,V)) = nw(S) if ((a-l(S,Ac) ~ a1) =
```

```
   false /\ (a-l(S,Ac) ~ a2) = false) [metadata "CA-rslearn-nw6
   "] .
eq unique(r&s-learn(S,Pp1,Ac,N,V),Pp) = unique(S,Pp) .
eq vUser(r&s-learn(S,Pp1,Ac,N,V),Pp) = vUser(S,Pp) .
eq list-Ld(r&s-learn(S,Pp1,Ac,N,V),Pp) = list-Ld(S,Pp) .
ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = N if (a-l(S,Ac1) = a1) /\
   member(accept-m(Pp,N,V),nw(S)) /\ (N >= n-p(S,Ac)) /\ Ac1 =
   Ac [metadata "CA-rslearn-np1"] .
ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = N if (a-l(S,Ac1) = a2) /\
   member(accept-m(Pp,N,V),nw(S)) /\ (N >= n-p(S,Ac)) /\ Ac1 =
   Ac [metadata "CA-rslearn-np2"] .
ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-p(S,Ac) if (a-l(S,Ac1) =
    a1) /\ member(accept-m(Pp,N,V),nw(S)) /\ (N < n-p(S,Ac)) /\
   Ac1 = Ac [metadata "CA-rslearn-np3"] .
ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-p(S,Ac) if (a-l(S,Ac1) =
    a2) /\ member(accept-m(Pp,N,V),nw(S)) /\ (N < n-p(S,Ac)) /\
   Ac1 = Ac [metadata "CA-rslearn-np4"] .
ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-p(S,Ac) if (Ac1 ~ Ac) =
   false [metadata "CA-rslearn-np5"] .
ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-p(S,Ac) if not(member(
   accept-m(Pp,N,V),nw(S))) [metadata "CA-rslearn-np6"] .
ceq n-p(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-p(S,Ac) if ((a-l(S,Ac1)
   ~ a1) = false /\ (a-l(S,Ac1) ~ a2) = false) [metadata "CA-
   rslearn-np7"] .
ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = N if (a-l(S,Ac1) = a1) /\
   member(accept-m(Pp,N,V),nw(S)) /\ (N >= n-p(S,Ac)) /\ Ac1 =
   Ac [metadata "CA-rslearn-na1"] .
ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = N if (a-l(S,Ac1) = a2) /\
   member(accept-m(Pp,N,V),nw(S)) /\ (N >= n-p(S,Ac)) /\ Ac1 =
   Ac [metadata "CA-rslearn-na2"] .
ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-a(S,Ac) if (a-l(S,Ac1) =
    a1) /\ member(accept-m(Pp,N,V),nw(S)) /\ (N < n-p(S,Ac)) /\
   Ac1 = Ac [metadata "CA-rslearn-na3"] .
ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-a(S,Ac) if (a-l(S,Ac1) =
    a2) /\ member(accept-m(Pp,N,V),nw(S)) /\ (N < n-p(S,Ac)) /\
   Ac1 = Ac [metadata "CA-rslearn-na4"] .
ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-a(S,Ac) if (Ac1 ~ Ac) =
```

```
    false [metadata "CA-rslearn-na5"] .
  ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-a(S,Ac) if not(member(
    accept-m(Pp,N,V),nw(S))) [metadata "CA-rslearn-na6"] .
  ceq n-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = n-a(S,Ac) if ((a-l(S,Ac1)
    ~ a1) = false /\ (a-l(S,Ac1) ~ a2) = false) [metadata "CA-
    rslearn-na7"] .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = V if (a-l(S,Ac1) = a1) /\
    member(accept-m(Pp,N,V),nw(S)) /\ (N >= n-p(S,Ac)) /\ Ac1 =
    Ac [metadata "CA-rslearn-va1"] .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = V if (a-l(S,Ac1) = a2) /\
    member(accept-m(Pp,N,V),nw(S)) /\ (N >= n-p(S,Ac)) /\ Ac1 =
    Ac [metadata "CA-rslearn-va2"] .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = v-a(S,Ac) if (a-l(S,Ac1) =
     a1) /\ member(accept-m(Pp,N,V),nw(S)) /\ (N < n-p(S,Ac)) /\
    Ac1 = Ac [metadata "CA-rslearn-va3"] .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = v-a(S,Ac) if (a-l(S,Ac1) =
     a2) /\ member(accept-m(Pp,N,V),nw(S)) /\ (N < n-p(S,Ac)) /\
    Ac1 = Ac [metadata "CA-rslearn-va4"] .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = v-a(S,Ac) if (Ac1 ~ Ac) =
    false [metadata "CA-rslearn-va5"] .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = v-a(S,Ac) if not(member(
    accept-m(Pp,N,V),nw(S))) [metadata "CA-rslearn-va6"] .
  ceq v-a(r&s-learn(S,Pp,Ac1,N,V),Ac) = v-a(S,Ac) if ((a-l(S,Ac1)
    ~ a1) = false /\ (a-l(S,Ac1) ~ a2) = false) [metadata "CA-
    rslearn-va7"] .
  eq list-Ln(r&s-learn(S,Pp,Ac,N,V),Ln) = list-Ln(S,Ln) .
  eq v-d(r&s-learn(S,Pp,Ac,N,V),Ln) = v-d(S,Ln) .

--- Learner
  --- r-learn
  eq p-l(r-learn(S,Ac,N,V,Ln),Pp) = p-l(S,Pp) .
  eq a-l(r-learn(S,Ac1,N,V,Ln),Ac) = a-l(S,Ac) .
  ceq l-l(r-learn(S,Ac,N,V,Ln1),Ln) = l0 if l-l(S,Ln1) = l0 /\
    member(learn-m(Ac,N,V),nw(S)) /\ Ln = Ln1 [metadata "CA-learn
    -pl1"] .
  ceq l-l(r-learn(S,Ac,N,V,Ln1),Ln) = l-l(S,Ln) if (Ln ~ Ln1) =
    false [metadata "CA-learn-pl2"] .
```

```
ceq l-l(r-learn(S,Ac,N,V,Ln1),Ln) = l-l(S,Ln) if not(member(
   learn-m(Ac,N,V),nw(S))) [metadata "CA-learn-pl3"] .
ceq l-l(r-learn(S,Ac,N,V,Ln1),Ln) = l-l(S,Ln) if l-l(S,Ln1) ~ l0
    = false [metadata "CA-learn-pl4"] .
eq nw(r-learn(S,Ac,N,V,Ln)) = nw(S) .
eq unique(r-learn(S,Ac,N,V,Ln),Pp) = unique(S,Pp) .
eq vUser(r-learn(S,Ac,N,V,Ln),Pp) = vUser(S,Pp) .
eq list-Ld(r-learn(S,Ac,N,V,Ln),Pp) = list-Ld(S,Pp) .
eq n-p(r-learn(S,Ac1,N,V,Ln),Ac) = n-p(S,Ac) .
eq n-a(r-learn(S,Ac1,N,V,Ln),Ac) = n-a(S,Ac) .
eq v-a(r-learn(S,Ac1,N,V,Ln),Ac) = v-a(S,Ac) .
ceq list-Ln(r-learn(S,Ac,N,V,Ln1),Ln) = updateTri(< Ac ; N ; V
   >,list-Ln(S,Ln1)) if l-l(S,Ln1) = l0 /\ member(learn-m(Ac,N,V
   ),nw(S)) /\ (V ~ null) = false /\ Ln = Ln1 [metadata "CA-
   learn-list-Ln1"] .
ceq list-Ln(r-learn(S,Ac,N,V,Ln1),Ln) = list-Ln(S,Ln) if (Ln ~
   Ln1) = false [metadata "CA-learn-list-Ln2"] .
ceq list-Ln(r-learn(S,Ac,N,V,Ln1),Ln) = list-Ln(S,Ln) if V =
   null [metadata "CA-learn-list-Ln3"] .
ceq list-Ln(r-learn(S,Ac,N,V,Ln1),Ln) = list-Ln(S,Ln) if not(
   member(learn-m(Ac,N,V),nw(S))) [metadata "CA-learn-list-Ln4"]
    .
ceq list-Ln(r-learn(S,Ac,N,V,Ln1),Ln) = list-Ln(S,Ln) if l-l(S,
   Ln1) ~ l0 = false [metadata "CA-learn-list-Ln5"] .
eq v-d(r-learn(S,Ac,N,V,Ln1),Ln) = v-d(S,Ln) .

--- decide
eq p-l(decide(S,Ln),Pp) = p-l(S,Pp) .
eq a-l(decide(S,Ln),Ac) = a-l(S,Ac) .
ceq l-l(decide(S,Ln1),Ln) = l1 if l-l(S,Ln1) = l0 /\ Ln = Ln1 /\
    ((majN(ctoPL(list-Ln(S,Ln1))) * 2) >= (nAcceptor + 1)) /\ (
   decideV(list-Ln(S,Ln)) ~ null) = false [metadata "CA-decide-
   ll1"] .
ceq l-l(decide(S,Ln1),Ln) = l-l(S,Ln) if ((majN(ctoPL(list-Ln(S,
   Ln1))) * 2) >= (nAcceptor + 1)) = false [metadata "CA-decide-
   ll2"] .
ceq l-l(decide(S,Ln1),Ln) = l-l(S,Ln) if (Ln ~ Ln1) = false [
```

```
    metadata "CA-decide-ll3"] .
  ceq l-l(decide(S,Ln1),Ln) = l-l(S,Ln) if decideV(list-Ln(S,Ln))
    = null [metadata "CA-decide-ll4"] .
  ceq l-l(decide(S,Ln1),Ln) = l-l(S,Ln) if (l-l(S,Ln1) ~ l0) =
    false [metadata "CA-decide-ll5"] .
  eq nw(decide(S,Ln)) = nw(S) .
  eq unique(decide(S,Ln),Pp) = unique(S,Pp) .
  eq vUser(decide(S,Ln),Pp) = vUser(S,Pp) .
  eq list-Ld(decide(S,Ln),Pp) = list-Ld(S,Pp) .
  eq n-p(decide(S,Ln),Ac) = n-p(S,Ac) .
  eq n-a(decide(S,Ln),Ac) = n-a(S,Ac) .
  eq v-a(decide(S,Ln),Ac) = v-a(S,Ac) .
  eq list-Ln(decide(S,Ln1),Ln) = list-Ln(S,Ln) .
  ceq v-d(decide(S,Ln1),Ln) = decideV(list-Ln(S,Ln1)) if l-l(S,Ln1
    ) = l0  /\ (decideV(list-Ln(S,Ln1)) ~ null) = false /\ ((majN
    (ctoPL(list-Ln(S,Ln1))) * 2) >= (nAcceptor + 1)) /\ Ln = Ln1
    [metadata "CA-decide-vd1"] .
  ceq v-d(decide(S,Ln1),Ln) = v-d(S,Ln) if ((majN(ctoPL(list-Ln(S,
    Ln1))) * 2) >= (nAcceptor + 1)) = false [metadata "CA-decide-
    vd2"] .
  ceq v-d(decide(S,Ln1),Ln) = v-d(S,Ln) if (Ln ~ Ln1) = false [
    metadata "CA-decide-vd3"] .
  ceq v-d(decide(S,Ln1),Ln) = v-d(S,Ln) if decideV(list-Ln(S,Ln1))
     = null [metadata "CA-decide-vd4"] .
  ceq v-d(decide(S,Ln1),Ln) = v-d(S,Ln) if (l-l(S,Ln1) ~ l0) =
    false [metadata "CA-decide-vd5"] .
endfth)
```

Appendix E

Paxos Verification by CITP

```
load ui

(fth GOAL is
  inc PAXOS .
  var S : Sys .
  vars L L1 : Learner .
   ceq [lemma1] : majV(ctoPL(list-Ln(S,L))) = majV(ctoPL(list-Ln(S
      ,L1))) if ((majN(ctoPL(list-Ln(S,L))) * 2) >= (nAcceptor +
      1)) = true /\ ((majN(ctoPL(list-Ln(S,L1))) * 2) >= (
      nAcceptor + 1)) = true [nonexec] .
  ceq [lemma2] : ((majN(ctoPL(list-Ln(S,L))) * 2) >= (nAcceptor +
     1)) = true if (v-d(S,L) ~ null) = false [nonexec] .
  ceq [lemma3] : majV(ctoPL(list-Ln(S,L))) = v-d(S,L) if (v-d(S,L)
      ~ null) = false [nonexec] .
endfth)

 (goal GOAL |- ceq (v-d(S:Sys,L:Learner) ~ v-d(S:Sys,L1:Learner))
    = true if (v-d(S:Sys,L:Learner) ~ null) = false /\ (v-d(S:Sys,
    L1:Learner) ~ null) = false ; )

(set ind on S:Sys .)
(apply SI .)
(auto .)
--- decide
(apply TC CA .)
(apply IP RD .)
```

```
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(init lemma1 by (S:Sys <- x#1 ;) (L:Learner <- z#2 ;) (L1:Learner
   <- L1#4 ;) .)
(init lemma2 by (S:Sys <- x#1 ;) (L:Learner <- L1#4 ;) .)
(init lemma3 by (S:Sys <- x#1 ;) (L:Learner <- L1#4 ;) .)
(apply RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(init lemma1 by (S:Sys <- x#1 ;) (L:Learner <- z#2 ;) (L1:Learner
   <- L#3 ;) .)
(init lemma2 by (S:Sys <- x#1 ;) (L:Learner <- L#3 ;) .)
(init lemma3 by (S:Sys <- x#1 ;) (L:Learner <- L#3 ;) .)
(apply RD .)
(apply CA IP RD .)
--- case1
(cp eq l-l(x#1,z#2) = l0  . >< eq l0 ~ l-l(x#1,z#2) = false  .)
(equation .)
(apply RD .)
--- case2
(cp eq l-l(x#1,z#2) = l0  . >< eq l0 ~ l-l(x#1,z#2) = false  .)
(equation .)
(apply RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
```

```
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
(apply IP RD .)
--- others
(apply TC IP RD .)
(apply TC IP RD .)
(apply TC IP RD .)
(apply TC IP RD .)
(apply TC IP RD .)
(apply TC IP RD .)
(apply TC IP RD .)
```

Bibliography

[BBF+10] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.

[CBS09] Bernadette Charron-Bost and André Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[CDE+11] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.6)*, January 2011.

[CGJ+01] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, UK, 2001. Springer-Verlag.

[CGR07] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *In Proc. of PODC*, pages 398–407. ACM Press, 2007.

[CMMP95] Tim Coe, Terje Mathisen, Cleve Moler, and Vaughan Pratt. Computational aspects of the Pentium affair. 2(1):18–30, Spring 1995.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[DdM06] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for dpll(t). In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 81–94, Berlin, Heidelberg, 2006. Springer-Verlag.

[DF98] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ report : the language, proof techniques, and methodologies for object-oriented algebraic specification.* AMAST series in computing. World scientific, Singapore, River Edge, NJ, London, 1998.

[GS01] Rachid Guerraoui and André Schiper. The generic consensus service. *IEEE Trans. Software Eng.*, 27(1):29–41, 2001.

[GZCA13] Daniel Gâinâ, Min Zhang, Yuki Chiba, and Yasuhito Arimoto. Constructor-based inductive theorem prover. In *CALCO*, pages 328–333, 2013.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[Lam01a] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[Lam01b] Butler W. Lampson. The abcd's of paxos. In Ajay D. Kshemkalyani and Nir Shavit, editors, *PODC*, page 13. ACM, 2001.

[Lam02] Leslie Lamport. Paxos made simple, fast, and byzantine. In *OPODIS*, pages 7–9, 2002.

[MM13] Alessandro Mei Marzullo, Keith and Hein Meling. A simpler proof for paxos and fast paxos, 2013. Course notes.

[MRS] Leonardo De Moura, Harald Ruess, and Maria Sorea. Bounded model checking and induction: From refutation to verification (extended abstract, category a. In *Proceedings of the 15th International Conference on Computer Aided Verification, CAV 2003, volume 2725 of Lecture Notes in Computer Science*, pages 14–26. Springer.

[OF06] Kazuhiro Ogata and Kokichi Futatsugi. Some tips on writing proof scores in the ots/cafeobj method. In *Essays Dedicated to Joseph A. Goguen*, pages 596–615, 2006.

[TS11] Tatsuhiro Tsuchiya and André Schiper. Verification of consensus algorithms using satisfiability solving. *Distributed Computing*, 23(5-6):341–358, 2011.

[Wah13] Thomas Wahl. The k-induction principle, 2013. Notes.