# A Case Study: Proving Paxos with the IOA Toolkit [*]

Nicole Immorlica, Toh Ne Win [†]

May 25, 2002

**Abstract**

Paxos is an important distributed algorithm that implements consensus in the presence of stopping failures. It was introduced by Leslie Lamport in 1990 and published in 1998 [8]. In this paper, we present a formal safety proof of the Paxos algorithm using an interactive theorem prover. Using the I/O automaton [13] model of Paxos from Lynch and Shvartsman [11], we define a forward simulation from Paxos to the consensus specification using several intermediate automata and present and prove invariants of each automaton. Through this case study, we highlight the power and use of the IOA language and toolkit.

## 1 Introduction

Distributed consensus is an important problem that captures a core issue in many computer science applications such as consistent distributed databases. The problem addresses the situation in which there is a set of $n$ processes. Each process can propose a value, but eventually they all must agree on a common value. The consensus procedure must be *safe* at all times. That is, the common value must be a proposed value, there must be at most one common value, and no process should ever agree on a value different from the common value. Furthermore, the consensus procedure must be *live*. That is, eventually all processes should learn the value. The consensus procedure should work even in the presence of asynchronous processes, benign process failures, and message loss and duplication.

As an example, suppose there are $n$ terminals $t_1, \dots, t_n$ and a user at each terminal $t_i$. The terminals maintain a distributed database. Each user proposes a value $\pi(t_i)$ for database entry $\pi$. The terminals run a consensus procedure to decide which value $\pi(t_i)$ will be assigned to database entry $\pi$. The consensus procedure guarantees that the database will be consistent and complete (i.e. return the same value for $\pi$ no matter what terminal it is accessed from) even in the presence of benign terminal failures.

The general problem of consensus has been studied extensively in the literature [10, 4, 8, 9, 11]. As early as 1985, it was known that the consensus problem can not be solved at all in a completely asynchronous setting, even with at most one faulty process [4]. Thus, in order to design an algorithm for consensus, the consensus conditions had to be relaxed. In 1990, Lamport sacrificed the liveness condition and designed an algorithm known as Paxos to implement consensus safely [8, 9]. In 2002, Lynch and Shvartsman formalized the correctness proof of Paxos and provided a performance analysis under certain timing and failure assumptions [11].

[*]Project Report for 6.962/6.897

[†]{nickle, tohn}@theory.lcs.mit.edu

This paper presents a formal, mechanized proof of the safety properties of the Paxos algorithm. Via an interactive theorem prover toolkit, we prove the Paxos algorithm implements the consensus specification. That is, we show every possible externally observable outcome of the Paxos algorithm is also an externally observable outcome of the consensus specification. We first define both the Paxos algorithm and the consensus specification as input/output automata [13] using the IOA language [6]. To prove that Paxos implements consensus, we define a forward simulation relation from the Paxos automaton to the consensus automaton. We translate our automata and forward simulation conjecture into a form readable by the Larch Prover [5], using an automated translation tool IOA2LSL [2]. This work is largely based on the algorithm code and definitions introduced by Lynch and Shvartsman in [11]. Our main contribution is the automated proof of this complicated distributed algorithm, and the discovery of new invariants that are needed for the proof. We provide a complete and detailed proof that Paxos implements consensus and also demonstrate the performance of the IOA toolkit on a complicated distributed algorithm.

The rest of this paper is organized as follows. Section 2 gives a short introduction to most of the mathematical definitions and theorems that we need. Section 3 formalizes the consensus problem and Paxos solution and introduces the input/output automata specification of each one. Section 4 presents the formal proof and discusses the use of the IOA toolkit.

## 2 Mathematical Foundations

Much effort has been exerted during the years to formalize the notion of algorithms and distributed systems. There are several standard models such as Temporal Logic of Actions (TLA) [7] and input/output (I/O) automata [13]. In addition to these models, there are several standard proof methods for implementation theorems. These methods include composition [14] and simulations [12]. Here we give a brief overview of the models and methods used in our proofs and the automated tools designed to support them.

### 2.1 I/O automata

We will define all the algorithms we describe in terms of input/output or I/O automata. These automata reason about algorithms in terms of the their state machine representation.

**Definition 1** *An em I/O automaton $A$ is made up of four parts:*

- *$states(A)$ is a state space, usually written as a cross product of some variables.*
- *$start(A) \subseteq states(A)$ is a set of start states.*
- *$sig(A)$ is a signature that lists the actions of the automaton. The signature specifies the type of each action as either input, output, or internal.*
- *$trans(A) \subseteq states(A) \times actions(A) \times states(A)$ is a transition relation that tells which actions are enabled at which states, and the effects of the actions. Input actions are always enabled.*

An execution of an I/O automaton is a sequence of interleaved actions and states. The set of all possible executions is written as $execs(A)$. A trace of an execution is the sequence of all the external (i.e. input or output) actions in the execution. The set of all traces is written as $traces(A)$.

Often we would like to prove statements of the form "nothing bad happens" in the execution of an algorithm. For example, one might wish to prove that during the execution of Kruskal's minimum spanning tree algorithm, the graph that the algorithm is building is always a tree or a forest. Such properties are called *safety properties*.

**Definition 2** *A* safety property *$P$ is a set of traces, $traces(P)$ such that*

- *$traces(P)$ is nonempty.*
- *$traces(P)$ is prefix-closed: all finite prefixes of a trace in $traces(P)$ are also in $traces(P)$.*
- *$traces(P)$ is limit-closed: if an infinite sequence of traces $\beta_1, \beta_2, \ldots$ are in $traces(P)$ and each $\beta_i$ is a prefix of $\beta_{i+1}$, then the trace $\beta$ that is the limit of the sequence is also in $traces(P)$.*

One way to show that a safety property holds is through invariants. An invariant is a predicate of the states that holds at every point in every reachable execution. Another way to prove a saftey property is via a simulation relation. If automaton $B$ satisfies a safety property and $traces(A) \subseteq traces(B)$, then $A$ satisfies the safety property. We can show $traces(A) \subseteq traces(B)$ by showing that there exists a *forward simulation relation $f$* from an *implementation automaton $A$* to a *specification automaton $B$*.

**Definition 3** *A* forward simulation relation *$f$ from $A$ to $B$ is a relation from states of $A$ to states of $B$ that satisfies:*

- *Every start state of $A$ corresponds to a start state of $B$:*

$$\forall_{s \in start(A)} \exists_{u \in start(B)} f(s, u)$$

- *For every enabled transition $(s, a, s')$ of $A$ and every state $u$ of $B$ such that $f(s, u)$, there is a corresponding execution fragment $\beta$ of $B$ such that $trace(a) = trace(\beta)$ and $f(s', last(\beta))$ where $last(\beta)$ is the last state in the execution fragment $\beta$.*

If a forward simulation relation exists between $A$ and $B$, we write $A \rightarrow B$. In order to show $f(s', u')$, we usually use invariants[1] of $A$ and the hypothesis that $f(s, u)$. Sometimes the relation $f$ is difficult to define. In these cases, it can be useful to define one or several intermediate automata $C_1, \ldots, C_k$ and prove the $k+1$ forward simulation relations $A \rightarrow C_1, C_1 \rightarrow C_2, \ldots, C_k \rightarrow B$. This technique is know as *successive refinement.*

When reasoning about distributed systems, humans often find it easier to consider a system of automata. However, all the techniques we have developed for proving safety reason about single automata. Thus it is useful to define a formal way of combining separate automata that form a single system into a single automaton that represents that system. We would like to define this combination in such a way that safety properties of the combined automata imply safety properties of the individual automata. Although we do not take the time to define it formally here, the technique hinted at exists and is called *composition.* Basically composition requires that the signatures of the automata be compatible. It forms the combined (or composed) automaton by considering cross products of states of the component automata and allowing a transition whenever the projected transition is valid on every component automaton.

## 2.2 The IOA Toolkit

The IOA language allows I/O automata to be written as programs. The signature of the automata is declared at the begining of the program. The states are declared by listing the state variables, and the start state is implicit in the variable initializers. Each transition contains a (conjoined) set of preconditions. Transition effects may be specified declaratively (as a predicate on pre and post

[1]Technically, we also have to also show that $s$ and $u$ are reachable states. However, for the simulation, we are only interested in the reachable states, where the invariants have been proven to hold.

states) or imperatively (using assignments). Safety properties can be expressed as invariants and as simulation relations in the IOA code itself. These are checked during execution and are written as proof obligations for the theorem prover tool, Larch Prover (LP).

LP [5] is a theorem prover that uses multi-sorted first-order logic. In order to convert an IOA program and its invariants and simulation relations into first order logic for LP, we use a tool IOA2LSL. The I/O automaton's transitions become assertions in LP's body of knowledge about how pre and post states of the automaton relate.

The verification of safety properties involves the verification of invariants and the proof of a simulation relation. Bogdanov [2] developed standard ways to proceed with these proofs in LP. The proofs are by induction on the actions of the automaton. To prove an invariant $Inv$ holds in all reachable states, we first prove that $Inv$ holds in the start state. Then we prove that if $Inv$ holds on state $s$, and if $a$ is a valid action from $s$, $Inv$ also holds on the post state $s'$. In LP, we write

```
prove Start(s) => Inv(s)
prove Inv(s) /\ isStep(s, a, s') => Inv(s')
```

To prove a relation $f(s,u)$ defined in the IOA code is a forward simulation between the states $s$ of the implementation automaton $A$ and the states $u$ of the specification automaton $B$, we first prove the start state correspondence and then we show that every enabled action of $A$ has a corresponding execution fragment that maintains the relation. In LP, we write

```
prove Start(s) => \E u : States[UpperLevel] (f(s, u) /\ Start(u))
prove isStep(s, a, s') /\ f (s, u) =>
  \E beta : Execs[UpperLevel]
        (trace(beta) = trace(a)
     /\ f(s', last(u, beta)))
     /\ execFrag(u, beta)
```

where $beta$ is an execution fragment of the upper level automaton, $last(u, beta)$ is the last state of the fragment, and $execFrag(u, beta)$ is a predicate indicating that $beta$ is a valid execution from $u$.

Both invariants and simulation relation proofs are completed using induction on the actions of the implementation automaton $A$. In LP, we begin the proof by writing

```
resume by induction on a : Actions[A]
```

LP then produces a proof subgoal for each possible action the implementation automaton can take. Then one usually specifies the corresponding execution fragment of the specification automaton. Most of the creativity of the proofs lies in these steps, but most of the work lies in proving that the last state of the specification automaton's execution fragment does indeed correspond to the last state of the implementation automaton's transition.

## 3 The Problem and Solution

We would like to solve the problem of distributed consensus. Given a set of asynchronous processors connected via a network, we would like to design an algorithm that allows the processors to reach a consensus regarding some value. In order to design such an algorithm, we first must formalize the notion of distributed consensus. For simplicity of presentation, we postpone the treatment of node failures until our discussion of the automaton description of distributed consensus.

**Definition 4** *Suppose we are given a set of nodes $N$ and a set of proposed values $V(t)$, initially empty. At any moment $t$ in time, a value $v$ may be added to $V(t)$ (i.e. $V(t)$ grows monotonically). The nodes $N$ are said to satisfy* distributed consensus *if at every moment $t$ in time there is some chosen value $v \in V(t)$ such that each $n \in N$ either has outputed $v$ exactly once or has not output a value.*

Notice that distributed consensus is a safety property of an automaton Cons. Intuitively, this is because if Cons fails to satisfy distributed consensus, then it fails at some particular moment in time. Formally, let $init(n, v)$ be the input action that adds $v$ to the set $V(t)$ and for each $n \in N$, let $decide(n, v)$ be the output action that $n$ outputs $v$. Then distributed consensus is the set of all traces $(\alpha_1, \ldots, \alpha_k)$ for $0 \leq k \leq \infty$ where $\alpha_i = init(m_i, u_i)$ or $\alpha_i = decide(n_i, v_i)$ such that

- $v_i = v_j$ for all $i, j$ (i.e. the chosen value is consistent)
- for all $i$ there is some $j < i$ such that $v_i = u_j$ (i.e. the chosen value was proposed at some time in the past)
- $n_i \neq n_j$ for all $i \neq j$ (i.e. nodes only choose once)

Clearly, this is nonempty, prefix-closed, and limit-closed.

Notice the trivial automaton that simply has no output satisfies distributed consensus. This corresponds to the elimination of the liveness condition from our intuitive notion of consensus. As we argued in Section 1, it is unfortunately necessary to eliminate the liveness from consensus if we wish to find an implementation.

As distributed consensus is a safety property, we can define an automaton whose traces are exactly those of distributed consensus, except now we include node failures. We call this automaton Cons. Let *initiated*, *decided*, and $failed$ be sets of nodes and *proposed* and *chosen* be sets of values. The *initiated*, *proposed*, and *chosen* sets are self-explanatory. The *decided* set represents nodes that have outputed a value. The $failed$ set represents nodes that have failed. These sets are the variables of Cons and form its state space, so $states(\text{Cons}) = initiated \times decided \times failed \times proposed \times chosen$. There is just one start state $start(\text{Cons}) = \{initiated = \emptyset, decided = \emptyset, failed = \emptyset, proposed = \emptyset, chosen = \emptyset\}$. The signature consists of four actions $sig(\text{Cons}) = \{input\ init(n, v),\ input\ fail(n),\ output\ decide(n, v),\ internal\ chooseVal(v)\}$. The transition relation can be viewed as preconditions and effects of the actions and is as follows: the input action $init(n, v)$ has no precondition (in fact, input actions must always be enabled) and adds node $n$ to *initiated* and value $v$ to *proposed* unless node $n$ has failed in which case the state does not change. The input action $fail(n)$ adds node $n$ to $failed$. The internal action $chooseVal(v)$ has the precondition that *chosen* is empty and has the effect of adding $v$ to *chosen*. Finally, the output action $decide(n, v)$ has the precondition that $n \in initiated - failed$, $n \notin decided$, and $v \in chosen$ and adds $n$ to decided. Figure 3 shows the IOA description of this automaton.

We will refer to the consensus automaton Cons as the *specification* automaton. We now present the algorithm Paxos that solves the distributed consensus problem. First we provide a high-level description of the algorithm, and then we define an *implementation* automaton Paxos that describes this algorithm. The Paxos algorithm was first introduced by Lamport [8]. It is a three-phase algorithm that satisfies distributed consensus. As with Cons, processes in Paxos are initiated with proposed values and can fail.

The Paxos algorithm introduces two new concepts — ballots and quorums. Ballots have identification numbers and values. The identification numbers have a total ordering defined on them. Each process has a unique set of ballots which it can initiate. The total ordering of ballots allows all the processes of the Paxos algorithm to agree on the same ballot when they all receive multiple

```
automaton Cons

signature

  input init(i : Node, v : Value)
  input fail(i : Node)
  output decide (i : Node, v : Value)
  internal chooseVal (v : Value)

states
  initiated : Set[Node] := { },
  proposed : Set[Value] := { },
  chosen : Set[Value]   := { },
  decided : Set[Node]   := { },
  failed : Set[Node]    := { }

transitions

input init (i, v)
eff
  if ¬(i in failed) then
    initiated := initiated union {i};
    proposed := proposed union {v};
  else
    initiated := initiated;
  fi

internal chooseVal (v)
pre
  v in proposed and
  chosen = { }
eff
  chosen := {v};

output decide (i, v)
pre
  i in initiated and
  ¬(i in decided) and
  ¬(i in failed) and
  v in chosen
eff
  decided  := decided union {i};

input fail (i)
eff
  failed := failed union {i};
```

Figure 1: Cons Automaton IOA Description

ballots. Quorums are sets of nodes. There are two types of quorums — read quorums and write quorums. The quorums are designed such that for all read quorums $r$ and all write quorums $w$, $r \cap w \neq \emptyset$. For example, one feasible quorum design is to have one read quorum consisting of all the nodes and one write quorum also consisting of all the nodes. This design is instructive in understanding the algorithm. Another more optimal design is to arrange the nodes in a matrix and have the rows be the read quorums and the columns be the write quorums. Then any read quorum has an intersection of size one with a write quorum. The intersection property of quorums prevents two processes that have received different sets of ballots from deciding on different ballots.

**Algorithm 5** *Throughout the algorithm, processes gossip about each other. In particular, they pass around information concerning what values have been proposed, what ballots have been proposed, what ballots have been assigned what values, who has voted for what, and who has abstained from what. Also, a process may vote at any moment during any phase for a ballot that it has received and not abstained from, and it may abstain from a ballot if it has received a larger ballot. For clarity, we will talk about two kinds of processes — leaders and learners. Leaders propose ballots and assign values to ballots. Learners abstain from and vote on ballots. Note a process can be both a leader and a learner.*

1. *In the first phase of the algorithm, leaders propose ballots. Each learner which has heard about this ballot proposal through the gossip is now free to abstain from smaller ballots that it has not voted for.*
2. *In the second phase, a leader considers the votes of a read quorum. It finds the largest ballot $b$ from which a read quorum has not abstained. If there is no such $b$, then the leader knows all ballots less than its ballot have failed, so the leader assigns a proposed value to its ballot. If there is such a $b$, the leader takes the value of $b$ and assigns this value to its own ballot.*
3. *In the third phase, when a process hears that a write quorum has voted for a ballot, it may decide on that ballot's value.*

At this point, it is instructive to consider an example.

**Example 6** Suppose there are 3 processes, *dubya*, $ashcroft$, and $rumsfeld$, which have been initiated with the values "axis of evil", "fear of god", and "war on terror". We will let the set of read quorums and the set of write quorums consist of the single set $\{ashcroft,\ rumsfeld\}$. Suppose the universe of ballot identifiers is the integers with the usual ordering. WLOG assume *dubya* becomes a leader process. For brevity of exposition, we will pretend the gossip in this circle of processors is highly efficient, and processes learn each other's information immediately. We will not record this gossip in the Paxos transcription. Then a possible execution of Paxos is as follows:

1. *dubya* assigns value "fear of god" to ballot 1
2. *dubya* proposes ballot 2 to $ashcroft$ and $rumsfeld$. Now $ashcroft$ and $rumsfeld$ may abstain from ballot 1.
3. $ashcroft$ votes on ballot 2
4. *dubya* proposes ballot 3 to $ashcroft$ and $rumsfeld$. Now $rumsfeld$ may abstain from ballot 2 even though $ashcroft$ has already voted for ballot 2.
5. $rumsfeld$ and $ashcroft$ abstain from ballot 1. Now a read quorum has abstained from ballot 1, so ballot 2 can be assigned any value.
6. $rumsfeld$ abstains from ballot 2. Now ballot 2 can not succeed, but it can not fail either (i.e. it will never be the case that a read quorum or a write quorum agrees on whether to vote or abstain from this ballot). Therefore, ballot 2 must be assigned a value.

7. *dubya* assigns value "axis of evil" to ballot 2
8. $rumsfeld$ and $ashcroft$ vote for ballot 3. Notice processes can vote for ballots that don't have a value.
9. *dubya* assigns value "axis of evil" to ballot 3
10. all three processes decide on the value of ballot 3, "axis of evil". Note the processes had to wait for a value to be assigned to ballot 3 before they could decide on it.

This definition of Paxos arises quite naturally from the requirements of distributed consensus as argued by Lamport [8]. However, to formally prove that Paxos satisfies distributed consensus requires a bit more work. First we must define an *implementation* automaton Paxos that describes Paxos, and then we must prove that there is a simulation relation from Paxos to Cons. The definition of the Paxos automaton is presented in Appendix A.

We have arrived at this I/O automaton definition by composing all the node automata and all the channel automata. Every action and every state variable is indexed by the individual node automaton which the action/state variable corresponds to. Thus, if in the underlying system node 2 is initialized with value $d$, the Paxos automaton will have an action of the form $init(2, d)$ and will add $d$ to node 2's set of proposed values $proposed[2] \leftarrow proposed[2] \cup \{d\}$. This is different from the Cons automaton where there was just one global set of proposed values.

In the underlying system, nodes communicate to each other through channels. All the internal *send* and *recv* actions are artifacts of the channel automata. In the underlying system, there is one channel automaton for every pair of node automaton, and so the *send* and *recv* actions are indexed by two nodes. The underlying channel automaton may duplicate messages, reorder messages, and lose messages, but it may not create messages. We model these properties of the channel automaton by maintaining a set of messages $S$ in the channel. The *send* action adds its input message $s$ to the set $S$. The *recv* action has as a precondition $s \in S$.

Now we will argue the IOA description in Appendix A actually describes Algorithm 5, Paxos. The gossip is achieved via the *send* and *recv* actions. Voting is represented by the *vote* action and abstention by the *abstain* action. The conditions of voting and abstention are preconditions of the corresponding action. The ballot proposals from phase 1 are initiated by a *newBallot* action and completed by a *makeBallot* action. The presence of *newBallot*, *makeBallot*, and the *doMakeBallot* variable is a technical detail. We write Paxos in this way simply to make it composable with a timed version of Paxos in future work. The second phase of Algorithm 5 is encoded in the $assignValue(i, b, v)$ action. This lets automaton $i$ assigns value $v$ to ballot $b$ if

$$\left(\forall b' < b,\ b' \in\ dead\right) \vee \left(\exists b'' < b,\ val(b'') = v \wedge (\forall b',\ b'' < b' < b,\ b' \in\ dead)\right)$$

where $b'$ and $b''$ are *any* ballots in the universe of ballots and dead is the set of ballots from which process $i$ knows that a read quorum has abstained. This condition ensures that the value $v$ which $i$ assigns to $b$ is consistent with all smaller ballots, as Algorithm 5 states. The third and final phase of Algorithm 5 is encoded by the internal action, *internalDecide*, in which a process adds a ballot to its succeeded set, and the external output action *decide* in which a process decides on a value of a ballot in its succeeded set. We have encoded this third phase in two steps in order to allow more flexibility in our automaton — a process can decide internally long before it becomes inactive, and a process can decide internally on a ballot that doesn't have a value.

We have omitted a few technical details from our description of the IOA code. These details are not essential to an understanding of the algorithm and proof, but are necessary to actually run the proof in the Larch Prover. These details include the *mode* and $failed$ variables, the indexing of quorums on ballots, the implementation of *dead* mentioned above, the definition of *minBallot* for the smallest ballot and the *nil* value for ballots which have not been assigned a value.

## 4 The Proof

We prove that Paxos satisfies distributed consensus by defining a forward simulation from the Paxos automaton to the Cons automaton. The correctness of this proof follows from the discussion in Section 2. We will use two refinements in order to prove the forward simulation. The first refinement is a forward simulation from an automaton called Global1 to Cons. The second refinement is a forward simulation from an automaton called Global2 to Global1. These successive refinements allow us to prove the simulation relation incrementally by breaking up the proof into conceptual chunks. This has the advantage of making each individual proof easier and giving us more insight into the algorithm itself. In order for the forward simulations to work, we define Global1 and Global2 with the same input and output actions — $init(i, v)$, $fail(i)$, and $decide(i, v)$ — as Cons.

The Global1 automaton captures most of the essence of the Paxos automaton. It introduces a simplified notion of learning capabilities by defining internal *abstain* and *vote* actions. However, there are several major differences between Global1 and Paxos. Global1 is not a composition of node automaton. This means there is no node communication and so all the channel actions are missing from Global1. Furthermore, the *makeBallot*, *assignVal* and *internalDecide* actions are actions of the automaton as a whole. In terms of the algorithmic description of Paxos, Global1 encodes the first, second, and third phase in *makeBallot*, *assignVal*, and *internalDecide/decide*. The *makeBallot* action just ensures new ballots have a distinct identifier from old ballots. The *internalDecide* action just ensures succeeded ballots have a write quorum that has voted for them. These two actions are essentially identical to the corresponding Paxos automaton actions. The $assignVal(b, v)$ action is slightly different and in fact does not fully capture the corresponding Paxos automaton action. Instead of just checking the largest ballot $b' < b$ from which a read quorum has not abstained, it checks that *every* ballot $b' < b$ is either dead or has value $v$.

The second refinement is a forward simulation from an automaton called Global2 to an automaton called Global1. The Global2 automaton is exactly the same as the Global1 automaton except in the $assignVal(b, v)$ action. This transition is a full implementation of the second phase of the Paxos algorithm; it only checks that the largest ballot $b' < b$ from which a read quorum has not abstained has value $v$ if such a $b'$ exists.

As mentioned in Section 2.2, we prove the simulation relation for each implementation-specification automaton pair in LP using structural induction on the actions of the implementation automaton. The proofs use several invariants of the automata. We also prove these invariants in LP.

### 4.1 Global1 to Cons

Although not the longest in terms of length, this relation is conceptually the most important because it connects the decision of the consensus specification with the use of ballots. Intuitively, when a ballot is voted on by a quorum in Global1, the corresponding action in Cons is to choose a value to decide on. Thus, *internalDecide* should correspond to *chooseVal*, and the simulation relation we have is:

```
forward simulation from Global1 to Cons:

   Cons.initiated = Global1.initiated
∧  Cons.proposed  = Global1.proposed
∧  Cons.decided   = Global1.decided
∧  Cons.failed    = Global1.failed
∧  ∀ v : Value  ((∃ b : Ballot (b in Global1.succeeded ∧ Global1.val[b] = embed(v))) ⇒ v in Cons.chosen )
   ∧ ∀ v : Value (v in Cons.chosen ⇒ ∃ b : Ballot (b in Global1.succeeded ∧ Global1.val[b] = embed(v) ))
```

The last two clauses are the important ones — they are a biconditional in [11], but for convenience in LP, we choose to separate them. They say that the values of the ballots in *Global*1.*succeeded* are the same as those in *Cons.chosen*.

Even though this concept is clear, there are a few caveats that do not allow a direct correspondence. First, *internalDecide* can happen to more than one ballot (or more than once on the same ballot), while *chooseVal* requires that *Cons.chosen* be empty. Thus, the second time *internalDecide* happens in Global1, the corresponding execution in Cons is not *chooseVal* but the empty sequence. In LP, we handle this situation by doing a case analysis where the witness for the existentially quantified execution $\beta$ is different in each case.

Another caveat is that *Global*1 allows ballots to be voted and internally decided on before their values are assigned. This does not apparently affect the correctness of the algorithm[2], but makes the proof more complicated, because we need to take into account two new cases:

- When *internalDecide* fires on a ballot without a value, the corresponding $\beta$ execution is the empty sequence, even if *Global*1.*succeeded* was empty.
- When *assignVal* is fired, it could be assigning a value to a ballot already in *Global*1.*succeeded*. In this case, the corresponding $\beta$ is *chooseVal*.

Once these cases are handled, witness executions for internal actions of *Global*1 are as follows. Actions *vote*, *makeBallot* and *abstain* always have $\beta = \{\}$. Action *internalDecide*($b$) has $\beta = \{\}$ if there exists a ballot in *Global*1.*succeeded* that already has a value or if $b$ does not have a value. If $b$ has a value and *succeeded* does not, then $\beta = chooseVal(val[b])$. Lastly, *assignVal* has $\beta = \{\} = \{\}$ if it is assigning a value to a ballot not in succeeded, but corresponds to *chooseVal* otherwise.

### 4.1.1 Invariants Used

Lynch [11] mentioned four invariants necessary to prove the simulation relation:

- The set of voted ballots is disjoint from the set of abstained ballots.
- If $v$ is the value of a ballot, then $v$ was proposed.
- The set of succeeded ballots is disjoint from the set of dead ballots.
- If $b$ and $b'$ are two ballots such that $b$ has a value and $b' < b$, then either the value of $b'$ equals the value of $b$ or $b'$ is dead.

We have added another invariant used in the simulation relation proof:

- The set of succeeded ballots is a subset of the set of designated (i.e. made) ballots.

and two invariants used to prove the five main invariants themselves:

- If a ballot has succeeded, then a write quorum has voted for it.
- If a ballot is not designated (i.e. it has not been made), its value is nil.

The IOA description of these invariants follows.

```
invariant Inv1 of Global1: ∀ i : Node  (∀ b : Ballot (b in voted[i] ⇒ ¬(b in abstained[i])))
invariant Inv2 of Global1: ∀ b : Ballot (val[b] ≠ nil ⇒ val[b].val in proposed)
```

[2]It may reduce the fault tolerance specifications.

```
invariant Inv3 of Global1: ∀ b : Ballot (b in succeeded ⇒ ¬(b in dead(abstained)))

invariant Inv4 of Global1: ∀ b : Ballot ∀ b' : Ballot
   ((val[b] ≠ nil ∧ b' < b) ⇒ val[b'] = val[b] ∨ b' in dead(abstained))

invariant Inv5 of Global1: ∀ b : Ballot (b in succeeded ⇒ b in ballots)

invariant Inv6 of Global1:
  ∀ b_Inv6 : Ballot
  (b_Inv6 in succeeded ⇒
    ∃ b_qInv6 : Ballot
    ∀ n_Inv6 : Node
      (n_Inv6 in wquorums(b_qInv6) ⇒ b_Inv6 in voted[n_Inv6]))

invariant Inv7 of Global1:
  ∀ b_Inv7 : Ballot
  (¬(b_Inv7 in ballots) ⇒ val[b_Inv7] = nil)
```

Invariants 1 through 4 were the original ones. $Inv1$ and $Inv3$ are actually written as intersections in [11] but we rewrote them in terms of elements to better work with our set axioms in LP. We found that $Inv5$ was necessary because *makeBallot* assigns a value to the newly-created ballot, so we must ensure that succeeded ballots do not have their values changed. $Inv6$ was used in the proof of $Inv3$ and $Inv7$ in the proof of $Inv4$.

The property that quorums intersect was used in the simulation relation and the proof of $Inv3$.

## 4.2 Global2 to Global1

The state variables of the Global2 automaton were not different from those of the Global1 automaton. Thus the simulation relation was an equality mapping:

```
forward simulation from Global2 to Global1:
   Global1.initiated = Global2.initiated
∧  Global1.proposed  = Global2.proposed
∧  Global1.decided   = Global2.decided
∧  Global1.failed    = Global2.failed
∧  Global1.val       = Global2.val
∧  Global1.ballots   = Global2.ballots
∧  Global1.abstained = Global2.abstained
∧  Global1.voted     = Global2.voted
∧  Global1.succeeded = Global2.succeeded
```

The only non-trivial transition was $assignVal$. Even then, the LP proof was 9 lines for this transition.

We expected to use no invariants for proving this simulation relation, but we found that an equivalent of $Inv4$ was necessary. Nevertheless, the simulation relation was ultimately a trivial proof. The witness executions also had a one-to-one correspondence.

## 4.3 Paxos to Global2

Although the simulation relation proof from fully distributed Paxos to Global2 was longer than the previous two, this was mainly because Paxos had more transitions. Conceptually, the relation between the two automata was straightforward: the union of the data in the distributed Paxos is the state of Global2. In IOA, this was written as:

```
forward simulation from Paxos to Global2:
    ∀ i : Node (i in Global2.initiated ⇔ Paxos.mode[i] ≠ idle)
∧   ∀ v : Value (v in Global2.proposed ⇔ (∃ i : Node (v in Paxos.proposed[i])))
∧   ∀ i : Node (i in Global2.decided ⇔ Paxos.mode[i] = done)
∧   ∀ i : Node (i in Global2.failed ⇔ Paxos.failed[i])
∧   ∀ b : Ballot (b in Global2.succeeded ⇔ (∃ i : Node (b in Paxos.succeeded[i])))
∧   ∀ b : Ballot (Global2.val[b] = Paxos.val[b.procid][b])
∧   ∀ i : Node (Global2.voted[i] = Paxos.voted[i][i])
∧   ∀ i : Node (Global2.abstained[i] = Paxos.abstained[i][i])
∧   ∀ b : Ballot (b in Global2.ballots ⇔ (∃ i : Node (b in Paxos.ballots[i])))
```

Notice that there is no union operator appearing anywhere. This is because IOA does not support union over variables in a set (it only supports unions between two variables). However, the second conjunct is the equivalent of saying that *Global2.proposed* is the union of proposed values in each of the Paxos automata.

Note also that each automaton's program state (idle, active, done) and failure state (failed) directly mapped to Global2 variables after the changes we made in 5.

The witness executions of Global2 were again straightforward: every action done by a Paxos automaton had the same-named corresponding action in Global2, except for *doMakeBallot*, which had the empty execution. For example, a *vote* action in Paxos resulted in a *vote* action in Global2.

Of course, there are some actions in Global2, such as *internalDecide* that are not associated with a particular node. For these, whenever *internalDecide* was fired in Paxos, we fired *internalDecide* in Global2. This is possible because Global2 allows for repeats of previously performed internal actions, so multiple *internalDecide* on the same ballot by different nodes is acceptable.

We noticed that even though the simulation relation involved 9 clauses, no single action involved proving more than 4 of them. Most either went through immediately (in the case of *fail* or any of the channel sends) and the others mainly required 2-3 clauses. This is because LP notices which state variables change, and automatically proves the simulation relation conjunct for unchanged variables.

### 4.3.1 Invariants Used

There were 5 invariants needed for the proof, even though [11] mentioned 3 (listed as 1-3 here).

```
invariant DistInv1 of Paxos:  ∀ i : Node (∀ j : Node (abstained[j][i] subseteq abstained[i][i]))
invariant DistInv2 of Paxos:  ∀ i : Node (∀ j : Node (voted[j][i] subseteq voted[i][i]))
invariant DistInv3 of Paxos:  ∀ i : Node (∀ b : Ballot (val[i][b] ≠ nil ⇒ val[i][b] = val[b.procid][b]))
invariant DistInv4 of Paxos:  ∀ i : Node (∀ b : Ballot (b in ballots[i] ⇒ b in ballots[b.procid]))
invariant DistInv5 of Paxos:  ∀ b : Ballot (¬( b in ballots[b.procid]) ⇒ (val[b.procid])[b] = nil)
```

$DistInv4$ and $DistInv5$ were required for the last clause of the simulation relation in $makeBallot$ and the sixth clause in $assignVal$.

## 4.4 Miscellaneous proof details

There was only one place in the simulation relation where had to explicitly use quorums, and this was in *internalDecide*, where we had to prove that a write quorum existed in Global1 given that one existed in Global2. However, we never had to use read quorums because LP could use dead ballots without referring to quorums, and we did not need the property that read and write quorums intersect.

However, it must be noted that quorum intersection is needed for Paxos to properly implement Global1. It just happens that the property is not used in the proof because it is specified as an axiom. The advantage for us was that the proof of Paxos, which we expected to be more complex than than of Global1, was actually simpler in many ways.

Initially, we were unsure of how to implement the quorum specification in [11]. In the end, we settled on the idea of parameterizing quorums using a dummy variable as shown in Appendix B so that we could allow LP's first order logic to understand the concept "there exists a quorum" which would normally be a quantification over sets.

When we started trying to prove the simulation relations, however, we attempted to add another level of refinement between Global2 and Paxos, called Global3, so that Global1 and Global2 would use a single quorum and Global3 would expand to use different quorums that obeyed the intersection property. This, we thought, would make the proofs of Global1 and Global2 easier. What we found out, however, was the proving Global1 with multiple quorums was not difficult, but we were unable to find a simulation relation between Global3 and Global2. Thus, we changed the successive refinement back to the one presented in [11].

## 5 Conclusion

Using the IOA language and the Larch Prover, we were able to take the I/O automaton specification of Lamport's Paxos algorithm, written in [11] and prove its correctness. Some of the lessons learned for formal verification with LP include:

- When using channels, write the program so that the precondition of the *send* transition to the channel holds on channel contents at all times.
- Successive refinement is a useful technique for managing algorithm complexity. However, the size of proofs in simulation relations is not proportional to the length of the algorithms used, but rather to the conceptual differences between different abstraction levels.
- There is still too much extra work in using an interactive theorem prover. Much of our time was spent trying to understand what LP was trying to do rather than leading the tool towards a proof.

### 5.1 Further work

We suggest that Paxos's liveness properties could be proved using the current set of IOA tools and simple temporal logic in LP. We also consider the idea of reducing the work it takes to discover invariants.

#### 5.1.1 Liveness

The algorithm in [11] used timing to provide for liveness properties. However, the timing properties were only introduced in the Paxos automaton and were not part of the successive refinement as with the safety proofs. We were able to ignore the timed ballot trigger automaton because we were only proving safety properties.

Ideally, we would like to specify liveness properties at the Cons automaton level and use simulation relations as with safety to show that Paxos is live. The advantage of such a method is that proofs would be similar to safety proofs in that they reason over individual transitions rather than over executions. One way to prove liveness would be to use successive refinement with timed automata as described in [12]. However, the IOA language currently does not support timed automata.

[1] suggests a method for doing this using the standard I/O automaton model augmented with minimal temporal logic, using "liveness preserving simulation relations". A liveness preserving simulation relation is a standard simulation relation augmented by a liveness "lattice" function that, maps the liveness properties of the lower level automaton to the liveness properties of the upper level automaton. Liveness properties are always expressed in complemented-pairs form:

$\Box\Diamond A \rightarrow \Box\Diamond B$

which reads "always eventually $A$ implies always eventually $B$" where $A$ and $B$ are states of the I/O automaton.

Each complemented pair in the upper level automaton has to be satisfied by complemented pairs in the lower level automaton. This is done by providing a "lattice", or directed acyclic graph of complemented pairs in the lower automaton.

We would like to implement this as a standard method for proving liveness in IOA with LP or another prover. Implementing this method would involve a one-time cost of axiomatizing the complemented-pairs temporal logic in LP, followed by a modeling of the desired algorithms' liveness properties. With Paxos for example, a property we may wish to have in the Cons automaton is:

$\forall_i \Box\Diamond[i \in \mathit{initialized}] \rightarrow \Box\Diamond[\neg(i \in \mathit{failed}) \rightarrow i \in \mathit{decided}]$

That is, an initialized process eventually decides if it does not fail.

### 5.1.2 Invariant discovery

For the invariants of Paxos, we were given the important ones in [11], but these were not enough to fully prove the simulation relations. Discovering which invariants were needed using LP took time, and would have taken longer had we not already had some invariants given in [11]. One way to alleviate the problem would be to use runtime information to suggest invariants that may be true for the program. These invariants could then be human- or computer-filtered to be used in proofs of larger properties like simulation relations.

Daikon [3] is a tool that performs the dynamic, runtime analysis described above. Daikon can already process IOA data and output invariants in IOA syntax. It cannot discover all invariants, but the ones that it discovers are often enough to prove important program properties. Using Daikon for Paxos and other IOA programs is studied further in [15]. Presently, Daikon is able to discover the first five invariants in the simulation relation from Global1 to Cons. However, there may not yet be enough data to see how using Daikon would generalize.

## A Paxos IOA description

```
axioms AuxDist
axioms Null(Value)

type ModeType = enumeration of idle, active, done

automaton Global2

signature

  input init (i : Node, v : Value)
  input fail (i : Node)
  output decide (i : Node, v : Value)
  internal makeBallot (b : Ballot)
  internal abstain (i : Node, B : Set[Ballot])
  internal assignVal (b : Ballot, v : Value)
  internal vote (i : Node, b : Ballot)
  internal internalDecide (b : Ballot)

states
  initiated : Set[Node] := {},
  proposed : Set[Value] := {},
  decided : Set[Node]   := {},
  failed : Set[Node]    := {},
  ballots : Set[Ballot] := {},
  succeeded : Set[Ballot] := {},
  val : Array[Ballot, Null[Value]] := constant(nil),
  voted : Array[Node, Set[Ballot]] := constant({}),
  abstained  : Array[Node, Set[Ballot]] := constant({minBallot})

transitions

input init (i, v)
eff
  if ¬(i in failed) then
    initiated := initiated union {i};
    proposed := proposed union {v};
  else
    proposed := proposed;
  fi;

input fail (i)
eff
  failed := failed union {i}


internal makeBallot(b)
pre
  ∀ b' : Ballot (b' in ballots ⇒ (b' ≠ b)) /   b ≠ minBallot
eff
  ballots := ballots union {b};
  val[b] := nil;

internal assignVal (b, v)
pre
  b in ballots /   val[b] = nil /   v in proposed /   ((∀ b' : Ballot (b' < b ⇒ (b' in dead(abstained))))
   ∨
   (∃ b'': Ballot (val[b''] = embed(v) ∧ ∀ bd' : Ballot (b'' < bd' ⇒ bd' in dead(abstained))))
   )
eff
  val[b] := embed(v);

internal vote(i, b)
pre
  i in initiated /   ¬(i in failed) /   b in ballots /   ¬(b in abstained[i])
eff
  voted[i] := voted[i] union {b};

internal abstain (i, B)
pre
  i in initiated /   ¬(i in failed) /   voted[i] intersection B = {}
eff
  abstained[i] := abstained[i] union B;

internal internalDecide(b)
pre
  b in ballots /   ∃ b_qID : Ballot ∀ j : Node (j in wquorums(b_qID) ⇒ b in voted[j])
eff
```

```
transitions

input init (i_Me, v_Init)
eff
  if (mode[i_Me] = idle ∨ mode[i_Me] = active) then
    mode[i_Me] := active;
    proposed[i_Me] := proposed[i_Me] union {v_Init};
  else
    proposed[i_Me] := proposed[i_Me];
  fi;

internal newBallot (i_Me)
eff
  if (mode[i_Me] = active) then
    doMakeBallot[i_Me] := true;
  else
    doMakeBallot[i_Me] := doMakeBallot[i_Me];
  fi;

input fail (i_Me)
eff
  mode[i_Me] := failed;

internal makeBallot(i_Me, b_MakeBallot)
pre
  mode[i_Me] = active;
  doMakeBallot[i_Me];
  ∀ b'_MakeBallot : Ballot (b'_MakeBallot in ballots[i_Me] ⇒ b'_MakeBallot < b_MakeBallot);
  b_MakeBallot.procid = i_Me;
eff
  ballots[i_Me] := insert(b_MakeBallot, ballots[i_Me]);
  val[i_Me][b_MakeBallot] := nil;
  doMakeBallot[i_Me] := false;

internal assignVal (i_Me, b_AssignVal, v_AssignVal)
pre
  mode[i_Me] = active;
  b_AssignVal in ballots[i_Me];
  b_AssignVal.procid = i_Me;
  val[i_Me][b_AssignVal] = nil;
  v_AssignVal in proposed[i_Me];
  ((∀ b'_AssignVal : Ballot (b'_AssignVal < b_AssignVal ⇒
                            (b'_AssignVal in dead(abstained[i_Me]))))
   ∨
   (∃ b''_AssignVal: Ballot
      (val[i_Me][b''_AssignVal] = embed(v_AssignVal)
      and ∀ bd'_AssignVal : Ballot (b''_AssignVal < bd'_AssignVal ⇒
                                  bd'_AssignVal in dead(abstained[i_Me]))))
   )
eff
  val[i_Me][b_AssignVal] := embed(v_AssignVal);
```

Figure 3: Actions

```
internal vote(i_Me, b_Vote)
pre
  mode[i_Me] ≠ idle;
  mode[i_Me] ≠ failed;
  b_Vote in ballots[i_Me];
  val[i_Me][b_Vote] ≠ nil;
  ¬(b_Vote in abstained[i_Me][i_Me])
eff
  voted[i_Me][i_Me] := voted[i_Me][i_Me] union {b_Vote};

internal abstain (i_Me, B_Abstain)
pre
  mode[i_Me] ≠ idle;
  mode[i_Me] ≠ failed;
  ∀ b_Abstain : Ballot ((b_Abstain in B_Abstain) ⇒
     ∃ b'_Abstain : Ballot (b'_Abstain in ballots[i_Me] and b_Abstain < b'_Abstain));
  voted[i_Me][i_Me] intersection B_Abstain = {};
eff
  abstained[i_Me][i_Me] := abstained[i_Me][i_Me] union B_Abstain;

internal internalDecide(i_Me, b_InternalDecide)
pre
  mode[i_Me] = active;
  b_InternalDecide in ballots[i_Me];
  ∃ b_qID : Ballot ∀ j : Node (j in wquorums(b_qID) ⇒ b_InternalDecide in voted[i_Me][j])
eff
  succeeded[i_Me] := succeeded[i_Me] union {b_InternalDecide};

output decide(i_Me, v_Decide)
choose b_Decide : Ballot
pre
  mode[i_Me] = active;
  b_Decide in succeeded[i_Me];
  embed(v_Decide) = val[i_Me][b_Decide];
eff
  mode[i_Me] := done;
```

Figure 4: Actions

```
internal sendProposed (i_Me, j_You, v_SProposed)
pre
  mode[i_Me] = active;
  v_SProposed in proposed[i_Me];

internal sendBallot (i_Me, j_You, b_SBallot)
pre
  mode[i_Me] ≠ idle;
  b_SBallot in ballots[i_Me];

internal sendValue (i_Me, j_You, b_SValue, v_SValue)
pre
  mode[i_Me] ≠ idle;
  embed(v_SValue) = val[i_Me][b_SValue];

internal sendVote (i_Me, j_You, k_SVote, b_SVote)
pre
  mode[i_Me] ≠ idle;
  b_SVote in voted[i_Me][k_SVote];

internal sendAbstained (i_Me, j_You, k_SVote, B_SVote)
pre
  mode[i_Me] ≠ idle;
  B_SVote subset voted[i_Me][k_SVote];

internal recvProposed (i_Me, j_You, v_RProposed)

internal recvBallot (i_Me, j_You, b_RBallot)

internal recvValue (i_Me, j_You, v_RValue)

internal recvVote (i_Me, j_You, k_RVote, b_RVote)

internal recvAbstained (i_Me, j_You, k_RVote, B_RVote)
```

Figure 5: Actions

## B LSL Auxiliary Specifications

The following are the LSL specifications used as axioms for all three Paxos algorithms.

```
AuxDist : trait

includes TotalOrder(Ballot), TotalOrder(Node), Set(Ballot), Array(Node, Set[Ballot]), Set(Node), Integer

Ballot tuple of seqno : Int, procid : Node

introduces
  dummyNode : → Node,
  dummyValue : → Value,
  dummyBallot : → Ballot,
  minBallot : → Ballot,
  __<__ : Ballot, Ballot → Bool,
  __<__ : Node, Node → Bool,
  wquorums : Ballot → Set[Node],
  rquorums : Ballot → Set[Node],
  dead : Array[Node, Set[Ballot]] → Set[Ballot],
  haveRQuorum : Array[Node, Set[Ballot]], Ballot → Bool,
  haveWQuorum : Array[Node, Set[Ballot]], Ballot → Bool,
  haveQuorum : Array[Node, Set[Ballot]], Ballot → Bool
  haveNobody : Array[Node, Set[Ballot]], Ballot → Bool

asserts with
abstained, abs1, abs2 : Array[Node, Set[Ballot]], b_WQuorum, b_RQuorum,
b_DeadQuorum, b_Dead, b_NotMin : Ballot,
n_Quorum, n_rQuorum, n_wQuorum, n_Dead : Node,
voted : Array[Node, Set[Ballot]],
b_HaveWQuorum, b_qHaveWQuorum : Ballot,
n_HaveWQuorum : Node,
b_HaveRQuorum, b_qHaveRQuorum : Ballot,
n_HaveRQuorum : Node,
b_HaveQuorum, b_qHaveQuorum : Ballot,
n_HaveQuorum : Node,
a_HaveQuorum : Array[Node, Set[Ballot]],
a_HaveWQuorum : Array[Node, Set[Ballot]],
a_HaveRQuorum : Array[Node, Set[Ballot]],
b_Less, b_Greater : Ballot

  b_Less < b_Greater ⇔ (b_Less.seqno < b_Greater.seqno ∨
    (b_Less.seqno = b_Greater.seqno ∧ b_Less.procid < b_Less.procid));

  b_Less = b_Greater ⇔ (b_Less.seqno = b_Greater.seqno
    ∧ b_Less.procid = b_Greater.procid);

  ∀ b_RQuorum (∀ b_WQuorum (∃ n_Quorum : Node
    (n_Quorum in(rquorums(b_RQuorum) intersection wquorums(b_WQuorum)))));

  ∀ b_RQuorum  (∃ n_wQuorum : Node (n_wQuorum in (rquorums(b_RQuorum))));

  ∀ b_WQuorum (∃ n_rQuorum : Node (n_rQuorum in (wquorums(b_WQuorum))));

  b_Dead in dead (abstained)  ⇔ ∃ b_DeadQuorum  (∀ n_Dead : Node
     (n_Dead in rquorums(b_DeadQuorum) ⇒ b_Dead in abstained[n_Dead]));

  ∀ n_Dead : Node (abs1[n_Dead] subseteq abs2[n_Dead] ⇒ dead(abs1) subseteq dead(abs2));

  ∀ b_NotMin (b_NotMin ≠ minBallot ⇒ minBallot < b_NotMin);

  haveWQuorum (a_HaveWQuorum, b_HaveWQuorum) ⇔ ∃ b_qHaveWQuorum (∀ n_HaveWQuorum
    (n_HaveWQuorum in wquorums(b_qHaveWQuorum) ⇒ b_HaveWQuorum in a_HaveWQuorum[n_HaveWQuorum]));

  haveNobody (a_HaveQuorum, b_HaveQuorum) ⇔ (∀ n_HaveQuorum
  ¬ (b_HaveQuorum in a_HaveQuorum[n_HaveQuorum]));

haveRQuorum (a_HaveRQuorum, b_HaveRQuorum) ⇔ ∃ b_qHaveRQuorum (∀ n_HaveRQuorum
  (n_HaveRQuorum in rquorums(b_qHaveRQuorum) ⇒ b_HaveRQuorum in a_HaveRQuorum[n_HaveRQuorum]))
```

# C Proof Scripts

## C.1 Paxos to Global2

The following is the LP proof of the simulation relation from Paxos to Global2.

```
clear
thaw Paxos2Global2
forget

set name Z

decl vars z_s, z_s' : States[Paxos]
decl vars z_u, z_u' : States[Global2]
decl vars beta : ActionSeq[Global2]
decl vars v, vhack : Value

decl op sk_b : -> Ballot
decl op sk_bn : -> Ballot
decl op sk_i : -> Node

pr (F(z_s, z_u) /\ step (z_s, pi, z_s') /\ DistInv1(z_s)
/\ DistInv2(z_s) /\ DistInv3(z_s) /\ DistInv4(z_s) /\ DistInv5(z_s)
=> \E beta : ActionSeq[Global2] (execFrag(z_u, beta)
/\ F(z_s', last(z_u, beta)) /\ trace(beta) = trace(pi:Actions[Paxos])))
<>
  make immune con
  res by ind on pi : Actions[Paxos]

  <> init (n, v1)
    res by =>
    res by spec beta to init(n, v1) * {}
    % 3 requirements: failed/active; initiated; proposed
    res by /\
    <> for done
      res by cases z_sc.failed[n]
      <>
      []
      <>
        res by cases z_sc.mode[nc] = idle
        <>
          res by cases i = nc
        []
      []
    []

    <> for idle/active
      res by cases z_sc.failed[n]
      <>
      []
      <>
        res by cases i = nc
        <>
          res by cases z_sc.mode[nc] = idle
        []
        <>
          res by cases z_sc.mode[nc] = idle
        []
      []
    []
    <>
      res by cases z_sc.failed[n]
      <> Failed
        ex TacticPaxos2G2_1.lp
      []
      <> Not failed
        res by cases embed(v1) = embed(v)
        <> We're the one doing the inserting
          res by spec i to nc
        [] Someone else's value
        <>
          ex TacticPaxos2G2_1.lp
        []
      []
    [] res by /\
  [] init

  <> Fail (n)
    res by =>
    res by spec beta to fail(n) * {}
  []

  <> Decide (n, v1, b1)
    res by =>
    res by spec beta to decide (nc, v1c, b1c) * {}
    %     (z_sc.val[nc])[b1c] = z_uc.val[b1c]
    %       /\ \E i:Node (b1c \in z_sc.succeeded[i])
    %       /\ \A i:Node
  %              (z_sc.mode[i] = idle
  %                 <=> (if nc = i then done else z_sc.mode[i]) = idle)
  %        /\ \A i:Node
  %              (z_sc.mode[i] = done \/ i = nc
  %                 <=> (if nc = i then done else z_sc.mode[i]) = done)
  res by /\
  <> Level 5 subgoal for conjunct 1: (z_sc.val[nc])[b1c] = z_uc.val[b1c]
    % Value consistency
    pr (z_sc.val[nc])[b1c] ~= nil
    res by con
    inst i by nc, b by b1c in *Hyp
  []
  <>
    %      z_sc.mode[i] = idle
    %        <=> (if nc = i then done else z_sc.mode[i]) = idle
    res by cases nc = i
  []
  <>
    res by cases nc = i
  []
  <> Level 5 subgoal for conjunct 2: \E i:Node (b1c \in z_sc.succeeded[i])
    res by spec i to nc
  []
[]

<> enabled(z_s, newBallot(n))
  res by =>
  res by spec beta to {}
[]

<>     enabled(z_s, makeBallot(n, b1))
  res by =>
  res by spec beta to makeBallot (b1c) * {}
  res by /\
  <>           Current subgoal: ~(b1c \in z_sc.ballots[i])
    res by con
    inst b by b1c, i by ic in *Hyp
    % ZImpliesHyp.1.13.2: b1c \in z_sc.ballots[b1c.procid] -> true
    % Now this violates precondition of unique ballot
    inst b'_MakeBallot by b1c in *Hyp
  []
  <>
    %      b1c.seqno = b.seqno \/ \E i:Node (b \in z_sc.ballots[i])
    %        <=> \E i:Node
    %               (b
    %                  \in (if b1c.procid = i
    %                         then insert(b1c, z_sc.ballots[b1c.procid])
    %                         else z_sc.ballots[i]))

    res by cases b1c = b
    <>
      res by spec i to b1c.procid
    []
    <>
      res by <=>
      <>
        fix i as sk_i in *Hyp
        res by spec i to sk_i
        res by cases b1c.procid = sk_i
      []
      <>
        fix i as sk_i in *Hyp
        res by spec i to sk_i
        res by cases b1c.procid = sk_i
      []
    []

    <> Val
      set imm on
      res by cases b = b1c
      <>
      []
      <>
        res by cases b1c.procid = bc.procid
        <> Same
        []
        <> Different
          res by cases bc.seqno = b1c.seqno
          <> HACK
            ass nil = z_uc.val[bc]
        []
      []
      set imm off
    []
  [] /\
[] makeBallot

<>     enabled(z_s, abstain(n, s7))
  res by =>
  res by spec beta to abstain(nc, s7c) * {}
  % Just one clause, yay!
  res by cases nc = i
[]

<>     enabled(z_s, assignVal(n, b1, v1))
  res by =>
  res by spec beta to assignVal(b1c, v1c) * {}
  %          (\A b' (b'.seqno < b1c.seqno => b' \in dead(z_uc.abstained))
  %         \/ \E b''
  %               (z_uc.val[b''] = embed(v1c)
  %                  /\ \A bd'
  %                        (b''.seqno < bd'.seqno
  %                           => bd' \in dead(z_uc.abstained))))
  %        /\ \A b
  %              ((if b.seqno = b1c.seqno then embed(v1c) else z_uc.val[b])
  %                 = (if b1c.procid = b.procid
  %                       then assign(z_sc.val[b1c.procid], b1c, embed(v1c))
  %                       else z_sc.val[b.procid])
  %                     [b])
  %        /\ \E i:Node (b1c \in z_sc.ballots[i])
  %        /\ \E i:Node (v1c \in z_sc.proposed[i])
  res by /\
  <>
    res by cases \A b'_AssignVal (b'_AssignVal.seqno < b1c.seqno => b'_AssignVal \in dead(z_sc.abstained[b1c.procid]))
    <>
      pr \A b' (b'.seqno < b1c.seqno => b' \in dead(z_uc.abstained))
      res by =>
      inst b'_AssignVal by b'c in *Hyp
      set imm on
      pr \A i : Node ((z_sc.abstained[b1c.procid])[i] \subseteq z_uc.abstained[i])
      set imm off
      inst n_Dead by i, abs1 by z_sc.abstained[b1c.procid], abs2 by z_uc.abstained in Aux*
      res by cases dead(z_sc.abstained[b1c.procid]) = dead(z_uc.abstained)
      inst e by b'c, s1 by dead(z_sc.abstained[b1c.procid]), s2 by dead(z_uc.abstained) in Set
      inst e by b'c in Set
    []
    <>
      fix b''_AssignVal as sk_b in *Hyp

      pr \E b'' (z_uc.val[b''] = embed(v1c)/\ \A bd' (b''.seqno < bd'.seqno  => bd' \in dead(z_uc.abstained)))
      res by spec b'' to sk_b

      % Now show that (z_sc.val[b1c.procid])[sk_b] -> embed(v1c) = z_uc.val[sk_b]
      inst i by b1c.procid, b by sk_b in *Hyp
      % This removes first conjunct
      % Other goal:  sk_b.seqno < bd'.seqno => bd' \in dead(z_uc.abstained)

      res by =>
      inst bd'_AssignVal by bd'c in Z
      set imm on
      pr \A i : Node ((z_sc.abstained[b1c.procid])[i] \subseteq z_uc.abstained[i])
      set imm off
      inst n_Dead by i, abs1 by z_sc.abstained[b1c.procid], abs2 by z_uc.abstained in Aux*
      inst e by bd'c, s1 by dead(z_sc.abstained[b1c.procid]), s2 by dead(z_uc.abstained) in Set
      inst e by bd'c in Set
    []
    <>
      %    (if b.seqno = b1c.seqno then embed(v1c) else z_uc.val[b])
      %      = (if b1c.procid = b.procid
      %            then assign(z_sc.val[b1c.procid], b1c, embed(v1c))
      %            else z_sc.val[b.procid])
      %          [b]
      res by cases b.procid = b1c.procid
      <>
        res by cases bc.seqno = b1c.seqno
        <>
        []
        <>
          inst i by b1c.procid, b by bc in *Hyp
        []
      []
      <> Different proc IDs
      []
    []
    <> Level 5 subgoal for conjunct 3: \E i:Node (b1c \in z_sc.ballots[i])
      res by spec i to b1c.procid
    []
    <> Level 5 subgoal for conjunct 4: \E i:Node (v1c \in z_sc.proposed[i])
      res by spec i to b1c.procid
    []
  [] /\
[] assignVal

<> vote(n, b1)
  res by =>
  res by spec beta to vote(nc, b1c) * {}
  % Two conjunts
  res by /\
  <>        Level 5 subgoal for conjunct 1: \E i:Node (b1c \in z_sc.ballots[i])
    res by spec i to nc
  []
  <>
```

```
    %           (if nc = i
    %         then z_uc.voted[nc] \U insert(b1c, {})
    %         else z_uc.voted[i])
    %       = (if nc = i
    %             then assign(z_sc.voted[nc],
    %                         nc,
    %                         z_uc.voted[nc] \U insert(b1c, {}))
    %             else z_sc.voted[i])
    %           [i]
    res by cases i = nc
  []
[] vote

<>   enabled(z_s, internalDecide(n, b1))
  res by =>
  res by spec beta to internalDecide(b1c) * {}
  % 3 conjuncts
  res by /\
  <>
    %      ((b1c.seqno = b.seqno /\ b1c.procid = b.procid)
    %         \/ \E i:Node (b \in z_sc.succeeded[i])
    %         <=> \E i:Node
    %                (b
    %                   \in (if nc = i
    %                           then z_sc.succeeded[nc] \U insert(b1c, {})
    %                           else z_sc.succeeded[i])))
    res by cases             (b1c.seqno = b.seqno /\ b1c.procid = b.procid)
    % bc is the generic guy in the upper automaton
    <>
      res by spec i to nc
    []
    <>
      res by <=>
      <>
        fix i as sk_i in *Hyp
        res by spec i to sk_i
        res by cases nc = sk_i
      []
      <>
        fix i as sk_i in *Hyp
        res by spec i to sk_i
        res by cases nc = sk_i
      []
    []
  []
  <> \E b_qID \A j:Node (j \in wquorums(b_qID) => b1c \in z_uc.voted[j])
    % We have a quorum
    fix b_qID as sk_bn in *Hyp
    res by spec b_qID to sk_bn
    res by =>
    inst j by jc in Z
    res by cases z_uc.voted[jc] = (z_sc.voted[nc])[jc]
    inst i by jc, j by nc in *Hyp
    inst e by b1c, s1 by(z_sc.voted[nc])[jc], s2 by z_uc.voted[jc] in Set
    inst e by b1c in Set
  []
  <>         Level 5 subgoal for conjunct 3: \E i:Node (b1c \in z_sc.ballots[i])
    res by spec i to nc
  []
[] intl decide

<>   enabled(z_s, sendProposed(n, n1, v1))
  ex TacticPaxos2G2_emptyBeta
[]

<>   enabled(z_s, sendBallot(n, n1, b1))
  ex TacticPaxos2G2_emptyBeta
[]

<>   enabled(z_s, sendValue(n, n1, b1, v1)
  ex TacticPaxos2G2_emptyBeta
[]

<>   enabled(z_s, sendVote(n, n1, n2, b1))
  ex TacticPaxos2G2_emptyBeta
[]

<>   enabled(z_s, sendAbstained(n, n1, n2, s7))
  ex TacticPaxos2G2_emptyBeta
[]



<>   enabled(z_s, recvProposed(n, n1, v1))
  ex TacticPaxos2G2_emptyBeta
  %              \E i:Node (v \in z_sc.proposed[i])
  %            <=> \E i:Node
  %                   (v
  %                      \in (if ~(z_sc.mode[n1] = idle) /\ ~z_sc.failed[n1]
  %                               then assign(z_sc.proposed,
  %                                           n1,
  %                                           insert(v1c, z_sc.proposed[n1]))
  %                               else z_sc.proposed)
  %                             [i])
  res by <=>
  <>
    fix i as sk_i in *Hyp
    res by spec i to sk_i
    res by cases ~(z_sc.mode[n1c] = idle) /\ ~z_sc.failed[n1c]
    <>
      res by cases n1c = sk_i
    []
    <>
    []
  []
  <>
    set imm on
    res by cases vc = v1c
    <>
      set imm off
      res by spec i to nc
    []
    <>
      set imm off
      fix i as sk_i in *Hyp
      res by spec i to sk_i
      res by cases ~(z_sc.mode[n1c] = idle) /\ ~z_sc.failed[n1c]
      res by cases n1c = sk_i
      pr embed(v1c) ~= embed(vc)
      inst t by vc, t2 by v1c in Null
    []
  []
[] ugh

<>   enabled(z_s, recvBallot(n, n1, b1))
  ex TacticPaxos2G2_emptyBeta
  res by <=>
  <>
    fix i as sk_i in *Hyp
    res by spec i to sk_i

    res by cases ~(z_sc.mode[n1c] = idle) /\ ~z_sc.failed[n1c]
    <>
      res by cases n1c = sk_i
    []
    <>
    []
  []
  <>
    fix i as sk_i in *Hyp
    res by cases ~(z_sc.mode[n1c] = idle) /\ ~z_sc.failed[n1c]
    res by cases n1c = sk_i
    <>
      res by cases bc \in z_sc.ballots[sk_i]
      <>
        res by spec i to sk_i
      []
      <>
        res by spec i to nc
        res by con
        set imm on
        pr bc = b1c
      []
    []
    <>
      res by spec i to sk_i
    []
[]

<>   enabled(z_s, recvValue(n, n1, b1, v1))
  ex TacticPaxos2G2_emptyBeta
  res by cases ~(z_sc.mode[n1] = idle) /\ ~z_sc.failed[n1]
  set imm on
  res by cases b1c = b
  <>
    set imm off
    res by cases n1c = b1c.procid
    pr (z_sc.val[nc])[bc] ~= nil
    res by con
    inst b by bc, i by nc in *Hyp
  []
  <>
    set imm off
    res by cases n1c = bc.procid
    pr ~(bc.seqno = b1c.seqno /\ bc.procid = b1c.procid)
    inst b_Less by bc, b_Greater by b1c in Aux*
  []
[] recvValue

<>   enabled(z_s, recvVote(n, n1, n2, b1))
```

```
    ex TacticPaxos2G2_emptyBeta
    res by cases ~(z_sc.mode[n1] = idle) /\ ~z_sc.failed[n1]
    % Requirement: z_uc.voted[i] = z_sc.voted[i][i]
    % nc = sender, n1c = receiver, n2c = information about

    res by cases n1c ~= i
    % First case done
    % Now n1c = i, receiver data being changed
    res by cases n2c ~= ic
    % First case easy again
    % Now nc = ic, talking about info of sender

    pr b1c \in (z_sc.voted[ic])[ic]
    <>
      res by cases (z_sc.voted[nc])[ic] = z_uc.voted[ic]
      inst j by nc, i by ic in *Hyp
      inst e by b1c, s1 by (z_sc.voted[nc])[ic], s2 by z_uc.voted[ic] in Set
      inst e by b1c in Set
    []
    pr b1c \in z_uc.voted[ic]

    ass b \in s : Set[Ballot] => s =  insert(b, s)

    inst b by b1c, s by z_uc.voted[ic] in Z
  []

  <>   enabled(z_s, recvAbstained(n, n1, n2, s7))
    ex TacticPaxos2G2_emptyBeta

    res by cases ~(z_sc.mode[n1] = idle) /\ ~z_sc.failed[n1]
    % Requirement: z_uc.voted[i] = z_sc.voted[i][i]
    % nc = sender, n1c = receiver, n2c = information about

    res by cases n1c ~= i
    % First case done
    % Now n1c = i, receiver data being changed
    res by cases n2c ~= ic
    % First case easy again
    % Now nc = ic, talking about info of sender

    pr s7c \subseteq (z_sc.abstained[ic])[ic]
    <>
      ass s : Set[Ballot] \subseteq s1 : Set[Ballot] /\ s1 : Set[Ballot] \subseteq s2 : Set[Ballot] => s : Set[Ballot] \subseteq s2 : Set[Ballot]
     inst j by nc, i by ic in *Hyp
     inst s by s7c, s1 by (z_sc.abstained[nc])[ic], s2 by (z_sc.abstained[ic])[ic] in Z
     res by con
    []
    pr s7c \subseteq z_uc.abstained[ic]

    ass s1 : Set[Ballot] \subseteq s2 : Set[Ballot] => s1 : Set[Ballot] \U s2 = s2
    inst s1 by s7c, s2 by z_uc.abstained[ic] in Z

  []
[]

qed
```

## C.2 Global2 to Global2

The following is the LP proof of the simulation relation from Global2 to Global1. Notice that it is much simpler than the proof from Paxos to Global2 or from Global1 to Cons.

```
clear

thaw Global22Global1
forget

set name Z

decl vars z_s, z_s' : States[Global2]
decl vars z_u, z_u' : States[Global1]
decl vars beta : ActionSeq[Global1]
decl vars v, vhack : Value

decl op sk_b : -> Ballot
decl op sk_bn : -> Ballot
decl op sk_i : -> Node
decl op sk_z1 : -> States[Global1]
decl op sk_s1 : -> States[Global2]

decl op StartRel : States[Global2] -> States[Global1]

ass StartRel(z_s:States[Global2]) = [z_s.abstained, z_s.voted,
  z_s.val, z_s.succeeded, z_s.ballots, z_s.failed, z_s.decided,
  z_s.proposed, z_s. initiated]
```

```
pr start(z_s) => \E z_u (start(z_u) /\ F(z_s, z_u))
<> Start of proof
  make immune con
  res by =>
  res by spec z_u to StartRel(z_sc)
[]

pr (F(z_s, z_u) /\ step (z_s, pi, z_s') /\ Inv4(z_s) =>
  \E beta : ActionSeq[Global1] (execFrag(z_u, beta) /\
     F(z_s', last(z_u, beta)) /\ trace(beta) = trace(pi:Actions[Global2])))
make immune con
res by ind on pi : Actions[Global2]
<> Induction proof
  <> enabled(z_s, init(n, v))
    res by =>
    res by spec beta to init(n, v) * {}
  []

  <> enabled(z_s, fail(n))
    res by =>
    res by spec beta to fail(n) * {}
  []

  <> enabled(z_s, decide(n, v, b1))
    res by =>
    res by spec beta to decide (nc, vc, b1c) * {}
  []


  <> enabled(z_s, makeBallot(b1, s3))
    res by =>
    res by spec beta to makeBallot (b1c, quorums) * {}
  []


  <> enabled(z_s, abstain(n, s13))
    res by =>
    res by spec beta to abstain(nc, s13c) * {}
  []

  <> enabled(z_s, assignVal(b1, v))
    res by =>
    res by spec beta to assignVal(b1c, vc) * {}
    res by =>

    %  Level 5 subgoal for proof of =>:
    %      z_sc.val[b'c] = embed(vc)
    %        \/ \A j:Node (j \in quorums => b'c \in z_sc.abstained[j])

    inst b' by b'c in *Hyp
    res by cases \A j:Node (j \in quorums => b'c \in z_sc.abstained[j])

    <> First case easy
    []
    <> b'c isn't dead yet           remember, b'c < bc
      % Current subgoal: z_sc.val[b'c] = embed(vc)
      fix b'' as sk_b in *Hyp
      res by cases sk_b < b'c, sk_b = b'c, b'c < sk_b
      <> Easy case
        inst bd' by b'c, b' by b'c  in Z
        % Impossible
      []
      <> Easy case
      []
      <> Not so easy, use Inv4
        inst b by sk_b, b' by b'c in *Hyp
      []
    []
  [] assignVal

  <> enabled(z_s, vote(n, b1))
    res by =>
    res by spec beta to vote(nc, b1c) * {}
  []

  <> internalDecide(b1)
    res by =>
    res by spec beta to internalDecide(b1c) * {}
  []
[] End of induction
qed
```

## C.3 Global1 to Cons

This is the proof from Global1 to Cons. Although Global1 is a smaller program than Paxos, the proof is nearly as long as the simulation relation from Paxos to Global2.

```
clear
thaw Global12Cons
forget


set name Z

decl vars z_s, z_s' : States[Global1]
decl vars z_u, z_u' : States[Cons]
decl vars beta : ActionSeq[Cons]
decl vars v, vhack : Value

decl op sk_b : -> Ballot
decl op sk_bn : -> Ballot
decl op sk_i : -> Node


decl op StartRel : States[Global1] -> States[Cons]

ass StartRel(z_s:States[Global1]) = [{}, {}, {}, {}, {}] : States[Cons]
pr start(z_s) => \E z_u (start(z_u) /\ F(z_s, z_u))
<> Start of proof
  make immune con
  res by =>
  res by spec z_u to StartRel(z_sc)
[]

pr (F(z_s, z_u) /\ step (z_s, pi, z_s') /\
 Inv1(z_s) /\ Inv2(z_s) /\ Inv3(z_s) /\ Inv4(z_s) /\ Inv5(z_s) =>
\E beta : ActionSeq[Cons] (execFrag(z_u, beta) /\
   F(z_s', last(z_u, beta)) /\ trace(beta) = trace(pi:Actions[Global1])))

make immune con
res by ind on pi : Actions[Global1]
<> Induction proof
  <> enabled(z_s, init(n, v1))
    res by =>
    res by spec beta to init(n, v1) * {}
  []

  <> enabled(z_s, fail(n))
    res by =>
    res by spec beta to fail(n) * {}

  []


  <> enabled(z_s, decide(n, v1, b1))
    res by =>
    res by spec beta to decide (nc, v1c) * {}
    inst b by b1c, v by v1c in *Hyp
  []


  <> enabled(z_s, makeBallot(b1))
    res by =>
    res by spec beta to {}
    res by /\
    <>
      res by =>
      inst v by vc in *Hyp
      fix b as sk_b in con-op(vc)

      res by spec b to sk_b
      res by cases b1c = sk_b
      inst b by sk_b in *Hyp
    []

    <>
      res by =>
      fix b as sk_b in con-op(b1c)
      res by cases sk_b = b1c
      <> First case easy, impossible case
      []
      <> Second case, sk_b ~= b1c
        inst b by sk_b, v by vc in *Hyp
      [] Booyeah!
    []
  []


  <> enabled(z_s, abstain(n, s13))
    res by =>
    res by spec beta to {}
  [] ayup

  <> enabled(z_s, assignVal(b1, v1))
    res by =>

    res by cases \A b : Ballot (b \in z_sc.succeeded => z_sc.val[b] = nil)
    <> True case, all vals are nil
    res by cases ~(b1c \in z_sc.succeeded)
    <> ~(b1c  \in z_sc.succeeded)
      res by spec beta to {}
      res by /\
      <>
        ex TacticG2C_1
      []
      <>
        ex TacticG2C_2
      []
    []

    <> True.  New assigned ballot is in succeeded and nothing else has succeeded
      res by spec beta to chooseVal(v1c) * {}

      res by /\
      <> empty(z_uc.chosen)
        ex TacticG2C_5
      []
      <> v in z_sc.chosen => \E...
        res by =>
        res by spec b to b1c
      []
      <> \E.... => v in z_sc.chosen
        ex TacticG2C_4
      [] \E...
    [] b1c \in z_sc.succeeded
  [] true case for all ballots being nil

  <> false case, there  are some ballots not nil
    % Use precondition on < operator

    pr \E b : Ballot (b \in z_sc.succeeded /\ ~(z_sc.val[b] = nil))
    <> Some preliminaries
      set name temp
      pr  \E b ~(b \in z_sc.succeeded => z_sc.val[b] = nil)
      make immune con
      res by con
      fix b as sk_b in temp*

      res by spec b to sk_b
      res by cases z_sc.val[sk_b] = nil
    []

    fix b as sk_bn in Z
    % sk_bn is one of the ballots that aren't null

    res by spec beta to {}
    res by /\
    <>
      ex TacticG2C_1
    []

    <>
      res by =>
      fix b as sk_b in con-op(vc)
      res by cases b1c = sk_b

      % vc = value of inserted element b1c
      <> True case, show using < operator

        res by cases sk_b < sk_bn, sk_b = sk_bn, sk_bn < sk_b
        <> sk_b < sk_bn
          inst b' by sk_b, b by sk_bn in *Hyp
          inst b by sk_b in *Hyp
        [] Apparently impossible case, because of invariant
        <> sk_b = sk_bn
        []
        <> sk_bn < sk_b     Level 8
          % Something in ballots is < current insertion
          pr ~ (sk_bn \in dead(z_sc.abstained))
          <>
            make immune con
            inst b by sk_bn in *Hyp
          []
          inst b by sk_bn, b' by sk_bn, v by vc in *Hyp
        []
      [] b1c = sk_b
      <> b1c ~= sk_b
        inst b by sk_b, v by vc in *Hyp
        % Ppeviously in succeeded
      []
    [] sk_b ~= b1c
  []
[] assignVal



<> vote enabled(z_s, vote(n, b1))
  res by =>
  res by spec beta to {}
[] ayup


<> enabled(z_s, internalDecide(b1))
  res by =>
  res by cases b1c \in z_sc.succeeded
  <> Easy case
    res by spec beta to {}
    res by /\
    <>
      ex TacticG2C_3
    []
    <>
      ex TacticG2C_4

    [] Easy case
    <> b1c is actually new
    res by cases z_sc.val[b1c] = nil
    <> Another easy case - deciding on a null ballot
      res by spec beta to {}
      res by /\
      <>
        ex TacticG2C_3
      []
      <>
        ex TacticG2C_4
      []
    []
  []
  <> b1c has a value.  Now we're talking
    res by cases \E b : Ballot (b \in z_sc.succeeded /\ z_sc.val[b] ~= nil)
    <> True case, there are existing succeeded ballots with value
      fix b as sk_bn in *CaseHyp
      res by spec beta to {}
      res by /\
      <>
        ex TacticG2C_3
      []
      <>
        % Remember, goal is vc \in uc.chosen, where val[b = sk_b]] = vc
        res by =>
        fix b as sk_b in con-op(vc)
        res by cases b1c = sk_b
        % This case resumption is not necessary.  There exists sk_b
        % that already holds what we want.
        <> True case, show using < operator

          res by cases sk_b < sk_bn, sk_b = sk_bn, sk_bn < sk_b
           <> sk_b < sk_bn
             pr z_sc.val[sk_bn] = z_sc.val[sk_b]
             inst b' by sk_b, b by sk_bn in *Hyp

             % embed(vc) = z_sc.val[sk_bn]
             %       \/ sk_b \in dead(z_sc.abstained)
             %       -> true

             pr ~(sk_b \in dead(z_sc.abstained))
             make immune con
             rewrite con
             res by con

             decl op sk_i : Ballot, Ballot -> Node
             fix n_Quorum as sk_i(b_RQuorum, b_WQuorum) in Aux*

             decl op sk_bq : -> Ballot
             fix b_qID as sk_bq in *Hyp

             inst j by sk_i(b_DeadQuorumc, sk_bq), b_WQuorum by sk_bq in Z
             inst n_Dead by sk_i(b_DeadQuorumc, sk_bq) in *Hyp

             crit *Hyp with Z, *Hyp

            []
            inst b' by sk_b, b by sk_bn in *Hyp
            inst b by sk_b, v by vc in *Hyp
          [] sk_b < sk_bn
          <> sk_b = sk_bn
            % Impossible
          []
          <> sk_bn < sk_b
            inst b' by sk_bn, b by sk_b in *Hyp
            inst v by vc, b by sk_bn in *Hyp
          []
        [] sk_b = b1c
        <> sk_b ~= b1c
          inst b by sk_b, v by vc in *Hyp
        []
      []
    [] b1c is new
  [] ballot already chosen
```

```
    <> Ballot not yet chosen
      res by spec beta to chooseVal(z_sc.val[b1c].val) * {}
      res by /\
      <> z_sc.val[b1c].val \in z_sc.proposed
        inst b by b1c in *Hyp
      []
      <> empty(z_uc.chosen)
        ex TacticG2C_5
      []
      <> \A v (v \in z_uc.chosen)...
        res by =>
        res by spec b to b1c
      []
      <> \E....
        % v1c = newly assigned value
        % vc = value on right side of what we have to prove
        res by =>
        fix b as sk_b in *Hyp.2

        inst b by sk_b in *Hyp
      [] \E...
    [] new value chosen
  [] internalDecide
[] End of induction
qed
```

## References

[1] Paul Attie. Liveness preserving simulation relations. *PODC*, 1998.

[2] Andrej Bogdanov. Formal verification of simulations between I/O automata. Master of engineering thesis, September 2000.

[3] *The Daikon Invariant Detector User Manual*, December 7, 2001.

[4] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, (2):374–382, 1985.

[5] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation, Systems Research Center, 31 December 1991.

[6] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA: A language for specifying, programming, and validating distributed systems. Technical report, MIT Laboratory for Computer Science, 1997.

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

[8] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, pages 133–169, May 1998.

[9] Leslie Lamport. Paxos made simple. *to appear in SIGACT News*, Nov 2001.

[10] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, San Francisco, CA, 1996.

[11] Nancy Lynch and Alex Shvartsman. Paxos made even simpler (and formal).

[12] Nancy Lynch and Frits Vandraager. Forward and backward simulations, parts i and ii. http://theory.lcs.mit.edu/tds/papers/Lynch, October 1994.

[13] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[14] Roberto Segala, Rainer Gawlick, Jorgen Sogaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, 1998.

[15] Toh Ne Win and Michael Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. http://theory.lcs.mit.edu/tds/papers/Tohn, May 2002.