Verification of a Generic Consensus Protocol with PROMELA/SPIN—an Exercise in Specification and Verification

Jiang Liu

March 1, 2001

Abstract

In this study, a generic consensus protocol proposed by Mostefaoui and Raynal is verified with PROMELA/SPIN—a popular specification and model checking suite. The correctness of the model and the protocol are summarized in six properties, which are expressed either in Linear Temporal Logic formulae or using assert conditions. Due to the shortage of the computational resources, the verification is partially completed without finding any error. The PROMELA language is found to be expressive and well suited for specifying asynchronous systems; the SPIN model checker appears more capable in finding an error than in fully verifying a system.

1 Introduction

Automated verification with formal methods is the only reliable way to truly ascertain the correctness of an algorithm. It has been gaining popularity as the theory advances and the computational resources become increasingly available [2]. PROMELA/SPIN is one popular model checking software suite that allows simple and intuitive specification of distributed systems and performs model checking. It provides easy and intuitive syntax for message passing, asynchronous process execution, and Linear Temporal Logic (LTL) [5, 4]. In this report, the author uses this software package to verify a distributed computation protocol for solving a consensus problem.

Consensus problem is one of the most fundamental problems facing designers of reliable applications on top of an unreliable asynchronous system. In a typical setting, several processes, any one of which may crash, each propose a value. They can pass messages to each other through a reliable channel. Additionally, each process may suspect the correctness of any other processes through the action of a failure detector. Eventually, all the correct processes (i.e. noncrashed) are required to terminate and decide upon the same value chosen from the originally proposed set. Many important practical problems such as atomic broadcast, atomic multicast, etc. can be reduced to consensus [6, 1]. This problem becomes particularly difficult when the failure detectors are allowed to become unreliable which is usually the case in practice. Mostefaoui and Raynal [6] propose a generic quorum-based algorithm to solve a slightly restricted class of consensus problem with unreliable failure detectors—the Uniform Consensus Problem, which satisfies the following three properties:

- ◇ Termination: Every correct process eventually decides on some value.
- ◇ Validity: If a process decides on v, then v was proposed by some process.
- ◇ Uniform Agreement: No two processes (correct or not) decide differently.

In Chandra and Toueg's seminal paper [1], the unreliable failure detectors are classified into two classes— S and $\diamond S$. Both classes satisfy the same two properties: (a) "Eventually, every crashed process is suspected by every correct process" (Strong Completeness) and (b) "There is a correct process that is never suspected" (Weak Accuracy). However, the two classes differ in the way that (b) is satisfied perpetually in class S and eventually in class $\diamond S$ [1]. The failure detectors are unreliable because they can suspect a correct process or not suspect a failed process.

In this study, several properties pertaining to the liveness and safety of the protocol algorithm are checked with SPIN model checker for S class unreliable detectors. The $\diamond S$ class detectors may take arbitrarily many rounds for the protocol to decide [6]. Consequently, its verification is impractical and not attempted.

This report is organized as follows. Section 2 illustrates the consensus protocol with a simple example. Section 3 details the specification of the environment and system as well as the properties to be verified. In section 4, the results of the verification are presented and the modeling experience examined. Section 5 recapitulates the important findings of the report. Lastly, all the models and verification results can be found in the appendices.

2 Protocol

In order to understand the protocol, the environment must be introduced first. Generally, an asynchronous distributed system consists of a finite set of n, such that $n > 1$, processes, each of which behaves correctly according to the consensus protocol until it crashes—these processes are henceforth referred to as consensus processes. The original consensus protocol is given in Figure 1. The maximum number (f) of the processes that can crash must be less than n. Further, every pair of processes is connected by a channel through which they communicate and synchronize. The channels are not required to be first-in-first-out (FIFO); but they are reliable in the sense that they "do not create, alter, or lose message" [6]. In this protocol, two types of messages are exchanged: *EST*, which is the estimated consensus value, and *DECIDE*, which is the value decided upon by one of the processes.

The failure detectors belonging to class S satisfy two properties:

1. Strong Completeness: Eventually, every process that crashes is permanently suspected by every correct process.

2. Perpetual Weak Accuracy: Some correct process is never suspected.

The failure detector consists of a set of n modules, one attached to each process. Each module maintains a set of processes it suspects of having crashed. Any detector module is inherently unreliable and can suspect a correct process or fail to detect a crashed process for a finite length of time. Suspicions are not necessarily stable either: a process can be added or removed from the suspected list.

Each consensus process starts by proposing an arbitrary value (v) other than a default value ($\perp$) and passes it to the consensus protocol. The protocol consists of two parallel tasks (*T1* and *T2*). The completion of either task successfully terminates the protocol. *T2* (line 16) simply waits for the *DECIDE* value to arrive, terminates the protocol, and returns the *DECIDE*

```
Function Consensus(v_i)
cobegin
 (1) task T1: r_i ← 0; est_i ← v_i; % v_i ≠ ⊥ %
 (2)  while true do
 (3)    c ← (r_i mod n) + 1; est_from_c_i ← ⊥; r_i ← r_i + 1; % round r = r_i %
 (4)    case (i = c) then est_from_c_i ← est_i
 (5)         (i ≠ c) then wait ((EST(r_i, v) is received from p_c)∨(c ∈ suspected_i));
 (6)                   if (EST(r_i, v) has been received) then est_from_c_i ← v
 (7)    endcase;  % est_from_c_i = est_c or ⊥ %
 (8)    ∀j do send EST(r_i, est_from_c_i) to p_j enddo;
 (9)    wait until (∀p_j ∈ Q_i: EST(r_i, est_from_c) has been received from p_j);
                  % Q_i has to be a live and safe quorum %
                  % For  S:   Q_i is such that Q_i ∪ suspected_i = Π %
                  % For ◇S:   Q_i is such that |Q_i| = ⌈(n + 1)/2⌉ %
 (10)   let rec_i = {est_from_c | EST(r_i, est_from_c) is received at line 5 or 9};
                  % est_from_c = ⊥ or v with v = est_c %
                  % rec_i = {⊥} or {v} or {v, ⊥} %
 (11)   case (rec_i = {⊥}) then skip
 (12)        (rec_i = {v}) then ∀j ≠ i do send DECIDE(v) to p_j enddo; return(v)
 (13)        (rec_i = {v, ⊥}) then est_i ← v
 (14)   endcase
 (15) enddo

 (16) task T2: upon reception of DECIDE(v):
                   ∀j ≠ i do send DECIDE(v) to p_j enddo; return(v)
coend
```

Figure 1: The consensus protocol. Taken from A Mostefaoui and M Raynal [6].

value to the parent process. *T1* (lines 1–15) proceeds in rounds(r) and terminates whenever all the processes in a dynamically defined quorum agree on the same value, which is subsequently returned to its parent process. A detailed explanation can be found in Mostefaoui and Raynal [6]. Here the protocol is illustrated with a simple example.

Assume four processes attempt to reach a consensus. Number the processes 1 through 4 and denote them *P1* through *P4* accordingly. All processes initailize in round 1 (lines 1 and 3) and elects a coordinator by round robin (e.g. r *mod* 4). If *P1* is a correct process, it broadcasts its estimate(est, initially assigned value v) to all other processes (line 8). Meanwhile, *P2* through *P4* wait for the arrival of EST from *P1* (line 5). If none of them suspects *P1*, they all recieve EST from *P1* and broadcast the value to all the other processes. By now, each of the processes is waiting for EST messages to complete its quorum (line 9). The quorum consists of at least all the processes the process does not suspect. Once one of processes receives enough messages, it examines the set of received values and finds, in this case, them equal to v. It broadcasts v, exits, and returns v (line 12). All the other processes, upon reception of the *DECIDE* message, also exit and return the same value. It is clear that during any round, as long as the coordinator does not crash nor is it suspected erroneously by some others, all correct processes will exit successfully regardless of how many processes crash. However, if the coordinator, say *P1*, crashes, all of the remaining correct processes will broadcast $\perp$. Subsequently, they start a new round (line 11) and elect a new coordinator and repeat the procedure anew. In yet another scenario, *P1* is correct but erroneously suspected by some processes, say *P3*. *P3* will broadcast $\perp$ while the remaining correct processes broadcast EST they have received from *P1*. Then some processes may receive both EST and $\perp$. These processes then take the value v as their new estimate and start a new round. A formidable number of possible scenarios exist. Nonetheless, it may be observed that no process blocks perpetually while waiting for EST messages (lines 5 and 9) because of the eventual accuracy property of its failure detector.

3 Model

The model is specified in the PROMELA language. The system and the environment are carefully identified first. Then some important variables are defined. Finally, the models are given and discussed in detail. The full source code for the PROMELA models can be found in Appendices A and B.

3.1 The System and the Environment

In the model, the system is a collection of n processes ($n = 4$ in this case), correct or failed, executing the consensus protocol. The environment is the failure detector as well as the state variables that determine the combination of correct and crashed processes. The system, being the algorithm to be verified, has a well-defined mechanism. The environment is specified with as little mechanism as possible. It is mainly a stochastic process that performs random events appropriate at the moment. However, care is taken to minimize the number of choices necessary at each step to reduce the verification requirement. Ideally, the completeness and accuracy of the failure detector can be achieved with fairness conditions. In the absence of SPIN support for strong fairness (i.e. the ability to allow specification of fairness in some logic) [3], these properties have to be reflected in the specification of the environment.

The presiding modeling philosophy is to impose as few restrictions on the environment as allowed by the limited computational resources. Inevitably, sacrifices must be made in generality to improve efficiency. Further, the PROMELA language, though rich in syntactical varieties, can not possibly be tantamount to human language in expressiveness. These limitations are all reflected in modeling process.

3.2 Some Global State Variables and Message Channels

A few variables and message channels are defined globally so that they are visible to all processes and it is possible to formulate LTL expressions based on them. The processes are enumerated *0* through $n - 1$.

- *Proposed*: An array of 4 byte variables that contain the proposed value of each process. They can theoretically be any number or text message; here they are limited to a value randomly drawn from the set {10, 11, 12, 13}. These choices are sufficient to allow uniqueness of the proposed value by each of the four consensus processes.

- ◇ *Failure*: An array of 4 boolean variables that reflects the true state of each consensus process. While these flags are not required by the consensus processes themselves, they are used by the other processes to implement fairness conditions and to decide when to terminate[1].

- ◇ *Crash_Count*: A byte variable that counts the number of processes which have crashed. It is used to enforce the property that at least one process remains correct at all times.

- ◇ *Decided*: An array of 4 byte variables that serves the dual duty of flagging whether a process has decided and containing the decided value if it has.

- ◇ *Suspected*: An array of 16 boolean flags that emulates a four by four two dimensional array. Each module of the error detector uses a distinct set of four variables to store its current suspects. Each consensus process also negates the set allocated to itself to form the smallest quorum it is required to complete.

- ◇ *round*: An array of 4 byte variables that contain the designation of the round current to each process.

- ◇ *Msg*: An array of 4 message channels, each is associated with one receiving process and of a queue length of 17. The format of message is

  $$Type, From, Round, Value$$

  where *Type* can be *Dec* (i.e. $DECIDE$) or *Est*; *From* is the designation number of the sender, *Round* is the round of the sender at the time of transmission, and *Value* the actual transmitted message. In the case of decide message (*Dec*), the middle two arguments are ignored at the receiving end. The length of the queue is estimated by a theorem that a decision will be made in n rounds for S class failure detectors [6]. In the worst case scenario, a process suspects all the other processes all the time, is suspected by all other processes while it is the coordinator, and crashes by the end of the fourth round while a decision is made by another process, while no other process crashes. In the channel for this

[1] All the rest of the processes are written so that they terminate when all of the four consensus processes have terminated, either by failure or decision. Therefore, successful termination of all processes is a liveness property

process, there are three messages from each undecided round (because it won't suspect itself and will always receive its own message) and seven messages in the last round (three *Est* and three *Dec* from the other processes and one *Est* from itself) —totaling 16 messages. One extra space is also allocated so that the safety of the channel can be simply expressed as "The channel is never full". This claim is later verified as one of the properties (see Section 4.1, Safe 3).

3.3 The Failure Detector

The failure detector consists of a loop that only exits after all the processes have either crashed or decided. It does so by checking at the start of each loop whether the following condition is satisfied[2].

```
231>  (Decided[0] || Failure[0]) && (Decided[1] || Failure[1]) &&
        (Decided[2] || Failure[2]) && (Decided[3] || Failure[3])
        -> break;
```

Due to the inability of SPIN to allow specification of fairness conditions in temporal logic, the design of the failure detector has to incorporate all its completeness and accuracy properties. Although they are easily stated in LTL, the implementation in form of automata is less direct (see Figure 2).

In one complete loop, the set of *Suspected* flags for one process is first selected randomly, at most one flag of which is allowed to change. A reasonable assumption is made here that a correct process never suspects itself. Then, the flag for one correct process is chosen at random from the set, and assigned false; this ensures the Perpetual Accuracy requirement.

```
234> if
235>   :: Suspected[1] -> Suspected[1]=false;
236>   :: Suspected[2] -> Suspected[2]=false;
237>   :: Suspected[3] -> Suspected[3]=false;
238>   :: !Suspected[1] -> Suspected[1]=true;
239>   :: !Suspected[2] -> Suspected[2]=true;
240>   :: !Suspected[3] -> Suspected[3]=true;
241>   :: skip;
242> fi;
```

[2] The numbers correspond to the line numbers of source code for the original model (*c.f.* Appendix A). Since two versions of source codes exist, only the source codes taken from the original model are numbered to avoid confusion.

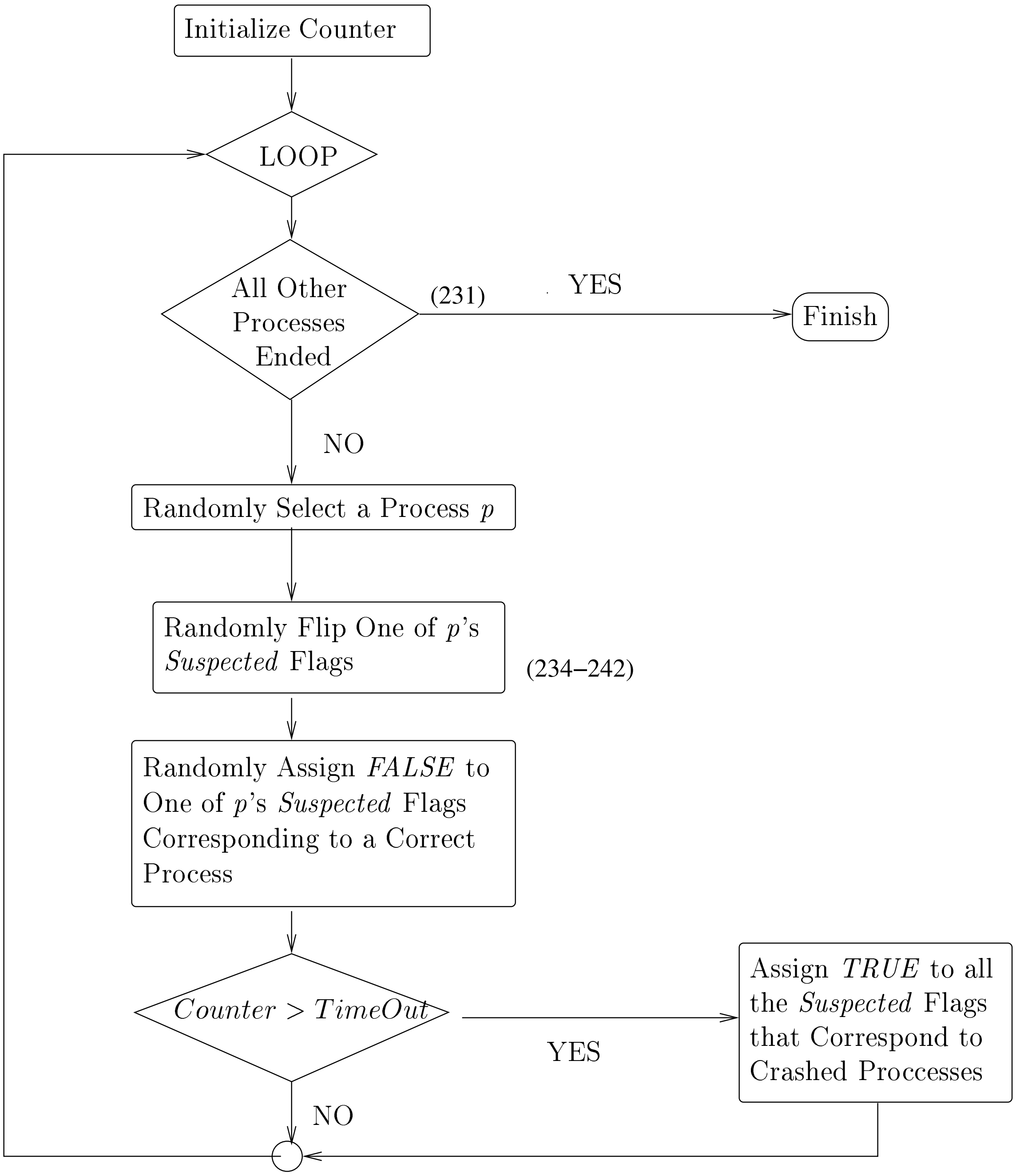

Figure 2: The failure detector. The numbers correspond to line numbers in the original source code (*c.f.* Appendix A). For ease of presentation, the figure only contains line numbers of the part of the source codes that are mentioned in this report.

The next step approximates the Strong Completeness. If the detector has completed more than *TimeOut*—a predefined value[3]—cycles, all the *Suspected* flags corresponding to all the crashed processes are set true. This mechanism approximates the time-out decision mechanism commonly used in practical applications. Of course, it will fail in a full search because it is possible that the *Detector* process does not get executed at all or not often enough to reach time out. One can circumvent this problem by defining a global counter that is incremented every time some process is run. Alternatively, the weak fairness option in SPIN can be dependent upon to enforce the frequent execution of all the processes.

All the three steps described above are executed in one atomic step to ensure that the list of suspects always satisfies the completeness and accuracy properties when they are visible to the consensus processes.

3.4 The Process with Consensus Protocol

A process executing the consensus protocol should theoretically consist of three parallel tasks. Two are required by the consensus protocol and one more to decide when the process should crash. The author has chosen one of the safety requirement to be that valid end states are always attained (i.e. all processes should end properly). The original construction, though intuitive, is difficult to realize because one of the processes must terminate the other two before any more messages (especially of *Dec* type) are broadcast by them. To the best of the author's knowledge, PROMELA does not offer a direct mechanism for one process to terminate another[4]. Of course, message passing can be used to this end; but significant amount of code has to be developed to enforce synchronization. Moreover, the search depth required in the verification will be much steeper and all the computational requirements increase accordingly.

In the author's specification, the three processes are merged into one. However, instead of a simple atomic step by atomic step interleaving, the randomized interleaving is only enabled at a few points. The validity of this

[3]The value of *TimeOut* is set to 16 rather arbitrarily. But this number is sufficient to allow all processes to suspect all other processes without the enforcement of the Strong Completeness.

[4]It is also unnecessary because in practice, one process has to communicate with another through a message channel. There does not yet exist any magical telepathic exchange of information.

implementation is based upon two observations. Firstly, only the state of the consensus processes matters in evaluating the protocol. Secondly, the state of one consensus process does not affect that of the others as long as it does not send any message to those processes[5]. The consequence of these facts is that a consensus process only needs to determine whether it has crashed before sending any message to the others. In other words, a consensus process only needs to be given a chance to crash before any writing to the message channels. The same argument applied to the response to *Dec* message concludes that a consensus process only needs to respond to a *Dec* message before it is deciding or while it is waiting for a message, either from the round coordinator or to complete a quorum (Figure 3).

The process itself is now examined in some detail in contrast to the original consensus protocol. The initialization and coordinator election are similar to the originals. One addition is that at the start of each loop, the *est_from_c* values are initialized to zero—a default value that also serves as a boolean flag for the reception of messages.

The wait for the arrival of *Est* message from the coordinator is implemented with a loop to allow the reception of *Dec* messages. Note that before receiving any message, the process always tests to ensure that there is indeed a message in the channel to receive. Otherwise, the statement will block until such a message does arrive even though there may be another type of receivable message arriving earlier.

```
do
 :: Msg[i]??[Dec(_, _, _)] ->
      Msg[i]??Dec(_, _, v); goto BROADCAST;
      /* Upon receiving Decide, broadcast it */
  :: else -> if
        :: Suspected[i*NoP+c] -> break;
           /* cur coord is suspected */
        :: Msg[i]??[Est, eval(c), eval(round[i]), _] ->
           Msg[i]??Est, eval(c), eval(round[i]), v;
             est_from_c[i]=v; break;
             /* received est from cur coord */
        :: else -> skip;
        fi;
od;
```

The loop breaks whenever a *Dec* message is received; *Dec* is given a higher priority to ensure that the message will be received. If one allows probabilistic

[5]This statement is not strictly true because if a process crashes, the effect will be felt eventually through the act of the failure detectors. However, a process ceases to send any more messages once it is crashed. Hence the argument that follows is still valid.

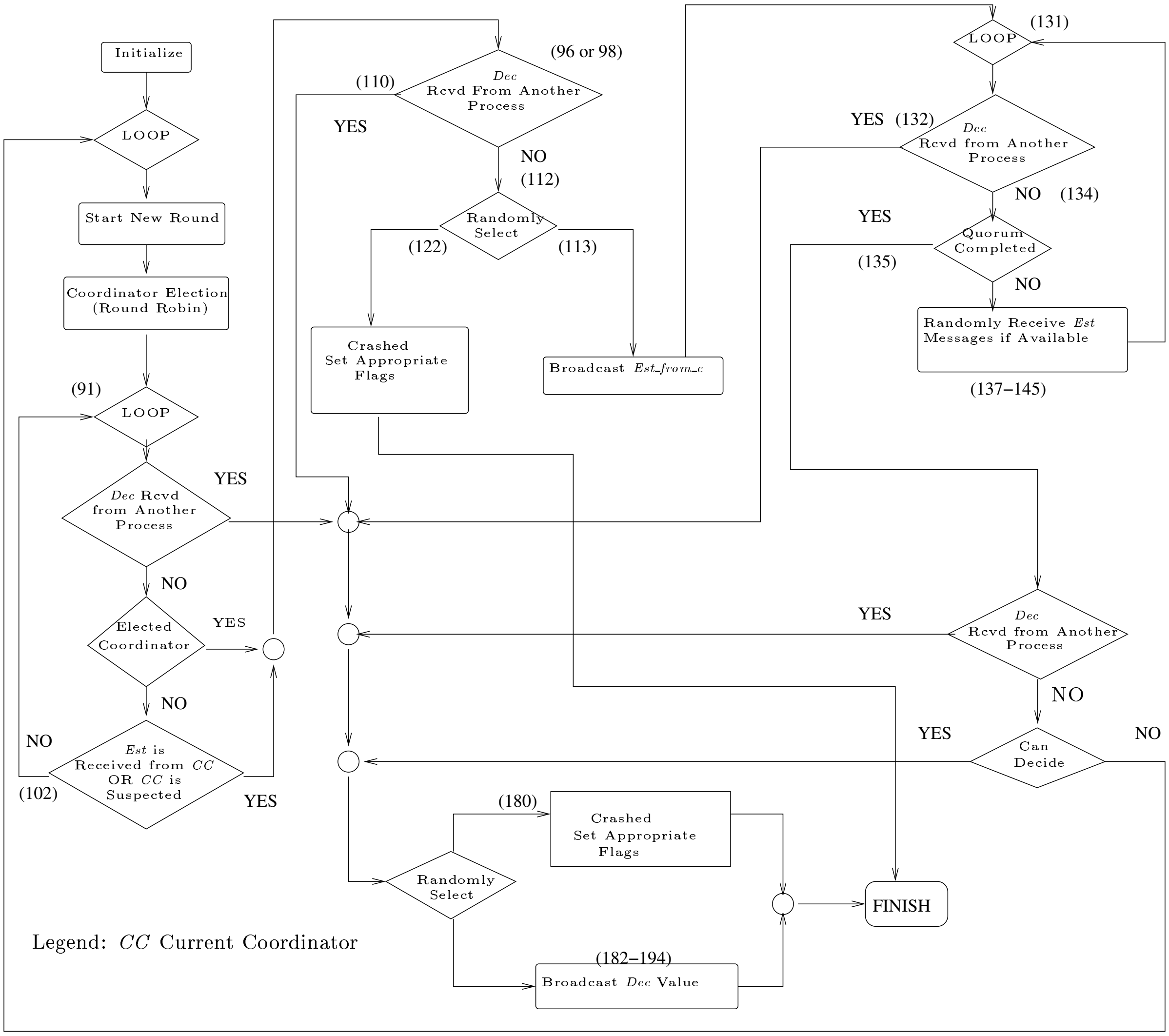

Figure 3: The interleaved consensus process. The numbers correspond to line numbers in original source code (*c.f.* Appendix A). For ease of presentation, the figure only contains line numbers of the part of the source codes that are mentioned in this report.

reception of *Dec* language, occasions may arise in which the *Dec* message is never received—a condition which violate the initial assumption about the communication channels.

This loop-based implementation is not efficient in the sense that between any two meaningful actions, SPIN can always insert an infinite number of skip loops (i.e. stutter actions). The weak fairness condition, which has to be enabled to keep the *Suspected* list up to date, reduces such instances. An alternative specification is made to improve the efficiency of the search.

```
(Msg[i]??[Decide(_, _, _)] || Suspected[i*NoP+c] ||
Msg[i]??[Est, eval(c), eval(round[i]), _]) -> if
    :: Msg[i]??[Decide(_, _, _)] ->
          Msg[i]??Decide(_, _, v); goto BROADCAST;
          /* Upon receiving Decideide, broadcast it */
    :: else -> if
          :: Suspected[i*NoP+c] -> skip;
             /* cur coord is suspected */
          :: Msg[i]??[Est, eval(c), eval(round[i]), _] ->
             Msg[i]??Est, eval(c), eval(round[i]), v;
             est_from_c[i]=v; skip;
             /* received est from cur coord */
          fi;
     fi;
```

The first statement blocks unless at least one of the three situations, which follow it, is executable. Effectively, the loop has been replaced by an if statement. This specification achieves the same objective that the process is allowed to perform any one of the three tasks in any order dictated by the sequence of arrival of the messages, while at the same time prevents the stutter actions from occurring; thus it can potentially lessen the time and computational resources required for verification.

The subsequent step in the protocol requires the process to broadcast its own *est_from_c* value. By the argument given at the beginning of the section, the process is also given the choice here to receive a *Dec* message or to crash. It is accomplished by these lines of codes.

```
if
   :: Msg[i]??[Dec(_,_,_)] ->
        Msg[i]??Dec(_, _, v); goto BROADCAST;
   :: else -> if
        :: true ->atomic {
                  Msg[0]!Est(i, round[i], est_from_c[i]);
                  Msg[1]!Est(i, round[i], est_from_c[i]);
                  Msg[2]!Est(i, round[i], est_from_c[i]);
                  Msg[3]!Est(i, round[i], est_from_c[i]);
                 };
           /* process may crash if less than 3 have crashed */
           /* Note that the crash_count var is updated as soon*/
           /* as the system decides to crash to ensure S class. */
        :: atomic {Crash_Count<3 -> Crash_Count++;Failure[i]=true;};
           goto CRASHED;
        fi;
fi;
```

It is notable that the process is only allowed to crash when there are less than 3 processes which have already crashed—the minimal condition required for the decidability of class S failure detectors [6]. Further, updating *Crash_Count* and *Failure* flag is performed in the same atomic step so that no other process may decide to crash before the appropriate global states are set. Moreover, because the latter two statements are always executable, so is the atomic step as long as the first condition is satisfied. Writing to all four channels is also done in one atomic step because it is sufficient to allow different sequences of event execution performed on the receiving end of a communication channel.

The third place of interest is where the quorum has to be completed. The protocol demands that for class S failure detectors, the quorum must be such that the union of the quorum and the *Suspected* set must give the complete set. Of course, if this condition is not satisfied, the process has to be allowed to keep on receiving *est_from_c* messages until the condition is satisfied. Moreover, the quorum must be live and safe [6]. Furthermore, it is desirable to allow the process to receive *Dec* messages. All these are accomplished with a loop.

```
do
   :: Msg[i]??[Dec(_,_,_)] ->
      Msg[i]??Dec(_, _, v); goto BROADCAST;
 :: else -> if
        :: ((Suspected[i*NoP+0] || (est_from_c[0]>0)) && (Suspected[i*NoP+1] ||
           (est_from_c[1]>0)) && (Suspected[i*NoP+2] || (est_from_c[2]>0)) &&
           (Suspected[i*NoP+3] || (est_from_c[3]>0))) -> break;
        :: else -> if
          :: Msg[i]??[Est, 0, eval(round[i]), _]->
               Msg[i]??Est, 0, eval(round[i]), est_from_c[0];
          :: Msg[i]??[Est, 1, eval(round[i]), _]->
               Msg[i]??Est, 1, eval(round[i]), est_from_c[1];
          :: Msg[i]??[Est, 2, eval(round[i]), _]->
               Msg[i]??Est, 2, eval(round[i]), est_from_c[2];
          :: Msg[i]??[Est, 3, eval(round[i]), _]->
             Msg[i]??Est, 3, eval(round[i]), est_from_c[3];
          :: else -> skip;
          fi
        fi
od;
```

The quorum is defined as any set that contains the complementary set of the *Suspected* set. At the start of each round, all the *est_from_c* variables are initialized to zero. In addition, all the possible values transmitted, which includes the default value (255), are positive. Hence, the *est_from_c* variables themselves also serve as a flag for whether a value has been received (i.e. assigned). In the specification, the loop breaks if each process is either suspected or heard from. Otherwise, of all the available messages in the channel, a random one is received. This construction allows the process to receive a message from a suspected process. Because no mechanism is assumed for the failure detector, such a scenario is perfectly conceivable.

Of course, this specification suffers from the same fault as the first one discussed. A similar improvement can be made. However, the loop is retained because the act of receiving an *Est* message may be performed up to four times.

```
define quorum_complete(i) ((Suspected[i*NoP+0] || (est_from_c[0]>0)) &&
                           (Suspected[i*NoP+1] || (est_from_c[1]>0)) &&
                           (Suspected[i*NoP+2] || (est_from_c[2]>0)) &&
                           (Suspected[i*NoP+3] || (est_from_c[3]>0)))
do
   :: (quorum_complete(i) || Msg[i]??[Decide(_,_,_)] ||
      Msg[i]??[Est, 0, eval(round[i]), _] || Msg[i]??[Est, 1, eval(round[i]), _] ||
      Msg[i]??[Est, 2, eval(round[i]), _] || Msg[i]??[Est, 3, eval(round[i]), _])
      -> if
         :: Msg[i]??[Decide(_,_,_)] ->
               Msg[i]??Decide(_, _, v); goto BROADCAST;
         :: else -> if
               :: quorum_complete(i) -> break;
               :: else -> if
                    :: Msg[i]??[Est, 0, eval(round[i]), _]->
                          Msg[i]??Est, 0, eval(round[i]), est_from_c[0];
                    :: Msg[i]??[Est, 1, eval(round[i]), _]->
                          Msg[i]??Est, 1, eval(round[i]), est_from_c[1];
                    :: Msg[i]??[Est, 2, eval(round[i]), _]->
                          Msg[i]??Est, 2, eval(round[i]), est_from_c[2];
                    :: Msg[i]??[Est, 3, eval(round[i]), _]->
                          Msg[i]??Est, 3, eval(round[i]), est_from_c[3];
                    fi;
                fi;
          fi;
od;
```

Lastly, a process is also allowed to receive *Dec* message before it is testing the conditions for decision or going to the next round. Before sending out *Dec* messages, a process is again given a chance to crash. The broadcast of *Dec* messages is not performed in one atomic step to allow scenarios in which more than one processes decide and broadcast simultaneously. One may raise the objection that the messages are delivered in a particular sequence. But any of the four consensus processes can decide in all the possible sequences—a fact which allows the verification of all possible scenarios[6]

[6] Well, maybe just most of the cases because the coordinator election by round robin always starts with process 0 and ends with 3.

```
if
   :: atomic {Crash_Count<3 -> Crash_Count++; Failure[i]=true;};
      goto CRASHED;
   :: true -> if
        ::i==0 -> Msg[1]!Dec(i, round[i], v);
                  Msg[2]!Dec(i, round[i], v);
                  Msg[3]!Dec(i, round[i], v);
        ::i==1 -> Msg[0]!Dec(i, round[i], v);
                  Msg[2]!Dec(i, round[i], v);
                  Msg[3]!Dec(i, round[i], v);
        ::i==2 -> Msg[0]!Dec(i, round[i], v);
                  Msg[1]!Dec(i, round[i], v);
                  Msg[3]!Dec(i, round[i], v);
        ::i==3 -> Msg[0]!Dec(i, round[i], v);
                  Msg[1]!Dec(i, round[i], v);
                  Msg[2]!Dec(i, round[i], v);
        fi;
fi;
Decided[i]=v; goto FINISH;
```

The crashable consensus process (*proctype cra_consensus*) is instantiated four times with designations 0 through 3 and randomized proposed values. Surveying the specification for the process, it is evident that no delay of message is explicitly modeled. This absence, however, does not necessary suggest that the message is passed instantaneously at all times. The author concedes that it is so when the recipient is waiting for the message. However, due to the interleaving fashion of executing several processes, all the possible sequences in which the message can be sent and received are possible. By the same token of reason, the processes receive a message of a type it desires whenever such one is present in the channel (using "??" to receive). The delay here, in one sense, is modeled by the arbitrariness of the sender in the sequence of execution. Nonetheless, care is given during the modeling to ensure that the specification is comprehensive.

4 Desired Properties

The author specified and verified six properties of the consensus protocol. They are listed and discussed in this section. In all the formulae presented, macros are defined to introduce conciseness with the understanding that SPIN does not support macros in parsing LTL formulae. In practice, the macros are fully expanded manually or used indirectly through additional *#define* definition.

4.1 Safety Properties

The first safety property (Safe 1) is that if a process decides, it decides upon one of the proposed values. The *Decided* flags are initialized to zero at the beginning and only accessed once in the consensus process—when the process terminates correctly. The macro *v(i)* is evaluated true in the case that *Decided* for the ith process equals to zero or one of the proposed values.

The LTL formula requires that at all times, each process either decides on one of the proposed values or remain undecided. Usually, this is a slightly stronger requirement than the original statement as it imposes constraints before a decision is reached. In this case, they are equivalent because one variable serves both as the decision flag and the holder of the decided value. However, neither statement stipulates that a process has to eventually decide. This is verified elsewhere (see Live 2).

Safe 1:

```
#define v(i) ( (Decided[i]==0) || (Decided[i]==Proposed[0]) ||
               (Decided[i]==Proposed[1]) || (Decided[i]==Proposed[2])
               || (Decided[0]==Proposed[3]) )

[]    ( v(0) && v(2) && v(3) && v(1))
```

The second safety property (Safe 2) is that all correct processes must decide on the same value. Two attempts of specification are made. In the first version, the LTL formula states that always, the sum of the four *Decided* variables has to be divisible by one of the possible proposed values. However, this statement is flawed because it is assuming that no combination of proposed values is divisible by one of the values. A quick counterexample is that $2 \times 12 + 2 \times 13$ is divisible by 10. To find a set of four positive integers between 1 and 60—the maximum sum has to be less than 255 or overflow will occur — that does not violate this property proves to be difficult.

Safe 2:(faulty)

```
#define sum (Decided[0]+Decided[1]+Decided[2]+Decided[3])
#define ndv(d) (sum%d)

[]    ( (! ndv(10)) || (! ndv(11)) || (! ndv(12)) || (! ndv(13)))
```

An alternative approach is taken to specify the same property. At all times, if one process has decided, each of the other processes has either not decided or decided on the same value.

Safe 2:

```
#define d(i) (Decided[i]>0)
#define pdc(i) ( (!d(0) || Decided[0]==Decided[i]) && (!d(1)
                 || Decided[1]==Decided[i]) && (!d(2)
                 || Decided[2]==Decided[i]) && (!d(3)
                 || Decided[3]==Decided[i]) )

[]( (d(0) -> pdc(0)) && (d(1) -> pdc(1)) && (d(2) ->pdc(2))
    && (d(3) -> pdc(3)) )
```

A theorem (Safe 3) is presented in Mostefaoui and Raynal's paper [6] that for class S failure detectors, the decision will be made within n rounds, where n is the number of consensus processes. It is specified by the following LTL formula.

Safe 3:

```
#define r(i) (round[i]<NoP+1)

[]  (r(0) && r(1) && r(2) && r(3))
```

The upper bound is set to be NoP (defined to be n) because the round starts at 1.

The author also specified another property (Safe 4) to test his claim that the length of the message queue is 16 at maximum. This property concerns the correctness of the model because if a buffered message channel is full, the protocol will block at broadcast stage and synchronization may occur unintentionally. The property is specified using an assert statement.

Safe 4:

```
proctype Chan_Check()
{
do
:: (Decided[0] || Failure[0]) && (Decided[1] ||
           Failure[1]) && (Decided[2] || Failure[2]) &&
           (Decided[3] || Failure[3]) -> break;
:: else ->  assert nfull(Msg[0]) && nfull(Msg[1])
           && nfull(Msg[2]) && nfull(Msg[3]);
od;
}
```

In the last case, the model is compiled without any LTL properties to verify that valid end states are always obtained. In this model, it implies that all the processes eventually terminate.

4.2 Liveness Property

The first liveness property (Live 1) is that all correct processes decide. Or if a process never fails, it is implied that the same process eventually decides.

Live 1:

```
#define d(i) (Decided[i]!=0)
#define f(i) (Failure[i])
#define cp(i) (([]!f(i)) -> <> d(i))

cp(0) && cp(1) && cp(2) && cp(3)
```

The second liveness property(Live 2) is the existential property: eventually, at least one process decides. It is easily stated in LTL formula.

Live 2:

```
#define d(i) (Decided[i]!=0)

<> ( d(0) || d(1) || d(2) || d(3) )
```

5 Results and Discussion

A few random simulations were performed on both the original and the improved models. The consensus processes all terminated correctly. Moreover, different combinations of crashed/decided states were obtained at the termination—a fact that bestowed confidence in the correctness of the specification (especially the rather artificial interleaving). The improved model terminated much sooner in the simulations.

All the pan.c source files for verifying LTL properties were compiled with flags *-D -DMEMLIM=900 -DMA=[378 or 410] -DNFAIR=4 -D_POSIX_SOURCE* and run with the flags *-a -f -m30000000 -w24 -s* . The one without LTL formula was compiled with *-DMEMLIM=900 -DCOLLAPSE -DSAFETY -D_POSIX_SOURCE* and run with the flags *-m30000000 -w24 -s* . Selected results from the more efficient model are tabulated; the full transcript of the session can be found in Appendix C. The original model only differs from the former in the amount of memory consumed and time required.

None of the verification runs was able to finish before the search depth constraint was violated. Further, if the verification was allowed to continue, the memory constraint inevitably failed. The depth (set at 30 million), however, could not be increased by a significant amount without bringing SPIN into a segmentation fault, the likely cause of which was that some value exceeds those supported by the machine. Of all the states and transitions visited, no violation to any of the properties was found. An error was reported just after pan aborts the verification. No violation was found upon the inspection of the resulting .trail file (see Appendix C.1). Because the error always occurred after the error on depth and an additional error caused

Property	Live 1	Live 2	Safety 1	Safety 2	Other 1	Other 2
Depth	29399872	29999999	29399872	29399872	29999999	29999999
Transitions:	9.8e+06	1.7866e+07	9.8e+06	9.8e+06	6.29276e+07	1.71111e+08
States	7e+06	8e+06	7e+06	7e+06	1.6e+07	2.4e+07
Mem(MB)	720.588	912.122	720.588	720.588	931.477	943.662
Time (Min)	100	138	102	100	418	660
Atomic Steps	1.57535e+07	3.68085e+07	1.60418e+07	1.60418e+07	1.31467e+08	3.09385e+08
Compression	0.08%	6.38%	0.26%	0.26%	3.24%	2.15%

Table 1: The computation results. All the runs terminated due to the lack of memory.

by the graph encoding algorithm, the author conjectures that the violation to the LTL properties was just an artifact.

Another effort was made to simplify the system even further by enforcing perpetual completeness. The failure detectors were made to suspect a process as soon as it fails so that it can only erroneously suspect the correct processes. This lead to a much simplified *Detector* proctype. Yet the verification still ran into violations on both the memory and search depth. The inability of SPIN to fully verify the model is far from being unexpected. From the inherent complexity of the protocol and the environment, the state space is certain to explode to a size unmanageable by ordinary computational devices. More scrupulous specification can improve the situation but not completely avoid it due to the enormous state space size (404-byte state vector including the states for fairness and never claim).

PROMELA as a modeling language is very expressive and well suited for specifying asynchronous processes. However, the author finds the fact that only weak fairness can be enforced unsatisfactory. Had SPIN the support for some strong fairness conditions in LTL, the specification of the environment would be much easier. But the implementation of such property may be very expensive. Given that it does a depth first search by default, SPIN appears to be a much better tool at fault-finding than full verification.

6 Conclusion

In this verification exercise, a generic consensus protocol is verified for the S class of unreliable failure detectors. The protocol is modeled with PRO-

MELA/SPIN, a popular specification and model checking suite. Due to the large size of the state space, the system is only partially verified. Both the constraints on memory and search depth are violated before the completion. The verification tool is noted for its expressiveness in modeling as well as its dissatisfactory ability to handle strong fairness.

7 Acknowledgement

The author would like to thank Marsha Chechik and Sam Toueg for suggesting the topic, and Chechik for her careful reading of the manuscript and many indispensable suggestions.

References

[1] Tushar D. Chandra and Sam Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems". *Jounal of the ACM*, 43(2):225–267, March 1996.

[2] Marsha Chechik. *"Automated Verfication Course Notes"*. Department of Computer Science, University of Toronto, Toronto, Canada, December 2000.

[3] Rob Gerth. "Concise Promela Reference". Technical report, Eindhoven University, http://cm.bell-labs.com/cm/cs/what/spin/Man/Quick.html, June 1997.

[4] Gerard J. Holzmann. *SPIN Online Reference*. Bell Labs, http://cm.bell-labs.com/cm/cs/what/spin/Man/index.html, August 1997.

[5] Gerard J. Holzmann. "The Model Checker SPIN". *IEEE Transaction on Software Engineering*, 23(5):1–17, May 1997.

[6] Achour Mostefaoui and Michel Raynal. "Solving Consensus Using Chandra-Toueg's Unreliable Failure Detectors: A General Quorum-Based Approach". In Prasad Jayanti, editor, *Distributed Computing, 13th International Symposium, DISC'99 Bratislava, Slovak Republic September 27-29, 1999 Proceedings*, pages 49–63, Berlin, Germany, 1999. Springer-Verlag.

A The Original PROMELA Model

The source code for the original PROMELA model is presented here. The description for of the model can be found in Section 3. This specification is not used in verification because of its inefficiency.

```
/* Automated Verification Final Project */
/* Copyright (c) Jiang Liu, 2001       */
/* SN: 961856390                       */

/* first version */

/* Consensus Protocol Using Chandra-Toueg's Unreliable Failure Detectors */
/* as discussed in Mostefaoui, A. and M Raynal, "Solving Consensus Using */
/* Chandra-Toueg's Unreliable Failure Detectors: A General Quorum-Based  */
/* Approach."  Proceedings of the 13th International Symposium on       */
/* Distributed Computing, September 1999, Bratislava, Slovak Republic,   */
/* Lecture Notes in Computer Science (Prasad Jayanti ed.), Springer-     */
/* Verlag, vol. 1693, 49-63.                                             */

/* Assumptions on the System */
/* 1. System consists of a finite set of n>1 processes: PI */
/* 2. Each process behaves correctly (according to spec) until it crashes. */
/*    By definition, a correct process is one that does not crash. */
/* 3. f denotes the maximum number of the processes that can crash: f<n */
/* 4. Every pair of processes is connected by a channel through which */
/*    they communicate and synchronize. */
/* 5. Channels are not required to be FIFO; but they are reliable in the */
/*    sense that they do not create, alter, or lose message.  */

#define NoP 4  /* total number of processes */
#define DEFAULT 255  /* default value for est */
#define SusLen 16  /* the lengh of suspected list */
#define BufLen 17 /* length of message buffer */
#define TimeOut 16

show bool Failure[NoP];
/* true state of the processes */

byte Crash_Count;  /* count the number of crashed processes */

show byte Decided[NoP];
/* Decision flag; used in reasoning */

show bool Suspected[SusLen];
/* suspected list */


mtype={Dec, Est};


chan Msg[NoP]=[BufLen] of {mtype, byte, byte, byte};
/* messgage format: Type, From, Round, v */

byte round[NoP]; /* current round of the processes */
```

```
byte Proposed[NoP]; /* proposed value by processes */
/* the Uniform Consensus problem is defined by three properties: */
/* Termination: Every correct process eventually decides on some value. */
/* Validy: If a process decides v, then v was proposed by some process. */
/* Uniform Agreement: No two proceeses (correct or not) decide differently. */
/* This spec for consensus may crash */
proctype cra_consensus(byte i; byte vi)
{
  byte est_from_c[NoP];
show  byte c;
  /* c is the current coordinator */
  byte est;
  byte v;
  /* v is the proposed value */
  atomic {
 round[i]=0;
 est=vi;
  };
  do
 ::
 atomic { est_from_c[0]=0; est_from_c[1]=0;
  est_from_c[2]=0; est_from_c[3]=0; };
 /* initialize est_from_c every time a new round starts*/
 atomic {
   c=(round[i] % NoP); est_from_c[i]=DEFAULT; round[i]++;
     /* Round robin coordinator election */
 };
 if
   :: i==c -> est_from_c[i]=est; /* is coordinator */
   :: else -> do
 :: Msg[i]??[Dec(_, _, _)] ->
 Msg[i]??Dec(_, _, v); goto BROADCAST;
 /* Upon receiving Decide, broadcast it */
  :: else -> if
 :: Suspected[i*NoP+c] -> break;
        /* cur coord is suspected */
 :: Msg[i]??[Est, eval(c), eval(round[i]), _] ->
    Msg[i]??Est, eval(c), eval(round[i]), v;
            est_from_c[i]=v; break;
        /* received est from cur coord */
 :: else -> skip;
 fi;
 od;
 /* blocks unless one of the two occurs */
 fi;
```

```

 /* broadcast est_from_c[i] */
progress: if
   :: Msg[i]??[Dec(_,_,_)] ->
 Msg[i]??Dec(_, _, v); goto BROADCAST;
   :: else -> if
 :: true ->atomic {
    Msg[0]!Est(i, round[i], est_from_c[i]);
    Msg[1]!Est(i, round[i], est_from_c[i]);
    Msg[2]!Est(i, round[i], est_from_c[i]);
    Msg[3]!Est(i, round[i], est_from_c[i]);
 };
 /* process may crash if less than 3 have crashed */
 /* Note that the crash_count var is updated as soon*/
 /* as the system decides to crash to ensure S class. */
 :: atomic {Crash_Count<3 -> Crash_Count++;Failure[i]=true;};
 goto CRASHED;
 fi;
 fi;

 /* wait until the quorum is formed */
 /* This is specified for class S, where the quorum has to */
 /* include at least all the processes that are not suspected */
 /* i.e. Q[i] union Suspected = entire space */
 do
   :: Msg[i]??[Dec(_,_,_)] ->
 Msg[i]??Dec(_, _, v); goto BROADCAST;
   :: else -> if
 :: ((Suspected[i*NoP+0] || (est_from_c[0]>0)) && (Suspected[i*NoP+1] ||
             (est_from_c[1]>0)) && (Suspected[i*NoP+2] || (est_from_c[2]>0)) &&
             (Suspected[i*NoP+3] || (est_from_c[3]>0))) -> break;
 :: else -> if
    :: Msg[i]??[Est, 0, eval(round[i]), _]->
      Msg[i]??Est, 0, eval(round[i]), est_from_c[0];
    :: Msg[i]??[Est, 1, eval(round[i]), _]->
    Msg[i]??Est, 1, eval(round[i]), est_from_c[1];
    :: Msg[i]??[Est, 2, eval(round[i]), _]->
    Msg[i]??Est, 2, eval(round[i]), est_from_c[2];
    :: Msg[i]??[Est, 3, eval(round[i]), _]->
    Msg[i]??Est, 3, eval(round[i]), est_from_c[3];
    :: else -> skip;
 fi
 fi
   od;
 /* This loop will only terminate when each process */
 /* either is suspected or has sent a message which */
 /* has been received. Note the spec allows the quorum */
 /* to change (performed by another parallel process) */
 /* by testing whether a message exists or not before */
 /* blocking to wait for the message. Further, the reception */
 /* of the message is random as the channel is assumed not to */
 /* be LILO. */

 /* grep a received value that is not default */
 if
   :: (est_from_c[0]>0) && (est_from_c[0]!=DEFAULT) -> v=est_from_c[0];
   :: (est_from_c[1]>0) && (est_from_c[1]!=DEFAULT) -> v=est_from_c[1];
```

```
   :: (est_from_c[2]>0) && (est_from_c[2]!=DEFAULT) -> v=est_from_c[2];
   :: (est_from_c[3]>0) && (est_from_c[3]!=DEFAULT) -> v=est_from_c[3];
   :: else -> skip;
 fi;
 /* make decisions */
 if
   :: Msg[i]??[Dec(_, _, _)] ->
 Msg[i]??Dec(_, _, v); goto BROADCAST;
   :: else -> if
    :: ((est_from_c[0]==v) || (est_from_c[0]==0)) && ((est_from_c[1]==v) ||
              (est_from_c[1]==0)) && ((est_from_c[2]==v) || (est_from_c[2]==0)) &&
              ((est_from_c[3]==v) || (est_from_c[3]==0)) -> goto BROADCAST;
    :: ((est_from_c[0]==DEFAULT) || (est_from_c[0]==0)) && ((est_from_c[1]==DEFAULT) ||
              (est_from_c[1]==0)) && ((est_from_c[2]==DEFAULT) || (est_from_c[2]==0)) &&
              ((est_from_c[3]==DEFAULT) || (est_from_c[3]==0)) -> skip;
    :: else -> est=v; /* Mixture */
 fi;
 fi;
  od;

  /* This part performs the reliable broadcast */
BROADCAST:  if
 :: atomic {Crash_Count<3 -> Crash_Count++; Failure[i]=true;};
    goto CRASHED;
 :: true -> if
   ::i==0 -> Msg[1]!Dec(i, round[i], v);
 Msg[2]!Dec(i, round[i], v);
 Msg[3]!Dec(i, round[i], v);
   ::i==1 -> Msg[0]!Dec(i, round[i], v);
 Msg[2]!Dec(i, round[i], v);
 Msg[3]!Dec(i, round[i], v);
   ::i==2 -> Msg[0]!Dec(i, round[i], v);
 Msg[1]!Dec(i, round[i], v);
 Msg[3]!Dec(i, round[i], v);
   ::i==3 -> Msg[0]!Dec(i, round[i], v);
 Msg[1]!Dec(i, round[i], v);
 Msg[2]!Dec(i, round[i], v);
   fi;
  fi;
  Decided[i]=v; goto FINISH;
CRASHED: goto FINISH;
  /* This is the end of the protocol */
FINISH: skip
};


/* Assumptions on the Failure Detector */
/* 1. A failure detector consists of a set of modules, one for each process. */
/* 2. The module attached to a process maintains a set of processes it  */
/*    suspects of having crashed.  */
/* 3. Any detector module is inherently unreliable. */
/* 4. Suspicions are not necessarily stable: a process can be added or */
/*    removed from the 'suspected' list. */

/* Properties of Failure Detectors */

/* 1. Strong Completeness: Eventually, every process that crashes is */
```

```
/*    permanently suspected by every correct process. */
/* 2. Perpectual Weak Accuracy: Some correct process is never suspected. */
/* 3. Eventual Weak Accuracy: There is a time after which some correct */
/*    process is never suspected by correct processes. */

/* Two classes of failure detectors: */
/* S: Strong failure detectors.  1+2 */



/* One fair assumption is that a process never suspects itself. */
/* The completeness and accuracy are enforced by the fairness condition. */
proctype Detector() priority 1
{
  int counter=0;
  do
 :: (Decided[0] || Failure[0]) && (Decided[1] || Failure[1]) && (Decided[2] ||
      Failure[2]) && (Decided[3] || Failure[3]) -> break;
 :: else -> atomic {
 if
     :: !Failure[0] ->if
   :: Suspected[1] -> Suspected[1]=false;
 :: Suspected[2] -> Suspected[2]=false;
 :: Suspected[3] -> Suspected[3]=false;
 :: !Suspected[1] -> Suspected[1]=true;
 :: !Suspected[2] -> Suspected[2]=true;
 :: !Suspected[3] -> Suspected[3]=true;
 :: skip;
      fi;
 /* ensures that at least one correct process is not */
 /* suspected. */
 if
 :: !Failure[0] -> Suspected[0]=false;
 :: !Failure[1] -> Suspected[1]=false;
 :: !Failure[2] -> Suspected[2]=false;
 :: !Failure[3] -> Suspected[3]=false;
 :: else -> skip;
 fi;
 /* enforce strong completeness here */
 if
 :: counter>TimeOut -> goto Timed_Out;
 :: else -> skip;
 fi;
 :: !Failure[1] ->if
   :: Suspected[NoP+0] -> Suspected[NoP+0]=false;
 :: Suspected[NoP+2] -> Suspected[NoP+2]=false;
 :: Suspected[NoP+3] -> Suspected[NoP+3]=false;
 :: !Suspected[NoP+0] -> Suspected[NoP+0]=true;
 :: !Suspected[NoP+2] -> Suspected[NoP+2]=true;
 :: !Suspected[NoP+3] -> Suspected[NoP+3]=true;
 :: skip;
     fi;
 if
 :: !Failure[0] -> Suspected[NoP+0]=false;
 :: !Failure[1] -> Suspected[NoP+1]=false;
 :: !Failure[2] -> Suspected[NoP+2]=false;
 :: !Failure[3] -> Suspected[NoP+3]=false;
```

```
 :: else -> skip;
 fi;
 if
 :: counter>TimeOut -> goto Timed_Out;
 :: else -> skip;
 fi;
 :: !Failure[2] -> if
   :: !Suspected[2*NoP+0]->Suspected[2*NoP+0]=true;
 :: !Suspected[2*NoP+1]->Suspected[2*NoP+1]=true;
 :: !Suspected[2*NoP+3]->Suspected[2*NoP+3]=true;
 :: Suspected[2*NoP+0]->Suspected[2*NoP+0]=false;
 :: Suspected[2*NoP+1]->Suspected[2*NoP+1]=false;
 :: Suspected[2*NoP+3]->Suspected[2*NoP+3]=false;
 :: skip;
     fi;
 if
 :: !Failure[0] -> Suspected[2*NoP+0]=false;
 :: !Failure[1] -> Suspected[2*NoP+1]=false;
 :: !Failure[2] -> Suspected[2*NoP+2]=false;
 :: !Failure[3] -> Suspected[2*NoP+3]=false;
 :: else -> skip;
 fi;
 if
 :: counter>TimeOut -> goto Timed_Out;
 :: else -> skip;
 fi;
 :: !Failure[3] -> if
   :: !Suspected[3*NoP+0]->Suspected[3*NoP+0]=true;
 :: !Suspected[3*NoP+1]->Suspected[3*NoP+1]=true;
 :: !Suspected[3*NoP+2]->Suspected[3*NoP+2]=true;
 :: Suspected[3*NoP+0]->Suspected[3*NoP+0]=false;
 :: Suspected[3*NoP+1]->Suspected[3*NoP+1]=false;
 :: Suspected[3*NoP+2]->Suspected[3*NoP+2]=false;
   :: skip;
     fi;
 if
 :: !Failure[0] -> Suspected[3*NoP+0]=false;
 :: !Failure[1] -> Suspected[3*NoP+1]=false;
 :: !Failure[2] -> Suspected[3*NoP+2]=false;
 :: !Failure[3] -> Suspected[3*NoP+3]=false;
 :: else -> skip;
 fi;
 if
 :: counter>TimeOut -> goto Timed_Out;
 :: else -> skip;
 fi;
    :: else -> skip

 fi;
 goto NEXT;
Timed_Out:
 if
   :: Failure[0] -> Suspected[0]=true;
    Suspected[NoP]=true;
    Suspected[2*NoP]=true;
    Suspected[3*NoP]=true;
   :: else -> skip;
```

```
 fi;
 if
    :: Failure[1] -> Suspected[1]=true;
     Suspected[NoP+1]=true;
     Suspected[2*NoP+1]=true;
     Suspected[3*NoP+1]=true;
    :: else -> skip;
 fi;
 if
    :: Failure[2] -> Suspected[2]=true;
     Suspected[NoP+2]=true;
     Suspected[2*NoP+2]=true;
     Suspected[3*NoP+2]=true;
    :: else -> skip;
 fi;
 if
    :: Failure[3] -> Suspected[3]=true;
     Suspected[NoP+3]=true;
     Suspected[2*NoP+3]=true;
     Suspected[3*NoP+3]=true;
    :: else -> skip;
 fi; };
NEXT: counter++;
  od;
};

proctype process_0 ()
{
  if
 :: Proposed[0]=10;
 :: Proposed[0]=11;
 :: Proposed[0]=12;
 :: Proposed[0]=13;
  fi;

  run cra_consensus(0, Proposed[0]);
};

proctype process_1()
{
  if
 :: Proposed[1]=10;
 :: Proposed[1]=11;
 :: Proposed[1]=12;
 :: Proposed[1]=13;
  fi;

  run cra_consensus(1, Proposed[1]);
};

proctype process_2()
{
  if
 :: Proposed[2]=10;
 :: Proposed[2]=11;
 :: Proposed[2]=12;
 :: Proposed[2]=13;
```

```
  fi;

  run cra_consensus(2, Proposed[2]);
};

proctype process_3()
{
  if
 :: Proposed[3]=10;
 :: Proposed[3]=11;
 :: Proposed[3]=12;
 :: Proposed[3]=13;
  fi;

  run cra_consensus(3, Proposed[3]);
}

proctype Chan_Check()
{
  do
 :: (Decided[0] || Failure[0]) && (Decided[1] || Failure[1]) && (Decided[2] ||
      Failure[2]) && (Decided[3] || Failure[3]) -> break;
 :: else ->  assert nfull(Msg[0]) && nfull(Msg[1]) && nfull(Msg[2]) && nfull(Msg[3]);
  od;
}

init {
 atomic {
  run process_1() priority 100; run process_0() priority 100;
  run process_2() priority 100; run process_3() priority 100;
  run Detector() priority 1; run Chan_Check() priority 1;
}
}
```

B The Improved Model

This specification is largely similar to the original model (Appendix A). Modifications are made at a few points to improve the efficiency (see Section 3.4). The improved model is used in verification.

```
/* Automated Verification Final Project */
/* Copyright (c) Jiang Liu, 2001       */

/* This is the improved version */

#define NoP 4 /* total number of processes */
#define DEFAULT 255 /* default value for est */
#define SusLen 16
#define BufLen 17
#define TimeOut 16
#define quorum_complete(i) ((Suspected[i*NoP+0] || (est_from_c[0]>0)) &&
        (Suspected[i*NoP+1] || (est_from_c[1]>0)) && (Suspected[i*NoP+2] ||
        (est_from_c[2]>0)) && (Suspected[i*NoP+3] || (est_from_c[3]>0)))

show bool Failure[NoP];
/* true state of the processes */
byte Crash_Count;

show byte Decided[NoP];
/* Decision flag; used in reasoning */

show bool Suspected[SusLen];

mtype={Decide, Est};


chan Msg[NoP]=[BufLen] of {mtype, byte, byte, byte};
/* messgage format: Type, From, Round, v */

byte round[NoP];
byte Proposed[NoP];


/* This spec for consensus may crash */
proctype cra_consensus(byte i; byte vi)
{
  /* *_rv's are temporary var that holds the value */
  /* recieved from the channel */
  byte est_from_c[NoP];
show  byte c;
  /* c is the current coordinator */

  byte est;

  byte v;

  /* v is the proposed value */

  atomic {
 round[i]=0;
```

```
 est=vi;
  };

  do
 ::
 atomic { est_from_c[0]=0; est_from_c[1]=0;
  est_from_c[2]=0; est_from_c[3]=0; };
 /* initialized est_from_c every time a new round starts*/

 atomic {
   c=(round[i] % NoP); est_from_c[i]=DEFAULT; round[i]++;
     /* Round robin coordinator election */
 };

 if
   :: i==c -> est_from_c[i]=est; /* is coordinator */
   :: else -> (Msg[i]??[Decide(_, _, _)] || Suspected[i*NoP+c] ||
                 Msg[i]??[Est, eval(c), eval(round[i]), _]) -> if
 :: Msg[i]??[Decide(_, _, _)] ->
 Msg[i]??Decide(_, _, v); goto BROADCAST;
 /* Upon receiving Decideide, broadcast it */
  :: else -> if
 :: Suspected[i*NoP+c] -> skip;
        /* cur coord is suspected */
 :: Msg[i]??[Est, eval(c), eval(round[i]), _] ->
    Msg[i]??Est, eval(c), eval(round[i]), v;
            est_from_c[i]=v; skip;
        /* received est from cur coord */
 fi;
 fi;
 /* blocks unless one of the two occurs */
 fi;

 /* broadcast est_from_c[i] */
progress: if
   :: Msg[i]??[Decide(_,_,_)] ->
 Msg[i]??Decide(_, _, v); goto BROADCAST;
   :: else -> if
 :: true ->atomic {
    Msg[0]!Est(i, round[i], est_from_c[i]);
    Msg[1]!Est(i, round[i], est_from_c[i]);
    Msg[2]!Est(i, round[i], est_from_c[i]);
    Msg[3]!Est(i, round[i], est_from_c[i]);
 };
 /* process may crash if less than 3 have crashed */
 /* Note that the crash_count var is updated as soon*/
 /* as the system decides to crash to ensure S class. */
 :: atomic {Crash_Count<3 -> Crash_Count++;Failure[i]=true;};
 goto CRASHED;
 fi;
 fi;

 /* wait until the quorum is formed */
 /* This is specified for class S, where the quorum has to */
 /* include at least all the processes that are not suspected */
 /* i.e. Q[i] union Suspected = entire space */
 do
```

```
   :: (quorum_complete(i) || Msg[i]??[Decide(_,_,_)]||
          Msg[i]??[Est, 0, eval(round[i]), _] || Msg[i]??[Est, 1, eval(round[i]), _] ||
          Msg[i]??[Est, 2, eval(round[i]), _] || Msg[i]??[Est, 3, eval(round[i]), _]) -> if
 :: Msg[i]??[Decide(_,_,_)] ->
 Msg[i]??Decide(_, _, v); goto BROADCAST;
    :: else -> if
   :: quorum_complete(i) -> break;
   :: else -> if
      :: Msg[i]??[Est, 0, eval(round[i]), _]->
      Msg[i]??Est, 0, eval(round[i]), est_from_c[0];
      :: Msg[i]??[Est, 1, eval(round[i]), _]->
    Msg[i]??Est, 1, eval(round[i]), est_from_c[1];
      :: Msg[i]??[Est, 2, eval(round[i]), _]->
    Msg[i]??Est, 2, eval(round[i]), est_from_c[2];
      :: Msg[i]??[Est, 3, eval(round[i]), _]->
    Msg[i]??Est, 3, eval(round[i]), est_from_c[3];
 fi;
 fi;
   fi;

   od;

 if
   :: (est_from_c[0]>0) && (est_from_c[0]!=DEFAULT) -> v=est_from_c[0];
   :: (est_from_c[1]>0) && (est_from_c[1]!=DEFAULT) -> v=est_from_c[1];
   :: (est_from_c[2]>0) && (est_from_c[2]!=DEFAULT) -> v=est_from_c[2];
   :: (est_from_c[3]>0) && (est_from_c[3]!=DEFAULT) -> v=est_from_c[3];
   :: else -> skip;
 fi;
 /* make decisions */
 if
   :: Msg[i]??[Decide(_, _, _)] ->
 Msg[i]??Decide(_, _, v); goto BROADCAST;
   :: else -> if
    :: ((est_from_c[0]==v) || (est_from_c[0]==0)) && ((est_from_c[1]==v) ||
              (est_from_c[1]==0)) && ((est_from_c[2]==v) || (est_from_c[2]==0)) &&
              ((est_from_c[3]==v) || (est_from_c[3]==0)) -> goto BROADCAST;
    :: ((est_from_c[0]==DEFAULT) || (est_from_c[0]==0)) && ((est_from_c[1]==DEFAULT) ||
              (est_from_c[1]==0)) && ((est_from_c[2]==DEFAULT) || (est_from_c[2]==0)) &&
              ((est_from_c[3]==DEFAULT) || (est_from_c[3]==0)) -> skip;
    :: else -> est=v; /* Mixture */
 fi;
 fi;
  od;

  /* This part performs the reliable broadcast */
BROADCAST:  if
 :: atomic {Crash_Count<3 -> Crash_Count++; Failure[i]=true;};
    goto CRASHED;
 :: true -> if
   ::i==0 -> Msg[1]!Decide(i, round[i], v);
 Msg[2]!Decide(i, round[i], v);
 Msg[3]!Decide(i, round[i], v);
   ::i==1 -> Msg[0]!Decide(i, round[i], v);
 Msg[2]!Decide(i, round[i], v);
 Msg[3]!Decide(i, round[i], v);
   ::i==2 -> Msg[0]!Decide(i, round[i], v);
```

```
 Msg[1]!Decide(i, round[i], v);
 Msg[3]!Decide(i, round[i], v);
   ::i==3 -> Msg[0]!Decide(i, round[i], v);
 Msg[1]!Decide(i, round[i], v);
 Msg[2]!Decide(i, round[i], v);
   fi;
  fi;
  Decided[i]=v; goto FINISH;
CRASHED: goto FINISH;
  /* This is the end of the protocol */
FINISH: skip
};



/* One fair assumption is that a process never suspects itself. */
/* The completeness and accuracy are enforced by the fairness condition. */
proctype Detector() priority 1
{
  int counter=0;
  do
 :: (Decided[0] || Failure[0]) && (Decided[1] || Failure[1]) &&
      (Decided[2] || Failure[2]) && (Decided[3] || Failure[3]) -> break;
 :: else -> atomic {
 if
     :: !Failure[0] ->if
   :: Suspected[1] -> Suspected[1]=false;
 :: Suspected[2] -> Suspected[2]=false;
 :: Suspected[3] -> Suspected[3]=false;
 :: !Suspected[1] -> Suspected[1]=true;
 :: !Suspected[2] -> Suspected[2]=true;
 :: !Suspected[3] -> Suspected[3]=true;
 :: skip;
      fi;
 /* ensures that at least one correct process is not */
 /* suspected. */
 if
 :: !Failure[0] -> Suspected[0]=false;
 :: !Failure[1] -> Suspected[1]=false;
 :: !Failure[2] -> Suspected[2]=false;
 :: !Failure[3] -> Suspected[3]=false;
 :: else -> skip;
 fi;
 if
 :: counter>TimeOut -> goto Timed_Out;
 :: else -> skip;
 fi;
 :: !Failure[1] ->if
   :: Suspected[NoP+0] -> Suspected[NoP+0]=false;
 :: Suspected[NoP+2] -> Suspected[NoP+2]=false;
 :: Suspected[NoP+3] -> Suspected[NoP+3]=false;
 :: !Suspected[NoP+0] -> Suspected[NoP+0]=true;
 :: !Suspected[NoP+2] -> Suspected[NoP+2]=true;
 :: !Suspected[NoP+3] -> Suspected[NoP+3]=true;
 :: skip;
     fi;
 if
```

```
 :: !Failure[0] -> Suspected[NoP+0]=false;
 :: !Failure[1] -> Suspected[NoP+1]=false;
 :: !Failure[2] -> Suspected[NoP+2]=false;
 :: !Failure[3] -> Suspected[NoP+3]=false;
 :: else -> skip;
 fi;
 if
 :: counter>TimeOut -> goto Timed_Out;
 :: else -> skip;
 fi;
 :: !Failure[2] -> if
   :: !Suspected[2*NoP+0]->Suspected[2*NoP+0]=true;
 :: !Suspected[2*NoP+1]->Suspected[2*NoP+1]=true;
 :: !Suspected[2*NoP+3]->Suspected[2*NoP+3]=true;
 :: Suspected[2*NoP+0]->Suspected[2*NoP+0]=false;
 :: Suspected[2*NoP+1]->Suspected[2*NoP+1]=false;
 :: Suspected[2*NoP+3]->Suspected[2*NoP+3]=false;
 :: skip;
     fi;
 if
 :: !Failure[0] -> Suspected[2*NoP+0]=false;
 :: !Failure[1] -> Suspected[2*NoP+1]=false;
 :: !Failure[2] -> Suspected[2*NoP+2]=false;
 :: !Failure[3] -> Suspected[2*NoP+3]=false;
 :: else -> skip;
 fi;
 if
 :: counter>TimeOut -> goto Timed_Out;
 :: else -> skip;
 fi;
 :: !Failure[3] -> if
   :: !Suspected[3*NoP+0]->Suspected[3*NoP+0]=true;
 :: !Suspected[3*NoP+1]->Suspected[3*NoP+1]=true;
 :: !Suspected[3*NoP+2]->Suspected[3*NoP+2]=true;
 :: Suspected[3*NoP+0]->Suspected[3*NoP+0]=false;
 :: Suspected[3*NoP+1]->Suspected[3*NoP+1]=false;
 :: Suspected[3*NoP+2]->Suspected[3*NoP+2]=false;
   :: skip;
     fi;
 if
 :: !Failure[0] -> Suspected[3*NoP+0]=false;
 :: !Failure[1] -> Suspected[3*NoP+1]=false;
 :: !Failure[2] -> Suspected[3*NoP+2]=false;
 :: !Failure[3] -> Suspected[3*NoP+3]=false;
 :: else -> skip;
 fi;
 if
 :: counter>TimeOut -> goto Timed_Out;
 :: else -> skip;
 fi;
    :: else -> skip

 fi;
 goto NEXT;
Timed_Out:
 if
   :: Failure[0] -> Suspected[0]=true;
```

```
    Suspected[NoP]=true;
    Suspected[2*NoP]=true;
    Suspected[3*NoP]=true;
   :: else -> skip;
 fi;
 if
    :: Failure[1] -> Suspected[1]=true;
     Suspected[NoP+1]=true;
     Suspected[2*NoP+1]=true;
     Suspected[3*NoP+1]=true;
    :: else -> skip;
 fi;
 if
    :: Failure[2] -> Suspected[2]=true;
     Suspected[NoP+2]=true;
     Suspected[2*NoP+2]=true;
     Suspected[3*NoP+2]=true;
    :: else -> skip;
 fi;
 if
    :: Failure[3] -> Suspected[3]=true;
     Suspected[NoP+3]=true;
     Suspected[2*NoP+3]=true;
     Suspected[3*NoP+3]=true;
    :: else -> skip;
 fi; };
NEXT: counter++;
  od;
};

proctype process_0 ()
{
  if
 :: Proposed[0]=10;
 :: Proposed[0]=11;
 :: Proposed[0]=12;
 :: Proposed[0]=13;
  fi;

  run cra_consensus(0, Proposed[0]);
};

proctype process_1()
{
  if
 :: Proposed[1]=10;
 :: Proposed[1]=11;
 :: Proposed[1]=12;
 :: Proposed[1]=13;
  fi;

  run cra_consensus(1, Proposed[1]);
};

proctype process_2()
{
  if
```

```
 :: Proposed[2]=10;
 :: Proposed[2]=11;
 :: Proposed[2]=12;
 :: Proposed[2]=13;
  fi;

  run cra_consensus(2, Proposed[2]);
};

proctype process_3()
{
  if
 :: Proposed[3]=10;
 :: Proposed[3]=11;
 :: Proposed[3]=12;
 :: Proposed[3]=13;
  fi;

  run cra_consensus(3, Proposed[3]);
}

proctype Chan_Check()
{
  do
 :: (Decided[0] || Failure[0]) && (Decided[1] || Failure[1]) && (Decided[2] ||
      Failure[2]) && (Decided[3] || Failure[3]) -> break;
 :: else ->  assert nfull(Msg[0]) && nfull(Msg[1]) && nfull(Msg[2]) && nfull(Msg[3]);
  od;
}

init {
 atomic {
  run process_1() priority 100; run process_0() priority 100;
  run process_2() priority 100; run process_3() priority 100;
  run Detector() priority 1; run Chan_Check() priority 1;
}
}
```

C Properties and Results

C.1 Live 1

The specification and verification result for Liveness property 1—”if a process never fails, it eventually decides” (see Section 4.2)—is presented here. The end state in the trail file generated by an error after is also included. Upon inspection, the error is caused by premature termination of the verification due to shortage of memory.

```
#define d0 (Decided[0]!=0)
#define d1 (Decided[1]!=0)
#define d2 (Decided[2]!=0)
#define d3 (Decided[3]!=0)
#define f0 (Failure[0])
#define f1 (Failure[1])
#define f2 (Failure[2])
#define f3 (Failure[3])

/*
 * Formula As Typed: ( (([]!f0) -> <> d0) && (([]!f1) -> <> d1) &&
                               (([]!f2) -> <> d2) && (([]!f3) -> <> d3) )
 * The Never Claim Below Corresponds
 * To The Negated Formula !(( (([]!f0) -> <> d0) && (([]!f1) -> <> d1) &&
                                   (([]!f2) -> <> d2) && (([]!f3) -> <> d3) ))
 * (formalizing violations of the original)
 */

never {    /* !(( (([]!f0) -> <> d0) && (([]!f1) -> <> d1) && (([]!f2) -> <> d2) &&
                (([]!f3) -> <> d3) )) */
accept_init:
T0_init:
if
:: (! ((d3)) && ! ((f3))) -> goto accept_S4
:: (! ((d2)) && ! ((f2))) -> goto accept_S7
:: (! ((d1)) && ! ((f1))) -> goto accept_S10
:: (! ((d0)) && ! ((f0))) -> goto accept_S13
fi;
accept_S4:
T0_S4:
if
:: (! ((d3)) && ! ((f3))) -> goto accept_S4
fi;
accept_S7:
T0_S7:
if
:: (! ((d2)) && ! ((f2))) -> goto accept_S7
fi;
accept_S10:
T0_S10:
if
:: (! ((d1)) && ! ((f1))) -> goto accept_S10
fi;
accept_S13:
T0_S13:
if
```

```
:: (! ((d0)) && ! ((f0))) -> goto accept_S13
fi;
accept_all:
skip
}

#ifdef NOTES

#endif
#ifdef RESULT
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
Depth= 4199872 States=   1e+06 Transitions= 1.4e+06 Nodes=    2528 Memory= 720.696
Depth= 8399872 States=   2e+06 Transitions= 2.8e+06 Nodes=    2528 Memory= 720.696
Depth= 12599872 States=   3e+06 Transitions= 4.2e+06 Nodes=    2528 Memory= 720.696
Depth= 16799872 States=   4e+06 Transitions= 5.6e+06 Nodes=    2525 Memory= 720.696
Depth= 20999872 States=   5e+06 Transitions=   7e+06 Nodes=    2528 Memory= 720.696
Depth= 25199872 States=   6e+06 Transitions= 8.4e+06 Nodes=    2528 Memory= 720.696
Depth= 29399872 States=   7e+06 Transitions= 9.8e+06 Nodes=    2528 Memory= 720.696
error: max search depth too small
pan: error, MA too small, recompile pan.c with -DMA=N with N>380
pan: aborting (at depth 29999999)
pan: error writing consensus.new.trail
(Spin Version 3.3.6 -- 23 November 1999)
Warning: Search not completed
+ Partial Order Reduction
+ Graph Encoding (-DMA=378)

Full statespace search for:
never-claim         +
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness enabled)
invalid endstates - (disabled by never-claim)

State-vector 408 byte, depth reached 29999999, errors: 1
7.16004e+06 states, stored
3.05627e+06 states, matched
1.02163e+07 transitions (= stored+matched)
1.57535e+07 atomic steps
hash conflicts: 0 (resolved)
(max size 2^24 states)

Stats on memory usage (in Megabytes):
2978.575 equivalent memory usage for states (stored*(State-vector + overhead))
2.335  actual memory usage for states (compression: 0.08%)
720.000 memory used for DFS stack (-m30000000)
722.437 total actual memory usage

5918.81u 9.64s 1:40:13.52 98.5%
\* The following was reported as a violation to the property. It is evidently not one.
\* It seems to be caused by the premature termination.
*/
processes: 7
                Failure[0] = 0
                Failure[1] = 0
                Failure[2] = 0
                Failure[3] = 0
```

```
                Crash_Count = 0
                Decided[0] = 0
                Decided[1] = 0
                Decided[2] = 0
                Decided[3] = 0
                Suspected[0] = 0
                Suspected[1] = 0
                Suspected[2] = 0
                Suspected[3] = 0
                Suspected[4] = 0
                Suspected[5] = 0
                Suspected[6] = 0
                Suspected[7] = 0
                Suspected[8] = 0

                Suspected[9] = 0
                Suspected[10] = 0
                Suspected[11] = 0
                Suspected[12] = 0
                Suspected[13] = 0
                Suspected[14] = 0
                Suspected[15] = 0
                queue 1 (Msg[0]):
                queue 2 (Msg[1]):
                queue 3 (Msg[2]):
                queue 4 (Msg[3]):
                round[0] = 0
                round[1] = 0
                round[2] = 0
                round[3] = 0
                Proposed[0] = 0
                Proposed[1] = 0

                Proposed[2] = 0
                Proposed[3] = 0
 31:    proc  5 (Chan_Check) line 358 "consensus.new" (state 5)
 31:    proc  4 (Detector) line 283 "consensus.new" (state 161)
 31:    proc  3 (process_3) line 346 "consensus.new" (state 5)
 31:    proc  2 (process_2) line 334 "consensus.new" (state 5)
 31:    proc  1 (process_0) line 310 "consensus.new" (state 5)
 31:    proc  0 (process_1) line 322 "consensus.new" (state 5)
 31:    proc  0 (:init:) line 370 "consensus.new" (state 8) <valid
endstate>


#endif
```

C.2 Live 2

This section presents the specification and verification result for Liveness property 2: "at least one process decides" (see Section 4.2).

```
#define d0 (Decided[0]!=0)
#define d1 (Decided[1]!=0)
#define d2 (Decided[2]!=0)
#define d3 (Decided[3]!=0)
/*
 * Formula As Typed: <>  (d0 || d1 || d2 || d3)
 * The Never Claim Below Corresponds
 * To The Negated Formula !(<>  (d0 || d1 || d2 || d3))
 * (formalizing violations of the original)
 */

never {    /* !(<>  (d0 || d1 || d2 || d3)) */
accept_init:
T0_init:
if
:: (! ((d0)) && ! ((d1)) && ! ((d2)) && ! ((d3))) -> goto accept_S1
fi;
accept_S1:
T0_S1:
if
:: (! ((d0)) && ! ((d1)) && ! ((d2)) && ! ((d3))) -> goto accept_S1
fi;
accept_all:
skip
}

#ifdef NOTES
Liveness: Eventually, at least one process decides.


#endif
#ifdef RESULT
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
Depth= 4199872 States=   1e+06 Transitions= 1.4e+06 Nodes=    2624 Memory= 720.729
Depth= 8399872 States=   2e+06 Transitions= 2.8e+06 Nodes=    2624 Memory= 720.729
Depth= 12599872 States=   3e+06 Transitions= 4.2e+06 Nodes=    2624 Memory= 720.729
Depth= 16799872 States=   4e+06 Transitions= 5.6e+06 Nodes=    2621 Memory= 720.729
Depth= 20999872 States=   5e+06 Transitions=   7e+06 Nodes=    2624 Memory= 720.729
Depth= 25199872 States=   6e+06 Transitions= 8.4e+06 Nodes=    2624 Memory= 720.729
Depth= 29399872 States=   7e+06 Transitions= 9.8e+06 Nodes=    2624 Memory= 720.729
error: max search depth too small
Depth= 29999999 States=   8e+06 Transitions= 1.7866e+07 Nodes= 5134986 Memory= 912.122
pan: out of memory
9.43663e+08 bytes used
102404 bytes more needed
9.43718e+08 bytes limit (2^MEMCNT)
hint: to reduce memory, recompile with
  -DCOLLAPSE # good, fast compression, or
  -DHC # hash-compaction, approximation
  -DBITSTATE # supertrace, approximation
(Spin Version 3.3.6 -- 23 November 1999)
Warning: Search not completed
```

```
+ Partial Order Reduction
+ Graph Encoding (-DMA=410)

Full statespace search for:
never-claim          +
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness enabled)
invalid endstates - (disabled by never-claim)

State-vector 420 byte, depth reached 29999999, errors: 0
8.18473e+06 states, stored
1.17373e+07 states, matched
1.9922e+07 transitions (= stored+matched)
3.68085e+07 atomic steps
hash conflicts: 0 (resolved)
(max size 2^24 states)

Stats on memory usage (in Megabytes):
3503.064 equivalent memory usage for states (stored*(State-vector + overhead))
223.560 actual memory usage for states (compression: 6.38%)
720.000 memory used for DFS stack (-m30000000)
943.663 total actual memory usage


#endif
```

C.3 Safe 1

This section presents the specification and verification result for Safety property 1: "if a process decides, it decides on one of the proposed values" (see Section 4.1).

```
#define v0 ( (Decided[0]==0) || (Decided[0]==Proposed[0]) ||
             (Decided[0]==Proposed[1]) || (Decided[0]==Proposed[2]) ||
             (Decided[0]==Proposed[3]) )
#define v1 ( (Decided[1]==0) || (Decided[1]==Proposed[0]) ||
             (Decided[1]==Proposed[1]) || (Decided[1]==Proposed[2]) ||
             (Decided[1]==Proposed[3]) )
#define v2 ( (Decided[2]==0) || (Decided[2]==Proposed[0]) ||
             (Decided[2]==Proposed[1]) || (Decided[2]==Proposed[2]) ||
             (Decided[2]==Proposed[3]) )
#define v3 ( (Decided[3]==0) || (Decided[3]==Proposed[0]) ||
             (Decided[3]==Proposed[1]) || (Decided[3]==Proposed[2]) ||
             (Decided[3]==Proposed[3]) )

/*
 * Formula As Typed: []    ( v0 && v2 && v3 && v1)
 * The Never Claim Below Corresponds
 * To The Negated Formula !([]    ( v0 && v2 && v3 && v1))
 * (formalizing violations of the original)
 */

never {    /* !([]    ( v0 && v2 && v3 && v1)) */
T0_init:
if
:: ((! ((v0)) || (! ((v1)) || (! ((v2)) || ! ((v3)))))) -> goto accept_all
:: (1) -> goto T0_init
fi;
accept_all:
skip
}

#ifdef NOTES
Safety: If a process decides, it decides upon one of the proposed values.



#endif
#ifdef RESULT
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
Depth= 4199872 States=   1e+06 Transitions= 1.4e+06 Nodes=    1437 Memory= 720.588
Depth= 8399872 States=   2e+06 Transitions= 2.8e+06 Nodes=    1437 Memory= 720.588
Depth= 12599872 States=   3e+06 Transitions= 4.2e+06 Nodes=    1437 Memory= 720.588
Depth= 16799872 States=   4e+06 Transitions= 5.6e+06 Nodes=    1434 Memory= 720.588
Depth= 20999872 States=   5e+06 Transitions=   7e+06 Nodes=    1437 Memory= 720.588
Depth= 25199872 States=   6e+06 Transitions= 8.4e+06 Nodes=    1437 Memory= 720.588
Depth= 29399872 States=   7e+06 Transitions= 9.8e+06 Nodes=    1437 Memory= 720.588
error: max search depth too small
pan: error, MA too small, recompile pan.c with -DMA=N with N>380
pan: aborting (at depth 29999999)
pan: error writing consensus.new.trail
(Spin Version 3.3.6 -- 23 November 1999)
```

```
Warning: Search not completed
+ Partial Order Reduction
+ Graph Encoding (-DMA=372)

Full statespace search for:
never-claim          +
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness enabled)
invalid endstates - (disabled by never-claim)

State-vector 408 byte, depth reached 29999999, errors: 1
7.19653e+06 states, stored
3.52736e+06 states, matched
1.07239e+07 transitions (= stored+matched)
1.60418e+07 atomic steps
hash conflicts: 0 (resolved)
(max size 2^24 states)

Stats on memory usage (in Megabytes):
2993.757 equivalent memory usage for states (stored*(State-vector + overhead))
7.654  actual memory usage for states (compression: 0.26%)
720.000 memory used for DFS stack (-m30000000)
727.756 total actual memory usage

6014.12u 10.13s 1:42:18.17 98.1%


#endif
```

C.4 Safe 2

This section presents the specification and verification result of Safety property 2: "all correct processes must decide on he same value" (see Section 4.1).

```
#define pdc(i) ((Decided[0]==0 || Decided[0]==Decided[i]) &&
               (Decided[1]==0 || Decided[1]==Decided[i]) &&
               (Decided[2]==0 || Decided[2]==Decided[i]) &&
               (Decided[3]==0 || Decided[3]==Decided[i]))
#define pdc0 pdc(0)
#define pdc1 pdc(1)
#define pdc2 pdc(2)
#define pdc3 pdc(3)
#define d0 (Decided[0]>0)
#define d1 (Decided[1]>0)
#define d2 (Decided[2]>0)
#define d3 (Decided[3]>0)
/*
 * Formula As Typed: []   ( (d0 -> pdc0) && (d1 -> pdc1) && (d2 ->pdc2) && (d3 -> pdc3))
 * The Never Claim Below Corresponds
 * To The Negated Formula !([]   ( (d0 -> pdc0) && (d1 -> pdc1) && (d2 ->pdc2) &&
                                  (d3 -> pdc3)))
 * (formalizing violations of the original)
 */

never {    /* !([]   ( (d0 -> pdc0) && (d1 -> pdc1) && (d2 ->pdc2) && (d3 -> pdc3))) */
T0_init:
if
:: (((! ((pdc1)) && ((((! ((pdc1)) && (d1)) || ((! ((pdc2)) &&
(d2)) || (! ((pdc3)) && (d3)))) && ((! ((pdc0)) && (d0)) || ((!
((pdc1)) && (d1)) || ((! ((pdc2)) && (d2)) || (! ((pdc3)) && (d3))))))
&& (d1))) || ((! ((pdc2)) && ((((! ((pdc2)) && (d2)) || (! ((pdc3)) &&
(d3))) && (((! ((pdc1)) && (d1)) || ((! ((pdc2)) && (d2)) || (!
((pdc3)) && (d3)))) && ((! ((pdc0)) && (d0)) || ((! ((pdc1)) && (d1))
|| ((! ((pdc2)) && (d2)) || (! ((pdc3)) && (d3))))))) && (d2))) || ((!
((pdc3)) && ((((! ((pdc2)) && (d2)) || (! ((pdc3)) && (d3))) && (((!
((pdc1)) && (d1)) || ((! ((pdc2)) && (d2)) || (! ((pdc3)) && (d3))))
&& ((! ((pdc0)) && (d0)) || ((! ((pdc1)) && (d1)) || ((! ((pdc2)) &&
(d2)) || (! ((pdc3)) && (d3))))))) && (d3))) || (! ((pdc0)) && ((d0)
&& ((! ((pdc0)) && (d0)) || ((! ((pdc1)) && (d1)) || ((! ((pdc2)) &&
(d2)) || (! ((pdc3)) && (d3))))))))))) -> goto accept_all
:: (1) -> goto T0_init
fi;
accept_all:
skip
}

#ifdef NOTES
Safety: All processes decide upon the same value.


#endif
#ifdef RESULT
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
Depth= 4199872 States=   1e+06 Transitions= 1.4e+06 Nodes=    1437 Memory= 720.588
```

```
Depth= 8399872 States=   2e+06 Transitions= 2.8e+06 Nodes=    1437 Memory= 720.588
Depth= 12599872 States=   3e+06 Transitions= 4.2e+06 Nodes=    1437 Memory= 720.588
Depth= 16799872 States=   4e+06 Transitions= 5.6e+06 Nodes=    1434 Memory= 720.588
Depth= 20999872 States=   5e+06 Transitions=   7e+06 Nodes=    1437 Memory= 720.588
Depth= 25199872 States=   6e+06 Transitions= 8.4e+06 Nodes=    1437 Memory= 720.588
Depth= 29399872 States=   7e+06 Transitions= 9.8e+06 Nodes=    1437 Memory= 720.588
error: max search depth too small
pan: error, MA too small, recompile pan.c with -DMA=N with N>380
pan: aborting (at depth 29999999)
pan: error writing consensus.new.trail
(Spin Version 3.3.6 -- 23 November 1999)
Warning: Search not completed
+ Partial Order Reduction
+ Graph Encoding (-DMA=372)

Full statespace search for:
never-claim          +
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness enabled)
invalid endstates - (disabled by never-claim)

State-vector 408 byte, depth reached 29999999, errors: 1
7.19653e+06 states, stored
3.52736e+06 states, matched
1.07239e+07 transitions (= stored+matched)
1.60418e+07 atomic steps
hash conflicts: 0 (resolved)
(max size 2^24 states)

Stats on memory usage (in Megabytes):
2993.757 equivalent memory usage for states (stored*(State-vector + overhead))
7.654  actual memory usage for states (compression: 0.26%)
720.000 memory used for DFS stack (-m30000000)
727.756 total actual memory usage

5951.88u 10.07s 1:40:35.60 98.7%


#endif
```

C.5 Safe 3

This section presents the specification and verification result for Safety property 3: "a decision must be made within n rounds, where n is the number of consensus processes" (see section 4.1).

```
#define r0 (round[0]<NoP+1)
#define r1 (round[1]<NoP+1)
#define r2 (round[2]<NoP+1)
#define r3 (round[3]<NoP+1)
/*
 * Formula As Typed: []  (r0 && r1 && r2 && r3)
 * The Never Claim Below Corresponds
 * To The Negated Formula !([]  (r0 && r1 && r2 && r3))
 * (formalizing violations of the original)
 */

never {    /* !([]  (r0 && r1 && r2 && r3)) */
T0_init:
if
:: ((! ((r0)) || (! ((r1)) || (! ((r2)) || ! ((r3)))))) -> goto accept_all
:: (1) -> goto T0_init
fi;
accept_all:
skip
}

#ifdef NOTES
Verify the claim that process will decide within NoP rounds.

#endif
#ifdef RESULT
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
Depth= 4199872 States=   1e+06 Transitions= 1.4e+06 Nodes=    1513 Memory= 720.627
Depth= 8399872 States=   2e+06 Transitions= 2.8e+06 Nodes=    1513 Memory= 720.627
Depth= 12599872 States=   3e+06 Transitions= 4.2e+06 Nodes=    1513 Memory= 720.627
Depth= 16799872 States=   4e+06 Transitions= 5.6e+06 Nodes=    1510 Memory= 720.627
Depth= 20999872 States=   5e+06 Transitions=   7e+06 Nodes=    1513 Memory= 720.627
Depth= 25199872 States=   6e+06 Transitions= 8.4e+06 Nodes=    1513 Memory= 720.627
Depth= 29399872 States=   7e+06 Transitions= 9.8e+06 Nodes=    1513 Memory= 720.627
error: max search depth too small
Depth= 29999999 States=   8e+06 Transitions= 1.74251e+07 Nodes= 2646892 Memory= 820.471
Depth= 29999999 States=   9e+06 Transitions= 2.16149e+07 Nodes= 4203632 Memory= 879.455
Depth= 29999999 States=   1e+07 Transitions= 2.76496e+07 Nodes= 5353866 Memory= 930.453
Depth= 29999999 States= 1.1e+07 Transitions= 3.48434e+07 Nodes= 5199973 Memory= 930.453
Depth= 29999999 States= 1.2e+07 Transitions= 3.7939e+07 Nodes= 4988745 Memory= 930.453
Depth= 29999999 States= 1.3e+07 Transitions= 4.57142e+07 Nodes= 5485744 Memory= 930.555
Depth= 29999999 States= 1.4e+07 Transitions= 5.01931e+07 Nodes= 4293712 Memory= 930.555
Depth= 29999999 States= 1.5e+07 Transitions= 5.60356e+07 Nodes= 5482863 Memory= 931.477
Depth= 29999999 States= 1.6e+07 Transitions= 6.29276e+07 Nodes= 5526447 Memory= 931.477
pan: out of memory
9.43663e+08 bytes used
102404 bytes more needed
9.43718e+08 bytes limit (2^MEMCNT)
hint: to reduce memory, recompile with
  -DCOLLAPSE # good, fast compression, or
```

```
  -DHC # hash-compaction, approximation
  -DBITSTATE # supertrace, approximation
(Spin Version 3.3.6 -- 23 November 1999)
Warning: Search not completed
+ Partial Order Reduction
+ Graph Encoding (-DMA=410)

Full statespace search for:
never-claim         +
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness enabled)
invalid endstates - (disabled by never-claim)

State-vector 420 byte, depth reached 29999999, errors: 0
1.61171e+07 states, stored
4.7929e+07 states, matched
6.40461e+07 transitions (= stored+matched)
1.31467e+08 atomic steps
hash conflicts: 0 (resolved)
(max size 2^24 states)

Stats on memory usage (in Megabytes):
6898.110 equivalent memory usage for states (stored*(State-vector + overhead))
223.560 actual memory usage for states (compression: 3.24%)
720.000 memory used for DFS stack (-m30000000)
943.663 total actual memory usage
25022.24u 19.97s 6:58:41.87 -71.-2%


#endif
```

C.6 Safe 4

This section presents the verification result of Safety property 4: “the message channels are never full and valid end states are eventually obtained” (see Section 4.1).

```
Depth= 3199873 States=   1e+06 Transitions= 1.4e+06 Nodes=    1507 Memory= 720.626
Depth= 6399873 States=   2e+06 Transitions= 2.8e+06 Nodes=    1507 Memory= 720.626
Depth= 9599873 States=   3e+06 Transitions= 4.2e+06 Nodes=    1507 Memory= 720.626
Depth= 12799873 States=   4e+06 Transitions= 5.6e+06 Nodes=    1504 Memory= 720.626
Depth= 15999873 States=   5e+06 Transitions=   7e+06 Nodes=    1507 Memory= 720.626
Depth= 19199873 States=   6e+06 Transitions= 8.4e+06 Nodes=    1507 Memory= 720.626
Depth= 22399873 States=   7e+06 Transitions= 9.8e+06 Nodes=    1507 Memory= 720.626
Depth= 25599873 States=   8e+06 Transitions= 1.12e+07 Nodes=    1504 Memory= 720.626
Depth= 28799873 States=   9e+06 Transitions= 1.26e+07 Nodes=    1507 Memory= 720.626
error: max search depth too small
Depth= 29999999 States=   1e+07 Transitions= 1.92028e+07 Nodes= 2043591 Memory= 797.531
Depth= 29999999 States= 1.1e+07 Transitions= 2.97743e+07 Nodes= 3480838 Memory= 860.100
Depth= 29999999 States= 1.2e+07 Transitions= 3.84516e+07 Nodes= 3570010 Memory= 860.100
Depth= 29999999 States= 1.3e+07 Transitions= 4.73283e+07 Nodes= 2445696 Memory= 860.100
Depth= 29999999 States= 1.4e+07 Transitions= 5.77412e+07 Nodes= 3645595 Memory= 860.202
Depth= 29999999 States= 1.5e+07 Transitions= 6.61998e+07 Nodes= 3726862 Memory= 869.009
Depth= 29999999 States= 1.6e+07 Transitions= 7.62698e+07 Nodes= 3504481 Memory= 869.009
Depth= 29999999 States= 1.7e+07 Transitions= 8.5544e+07 Nodes= 4093613 Memory= 878.021
Depth= 29999999 States= 1.8e+07 Transitions= 9.39876e+07 Nodes= 3281579 Memory= 878.021
Depth= 29999999 States= 1.9e+07 Transitions= 1.04902e+08 Nodes= 4444952 Memory= 886.827
Depth= 29999999 States=   2e+07 Transitions= 1.13283e+08 Nodes= 4250433 Memory= 886.930
Depth= 29999999 States= 2.1e+07 Transitions= 1.26498e+08 Nodes= 4934812 Memory= 908.742
Depth= 29999999 States= 2.2e+07 Transitions= 1.40627e+08 Nodes= 4874616 Memory= 910.278
Depth= 29999999 States= 2.3e+07 Transitions= 1.5544e+08 Nodes= 5143886 Memory= 919.494
Depth= 29999999 States= 2.4e+07 Transitions= 1.71111e+08 Nodes= 5439788 Memory= 930.759
pan: out of memory
9.43662e+08 bytes used
102404 bytes more needed
9.43718e+08 bytes limit (2^MEMCNT)
hint: to reduce memory, recompile with
  -DCOLLAPSE # good, fast compression, or
  -DHC # hash-compaction, approximation
  -DBITSTATE # supertrace, approximation
(Spin Version 3.3.6 -- 23 November 1999)
Warning: Search not completed
+ Partial Order Reduction
+ Graph Encoding (-DMA=410)

Full statespace search for:
never-claim          - (none specified)
assertion violations +
cycle checks        - (disabled by -DSAFETY)
invalid endstates +

State-vector 412 byte, depth reached 29999999, errors: 0
2.47398e+07 states, stored
1.57465e+08 states, matched
1.82204e+08 transitions (= stored+matched)
3.09385e+08 atomic steps
hash conflicts: 0 (resolved)
(max size 2^24 states)
```

```
Stats on memory usage (in Megabytes):
10390.719 equivalent memory usage for states (stored*(State-vector + overhead))
223.559 actual memory usage for states (compression: 2.15%)
720.000 memory used for DFS stack (-m30000000)
943.662 total actual memory usage
```