Verifying correctness of a concurrent linked list insertion algorithm

Mrityunjay Kumar
mrityunjay.k@research.iiit.ac.in
IIIT Hyderabad

[ ]

Abstract

Concurrent algorithms are hard to design correctly. One of the key reasons is that it is easy to make mistakes when specifying the algorithm, and such a mistake at spec level can be almost impossible to find out via debugging. Debugging is most suited to catch implementation bugs, not specification bugs. To ensure the specification is correct, a model checker can be used that verifies the specification against a model of the algorithm. SPIN is one of the leading model checkers in use. We report the results of the use of SPIN to verify the model of the concurrent linked list insertion algorithm.

Usual implementations of such an algorithm uses locks to achieve concurrency. However, we use an algorithm that leverages the notion of a controller for orchestration amongst processes to ensure there is safe access to the shared resources - the nodes in the linked list in this case. [1] describes such an algorithm with fine-grained synchronization, adapted from [2]. Transition Systems[3] are used to describe these processes and the controller. Defining them as inter-connected systems allows the processes to focus on their actual semantics and let the controller worry about concurrency.

We report on the experience in creating a model for this algorithm and then proving its correctness using SPIN model checker. We also point out a gap uncovered during the verification and suggest other recommendations to make the algorithm more robust.

1 Introduction

Creating and implementing algorithms for concurrent systems hard, as evidenced by bugs discovered in well-known implementations[4], [5]. There are many reasons for this, including the fact that most computer science courses teach concurrent programming as a special skill, most of computer science is taught through sequential programming paradigm. This is in spite of the fact that most of the systems in computer science are concurrent - operating systems, networks, etc.

Two key reasons that make this hard are:

1. Programming complexity
2. Testing complexity

This paper discusses the approaches for addressing these complexities by demonstrating a simple concurrent algorithm - inserting numbers in a sorted linked list(Figure 1). This is an example of concurrent data structure access problem, and demonstrates the challenges very well. We present the challenges and our solution to these in following subsections.

For simplicity of description (without losing generality), we assume that only positive numbers are being inserted, and the list always has a head and a tail with 0 and ∞. As processes add nodes to the linked list, these 2 conditions must hold at all times.

- There must be no duplicate entries in the linked list.
- The numbers in the list must be arranged in strictly increasing order, from head to tail.

We call these conditions invariants.

A concurrent algorithm is considered correct if satisfies two key properties[6]:

1. Safety : Something (bad) will *not* happen
2. Liveness : Something (good) *must* happen

This can be restated in our context:

1. Safety : The invariants (listed above) **will** be preserved during the operation of the algorithm
2. Liveness : A process that tries to insert a number **will** eventually succeed in doing so

To show that our suggested algorithm is correct, we need to show that it satisfies these properties.

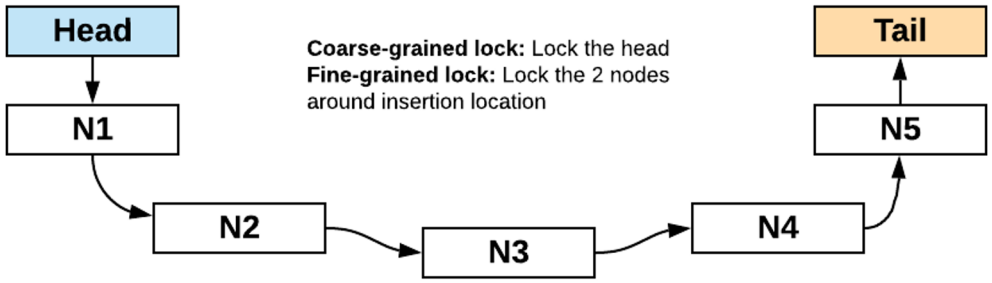

Figure 1: Linked List

1.1 Programming Complexity

Most concurrent algorithms implement concurrency controls by leveraging operating system constructs like critical section, locks and mutexes[7]. This poses two challenges:

- It increases the complexity of the program and the cognitive overload on the developer who now has to address these concerns around concurrency control, in addition to the domain logic of the program.
- It makes the program less modular since it now depends on the construct(s) available on the platform. Any change in the platform or the construct forces a change in the program - thus making it rigid and less adaptable.

There is also emphasis on designing lock-free algorithms as well, algorithms that use various non-blocking progress conditions - wait-freedom, lock-freedom, or obstruction-freedom[7], but these algorithms tend to be more complex as well. An approach to design simpler, lock-free algorithms is to make the programs controller-driven - programs give up their intelligence about concurrency and rely on an external controller to let them know whether it is safe to proceed to the next step or not. This is akin to a traffic controller (or light) who doesn't understand what each car intends to accomplish, but still is able to regulate the access to shared resource (roads), just by controlling who can go which way and who can't. Feedback control and transition systems are key concepts that are naturally suited to this kind of controller mechanism.

Choppella et.al. in [1] present a feedback control approach to concurrency problems that takes this approach to make the solutions more modular and doesn't require changes in the processes to make them concurrency-aware.

1.2 Testing complexity

A challenge in any software is to ensure the correctness of the algorithm. While defects in a sequential algorithm can be caught during implementation testing, it is almost impossible to do so for concurrent algorithms because of a large number of permuations possible through the code paths due to the concurrent nature of the process interactions. To ensure correctness, we need to ensure that the specification of the algorithm is correct. This can be achieved by creating a model of the solution and then proving its correctness using one of the model checkers.

Model checking is an important area of research and development. Formal proof of correctness of an algorithm before investing significant time and money in actual implementation can be cost-effective. In some cases like mission-critical software where cost of a bug in production code is too high, it may be the only reliable way of ensuring quality . In other cases, it can help identify issues that are almost impossible to catch via testing cycle during implementation phase.

Specifically, we chose SPIN to perform model checking of this algorithm.

1.3 Hypothesis

Our hypothesis is that the feedback-controlled back algorithm satisfies the correctness conditions of concurrent algorithms - safety and starvation-freedom. We validate this by programming the model in Promela (specification language in SPIN) and then running the verification on a set of test data patterns using SPIN.

1.4 Paper outline

Section 2 presents related work in the field of concurrent programming. Section 3 provides more details on the concurrent linked list solution - with locks and with feedback control. Section 4 presents the SPIN model checker tool that we use for ensuring that the specification of the feedback control solution is correct. It also describes the design of the experiment and the research questions that we want to answer. Section 5 presents implementation details as well as the finding from the implementation and verification activities. Section 6 presents the conclusion and future work.

2 Related Work

Concurrent algorithms are hard to get right, well-known algorithms have been shown to have bugs in specification[4] and formal techniques used to discover these bugs[5]. Therefore it makes sense to use tools that can formally verify the correctness. it can be done using a theorem solvers like PVS [8]–[10] (interestingly, these also used SPIN for verifying the model in addition to using PVS) or a model checker like SPIN[11]. [12] compared 6 model checking tools, and found Alloy and SPIN to be better than others. Amazon has adopted TLA+ [13]. [14] compared Alloy and SPIN on a single, complex networking project and concluded that while Alloy is marginally better for graph-type verification needs, SPIN/Promela was better in terms of writing the functional specification. Work has also been done to verify the correctness of the implementation once the model has been proven to be correct[15].

We picked SPIN for verifying the model of a concurrent linked list insertion algorithm.

3 Solving programming complexity

As described above, programming complexity comes from the fact that it becomes harder for the programmer to understand a new concept, and it makes the program dependent on the specifics of the concurrency constructs, thus sacrificing flexibility. In this section, we take the example of the concurrent linked list and presents the solution in the traditional way (using locks) and then using feedback control. The paper doesn't focus on demonstrating that the latter solution is more modular than former, it relies on the original paper to make the claim.

3.1 Design with locks

Here is the algorithm from the book that implements the solution in the traditional manner - using locks (Figure 2).

```
    public boolean add(T item) {
    int key = item.hashCode();
    head.lock();
    Node pred = head;
    try {
        Node curr = pred.next;
        curr.lock();
        try {
            while (curr.key < key) {
                pred.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock();
            }
            if (curr.key == key) {
                return false;
            }
            Node newNode = new Node(item);
            newNode.next = curr;
            pred.next = newNode;
            return true;
        } finally {
            curr.unlock();
        }
    } finally {
        pred.unlock();
    }
}
```

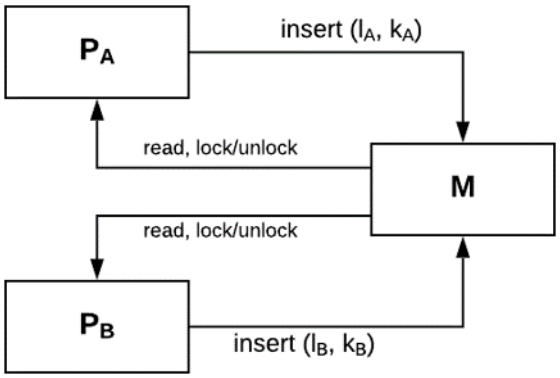

Figure 2: Architecture with lock

Here is how the algorithm operates: Each process initially tries to acquire the lock of the head node, and subsequently the next node. In each step, it releases the lock on the back node, and tries to lock the node next to the front node. This continues till it reaches a point where it finds the appropriate position to insert the value. Thus, the operation is designed in such a way that at any point of time, a process that is doing an insert operation requires the lock on just 2 adjacent nodes.

3.2 Design with feedback control

This algorithm presents a safe insertion algorithm for linked list using fine-grained synchronization (Figure 3). It is adapted from the algorithm described in the previous section.

A hub controller is introduced that observes the states of all the processes and computes whether a particular process should proceed to next stage or wait in the current state. Processes only react to the messages from the hub controller and decide to go to next state or stay in the current state according to the message. This keeps the process agnostic of the concurrency control mechanism and they can implement their actual functionality as if there is no concurrency involved. Each of the process and the hub controller are designed as transition systems. Following sections describe these systems.

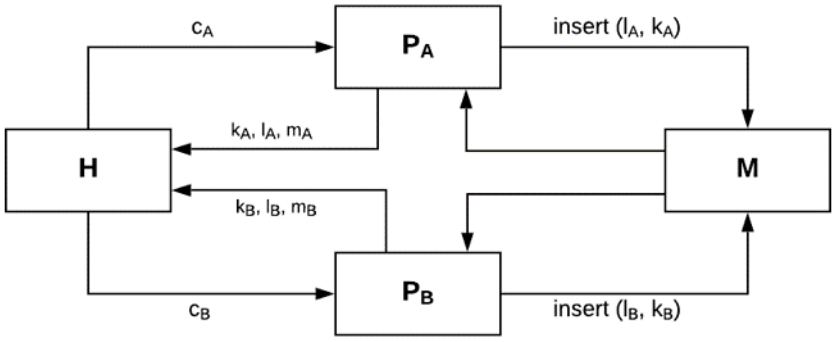

Figure 3: Architecture with feedback control

3.2.1 Transition System of a process

The transition system of a process can be written as $< X_P, X_P^0, U_P, f_P >$, where

$$
\begin{aligned}
X_P &: N_\perp \times Loc_\perp \times Mode \\
X_P^0 &: (\perp, \perp, Done) \\
U_P &: Cmd \times Oper_\perp \\
f_P &: X_P \times U_P \rightarrow X_P
\end{aligned}
$$

The transition function f_P is defined as follows:
f_P ((k, l, m), (c, o))

```
= (k, l, m) if c = 0 or (m = Done and o = ⊥)
= (k', head, Search) if m = Done and o = insert(k')
= (k, l.next, m), if m = Search and l.next.key < key
= (k, l, Insert), if m = Search and l.next.key > key
= (⊥, ⊥, Done), if m = Insert
```

3.2.2 Transition system of the Hub Controller

The transition system of the hub controller can be written as $< X_H, X_H^0, U_H, f_H >$, where

$$
\begin{aligned}
X_H &: Cmd \times Cmd \\
X_H^0 &: (1, 1) \\
U_H &: (N_\perp \times Loc_\perp \times Mode)^2 \\
f_H &: X_H \times U_H \rightarrow X_H
\end{aligned}
$$

The transition function can be written as follows:
$f_H((c_A, c_B), ((k_A, l_A, m_A), (k_B, l_B, m_B))$

$= (0, 1),\ if\ m_A = m_B = Insert,\ l_A = l_B\ and\ k_A < k_B$
$= (1, 0),\ if\ m_A = m_B = Insert,\ l_A = l_B\ and\ k_A > k_B$
$= (1, 1),\ otherwise$

4 Solving testing complexity

As mentioned above, ensuring the correctness of the model is key in concurrent algorithms. In the example problem described in the previous section, we want to prove the correctness of its model using a model checker tool. To do this, we implement the model of the transition systems described above and then verify it for safety and starvation-freedom (the properties of a correct concurrent algorithm model). We have selected SPIN model checker as our tool. Our hypotheses are as follows:

1. It is always true that the list items are in ascending order (safety)

2. It is always true that thre are no duplicates in the list (safety)

3. Eventually, a process will get the chance to insert an item if it has an item to insert (liveness)

4. The model is not biased towards any process when allowing to insert an item in the list (fairness)

4.1 SPIN Model Checker

SPIN[11] is one of the foremost model checkers[16] for distributed systems.

> SPIN is a general tool for verifying the correctness of concurrent software models in a rigorous and mostly automated fashion. It was written by Gerard J. Holzmann and others in the original Unix group of the Computing Sciences Research Center at Bell Labs, beginning in 1980. The software has been available freely since 1991, and continues to evolve to keep pace with new developments in the field. - Wikipedia

SPIN uses PROMELA (a restrictive, C-like programming language) for implementing the model to be verified. To verify, SPIN generates a C program specific to the given model, which when executed verifies the program. This approach makes the verification process very efficient. Properties to be verified are expressed as Linear Temporal Logic (LTL) formulas. SPIN also operates as a simulator (one random path) to help during development of the model. It also offers interactive simulation, letting you choose simulation paths, as well as guided simulation to trace error paths.

4.1.1 Implementation in PROMELA

To implement this model, we had to work around a few restrictions in PROMELA.

1. PROMELA doesn't have dynamic allocation of memory, so we had to implement an array-based linked list using pre-allocated array as a memeory manager.

2. There is no support for functions, so the implementation had to rely on inlines and global variables.

3. LTL can evaluate only one statement when checking for correctness, so all safety invariants had to be transformed into a single variable check.

The code repository can be found in Appendix A.

4.1.2 Verification in SPIN

To verify, we have to write the safety and liveness properties in SPIN LTL.

4.1.2.1 Safety property in SPIN LTL

We had to map the safety invariants into a single-statement check. To do that, following strategy was employed: We construct a number that is the product of the difference of consecutive numbers in the linked list. If the numbers in the list are not in order, there will be at least one consecutive pair that is not in order, which means their difference will be negative. Similarly, if there are duplicate numbers, they will either occur back to back (which means one of the differences will be 0) or at different places which means the list is out of order (and hence one of the difference will be negative). Every time we insert, we update this number using the following logic: Assume we are inserting c between a and b. So the number will have a factor (b-a) already. We divide by (b-a) and multiply by (c - a) and (b - a) to get the new value of the number.

This is not a fool-proof approach - for ex, 2 out of order pairs can still make the product positive. We assume that as soon as there is a path that causes one such pair to be formed, SPIN will catch it, we will never have a path in which there is more than one violation. Also, the sequence of division and multiplication is important since PROMELA doesn't support floating-point arithmatic. Under these assumptions (which is ensured by SPIN), above approach works.

Given this, safety property becomes $(critical\_val > 0)$ which is used as follows in SPIN:

```
Command-line:
$ spin -a -f '![](critical_val>0)' <pml file>
$ gcc -DSAFETY -o pan pan.c
$ ./pan
```

See Appendix C.1, C.2, and C.3 for the output of Safety verification.

4.1.2.2 Liveness property in SPIN LTL

To verify liveness property, we need to show that if a process is ready to insert an element in the list, it will eventually get the chance to do so. To do this, we keep a counter which holds the number of inserts done by each process. If liveness property is to hold, the count should become ¿ 0 eventually, for each of the process. In fact, it should be same as the number of inserts attempted if the program runs for sufficiently long time.

Given this, liveness property becomes $(insert_counter[0] > 0)(insert_counter[1] > 0)$ which is used as follows in SPIN:

```
Command-line:
$ spin -a -f '!<>((insert_counter[0]>0)&&(insert_counter[1]>0))' <pml file>
$ gcc -o pan pan.c
$ ./pan -a -f
```

See Appendix C.1, C.2, and C.3 for the output of Safety verification.

4.1.2.3 Fairness property

Fairness comes from strategy. Here is the strategy in the hub control to ensure processes get a fair chance: whenever there is a situation where only one of the process can be given go-ahead while both of them are ready, we probabilistically pick the process to give go-ahead. This ensures that both the processes have equal chance and no bias exists in the Hub logic.

4.1.2.4 Different test data

To verify, we run the verification through 3 types of test data with each process:

1. Insert same numbers in the same sequence
2. Insert some same numbers in the same sequence, some different ones

3. Insert different numbers in different sequences

See Appendix C for test data used.

5 Analysis and Results

A standard linux machine was used for development and running the verification on SPIN. Appendix sections have the detailed output from the execution of the program and verification.

There were two kinds of results. We identified a missing scenario which caused the verification to fail. We also identified a few areas where the specification could be more clear.

5.1 Missing scenario

Verification discovered a trace which resulted in one of the safety invariants to fail - the trace created an out of order list. Here is the scenario when it happened:

1. P_A was ready to insert number 1 right after the head (so its current location was HEAD). P_B was ready to proceed on searching to find the place to insert the number 2.

2. According to the algorithm, both of them were given go ahead ($C_A = 1, C_B = 1$).

3. P_A inserted 1 right after HEAD in the next cycle.

4. While P_A was inserting, P_B searched and found that it needs to insert right after HEAD as well (which was true before P_A inserted 1, but not after)

5. P_B got the go-ahead to insert and ended up in a state where it would have inserted 2 before 1, thus violating the order. SPIN terminated the verification run.

Appendix B shows the trace where safety verification fails.

This can be fixed by adding the following transition to the hub controller:

$= (0, 1),\ if\ l_A = l_B\ and\ m_A = Search,\ m_B = Insert$

$= (1, 0),\ if\ l_A = l_B\ and\ m_A = Insert,\ m_B = Search$

In other words, if two processes are on the same location, and one of them has Insert as the mode, the one with Search has to wait.

We can try to be aggressive and let P_B with Search mode go ahead if $k_A > k_B$. However, given that search and insert will proceed in parallel, there is no guarantee that the lookup (to compare k_B with next node value) will happen after or before P_A has inserted the node. So a safer approach is what is described above.

We made this change and ran the verification again which passed.

5.2 Unclear specification

There were a few areas where the specification was not clear, which made modeling difficult. We assumed a path and proceeded, and those are offered as recommendations here.

5.2.1 Bootstrapping

It is not clear how does this system start. One option is for the hub to send a message to start each of the process. However, this requires the hub to know which processes are connected to it. Another option could be to define a hub discovery and registration process using some well-known channel.

For our model, we assume that a god controller notifies the hub controller about the processes that need to be controlled, and hub controller starts by sending a 1 to each of these processes.

Observations	Assumptions/Recommendations
Processing control	The process should have the control over what to insert
Bootstrapping	Assume the presence of a god controller
Shutdown	Process de-registers from the hub when it is done

Table 1: Observations and Recommendations

5.2.2 Processing

In the current definition of the algorithm, the hub tells the process which number to insert next. It is not clear why hub has control over this. Ideally, the process should control this and the hub only controls whether the process can go ahead with this action or not.

In the implementation of the model, we assume that the hub doesn't control which number is to be inserted, it just says whether to go ahead or not go ahead. This seems to be a more realistic description of the problem.

5.2.3 Shutdown

It is not clear from the system specification what happens when one of the process is done - it has nothing else to do. This arises because of our assumption of the point 1 above - that processes control what they need to do, not the controller. One option is for the process to send an OVER message so that controller doesn't keep sending it messages. Another option could be to let the controller keep sending the INSERT message but the client knows it has nothing to insert, so it stays in DONE state.

We assume the first option in our model.

6 Conclusion and Future Work

We started this work by analyzing the feedback control based algorithm and creating a model for the same. This model was implemented in PROMELA, the language for SPIN model checker. Verification was done on different data sets for the safety and liveness properties, and a few recommendations provided (Table 1). The results suggest that the algorithm is safe (after the identified gap is plugged in the specification). In future, work should be done to extend the model to more processes, as well as other operations on the linked list (delete, update) to demonstrate the applicability of feedback control based algorithms to all the needs of a concurrent data structure like linked list.

Other model checkers like TLA+ or Alloy can be tried to see the ease of model implementation and verification across various tools.

References

[1] V. Choppella, K. Viswanath, A. Sanjeev, and B. Jayaraman, "Modular feedback control approach to concurrency," p. 32,

[2] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2008, 508 pp., OCLC: ocn181602117, ISBN: 978-0-12-370591-4.

[3] P. Tabuada, *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media, 2009.

[4] M. M. Michael and M. L. Scott, "Correction of a memory management method for lock-free data structures.," ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE, Tech. Rep., 1995.

[5] S. Burckhardt, R. Alur, and M. M. Martin, "Checkfence: Checking consistency of concurrent data types on relaxed memory models," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007, pp. 12–21.

[6] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE transactions on software engineering*, no. 2, pp. 125–143, 1977.

[7] M. Moir and N. Shavit, *Concurrent data structures.*

[8] R. Colvin and L. Groves, "Formal verification of an array-based nonblocking queue," in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, Shanghai: IEEE, 2005, pp. 507–516, ISBN: 978-0-7695-2284-5. DOI: 10.1109/ICECCS.2005.49. [Online]. Available: https://ieeexplore.ieee.org/document/1467933/ (visited on 05/05/2020).

[9] R. Colvin, L. Groves, V. Luchangco, and M. Moir, "Formal verification of a lazy concurrent list-based set algorithm," in *Computer Aided Verification*, T. Ball and R. B. Jones, Eds., red. by D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, and G. Weikum, vol. 4144, Series Title: Lecture Notes in Computer Science, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 475–488, ISBN: 978-3-540-37406-0 978-3-540-37411-4. DOI: 10.1007/11817963_44. [Online]. Available: http://link.springer.com/10.1007/11817963_44 (visited on 05/05/2020).

[10] P. Černy, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur, "Model checking of linearizability of concurrent list implementations," in *International Conference on Computer Aided Verification*, Springer, 2010, pp. 465–479.

[11] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997, ISSN: 00985589. DOI: 10.1109/32.588521. [Online]. Available: http://ieeexplore.ieee.org/document/588521/ (visited on 05/04/2020).

[12] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar, "Comparison of model checking tools for information systems," in *International Conference on Formal Engineering Methods*, Springer, 2010, pp. 581–596.

[13] C. Newcombe, "Why amazon chose tla+," in *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, Springer, 2014, pp. 25–39.

[14] P. Zave, "A practical comparison of alloy and spin," *Formal Aspects of Computing*, vol. 27, no. 2, pp. 239–253, 2015.

[15] S. Tasiran and S. Qadeer, "Runtime refinement checking of concurrent data structures," *Electronic Notes in Theoretical Computer Science*, vol. 113, pp. 163–179, Jan. 2005, ISSN: 15710661. DOI: 10.1016/j.entcs.2004.01.028. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1571066104052582 (visited on 05/04/2020).

[16] M. (Ben-Ari, "A primer on model checking," *ACM Inroads*, vol. 1, no. 1, pp. 40–47, Mar. 2010, ISSN: 2153-2184, 2153-2192. DOI: 10.1145/1721933.1721950. [Online]. Available: https://dl.acm.org/doi/10.1145/1721933.1721950 (visited on 05/04/2020).

Appendix

A Code for the verification

The code can be found here: Github

B Example flow for the missing scenario

```
CP(1) Started.
CP(2) Started.
HC Started with 2 processes 1 and 2.
HC: Message sent (1, O_INSERT) to CP(1)
HC: Message sent (1, O_INSERT) to CP(2)
CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(50,1,S_SEARCH)
CP(2): Message sent (50, 1, S_SEARCH)
CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(50,1,S_SEARCH)
CP(1): Message sent (50, 1, S_SEARCH)
HC: [(1,O_INSERT,0),(1,O_INSERT,0)]----[(50,1,S_SEARCH),(50,1,S_SEARCH)]---->[(1,
    O_CONTINUE,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): (50,1,S_SEARCH)----(1,O_CONTINUE)---->(50,1,S_INSERT)
CP(2): Message sent (50, 1, S_INSERT)
CP(1): (50,1,S_SEARCH)----(1,O_CONTINUE)---->(50,1,S_INSERT)
CP(1): Message sent (50, 1, S_INSERT)
HC: [(1,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(50,1,S_INSERT),(50,1,S_INSERT)
    ]---->[(1,O_CONTINUE,0),(0,O_CONTINUE,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (0, O_CONTINUE) to CP(2)
CP(2): (50,1,S_INSERT)----(0,O_CONTINUE)---->(50,1,S_INSERT)
CP(2): Message sent (50, 1, S_INSERT)              CP(1): Critical Value (before):
    255, a: 0, b: 255,  c:50
CP(1): (50,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(1): Message sent (0, 0, S_IDLE)
HC: [(1,O_CONTINUE,0),(0,O_CONTINUE,0)]----[(0,0,S_IDLE),(50,1,S_INSERT)]---->[(1,
    O_INSERT,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_INSERT) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): 50 already exists at index 3. Skipping the insert
CP(2): (50,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(2): Message sent (0, 0, S_IDLE)
CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(51,1,S_SEARCH)
CP(1): Message sent (51, 1, S_SEARCH)
HC: [(1,O_INSERT,0),(1,O_CONTINUE,0)]----[(51,1,S_SEARCH),(0,0,S_IDLE)]---->[(1,
    O_CONTINUE,0),(1,O_INSERT,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_INSERT) to CP(2)
CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(51,1,S_SEARCH)
CP(2): Message sent (51, 1, S_SEARCH)
CP(1): (51,1,S_SEARCH)----(1,O_CONTINUE)---->(51,3,S_SEARCH)
CP(1): Message sent (51, 3, S_SEARCH)
HC: [(1,O_CONTINUE,0),(1,O_INSERT,0)]----[(51,3,S_SEARCH),(51,1,S_SEARCH)]---->[(1,
    O_CONTINUE,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): (51,1,S_SEARCH)----(1,O_CONTINUE)---->(51,3,S_SEARCH)
CP(2): Message sent (51, 3, S_SEARCH)
CP(1): (51,3,S_SEARCH)----(1,O_CONTINUE)---->(51,3,S_INSERT)
CP(1): Message sent (51, 3, S_INSERT)
HC: [(1,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(51,3,S_INSERT),(51,3,S_SEARCH)
    ]---->[(1,O_CONTINUE,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): (51,3,S_SEARCH)----(1,O_CONTINUE)---->(51,3,S_INSERT)
CP(2): Message sent (51, 3, S_INSERT)              CP(1): Critical Value (before):
    10250, a: 50, b: 255,  c:51
CP(1): (51,3,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(1): Message sent (0, 0, S_IDLE)
```

```
HC: [(1,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(0,0,S_IDLE),(51,3,S_INSERT)]---->[(1,
    O_INSERT,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_INSERT) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): 51 already exists at index 4. Skipping the insert
CP(2): (51,3,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(2): Message sent (0, 0, S_IDLE)
CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(1,1,S_SEARCH)
CP(1): Message sent (1, 1, S_SEARCH)
HC: [(1,O_INSERT,0),(1,O_CONTINUE,0)]----[(1,1,S_SEARCH),(0,0,S_IDLE)]---->[(1,
    O_CONTINUE,0),(1,O_INSERT,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_INSERT) to CP(2)
CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(2,1,S_SEARCH)
CP(2): Message sent (2, 1, S_SEARCH)
CP(1): (1,1,S_SEARCH)----(1,O_CONTINUE)---->(1,1,S_INSERT)
CP(1): Message sent (1, 1, S_INSERT)
HC: [(1,O_CONTINUE,0),(1,O_INSERT,0)]----[(1,1,S_INSERT),(2,1,S_SEARCH)]---->[(1,
    O_CONTINUE,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): (2,1,S_SEARCH)----(1,O_CONTINUE)---->(2,1,S_INSERT)
CP(2): Message sent (2, 1, S_INSERT)              CP(1): Critical Value (before):
    10200, a: 0, b: 50,  c:1
CP(1): (1,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(1): Message sent (0, 0, S_IDLE)
HC: [(1,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(0,0,S_IDLE),(2,1,S_INSERT)]---->[(1,
    O_INSERT,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_INSERT) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): Critical Value (before): 9996, a: 0, b: 1,

****Trail ends after 939 steps (Critical_val goes negative ****
```

Listing 1: Missing scenario log

C Test Results

We used 3 sets of data as mentioned above. For each set, we generated 3 outputs:

1. Verification of safety property
2. Verification of liveness property
3. A simulation run

The verification runs show 0 errors, thus proving the property. They do show other details around memory requirements and state traversed. The simulation run shows the entire flow of the process and hub steps through print statements. These are large outputs so I am showing only one of them (last sub-section here) as illustration.

C.1 Test Data

```
Data sets for testing.


1. Set 1 - same sequence, same numbers
A: 100, 99, 98, 97, 96, 95
B: 100, 99, 98, 97, 96, 95

2. Set 2 - same sequence, some different numbers
A: 50, 51, 1, 2, 60, 61
B: 50, 51, 2, 3, 60, 61

3. Set 3 - different sequences of numbers
A: 11, 12, 13, 14, 15, 16
B: 16, 15, 14, 13, 12, 11
```

```
15
16
17
```

Listing 2: Test data used

C.2 Set 1 results

```
1  Running safety verification on swf_termpaper.pml
2  warning: for p.o. reduction to be valid the never claim must be stutter-
       invariant
3  (never claims generated from LTL formulae are stutter-invariant)
4
5  (Spin Version 6.5.1 -- 20 December 2019)
6    + Partial Order Reduction
7
8  Full statespace search for:
9    never claim           + (never_0)
10   assertion violations  + (if within scope of claim)
11   cycle checks        - (disabled by -DSAFETY)
12   invalid end states  - (disabled by never claim)
13
14 State-vector 736 byte, depth reached 2197, errors: 0
15      8412 states, stored
16      1611 states, matched
17     10023 transitions (= stored+matched)
18         0 atomic steps
19 hash conflicts:         0 (resolved)
20
21 Stats on memory usage (in Megabytes):
22     6.129 equivalent memory usage for states (stored*(State-vector + overhead))
23     6.983 actual memory usage for states
24   128.000 memory used for hash table (-w24)
25     0.534 memory used for DFS stack (-m10000)
26   135.175 total actual memory usage
27
28
29 unreached in proctype ChildProcess
30   swf_termpaper.pml:230, state 24, "p_state.l = l"
31   swf_termpaper.pml:231, state 25, "p_state.m = m"
32   swf_termpaper.pml:215, state 58, "mnext = S_SEARCH"
33   (3 of 112 states)
34 unreached in proctype HubController
35   swf_termpaper.pml:417, state 29, "c2 = 1"
36   swf_termpaper.pml:420, state 32, "c2 = 0"
37   swf_termpaper.pml:417, state 134, "c2 = 1"
38   swf_termpaper.pml:420, state 137, "c2 = 0"
39   (4 of 235 states)
40 unreached in proctype parent
41   linkedlistlib.pml:47, state 16, "printf('\n--(%d, %d, %d)\n',i,list[i].value,
       list[i].next)"
42   linkedlistlib.pml:49, state 17, "assert((list[i].value!=list[next].value))"
43   linkedlistlib.pml:58, state 25, "i = next"
44   (3 of 85 states)
45 unreached in claim never_0
46   swf_termpaper.pml.nvr:10, state 10, "-end-"
47   (1 of 10 states)
48
49 pan: elapsed time 0.15 seconds
50 pan: rate     56080 states/second
51 Done running safety verification on swf_termpaper.pml
```

Listing 3: Safety verification output

```
Running liveness verification on swf_termpaper.pml
warning: for p.o. reduction to be valid the never claim must be stutter-
    invariant
(never claims generated from LTL formulae are stutter-invariant)

(Spin Version 6.5.1 -- 20 December 2019)
  + Partial Order Reduction

Full statespace search for:
  never claim           + (never_0)
  assertion violations  + (if within scope of claim)
  acceptance   cycles   + (fairness enabled)
  invalid end states  - (disabled by never claim)

State-vector 744 byte, depth reached 380, errors: 0
     1009 states, stored (6379 visited)
     2926 states, matched
     9305 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.743 equivalent memory usage for states (stored*(State-vector + overhead))
    0.934 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  129.413 total actual memory usage


unreached in proctype ChildProcess
  swf_termpaper.pml:230, state 24, "p_state.l = l"
  swf_termpaper.pml:231, state 25, "p_state.m = m"
  swf_termpaper.pml:238, state 30, "newk = 0"
  swf_termpaper.pml:230, state 40, "p_state.l = 0"
  swf_termpaper.pml:231, state 41, "p_state.m = 1"
  swf_termpaper.pml:215, state 58, "mnext = S_SEARCH"
  swf_termpaper.pml:175, state 74, "printf('\nCP(%d): %d already exists at index
     %d. Skipping the insert',id,k,next)"
  swf_termpaper.pml:388, state 112, "-end-"
  (8 of 112 states)
unreached in proctype HubController
  swf_termpaper.pml:417, state 29, "c2 = 1"
  swf_termpaper.pml:420, state 32, "c2 = 0"
  swf_termpaper.pml:433, state 46, "c2 = 0"
  swf_termpaper.pml:437, state 49, "c2 = 1"
  swf_termpaper.pml:502, state 102, "isOver1 = 0"
  swf_termpaper.pml:498, state 103, "((h_input.proc[0].m==S_OVER))"
  swf_termpaper.pml:498, state 103, "else"
  swf_termpaper.pml:512, state 108, "o1 = O_CONTINUE"
  swf_termpaper.pml:508, state 109, "((h_input.proc[0].m==S_IDLE))"
  swf_termpaper.pml:508, state 109, "else"
  swf_termpaper.pml:522, state 113, "h_state.proc[0].o = o1"
  swf_termpaper.pml:523, state 114, "h_state.proc[0].isOver = isOver1"
  swf_termpaper.pml:494, state 115, "c1 = 1"
  swf_termpaper.pml:530, state 118, "ch_HtoP[(id1-1)]!id1,h_send.c,h_send.o"
  swf_termpaper.pml:531, state 119, "printf('\nHC: Message sent (%d, %e) to CP(%
    d)',h_send.c,h_send.o,id1)"
  swf_termpaper.pml:528, state 120, "h_send.c = h_state.proc[0].c"
  swf_termpaper.pml:417, state 134, "c2 = 1"
  swf_termpaper.pml:420, state 137, "c2 = 0"
  swf_termpaper.pml:433, state 151, "c2 = 0"
  swf_termpaper.pml:437, state 154, "c2 = 1"
  swf_termpaper.pml:502, state 207, "isOver1 = 0"
  swf_termpaper.pml:498, state 208, "((h_input.proc[1].m==S_OVER))"
  swf_termpaper.pml:498, state 208, "else"
  swf_termpaper.pml:512, state 213, "o1 = O_CONTINUE"
  swf_termpaper.pml:508, state 214, "((h_input.proc[1].m==S_IDLE))"
  swf_termpaper.pml:508, state 214, "else"
```

```
  swf_termpaper.pml:522, state 218, "h_state.proc[1].o = o1"
  swf_termpaper.pml:523, state 219, "h_state.proc[1].isOver = isOver1"
  swf_termpaper.pml:494, state 220, "c1 = 1"
  swf_termpaper.pml:530, state 223, "ch_HtoP[(id2-1)]!id2,h_send.c,h_send.o"
  swf_termpaper.pml:531, state 224, "printf('\nHC: Message sent (%d, %e) to CP(%
    d)',h_send.c,h_send.o,id2)"
  swf_termpaper.pml:528, state 225, "h_send.c = h_state.proc[1].c"
  swf_termpaper.pml:608, state 235, "-end-"
  (29 of 235 states)
unreached in proctype parent
  linkedlistlib.pml:47, state 16, "printf('\n--(%d, %d, %d)\n',i,list[i].value,
    list[i].next)"
  linkedlistlib.pml:49, state 17, "assert((list[i].value!=list[next].value))"
  linkedlistlib.pml:58, state 25, "i = next"
  linkedlistlib.pml:41, state 51, "printf('\nBegin List\n')"
  linkedlistlib.pml:44, state 53, "next = list[i].next"
  linkedlistlib.pml:47, state 55, "printf('\n--(%d, %d, %d)\n',i,list[i].value,
    list[i].next)"
  linkedlistlib.pml:49, state 56, "assert((list[i].value!=list[next].value))"
  linkedlistlib.pml:51, state 58, "printf('\n--Head Node\n')"
  linkedlistlib.pml:46, state 59, "((i!=HEAD))"
  linkedlistlib.pml:46, state 59, "else"
  linkedlistlib.pml:58, state 64, "i = next"
  linkedlistlib.pml:55, state 65, "((next==TAIL))"
  linkedlistlib.pml:55, state 65, "else"
  linkedlistlib.pml:43, state 67, "(1)"
  linkedlistlib.pml:39, state 71, "i = HEAD"
  linkedlistlib.pml:66, state 73, "printf('\nStart Memory print\n')"
  linkedlistlib.pml:67, state 74, "i = HEAD"
  linkedlistlib.pml:68, state 76, "printf('\nLocation: %d, Value: %d, Next: %d',
    i,list[i].value,list[i].next)"
  linkedlistlib.pml:67, state 77, "i = (i+1)"
  linkedlistlib.pml:65, state 84, "i = HEAD"
  swf_termpaper.pml:626, state 85, "-end-"
  (19 of 85 states)
unreached in claim never_0
  swf_termpaper.pml.nvr:8, state 6, "-end-"
  (1 of 6 states)

pan: elapsed time 0.03 seconds
pan: rate 212633.33 states/second
Done running liveness verification on swf_termpaper.pml
```

Listing 4: Liveness verification output

C.3 Set 2 results

```
Running safety verification on swf_termpaper.pml
warning: for p.o. reduction to be valid the never claim must be stutter-
    invariant
(never claims generated from LTL formulae are stutter-invariant)

(Spin Version 6.5.1 -- 20 December 2019)
  + Partial Order Reduction

Full statespace search for:
  never claim           + (never_0)
  assertion violations  + (if within scope of claim)
  cycle checks        - (disabled by -DSAFETY)
  invalid end states  - (disabled by never claim)

State-vector 736 byte, depth reached 3137, errors: 0
    15122 states, stored
     3019 states, matched
    18141 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         1 (resolved)
```

```
Stats on memory usage (in Megabytes):
   11.018 equivalent memory usage for states (stored*(State-vector + overhead))
   11.672 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  139.862 total actual memory usage


unreached in proctype ChildProcess
  swf_termpaper.pml:230, state 24, "p_state.l = l"
  swf_termpaper.pml:231, state 25, "p_state.m = m"
  (2 of 112 states)
unreached in proctype HubController
  swf_termpaper.pml:372, state 29, "c2 = 1"
  swf_termpaper.pml:375, state 32, "c2 = 0"
  swf_termpaper.pml:372, state 134, "c2 = 1"
  swf_termpaper.pml:375, state 137, "c2 = 0"
  (4 of 235 states)
unreached in proctype parent
  linkedlistlib.pml:47, state 16, "printf('\n--(%d, %d, %d)\n',i,list[i].value,
    list[i].next)"
  linkedlistlib.pml:49, state 17, "assert((list[i].value!=list[next].value))"
  linkedlistlib.pml:58, state 25, "i = next"
  (3 of 85 states)
unreached in claim never_0
  swf_termpaper.pml.nvr:10, state 10, "-end-"
  (1 of 10 states)

pan: elapsed time 0.08 seconds
pan: rate    189025 states/second
Done running safety verification on swf_termpaper.pml
```

Listing 5: Safety verification output

```
Running liveness verification on swf_termpaper.pml
warning: for p.o. reduction to be valid the never claim must be stutter-
    invariant
(never claims generated from LTL formulae are stutter-invariant)

(Spin Version 6.5.1 -- 20 December 2019)
  + Partial Order Reduction

Full statespace search for:
  never claim           + (never_0)
  assertion violations  + (if within scope of claim)
  acceptance   cycles   + (fairness enabled)
  invalid end states  - (disabled by never claim)

State-vector 744 byte, depth reached 380, errors: 0
     1009 states, stored (6379 visited)
     2926 states, matched
     9305 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.743 equivalent memory usage for states (stored*(State-vector + overhead))
    0.934 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  129.413 total actual memory usage


unreached in proctype ChildProcess
  swf_termpaper.pml:230, state 24, "p_state.l = l"
  swf_termpaper.pml:231, state 25, "p_state.m = m"
```

```
  swf_termpaper.pml:238, state 30, "newk = 0"
  swf_termpaper.pml:230, state 40, "p_state.l = 0"
  swf_termpaper.pml:231, state 41, "p_state.m = 1"
  swf_termpaper.pml:215, state 58, "mnext = S_SEARCH"
  swf_termpaper.pml:175, state 74, "printf('\nCP(%d): %d already exists at index
    %d. Skipping the insert',id,k,next)"
  swf_termpaper.pml:343, state 112, "-end-"
  (8 of 112 states)
unreached in proctype HubController
  swf_termpaper.pml:372, state 29, "c2 = 1"
  swf_termpaper.pml:375, state 32, "c2 = 0"
  swf_termpaper.pml:388, state 46, "c2 = 0"
  swf_termpaper.pml:392, state 49, "c2 = 1"
  swf_termpaper.pml:457, state 102, "isOver1 = 0"
  swf_termpaper.pml:453, state 103, "((h_input.proc[0].m==S_OVER))"
  swf_termpaper.pml:453, state 103, "else"
  swf_termpaper.pml:467, state 108, "o1 = O_CONTINUE"
  swf_termpaper.pml:463, state 109, "((h_input.proc[0].m==S_IDLE))"
  swf_termpaper.pml:463, state 109, "else"
  swf_termpaper.pml:477, state 113, "h_state.proc[0].o = o1"
  swf_termpaper.pml:478, state 114, "h_state.proc[0].isOver = isOver1"
  swf_termpaper.pml:449, state 115, "c1 = 1"
  swf_termpaper.pml:485, state 118, "ch_HtoP[(id1-1)]!id1,h_send.c,h_send.o"
  swf_termpaper.pml:486, state 119, "printf('\nHC: Message sent (%d, %e) to CP(%
    d)',h_send.c,h_send.o,id1)"
  swf_termpaper.pml:483, state 120, "h_send.c = h_state.proc[0].c"
  swf_termpaper.pml:372, state 134, "c2 = 1"
  swf_termpaper.pml:375, state 137, "c2 = 0"
  swf_termpaper.pml:388, state 151, "c2 = 0"
  swf_termpaper.pml:392, state 154, "c2 = 1"
  swf_termpaper.pml:457, state 207, "isOver1 = 0"
  swf_termpaper.pml:453, state 208, "((h_input.proc[1].m==S_OVER))"
  swf_termpaper.pml:453, state 208, "else"
  swf_termpaper.pml:467, state 213, "o1 = O_CONTINUE"
  swf_termpaper.pml:463, state 214, "((h_input.proc[1].m==S_IDLE))"
  swf_termpaper.pml:463, state 214, "else"
  swf_termpaper.pml:477, state 218, "h_state.proc[1].o = o1"
  swf_termpaper.pml:478, state 219, "h_state.proc[1].isOver = isOver1"
  swf_termpaper.pml:449, state 220, "c1 = 1"
  swf_termpaper.pml:485, state 223, "ch_HtoP[(id2-1)]!id2,h_send.c,h_send.o"
  swf_termpaper.pml:486, state 224, "printf('\nHC: Message sent (%d, %e) to CP(%
    d)',h_send.c,h_send.o,id2)"
  swf_termpaper.pml:483, state 225, "h_send.c = h_state.proc[1].c"
  swf_termpaper.pml:563, state 235, "-end-"
  (29 of 235 states)
unreached in proctype parent
  linkedlistlib.pml:47, state 16, "printf('\n--(%d, %d, %d)\n',i,list[i].value,
    list[i].next)"
  linkedlistlib.pml:49, state 17, "assert((list[i].value!=list[next].value))"
  linkedlistlib.pml:58, state 25, "i = next"
  linkedlistlib.pml:41, state 51, "printf('\nBegin List\n')"
  linkedlistlib.pml:44, state 53, "next = list[i].next"
  linkedlistlib.pml:47, state 55, "printf('\n--(%d, %d, %d)\n',i,list[i].value,
    list[i].next)"
  linkedlistlib.pml:49, state 56, "assert((list[i].value!=list[next].value))"
  linkedlistlib.pml:51, state 58, "printf('\n--Head Node\n')"
  linkedlistlib.pml:46, state 59, "((i!=HEAD))"
  linkedlistlib.pml:46, state 59, "else"
  linkedlistlib.pml:58, state 64, "i = next"
  linkedlistlib.pml:55, state 65, "((next==TAIL))"
  linkedlistlib.pml:55, state 65, "else"
  linkedlistlib.pml:43, state 67, "(1)"
  linkedlistlib.pml:39, state 71, "i = HEAD"
  linkedlistlib.pml:66, state 73, "printf('\nStart Memory print\n')"
  linkedlistlib.pml:67, state 74, "i = HEAD"
  linkedlistlib.pml:68, state 76, "printf('\nLocation: %d, Value: %d, Next: %d',
    i,list[i].value,list[i].next)"
  linkedlistlib.pml:67, state 77, "i = (i+1)"
```

```
  linkedlistlib.pml:65, state 84, "i = HEAD"
  swf_termpaper.pml:581, state 85, "-end-"
  (19 of 85 states)
unreached in claim never_0
  swf_termpaper.pml.nvr:8, state 6, "-end-"
  (1 of 6 states)

pan: elapsed time 0.04 seconds
pan: rate    159475 states/second
Done running liveness verification on swf_termpaper.pml
```

Listing 6: Liveness verification output

C.4 Set 3 results

```
Running safety verification on swf_termpaper.pml
warning: for p.o. reduction to be valid the never claim must be stutter-
    invariant
(never claims generated from LTL formulae are stutter-invariant)

(Spin Version 6.5.1 -- 20 December 2019)
  + Partial Order Reduction

Full statespace search for:
  never claim           + (never_0)
  assertion violations  + (if within scope of claim)
  cycle checks        - (disabled by -DSAFETY)
  invalid end states  - (disabled by never claim)

State-vector 736 byte, depth reached 2795, errors: 0
     5775 states, stored
     1113 states, matched
     6888 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    4.208 equivalent memory usage for states (stored*(State-vector + overhead))
    5.026 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  133.222 total actual memory usage


unreached in proctype ChildProcess
  swf_termpaper.pml:230, state 34, "p_state.l = l"
  swf_termpaper.pml:231, state 35, "p_state.m = m"
  (2 of 122 states)
unreached in proctype HubController
  swf_termpaper.pml:420, state 32, "c2 = 0"
  swf_termpaper.pml:425, state 36, "c2 = 0"
  swf_termpaper.pml:428, state 39, "c1 = 0"
  swf_termpaper.pml:433, state 46, "c2 = 0"
  swf_termpaper.pml:420, state 137, "c2 = 0"
  swf_termpaper.pml:425, state 141, "c2 = 0"
  swf_termpaper.pml:428, state 144, "c1 = 0"
  swf_termpaper.pml:433, state 151, "c2 = 0"
  swf_termpaper.pml:502, state 207, "isOver1 = 0"
  swf_termpaper.pml:498, state 208, "((h_input.proc[1].m==S_OVER))"
  swf_termpaper.pml:498, state 208, "else"
  swf_termpaper.pml:512, state 213, "o1 = O_CONTINUE"
  swf_termpaper.pml:508, state 214, "((h_input.proc[1].m==S_IDLE))"
  swf_termpaper.pml:508, state 214, "else"
  swf_termpaper.pml:522, state 218, "h_state.proc[1].o = o1"
  swf_termpaper.pml:523, state 219, "h_state.proc[1].isOver = isOver1"
  swf_termpaper.pml:494, state 220, "c1 = 1"
  swf_termpaper.pml:530, state 223, "ch_HtoP[(id2-1)]!id2,h_send.c,h_send.o"
```

```
  swf_termpaper.pml:531, state 224, "printf('\nHC: Message sent (%d, %e) to CP(%
    d)',h_send.c,h_send.o,id2)"
  swf_termpaper.pml:528, state 225, "h_send.c = h_state.proc[1].c"
  (18 of 235 states)
unreached in proctype parent
  linkedlistlib.pml:47, state 16, "printf('\n--(%d, %d, %d)\n',i,list[i].value,
    list[i].next)"
  linkedlistlib.pml:49, state 17, "assert((list[i].value!=list[next].value))"
  linkedlistlib.pml:58, state 25, "i = next"
  (3 of 85 states)
unreached in claim never_0
  swf_termpaper.pml.nvr:10, state 10, "-end-"
  (1 of 10 states)

pan: elapsed time 0.03 seconds
pan: rate    192500 states/second
Done running safety verification on swf_termpaper.pml
```

Listing 7: Safety verification output

```
Running liveness verification on swf_termpaper.pml
warning: for p.o. reduction to be valid the never claim must be stutter-
    invariant
(never claims generated from LTL formulae are stutter-invariant)

(Spin Version 6.5.1 -- 20 December 2019)
  + Partial Order Reduction

Full statespace search for:
  never claim           + (never_0)
  assertion violations  + (if within scope of claim)
  acceptance   cycles   + (fairness enabled)
  invalid end states  - (disabled by never claim)

State-vector 744 byte, depth reached 382, errors: 0
      675 states, stored (4223 visited)
     1873 states, matched
     6096 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.497 equivalent memory usage for states (stored*(State-vector + overhead))
    0.639 actual memory usage for states
  128.000 memory used for hash table (-w24)
    0.534 memory used for DFS stack (-m10000)
  129.120 total actual memory usage


unreached in proctype ChildProcess
  swf_termpaper.pml:230, state 34, "p_state.l = l"
  swf_termpaper.pml:231, state 35, "p_state.m = m"
  swf_termpaper.pml:238, state 40, "newk = 0"
  swf_termpaper.pml:230, state 50, "p_state.l = 0"
  swf_termpaper.pml:231, state 51, "p_state.m = 1"
  swf_termpaper.pml:215, state 68, "mnext = S_SEARCH"
  swf_termpaper.pml:175, state 84, "printf('\nCP(%d): %d already exists at index
     %d. Skipping the insert',id,k,next)"
  swf_termpaper.pml:388, state 122, "-end-"
  (8 of 122 states)
unreached in proctype HubController
  swf_termpaper.pml:420, state 32, "c2 = 0"
  swf_termpaper.pml:425, state 36, "c2 = 0"
  swf_termpaper.pml:428, state 39, "c1 = 0"
  swf_termpaper.pml:433, state 46, "c2 = 0"
  swf_termpaper.pml:437, state 49, "c2 = 1"
  swf_termpaper.pml:502, state 102, "isOver1 = 0"
```

```
  swf_termpaper.pml:498, state 103, "((h_input.proc[0].m==S_OVER))"
  swf_termpaper.pml:498, state 103, "else"
  swf_termpaper.pml:512, state 108, "o1 = O_CONTINUE"
  swf_termpaper.pml:508, state 109, "((h_input.proc[0].m==S_IDLE))"
  swf_termpaper.pml:508, state 109, "else"
  swf_termpaper.pml:522, state 113, "h_state.proc[0].o = o1"
  swf_termpaper.pml:523, state 114, "h_state.proc[0].isOver = isOver1"
  swf_termpaper.pml:494, state 115, "c1 = 1"
  swf_termpaper.pml:530, state 118, "ch_HtoP[(id1-1)]!id1,h_send.c,h_send.o"
  swf_termpaper.pml:531, state 119, "printf('\nHC: Message sent (%d, %e) to CP(%
    d)',h_send.c,h_send.o,id1)"
  swf_termpaper.pml:528, state 120, "h_send.c = h_state.proc[0].c"
  swf_termpaper.pml:420, state 137, "c2 = 0"
  swf_termpaper.pml:425, state 141, "c2 = 0"
  swf_termpaper.pml:428, state 144, "c1 = 0"
  swf_termpaper.pml:433, state 151, "c2 = 0"
  swf_termpaper.pml:437, state 154, "c2 = 1"
  swf_termpaper.pml:502, state 207, "isOver1 = 0"
  swf_termpaper.pml:498, state 208, "((h_input.proc[1].m==S_OVER))"
  swf_termpaper.pml:498, state 208, "else"
  swf_termpaper.pml:512, state 213, "o1 = O_CONTINUE"
  swf_termpaper.pml:508, state 214, "((h_input.proc[1].m==S_IDLE))"
  swf_termpaper.pml:508, state 214, "else"
  swf_termpaper.pml:522, state 218, "h_state.proc[1].o = o1"
  swf_termpaper.pml:523, state 219, "h_state.proc[1].isOver = isOver1"
  swf_termpaper.pml:494, state 220, "c1 = 1"
  swf_termpaper.pml:530, state 223, "ch_HtoP[(id2-1)]!id2,h_send.c,h_send.o"
  swf_termpaper.pml:531, state 224, "printf('\nHC: Message sent (%d, %e) to CP(%
    d)',h_send.c,h_send.o,id2)"
  swf_termpaper.pml:528, state 225, "h_send.c = h_state.proc[1].c"
  swf_termpaper.pml:608, state 235, "-end-"
  (31 of 235 states)
unreached in proctype parent
  linkedlistlib.pml:47, state 16, "printf('\n--(%d, %d, %d)\n',i,list[i].value,
    list[i].next)"
  linkedlistlib.pml:49, state 17, "assert((list[i].value!=list[next].value))"
  linkedlistlib.pml:58, state 25, "i = next"
  linkedlistlib.pml:41, state 51, "printf('\nBegin List\n')"
  linkedlistlib.pml:44, state 53, "next = list[i].next"
  linkedlistlib.pml:47, state 55, "printf('\n--(%d, %d, %d)\n',i,list[i].value,
    list[i].next)"
  linkedlistlib.pml:49, state 56, "assert((list[i].value!=list[next].value))"
  linkedlistlib.pml:51, state 58, "printf('\n--Head Node\n')"
  linkedlistlib.pml:46, state 59, "((i!=HEAD))"
  linkedlistlib.pml:46, state 59, "else"
  linkedlistlib.pml:58, state 64, "i = next"
  linkedlistlib.pml:55, state 65, "((next==TAIL))"
  linkedlistlib.pml:55, state 65, "else"
  linkedlistlib.pml:43, state 67, "(1)"
  linkedlistlib.pml:39, state 71, "i = HEAD"
  linkedlistlib.pml:66, state 73, "printf('\nStart Memory print\n')"
  linkedlistlib.pml:67, state 74, "i = HEAD"
  linkedlistlib.pml:68, state 76, "printf('\nLocation: %d, Value: %d, Next: %d',
    i,list[i].value,list[i].next)"
  linkedlistlib.pml:67, state 77, "i = (i+1)"
  linkedlistlib.pml:65, state 84, "i = HEAD"
  swf_termpaper.pml:626, state 85, "-end-"
  (19 of 85 states)
unreached in claim never_0
  swf_termpaper.pml.nvr:8, state 6, "-end-"
  (1 of 6 states)

pan: elapsed time 0.02 seconds
pan: rate    211150 states/second
Done running liveness verification on swf_termpaper.pml
```

Listing 8: Liveness verification output

C.5 One simulation run result

This is the output of one simulation path run. Print statements illustrate the state transitions for the processes (CP1 and CP2), and for the Hub Controller (HC).

```
1
2  Critical Value: 255
3  Begin List
4
5  --Head Node
6
7  End List
8
9  Start Memory print
10
11 Location: 1, Value: 0, Next: 2
12 Location: 2, Value: 255, Next: 0
13 Location: 3, Value: 0, Next: 0
14 End Memory print
15
16 HC Started with 2 processes 1 and 2.
17 CP(1) Started.
18 CP(2) Started.
19 HC: Message sent (1, O_INSERT) to CP(1)
20 HC: Message sent (1, O_INSERT) to CP(2)
21 CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(100,1,S_SEARCH)
22 CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(100,1,S_SEARCH)
23 CP(1): Message sent (100, 1, S_SEARCH)
24 CP(2): Message sent (100, 1, S_SEARCH)
25 HC: [(1,O_INSERT,0),(1,O_INSERT,0)]----[(100,1,S_SEARCH),(100,1,S_SEARCH)
       ]---->[(1,O_CONTINUE,0),(1,O_CONTINUE,0)]
26 HC: Message sent (1, O_CONTINUE) to CP(1)
27 HC: Message sent (1, O_CONTINUE) to CP(2)
28 CP(1): (100,1,S_SEARCH)----(1,O_CONTINUE)---->(100,1,S_INSERT)
29 CP(1): Message sent (100, 1, S_INSERT)
30 CP(2): (100,1,S_SEARCH)----(1,O_CONTINUE)---->(100,1,S_INSERT)
31 CP(2): Message sent (100, 1, S_INSERT)
32 HC: [(1,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(100,1,S_INSERT),(100,1,S_INSERT)
       ]---->[(0,O_CONTINUE,0),(1,O_CONTINUE,0)]
33 HC: Message sent (0, O_CONTINUE) to CP(1)
34 HC: Message sent (1, O_CONTINUE) to CP(2)
35 CP(1): (100,1,S_INSERT)----(0,O_CONTINUE)---->(100,1,S_INSERT)
36 CP(1): Message sent (100, 1, S_INSERT)              CP(2): Critical Value (
       before): 255, a: 0, b: 255,  c:100
37 CP(2): (100,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
38 CP(2): Message sent (0, 0, S_IDLE)
39 HC: [(0,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(100,1,S_INSERT),(0,0,S_IDLE)
       ]---->[(1,O_CONTINUE,0),(1,O_INSERT,0)]
40 HC: Message sent (1, O_CONTINUE) to CP(1)
41 HC: Message sent (1, O_INSERT) to CP(2)
42 CP(1): 100 already exists at index 3. Skipping the insert
43 CP(1): (100,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
44 CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(99,1,S_SEARCH)
45 CP(1): Message sent (0, 0, S_IDLE)
46 CP(2): Message sent (99, 1, S_SEARCH)
47 HC: [(1,O_CONTINUE,0),(1,O_INSERT,0)]----[(0,0,S_IDLE),(99,1,S_SEARCH)]---->[(1,
       O_INSERT,0),(1,O_CONTINUE,0)]
48 HC: Message sent (1, O_INSERT) to CP(1)
49 HC: Message sent (1, O_CONTINUE) to CP(2)
50 CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(99,1,S_SEARCH)
51 CP(2): (99,1,S_SEARCH)----(1,O_CONTINUE)---->(99,1,S_INSERT)
52 CP(2): Message sent (99, 1, S_INSERT)
53 CP(1): Message sent (99, 1, S_SEARCH)
54 HC: [(1,O_INSERT,0),(1,O_CONTINUE,0)]----[(99,1,S_SEARCH),(99,1,S_INSERT)
       ]---->[(0,O_CONTINUE,0),(1,O_CONTINUE,0)]
55 HC: Message sent (0, O_CONTINUE) to CP(1)
56 HC: Message sent (1, O_CONTINUE) to CP(2)
57 CP(1): (99,1,S_SEARCH)----(0,O_CONTINUE)---->(99,1,S_SEARCH)              CP(2):
```

```
     Critical Value (before): 15500, a: 0, b: 100,  c:99
CP(1): Message sent (99, 1, S_SEARCH)
CP(2): (99,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(2): Message sent (0, 0, S_IDLE)
HC: [(0,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(99,1,S_SEARCH),(0,0,S_IDLE)
    ]---->[(1,O_CONTINUE,0),(1,O_INSERT,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_INSERT) to CP(2)
CP(1): (99,1,S_SEARCH)----(1,O_CONTINUE)---->(99,1,S_INSERT)
CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(98,1,S_SEARCH)
CP(1): Message sent (99, 1, S_INSERT)
CP(2): Message sent (98, 1, S_SEARCH)
HC: [(1,O_CONTINUE,0),(1,O_INSERT,0)]----[(99,1,S_INSERT),(98,1,S_SEARCH)
    ]---->[(1,O_CONTINUE,0),(0,O_CONTINUE,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (0, O_CONTINUE) to CP(2)
CP(2): (98,1,S_SEARCH)----(0,O_CONTINUE)---->(98,1,S_SEARCH)
CP(2): Message sent (98, 1, S_SEARCH)
CP(1): 99 already exists at index 4. Skipping the insert
CP(1): (99,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(1): Message sent (0, 0, S_IDLE)
HC: [(1,O_CONTINUE,0),(0,O_CONTINUE,0)]----[(0,0,S_IDLE),(98,1,S_SEARCH)
    ]---->[(1,O_INSERT,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_INSERT) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(98,1,S_SEARCH)
CP(2): (98,1,S_SEARCH)----(1,O_CONTINUE)---->(98,1,S_INSERT)
CP(1): Message sent (98, 1, S_SEARCH)
CP(2): Message sent (98, 1, S_INSERT)
HC: [(1,O_INSERT,0),(1,O_CONTINUE,0)]----[(98,1,S_SEARCH),(98,1,S_INSERT)
    ]---->[(0,O_CONTINUE,0),(1,O_CONTINUE,0)]
HC: Message sent (0, O_CONTINUE) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(1): (98,1,S_SEARCH)----(0,O_CONTINUE)---->(98,1,S_SEARCH)
CP(1): Message sent (98, 1, S_SEARCH)              CP(2): Critical Value (before
    ): 15345, a: 0, b: 99,  c:98
CP(2): (98,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(2): Message sent (0, 0, S_IDLE)
HC: [(0,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(98,1,S_SEARCH),(0,0,S_IDLE)
    ]---->[(1,O_CONTINUE,0),(1,O_INSERT,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_INSERT) to CP(2)
CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(97,1,S_SEARCH)
CP(2): Message sent (97, 1, S_SEARCH)
CP(1): (98,1,S_SEARCH)----(1,O_CONTINUE)---->(98,1,S_INSERT)
CP(1): Message sent (98, 1, S_INSERT)
HC: [(1,O_CONTINUE,0),(1,O_INSERT,0)]----[(98,1,S_INSERT),(97,1,S_SEARCH)
    ]---->[(1,O_CONTINUE,0),(0,O_CONTINUE,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (0, O_CONTINUE) to CP(2)
CP(1): 98 already exists at index 5. Skipping the insert
CP(1): (98,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(2): (97,1,S_SEARCH)----(0,O_CONTINUE)---->(97,1,S_SEARCH)
CP(2): Message sent (97, 1, S_SEARCH)
CP(1): Message sent (0, 0, S_IDLE)
HC: [(1,O_CONTINUE,0),(0,O_CONTINUE,0)]----[(0,0,S_IDLE),(97,1,S_SEARCH)
    ]---->[(1,O_INSERT,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_INSERT) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): (97,1,S_SEARCH)----(1,O_CONTINUE)---->(97,1,S_INSERT)
CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(97,1,S_SEARCH)
CP(2): Message sent (97, 1, S_INSERT)
CP(1): Message sent (97, 1, S_SEARCH)
HC: [(1,O_INSERT,0),(1,O_CONTINUE,0)]----[(97,1,S_SEARCH),(97,1,S_INSERT)
    ]---->[(0,O_CONTINUE,0),(1,O_CONTINUE,0)]
HC: Message sent (0, O_CONTINUE) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(1): (97,1,S_SEARCH)----(0,O_CONTINUE)---->(97,1,S_SEARCH)
```

```
CP(1): Message sent (97, 1, S_SEARCH)              CP(2): Critical Value (before
    ): 15190, a: 0, b: 98,  c:97
CP(2): (97,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(2): Message sent (0, 0, S_IDLE)
HC: [(0,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(97,1,S_SEARCH),(0,0,S_IDLE)
    ]---->[(1,O_CONTINUE,0),(1,O_INSERT,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_INSERT) to CP(2)
CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(96,1,S_SEARCH)
CP(1): (97,1,S_SEARCH)----(1,O_CONTINUE)---->(97,1,S_INSERT)
CP(2): Message sent (96, 1, S_SEARCH)
CP(1): Message sent (97, 1, S_INSERT)
HC: [(1,O_CONTINUE,0),(1,O_INSERT,0)]----[(97,1,S_INSERT),(96,1,S_SEARCH)
    ]---->[(1,O_CONTINUE,0),(0,O_CONTINUE,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (0, O_CONTINUE) to CP(2)
CP(2): (96,1,S_SEARCH)----(0,O_CONTINUE)---->(96,1,S_SEARCH)
CP(2): Message sent (96, 1, S_SEARCH)
CP(1): 97 already exists at index 6. Skipping the insert
CP(1): (97,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(1): Message sent (0, 0, S_IDLE)
HC: [(1,O_CONTINUE,0),(0,O_CONTINUE,0)]----[(0,0,S_IDLE),(96,1,S_SEARCH)
    ]---->[(1,O_INSERT,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_INSERT) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): (96,1,S_SEARCH)----(1,O_CONTINUE)---->(96,1,S_INSERT)
CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(96,1,S_SEARCH)
CP(2): Message sent (96, 1, S_INSERT)
CP(1): Message sent (96, 1, S_SEARCH)
HC: [(1,O_INSERT,0),(1,O_CONTINUE,0)]----[(96,1,S_SEARCH),(96,1,S_INSERT)
    ]---->[(0,O_CONTINUE,0),(1,O_CONTINUE,0)]
HC: Message sent (0, O_CONTINUE) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(1): (96,1,S_SEARCH)----(0,O_CONTINUE)---->(96,1,S_SEARCH)              CP(2):
     Critical Value (before): 15035, a: 0, b: 97,  c:96
CP(1): Message sent (96, 1, S_SEARCH)
CP(2): (96,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(2): Message sent (0, 0, S_IDLE)
HC: [(0,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(96,1,S_SEARCH),(0,0,S_IDLE)
    ]---->[(1,O_CONTINUE,0),(1,O_INSERT,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_INSERT) to CP(2)
CP(1): (96,1,S_SEARCH)----(1,O_CONTINUE)---->(96,1,S_INSERT)
CP(1): Message sent (96, 1, S_INSERT)
CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(95,1,S_SEARCH)
CP(2): Message sent (95, 1, S_SEARCH)
HC: [(1,O_CONTINUE,0),(1,O_INSERT,0)]----[(96,1,S_INSERT),(95,1,S_SEARCH)
    ]---->[(1,O_CONTINUE,0),(0,O_CONTINUE,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (0, O_CONTINUE) to CP(2)
CP(1): 96 already exists at index 7. Skipping the insert
CP(1): (96,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(1): Message sent (0, 0, S_IDLE)
CP(2): (95,1,S_SEARCH)----(0,O_CONTINUE)---->(95,1,S_SEARCH)
CP(2): Message sent (95, 1, S_SEARCH)
HC: [(1,O_CONTINUE,0),(0,O_CONTINUE,0)]----[(0,0,S_IDLE),(95,1,S_SEARCH)
    ]---->[(1,O_INSERT,0),(1,O_CONTINUE,0)]
HC: Message sent (1, O_INSERT) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): (95,1,S_SEARCH)----(1,O_CONTINUE)---->(95,1,S_INSERT)
CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(95,1,S_SEARCH)
CP(2): Message sent (95, 1, S_INSERT)
CP(1): Message sent (95, 1, S_SEARCH)
HC: [(1,O_INSERT,0),(1,O_CONTINUE,0)]----[(95,1,S_SEARCH),(95,1,S_INSERT)
    ]---->[(0,O_CONTINUE,0),(1,O_CONTINUE,0)]
HC: Message sent (0, O_CONTINUE) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(1): (95,1,S_SEARCH)----(0,O_CONTINUE)---->(95,1,S_SEARCH)              CP(2):
```

```
     Critical Value (before): 14880, a: 0, b: 96,  c:95
CP(1): Message sent (95, 1, S_SEARCH)
CP(2): (95,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(2): Message sent (0, 0, S_IDLE)
HC: [(0,O_CONTINUE,0),(1,O_CONTINUE,0)]----[(95,1,S_SEARCH),(0,0,S_IDLE)
    ]---->[(1,O_CONTINUE,0),(1,O_INSERT,0)]
HC: Message sent (1, O_CONTINUE) to CP(1)
CP(2): No more insertions!
CP(2): (0,0,S_IDLE)----(1,O_INSERT)---->(0,0,S_OVER)
HC: Message sent (1, O_INSERT) to CP(2)
CP(1): (95,1,S_SEARCH)----(1,O_CONTINUE)---->(95,1,S_INSERT)
CP(2): Message sent (0, 0, S_OVER)
CP(1): Message sent (95, 1, S_INSERT)
HC: [(1,O_CONTINUE,0),(1,O_INSERT,0)]----[(95,1,S_INSERT),(0,0,S_OVER)]---->[(1,
    O_CONTINUE,0),(1,O_CONTINUE,1)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: Message sent (1, O_CONTINUE) to CP(2)
CP(2): Received message in S_OVER state,exiting now..
CP(2) Exited.
CP(1): 95 already exists at index 8. Skipping the insert
CP(1): (95,1,S_INSERT)----(1,O_CONTINUE)---->(0,0,S_IDLE)
CP(1): Message sent (0, 0, S_IDLE)
HC: [(1,O_CONTINUE,0),(S_OVER)]----[(0,0,S_IDLE),(S_OVER))]---->[(1,O_INSERT,0)
    ,(S_OVER)]
HC: Message sent (1, O_INSERT) to CP(1)
CP(1): No more insertions!
CP(1): (0,0,S_IDLE)----(1,O_INSERT)---->(0,0,S_OVER)
CP(1): Message sent (0, 0, S_OVER)
HC: [(1,O_INSERT,0),(S_OVER)]----[(0,0,S_OVER),(S_OVER))]---->[(1,O_CONTINUE,1)
    ,(S_OVER)]
HC: Message sent (1, O_CONTINUE) to CP(1)
HC: We should exit - both processes are done!
HC Exited
CP(1): Received message in S_OVER state,exiting now..
CP(1) Exited.
Begin List

--Head Node

--(8, 95, 7)

--(7, 96, 6)

--(6, 97, 5)

--(5, 98, 4)

--(4, 99, 3)

--(3, 100, 2)

End List

Start Memory print

Location: 1, Value: 0, Next: 8
Location: 2, Value: 255, Next: 0
Location: 3, Value: 100, Next: 2
Location: 4, Value: 99, Next: 3
Location: 5, Value: 98, Next: 4
Location: 6, Value: 97, Next: 5
Location: 7, Value: 96, Next: 6
Location: 8, Value: 95, Next: 7
Location: 9, Value: 0, Next: 0
End Memory print
4 processes created
```

Listing 9: Simulation output