Proving correctness of concurrent linked list algorithm using SPIN model checker

Mrityunjay Kumar (2018801001)

Software foundations – IIITH (Spring ‘20)

Agenda

- Problem Statement
- About SPIN model checker
- Implementation details
- Work so far and next steps

Problem Statement

Given a model of fine-grained concurrent linked list[1] using feedback control and transition systems[2],

Demonstrate that this model is correct, which means it describes a system that satisfies following properties:

Safety	Liveness	Fairness

1. Herlihy, Maurice, and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011, Page 223
2. Paulo Tabuada. 2009.*Verification and control of hybrid systems*: *a symbolic approach*. Springer Science & Business Media

Motivation – Why model checking?

Formal proof of correctness of an algorithm before investing significant time and money in actual implementation can be cost-effective

Provides much-needed quality control and confidence in key components of mission-critical software where cost of a bug in production code is too high

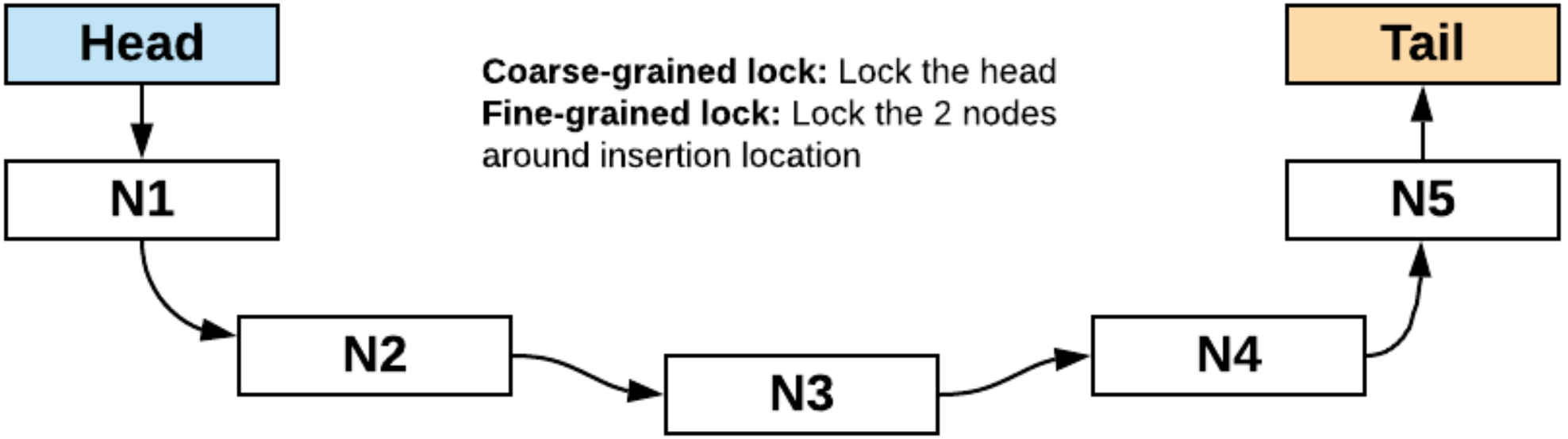
Helps identify issues that are almost impossible to catch via testing cycle

Insights from reviewing related work

Custom tools have been built to prove correctness of models

Among the established tools, SPIN seems to be the most referred/used one.

Locking in concurrent linked list



The fine-grained concurrent linked list algorithm[1]

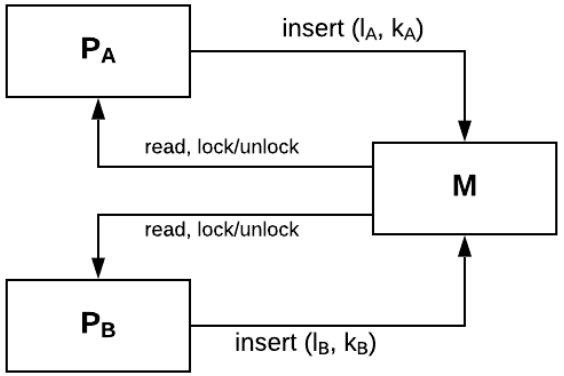
- Each process initially tries to acquire the lock of the head node, and subsequently the next node.
- In each step, it releases the lock on the back node, and tries to lock the node next to the front node.
- This continues till it reaches a point where it finds the appropriate position to insert the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

```
1   public boolean add(T item) {
2     int key = item.hashCode();
3     head.lock();
4     Node pred = head;
5     try {
6       Node curr = pred.next;
7       curr.lock();
8       try {
9         while (curr.key < key) {
10          pred.unlock();
11          pred = curr;
12          curr = curr.next;
13          curr.lock();
14        }
15        if (curr.key == key) {
16          return false;
17        }
18        Node newNode = new Node(item);
19        newNode.next = curr;
20        pred.next = newNode;
21        return true;
22      } finally {
23        curr.unlock();
24      }
25    } finally {
26      pred.unlock();
27    }
28  }
```

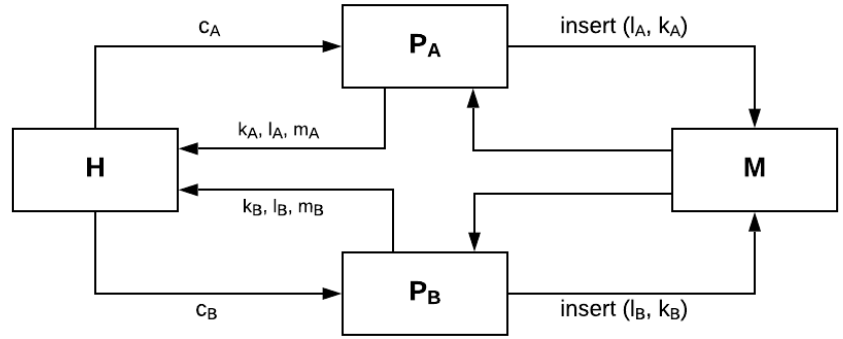
1. Herlihy, Maurice, and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011, Page 223

Problem Detail

With Lock



With Feedback control



P_A and P_B be 2 processes accessing the linked list	M is the memory	H is the controller

Processes try to add nodes to the linked list with these invariants.	**Assumptions**	Properties to be satisfied
• 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.	• Only positive numbers are being inserted • The list always has a head and a tail with 0 and ∞	• **Safety** : Invariants should be preserved during add • **Starvation-freedom** : Any process that tries to do an operation must eventually succeed in doing so

Using feedback control to avoid lock/unlock semantics[3]

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

3. Modular Feedback Control Approach to Concurrency, Venkatesh Choppella, Kasturi Viswanath, Arjun Sanjeev and Bharat Jayaraman

SPIN[4] Model Checker ACM Software system award 2001[5]

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

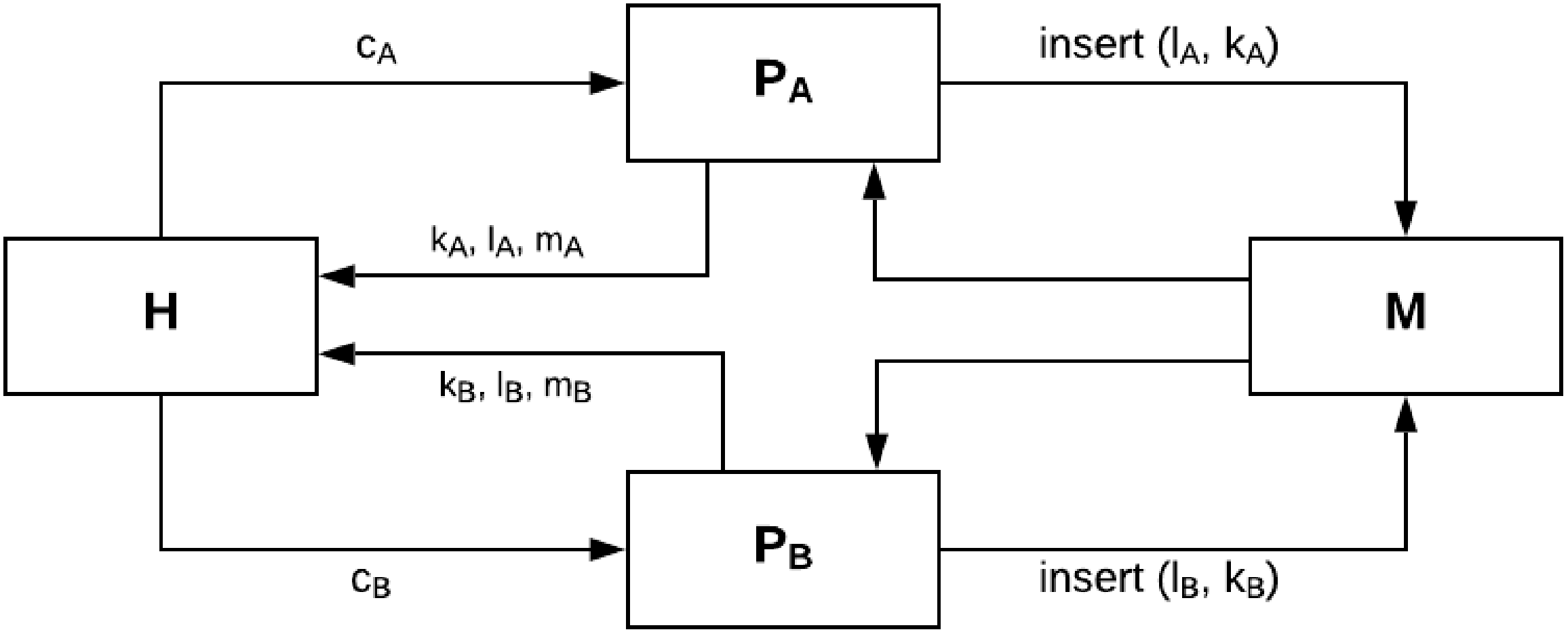
4. Holzmann, Gerard J. "The model checker SPIN." *IEEE Transactions on software engineering* 23, no. 5 (1997): 279-295.
5. Software System Award: ACM CITES TOOL TO DETECT SOFTWARE "BUGS" FOR PRESTIGIOUS AWARD. Bell Labs Researcher Developed "SPIN" to Make Computers More Reliable, https://awards.acm.org/award_winners/holzmann_1625680

What does SPIN Model Checker do

"Spin is one of the foremost model checkers".[6]

- Uses Promela (C-like) for describing the system to be verified
- Properties to be verified are expressed as LTL formulas
- Also operates as simulator (one random path, interactive or guided simulation to trace error path)
- Performs model checking by generating C source for a problem-specific model checker

6. Ben-Ari, Mordechai. "A primer on model checking." *ACM Inroads* 1, no. 1 (2010): 40-47.



Execution Model using controller

Transition Systems for the components

Processes P_A and P_B

$\langle X_P, X^0_P, U_P, f_P \rangle$

X_P: Number x Location x Mode

X^0_P : {(_,_, *Done*)}

U_P: Command x Operation

f_P: X_P x U_P → X_P

$$
\begin{aligned}
&f((k,l,m),(c,o)) \\
&= (k,l,m), && \text{if } c = 0 \text{ or } (m = Done \text{ and } o = \bot), \\
&= (k', head, Search), && \text{if } m = Done \text{ and } o = insert(k'), \\
&= (k, l.next, m), && \text{if } m = Search \text{ and } l.next.key < key, \\
&= (k, l, Insert), && \text{if } m = Search \text{ and } l.next.key > key, \\
&= (\bot, \bot, Done), && \text{if } m = Insert
\end{aligned}
$$

Hub controller H

$\langle X_H, X^0_H, U_H, f_H \rangle$

X_H: Command x Command

X^0_H : {(1,1}

U_H: (Number x Location x Mode)^2

f_H: X_H x U_H → X_H

$$
\begin{aligned}
&f((c_A,c_B),((k_A,l_A,m_A),(k_B,l_B,m_B))) \\
&= (0,1), && \text{if } m_A = m_B = Insert, l_A = l_B \text{ and } k_A < k_B, \\
&= (1,0), && \text{if } m_A = m_B = Insert, l_A = l_B \text{ and } k_A > k_B, \\
&= (1,1), && \text{otherwise}
\end{aligned}
$$

Implementation of the model in Promela

Two Processes which try to insert 2 different (partially overlapping) series of numbers

- A: **1, 3**, 5, 6, **8, 10**, 12
- B: **1, 3**, 4, 7, **8, 10**, 11

Implementation choices

- Linked list as array
- A large array for managing memory (use atomic operation to get a free node to create a new list element added)
- Inter-process communication through messages

Processes

```
inline initialize() {

    HEAD = MIN_INDEX + 1
    TAIL = MIN_INDEX + 2
    list[HEAD].value = MIN_VAL
    list[HEAD].next = TAIL
    list[TAIL].value = MAX_VAL
    list[TAIL].next = MIN_INDEX

    }


proctype child() {
    int i, j;
    int current;
    int next;

    for (j: 1 .. 5) {
        insert ( j )

    }

}

active proctype parent() {
    int i;
    int next;
    pid n;
    n = _nr_pr;

    initialize ()

    run child();
    run child();
    (n == _nr_pr);

    print_list ()
}
```

Insert Logic

```
inline insert(val) {
    /* Keep traversing. when the next item has a value higher than what we want to insert, we insert there */
    i = HEAD;

    printf("\nTo insert: %d\n", val)
    do
        :: true ->
            next = list[i].next
            if
                :: (i != HEAD) ->
                    printf("\nCurrent: (%d, %d, %d)", i, list[i].value, list[i].next)
                    printf(" Next: (%d, %d, %d)\n", next, list[next].value, list[next].next)
                :: else
                    printf("\nCurrent: Head Node")
                    printf(" Next: (%d, %d, %d)\n", next, list[next].value, list[next].next)
            fi

            if
                :: (list[next].value > val) ->
                    /* Get a free node to use for this insertion */
                        d_step {
                            current = EMPTY;
                            EMPTY = EMPTY + 1;
                        }
                    /* Insert after the current node */
                        list[current].value = val;
                        list[current].next = next;
                        list[i].next = current;
                        break;
                :: (list[next].value == val) ->
                        printf("Same value found!")
                        break
                :: else
                    i = next
            fi;
    od;
    printf("\nInserted.\n")
}
```

Constraints implementation in SPIN

Property	Constraints	Implementation
Safety	1. There must be no duplicate entries in the linked list. 2. The numbers in the list must be arranged in strictly increasing order, from head to tail. *3. (VC: Look at what standard algo does – should be same)*	*1. TBD – how to state as [] (always) claim in SPIN?*
Liveness	1. Either P_A or P_B will be able to eventually insert an element in the array 2. All numbers will be entered in the list before the processes terminates) 3. Both P_A and P_B are able to eventually insert all their numbers 4. Numbers that only P_A wants to insert are inserted 5. Numbers that only P_B wants to insert are inserted	1. Eventually, an insert statement is called *1. <> PA_insert_statement* *2. <> PB_insert_statement* 2. Eventually, last statement of P_A and P_B are reached *1. <> PA_last_statement* *2. <> PA_last_statement*)
Fairness	1. If PA has a number, it does get its turn to insert. (*VC: Show that the strategy is fair)*	Assert that each of the number is in the list after P_A and P_B terminate

Challenges in Promela/SPIN

No support for dynamic memory allocation

No support for functions

LTL doesn’t evaluate function, only statements

Progress so far

Work done	Next Steps
• Designed the model for the problem • Implemented the concurrent linked list (using array) • Implemented messaging structure	• Implement the properties • Run verification for multiple datasets • Write report

References

1. Herlihy, Maurice, and Nir Shavit. The art of multiprocessor programming. Morgan Kaufmann, 2011, Page 223
2. Paulo Tabuada. 2009.Verification and control of hybrid systems: a symbolic approach. Springer Science & Business Media
3. Modular Feedback Control Approach to Concurrency, Venkatesh Choppella, Kasturi Viswanath, Arjun Sanjeev and Bharat Jayaraman
4. Holzmann, Gerard J. "The model checker SPIN." IEEE Transactions on software engineering 23, no. 5 (1997): 279-295.
5. Software System Award: ACM CITES TOOL TO DETECT SOFTWARE "BUGS" FOR PRESTIGIOUS AWARD. Bell Labs Researcher Developed "SPIN" to Make Computers More Reliable, https://awards.acm.org/award_winners/holzmann_1625680
6. Černý, Pavol, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. "Model checking of linearizability of concurrent list implementations." In *International Conference on Computer Aided Verification*, pp. 465-479. Springer, Berlin, Heidelberg, 2010.
7. Colvin, Robert, Simon Doherty, and Lindsay Groves. "Verifying concurrent data structures by simulation." *Electronic Notes in Theoretical Computer Science* 137, no. 2 (2005): 93-110.
8. Norris, Brian, and Brian Demsky. "CDSchecker: checking concurrent data structures written with C/C++ atomics." In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pp. 131-150. 2013
9. Tasiran, Serdar, and Shaz Qadeer. "Runtime refinement checking of concurrent data structures." *Electronic notes in theoretical computer science* 113 (2005): 163-179.


Thank you

Appendix

Why is this an important problem?

Concurrent data structures are key to distributed systems

Transition systems are an elegant way to solve concurrency problems

The approach used here can be applied to other concurrent problems solved using transition systems

Why SPIN model checker

- Established model checker
- Easier implementation
- Reviewed model checking and tools (report a bit on them)

Starvation-freedom

Any process that tries to do an operation must eventually succeed in doing so

- Liveness
  - Either P_A and P_B will be able to eventually insert an element in the array
  - All numbers will be entered in the list before the processes terminates(liveness or fairness?)
- Fairness
  - Both P_A and P_B are able to eventually insert all their numbers
  - Numbers that only P_A wants to insert are inserted
  - Numbers that only P_B wants to insert are inserted

If we try to insert and find it is already inserted, it is considered an opportunity provided.

Safety property

- 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.

Steps

- Implement the logic of the model in Promela
- Describe the properties in LTL
- Run verification