# Verifying Safety Properties Related to Reachability Problems in Software Programs

Era Johri
Assistant Professor
K J Somaiya College of Engineering, Mumbai

## ABSTRACT

With the success of formal verification techniques like equivalence checking and model checking for hardware designs, there has been growing interest in applying such techniques for formal analysis and automatic verification of software programs. The majority of work carried out in the formal methods community throughout the last three decades has been devoted to special languages designed to make it easier to experiment with mechanized formal methods such as theorem provers, proof checkers and model checkers. With this philosophy, in this paper, it is proposed to develop a verification and testing environment for checking user given safety properties in C Programs, which uses model checking. The reachability properties for verification have to be considered in particular whether certain labeled statements are reachable or not in a program. Also, checkers are included for a set of standard programming bugs such as array bound violations, NULL pointer dereferences, use of uninitialized variables, memory leaks, lock/unlock violations, division by zero etc.

## General Terms

SPIN, Linear Temporal Logic, Formal Methods, Abstraction, Partial Order Reduction

## Keywords

Lex and Yaac, SPIN, Formal Methods.

## 1. INTRODUCTION

Verification of system requirements is very important design aspects of safety critical systems. Safety critical systems, such as nuclear reactors and many other are important in regard to their safety as their failure may lead to crashing of the whole system. In such systems it is required to prove that the program meets all the explicitly stated functional requirements. There are certain requirements which enforce the correct functioning of the software. Software meeting those requirements is very difficult to establish by testing. Many of these requirements map to reachability control points establishes that the program is safe.

Testing of these systems is not possible due to the infinite or large state space of the system. Therefore, we aim to use formal methods to overcome this state explosion problem. FM employ
different techniques like abstraction, partial order reduction to deal with the large state space problems. FM enable us to cope up with two major problems like time factor and memory constraints.

In this paper, we propose an idea that accepts the C Program and a temporal logic specifying the sequence of control points inputs from the user and to generate a modified C Program along with the model extraction information. This modified code is then annotated by using a test harness file. Test harness file contains all the abstractions to abstract the code. The abstracted code will cover all the areas required in a program for verification. This code then input to the Model Checker which will check for the reachability problems associated with the code. Model Checker itself will abstract the code in order to reduce the state space of the system.

For example, in Mutual Exclusion Problem, the resources once lock by a process should be unlock in order to prevent deadlocks. That is, the resources acquired by a process should be freed after completion of its process lacking which another process will go to an infinite waiting state. We can determine the states of the processes to be in Lock state and Unlock state. In between a process may attain any intermediately state as state 1, state 2, and so on.

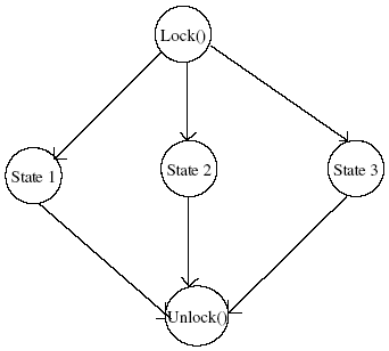


Figure 1: State Diagram of Mutual Exclusion

For instance,

```
Ex a m p l e ( ) {
1 : do {
        lock();
        old = new;
2: if (old= = new ) {
3: unlock();
        new + + ; }}
4 : while ( new != old);
5 : unlock ();
        return ;
        }
```

*Example 1*

In the above given Example 1 function, process calls a lock function to lock the resources and after this call it is eventually calling the unlock function to release the resources. The resource should be released as soon as the condition new==old is satisfied or while the process terminates. The state diagram in *Figure 1* indicate the control flow of the process.
We try to keep a track that whatever state a process attains after acquiring lock(), but finally it should reach to an unlock() state failing which the resource will not be free and case like Mutual Exclusion problem this may lead to deadlock. This paper aim to verify and check that whenever a lock() is called it eventually leads to an unlock(). This is represented diagrammatically in *Figure 2*.

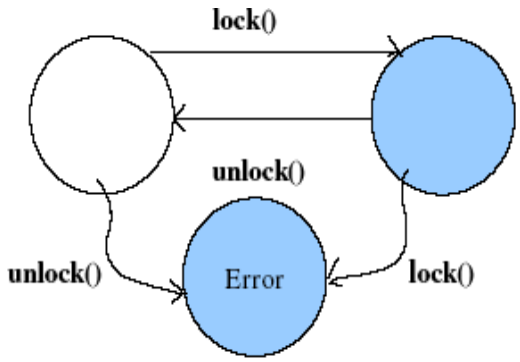


Figure 2: Function Lock and Unlock

## 2. SPIN

SPIN is a simple PROMELA Interpreter and a generic verification system that supports the design and verification of asynchronous process systems. It tries to abstract as much as possible from internal sequential computations and aims at proving the correctness of the process interactions. It can be used as a tool for analyzing the logical consistency of a system. It is used to stimulate (Finding deadlocks, unspecified reception of messages, and unexecutable code) and verify (Correctness of system invariants and finding out non progress cycles) the system behavior. It is based on on the fly verification. The advantage of using SPIN is that its notations are chosen in such away that the logical consistency of design can be demonstrated mechanically by the tool. SPIN accepts the design specifications written in PROMELA and accepts the correctness claims specified in the syntax of Linear Programming Logic. A Buchi‘s Automaton is constructed for the system which accepts the system execution if and only if the execution forces it to pass through one or more of systems accepting states infinitely often. It is a way of notation of acceptance to infinite runs.

### 2.1 Formal Definition for Buchi's Automata

An infinite run of finite state automaton { S, s0, L, T, F} is accepted iff at least one state from σ set F appears infinitely often in σ.

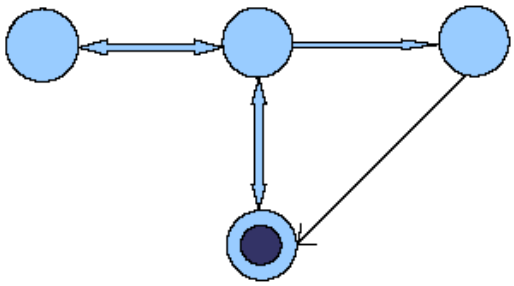

Figure 3: Buchi’s Automata

SPIN does the whole verification in a complete single procedure when simulated, using nested depthfirst search algorithm.

The main errors encountered in SPIN are:

1. Deadlock

2. Livelock, Starvation

3. Under Specification

- Unexpected reception of messages

4. Over specification

- Dead code

### 2.2 LINEAR TEMPORAL LOGIC

To express the properties to be verified for the given C Programs we have to adopt a notation for specifying these properties. Such properties are specified in the form of Linear Temporal Logic Formulas. They can be used to formally state the properties of system executions. Linear temporal logic consists of Atomic Prepositions, Standard Boolean and Temporal Operators. We use certain notations to define conditions to be checked as,
Weak Until (U), Strong Until (U), Always ([ ] ), Eventually (◊) etc.

## 3. IMPLEMENTATION

Suppose we take the above code as specified in Example 1 and write the code in C. Now we would like to check weather there is Unlock() after every Lock(). To check for this condition in SPIN we provide the Linear Temporal Logic as input defined by user as:

[ ]([ ]func_lock() ><> func_unlock())

i.e. **always a lock is followed by an unlock**. This specification has to be specified by the user that describes the property to be tested for the system.

```
File  Edit  View  Terminal  Tabs  Help
[era@calculus testltl]$ ./a.out
Enter the string
[](func_lock-><>func_unlock1)

The first variable is: func_lock
The second variable is func_unlock1

Sucessfully Parsed
[era@calculus testltl]$
```

Figure 4: Parsing the LTL formula

The model is then verified on behalf of the LTL property stated by the user which is now been parsed using Lex and Yaac parsers and the two states in program are identified. The LTL property is negated and SPIN will try to map this property to the system state space. If any intersection between the property or the state space is found then it is considered that the system is faulty and report is generated revealing that the system does not satisfy the property as shown in Figure 5. Else if no intersection is found then it is assumed that the property given by the user satisfies the code and the code is correct.

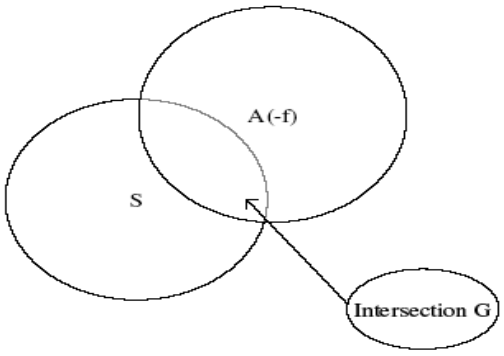


Figure 5: *Intersection G of System S and Automaton A derived from LTL Formula ¬f.*

The code is compiled using Exhaustive search. The program can be compiled simply as:

$ gcc o pan pan.c

and the output is written in pan file which is stored with the program for future. All the states are reachable and the system verifies the safety and reachability problems in the code.

The executable program pan can now be executed to perform the verification. The verification is truly exhaustive: it tests all possible event sequences in all possible orders. It can also assertion violations.

## Output

$ ./pan
assertion violated (i == last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
61 states, stored
5 states, linked
1 states, matched
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/5)

The first line of the output announces the assertion violation and attempts to give a first indication of the invariant that was violated. The violation was found after 61 states had been generated. Hash "conflicts" gives the number of hash collisions that happened during access to the state space. As indicated, all collisions are resolved in full search mode, since all states are placed in a linked list. The most relevant piece of output in this case, however, is on the third line which tells us that a trail file was created that can be used in combination with the simulator to recreate the error sequence.

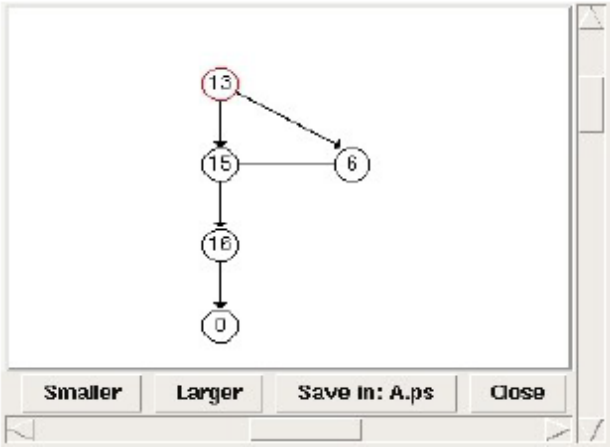


Figure 6: State Space generated by SPIN

Hence it can be seen clearly from Figure 6 that all the states are reachable and the system verifies the safety and reachability problems in the code.

## 4. CONCLUSION

In this paper we have taken a particular aspect regarding the verification of acquiring and releasing of resources by a process by verifying it in terms of function lock and unlock mechanism in a program which can be further extended to verification of various other objects of a program. This paper tried to focus on some aspects like deadlock, livelocks and starvation following Buchi's Automata Approach. In this paper we followed a different approach of Model Checking. Earlier systems use Classic Model Checkers which can only check a program syntactically. In this paper we proposed Modern Checkers which can do on the fly verification of programs as shown in Figure 7. Mutual Exclusion programs are verified by doing semantically analysis of a program in this paper.

## 5. REFRENCES

[1] Dijkstra, E.W., ``Guarded commands, nondeterminacy and formal derivation of programs." *CACM* 18, 8 (1975), 453457.

[2] Holzmann, G.J., ``Algorithms for automated protocol verification"

[2] Dijkstra, E.W., ``Solution to a problem in concurrent programming control." CACM 8, 9 (1965), 569.

[3] Holzmann, G.J., ``Algorithms for automated protocol verification." AT&T Technical Journal 69, 1(Jan/Feb 1990).

[4] Holzmann, G.J., The Spin Model Checker: Primer and Reference Manual (2003) AddisonWesley

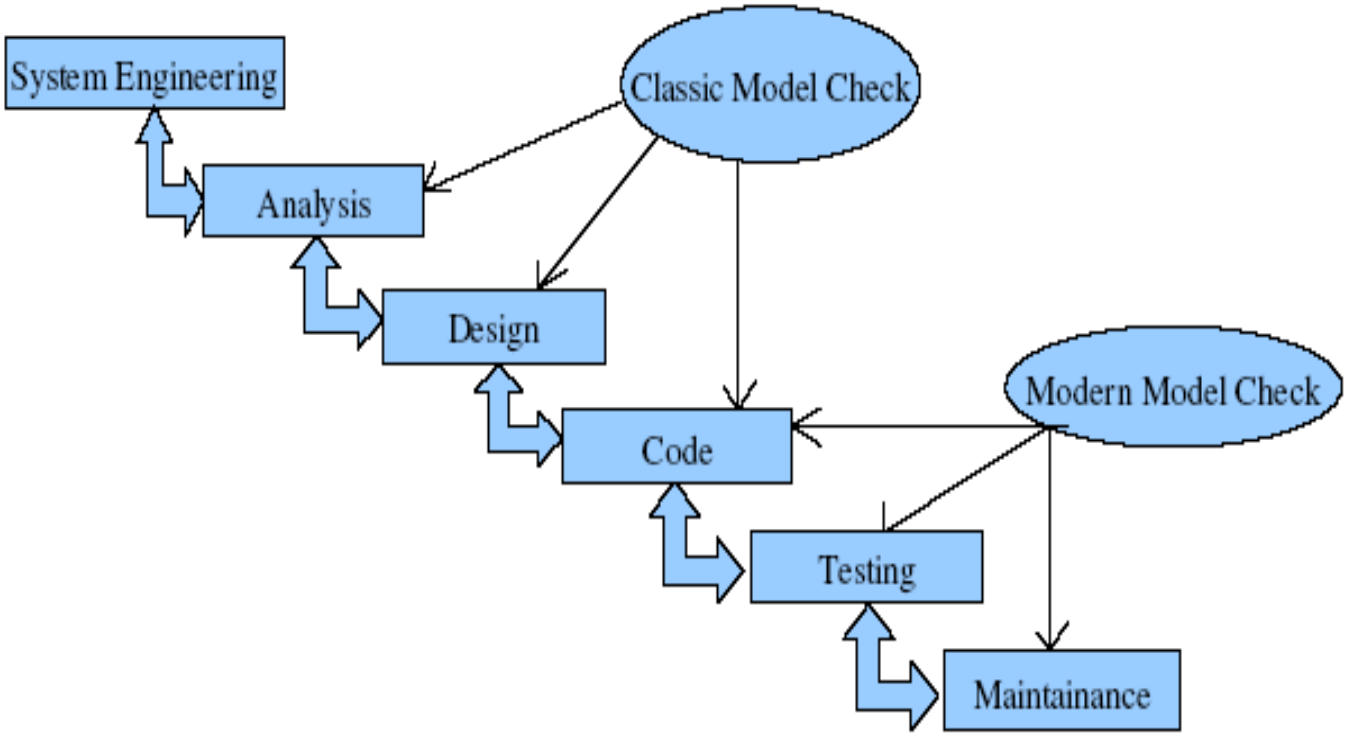


Figure 7: Model Checking at different phases of software development