Verification of mutual exclusion algorithms with SMV System

Nikola Bogunović, Edgar Pek
Faculty of Electrical Engineering and Computing
Unska 3
Croatia
email: nikola.bogunovic@fer.hr, edgar.pek@fer.hr

Abstract—**Mutual exclusion algorithms can exhibit intricate behavior for which correctness can be hard to establish. We demonstrate automatic verification of five algorithms by symbolic model checking. We used SMV tool which enables property specification in computation tree logic and allow us to impose fairness constraints on a model. For each of the algorithm we verify safety, liveness, non-blocking and no strict ordering properties.**

Keywords—**Formal verification, Model checking, Mutual exclusion algorithms, CTL, SMV.**

I. Introduction

WHEN concurrent processes share a resource such as a file on a disk or a database entry, it may be necessary to ensure that they do not have access to it at the same time. Several processes simultaneously editing the same file would not be desirable. Access to shared data is an essential part of distributed and concurrent computing, and algorithms to arbitrate concurrent accesses are at the heart of all operating systems. Sound and safe operation of safety-critical systems depends in the first place on validated and verified procedures that manipulate shared resources.

Formal methods applied in developing computer-based systems are mathematical procedures for describing and verifying system properties and its behavior. Hence, formal verification is the process of ensuring that formal model of the design (*Imp* - implementation) satisfies a formal specification (*Spec*) with mathematical certainty, or plainly that an implementation conforms to specification. The possible conformance relations are equivalencies, various simulation relations, (logical) satisfaction, (logical) implication, refinement, etc. Formal verification guarantees beyond any doubt the correct relation between the mathematical objects *Imp* and *Spec*. However, it does not establish the correctness of the real implementation abstracted by *Imp*.

Formally verifying computing algorithms and protocols are know to be a difficult task since interesting systems, modeled by a transition structure, will have millions of states. In the classical proof-based verification, the system description is a set of formulas Γ (in a suitable logic) and the specification is another formula φ. The verification method consists of trying to find a proof that $\Gamma \vdash \varphi$ (i.e. by applying allowable rules to the set Γ, derive φ). In a model-based approach [1], a finite model M in an appropriate logic represents the system. The specification is again represented by formula φ. The verification method consists of computing whether a model M logically satisfies φ (i.e. $M \models \varphi$). The model-based approach is potentially simpler that the proof-based approach, for it is based on a single model M, rather than a possibly infinite class of them. In model checking one is not concerned with semantic entailment ($\Gamma \models \varphi$), or with proof theory ($\Gamma \vdash \varphi$), but rather with the notion of *satisfaction*, i.e. the satisfaction relation between the model and a formula ($M \models \varphi$).

The paper considers formal verification of mutual-exclusion algorithms taking the model checking approach. In the next section mutual exclusion algorithms are informally introduced. Several desired properties of such algorithms are explicated and a brief record of proposed solutions given. In the following section the implementation modeling formalism (SMV) is presented, supported by the formal specification notion that is denoted in the branching-time logic (CTL). Actual verification procedures and results are described in the subsequent section. Finally, a conclusion gives some valuable comments on the conducted experiments and obtained results.

II. The Mutual exclusion algorithms

We identify critical sections of each process' code and arrange that only one process can be in its critical section. An entry section at the beginning and an exit section at the end frame a critical section of code; these sections act to grab and release the "lock" on that section. To simplify the model, we will assume that there is only one critical section, and not multiple sections of code that have to be mutually exclusive. This simplification does not weaken the model [2]. The problem that we are faced with is to find a protocol for determining which process is allowed to enter its critical section at which time. Once we have found one which we think works, we verify the solution by checking that it has some expected properties. When discussing distributed and concurrent algorithms, there are two kinds of properties with which we must be concerned. A *safety property* is one which guarantees that a bad thing will not happen. These are fairly easy to prove about algorithms because only the static state of the system at any particular time is taken into account. The other kind is a *liveness property*, which guarantees that a good thing will eventually happen. This is much more difficult to prove since time must be taken into account. Proving the correctness of programs is essential with concurrent algorithms because our intuitive notion of program flow is violated by the fact that the value of a variable may not be the same from one statement to the next, even if the process we are tracing does not modify it. Only through formal techniques can we be guaranteed that the above properties will be satisfied. With the problem of mutual exclusion, one safety property is that of *mu-*

tual exclusion; that is, not more than one process should have its program counter in the critical code at the same time. To prove this, it suffices to show that one process cannot leave the entry section while other process is between the entry section and the exit section. Another desirable safety property is that of *freedom from deadlock*. This property guarantees that not all processes will be stuck in the entry section waiting for another process to enter the critical section. There are two liveness properties that are desirable with mutual-exclusion algorithms. The first is that of *freedom from livelock*. This property can be phrased as, "If some process wants to enter the critical section, then *some* process will eventually enter the critical section." The other liveness property, *freedom from starvation*, is stronger than the first and is phrased as, "If some process wants to enter the critical section, then that process will eventually enter the critical section." The latter property implies the former, but is more difficult to guarantee. Notice also that freedom from livelock implies freedom from deadlock.

The problem of mutual-exclusion was first proposed in 1962 by T. Dekker, but no correct solution for more than two processes was published until 1965, when Edsger Dijkstra wrote his one-page article [3] giving a working but complicated solution to the problem. His solution guaranteed mutual exclusion, freedom from deadlock, and freedom from livelock, but allowed the possibility of one process being forever stuck in the entry section while other processes are allowed into the critical section. This paper was quickly followed by Donald Knuth's equally complicated solution [4], which did guarantee freedom from starvation.

In 1981, seven years after Knuth's article, Gary Peterson published his algorithm [5] for two processes, and also gave an algorithm for more than two processes. The two-process algorithm was so simple that he felt that a proof of correctness was not necessary. Both in this algorithm and in all of the previous ones, the processes busy-wait until they are allowed to enter the critical section. The overhead of context-switching is usually large enough that busy-waiting is more efficient.

III. Property specification in CTL

A formal specification language S provides a notation (syntactic domain), a universe of objects (semantic domain), and a precise rule defining which objects satisfy each specification. S is a triple $< Syn, Sem, Sat >$, where Syn and Sem are sets and $Sat \subseteq SynSem$ is a relation between them. The more explicit specification language implies the better-defined formal method. Specification language that is fundamentally advantageous in specifying system behavior should belong to the family of temporal logic. In this paper we concentrate on the branching discrete time temporal logic, i.e. computation tree logic (CTL) formulae [6].

Formula in CTL temporal logic can be true for some states of the underpinning model (Kripke structure) and false for other states. The Kripke structure, or the model M, is defined as a triple $M = \{S, R, L\}$, where S is a set of states, R is a transition relation in the structure, and L is a labeling function that assigns atomic propositional formulas to every state in S. Formulae may change their truth values as the system evolves from state to state. CTL formulae are built from atomic propositions, standard Boolean operators, and temporal operators. Atomic propositions express state properties of the system (e.g. if in the particular state a process requests entry in its critical section, then in this state the proposition $request\_to\_enter\_critical\_section$ is asserted). Temporal operators describe when these properties have to be satisfied. Each temporal operator consists of a path quantifier and a temporal modality. The path quantifier A indicates that a property is true for all computations (traces, paths), and E denotes that it is true for some computation (trace, path). The temporal modality describes ordering of events in time of computation, and can be one of the following: F (in some future and possibly current state), G (in every state), X (in the next state), U (strong until). In CTL one can easily express properties such as: "For any state, if a process requests entering into the critical section, it will eventually be granted":

$$AG[(request = true) \Rightarrow AF(critical\_section)]. \quad (1)$$

Any CTL formula φ could be interpreted within the given Kripke structure M as denoting a set of states, namely those states Q for which $(M, s \models \varphi)$ holds, or formally:

$$Q(\varphi) \subseteq S \text{ is a set of states: } Q(\varphi) = s \mid M, s \models \varphi. \quad (2)$$

Usually we require that the initial state is included, i.e.$s_0 \in Q(\varphi)$. Clearly, the verification procedure encompasses exploration of states as the system evolves from the initial state s_0 in discrete time steps.

The kind of practically relevant properties that need to be checked fall in to three categories. *Safety* (invariant) properties ensure that nothing bad ever happens in the considered system. A failure can be demonstrated by a finite transition sequence. E.g.: "From any state it is possible to get to the state where restart=True":
AG (EF restart=true)

Liveness (progress) properties ensure that something good should eventually happen. A failure can be demonstrated only by an infinite transition sequence. An example of liveness property is given by the Eq. 2. The verification of $M \models \varphi$ might fail because the model M may contain unrealistic behavior that is guaranteed not to occur in the actual system. Instead of refining the model M, one can impose a filter on the model check of the original M. On top of the described transition system one may impose *fairness* constraint (in CTL syntax) stating that a given formula is true infinitely often along every computation path. The meaning of this constraint is that CTL specifications (safety and liveness) are evaluated only over such *fair computations*, i.e. A and E connectives range only over fair paths. As an example one assumes that a process can stay in the critical section as long as it needs, but that it will eventually exit after some finite time (will not stay there forever). Such a fairness constraint is given by: $FAIRNESS\ !(process\_state = in\_critical\_section)$

IV. Implementation modeling in SMV

The SMV system is a tool for checking finite state systems against specifications in the temporal logic CTL [7]. The input language of SMV is designed to allow the description of finite state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract. The

language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only data types in the language are finite ones.

The primary purpose of the SMV input language is to describe the transition relation of a finite Kripke structure. The input file describes both the model and the specification (with possible fairness constraints). The states are defined by a collection of state variables, which may be of Boolean or scalar type. The transition relation of the Kripke structure is determined by a collection of parallel assignments, which are introduced by a keyword ASSIGN. The semantics of assignment in SMV is similar to that of a single assignment data flow languages. A program can be viewed as a system of simultaneous equations, whose solutions determine the next state. When a set is assigned to a variable, the result is a non-deterministic choice among elements of the set. Non-deterministic choices are useful for describing systems which are not yet fully implemented, or abstract models of complex protocols, where the value of some state variables cannot be completely determined.

V. Algorithmic Implementation and Evaluation

In this section we present an algorithmic description of mutual exclusion algorithms as in [8]. As an example of the SMV description we provide implementation of the Dekker's algorithm.

For all algorithms we check following properties expressed in CTL [9]:

1. *Safety:* $\phi_1 \stackrel{def}{=} AG(proc_1.crtical \wedge proc_2.critical)$.
2. *Liveness:* $\phi_2 \stackrel{def}{=} AG(proc_i.trying \rightarrow AFproc_i.critical)$.
3. *Non-blocking:*
$\phi_3 \stackrel{def}{=} AG(proc_i.noncritical \rightarrow AFproc_i.trying)$.
4. *No strict sequencing:*
$\phi_4 \stackrel{def}{=}$
$EF(proc_1.c \wedge E\,[proc_1.c\,U\,(proc_1.c \wedge E[proc_2.c\,U\,proc_1.c])])$.
Observe, that in this context $proc_i.c$ is same as $proc_i.critical$.

Alternatively, we can write safetey property as:
$\phi_1 \stackrel{def}{=} EF(proc_1.crtical \wedge proc_2.critical)$.
In the first case we can get counterexamples from modelchecker, while in second we can just see if a property is violated (CTL property is $true$). Note, that we check only the stronger liveness property (see section II).

A. Dekker's algorithm

Dekker's algorithm considers two processes, there are two boolean variables b_1 and b_2 whose initial values are $false$ and variable k which can arbitrary take value 1 or 2.

In Figure 1 we present SMV implementation for Dekker's algorithm. There are two modules, the first module `main` has a special meaning in SMV, while the second module `proc` represents process as described in Algorithm 1. In module `main` we define state variables `b1`, `b2` and `k`, which correspond to shared variables for processes. Further, variables `pr1` and `pr2` are instances of module `proc`. This variables are instantiated using keyword *process* which means that `pr1` and `pr2` are parallel processes, whose actions are interleaved in execution sequence of the program [7].

Algorithm 1 Dekker's algorithm

```
while true do
  <noncritical section>;
  b_i := true;
  while b_j do
    if k = j then
      b_i := false;
      while k = j do
        skip;
      end while
      b_i := true;
    end if
  end while
  <critical section>;
  k := j;
  b_i := false;
end while
```

The specification of the system φ as CTL formula is introduced in SMV system using keyword `SPEC`. CTL specifications described in the begining of this section are implemented as shown in Figure 2.

In modelling mutual exclusion algorithm it is important to assure fair access to resources as it was explained in section III. In SMV system this is achieved using *fairness constraints*, which restrict search tree to execution paths along which an CTL formula is true infinitely often. The fairness constraint is introduced using keyword `FAIRNESS`. In our model of mutual exclusion we have defined three fairness constraints. The first is associated with special variable `running` which is true only if that process is curently executing. We force that both instances to execute infinitely often using that constraint. The second constraint ensures that process can't stay in the critical section forever. The third constraint assures that process doesn't stay in the noncritical section forever. Fairness declaration can be seen in Figure 3.

Using SMV model checker we have obtained that model M of Dekker's algorithm (Figure 1) has satisfied desired system specification φ (Figure 2).

B. Dijkstra's algorithm

Dijkstra's algorithm considers n processes where $n \geq 2$. Shared variables are:
b, c : array[1..n] of boolean
k : integer.
Initially, all components of b and c array have value $true$, and the value of k is arbitrary chosen from interval $[1, n]$.

Our model of Dijkstra's algorithm considers two processes, so there are some obvious simplification related to variables and `for` loop. System specification and fairness constraint are implemented identically as in Dekker's model. Evaluation of system specification reveals that model of Dijkstra's algorithm doesn't satisfy liveness property. Using SMV generated counterexample we can see why propery is violated. Suppose that initially k is 2, and process 1 executes, and sets b_1 variable to $false$. Process 2 executes and sets b_2 to $false$, then process 1 sets $c1$ to $true$, because $k \neq 1$. Process 2 sets variable c_1 to $false$ ($k = 2$), then proceeds to critical and noncritical section.

```
MODULE main
VAR
  b1  : {true, false};
  b2  : {true, false};
  k   : {1, 2};
  pr1 : process proc(k, b1, b2, 1);
  pr2 : process proc(k, b2, b1, 2);
ASSIGN
  init(b1) := false;
  init(b2) := false;
  init(k)  := {1, 2};

MODULE proc(k, bi, bj, i)
VAR
  state : {noncritical, test_bj, ftest_k,
          stest_k, critical};

DEFINE
  j :=
    case
       i = 1 : 2;
       i = 2 : 1;
     esac;
ASSIGN
init(state) := noncritical;
next(state) :=
  case
    state = noncritical : {noncritical, test_bj};
    state = test_bj & (bj = false) : critical;
    state = test_bj & (bj = true) : ftest_k;
    state = ftest_k & (k = j) : stest_k;
    state = ftest_k & (k != j) : test_bj;
    state = stest_k & (k = j) : stest_k;
    state = stest_k & (k !=j) : test_bj;
    state = critical : {critical, noncritical};
  esac;

next(bi) :=
  case
    state = noncritical &
            next(state) = test_bj : true;
    state = ftest_k & (k = j) : false;
    state = stest_k & (k != j) : true;
    state = critical &
            next(state) = noncritical : false;
    1 : bi;
  esac;

next(k)  :=
  case
    state = critical &
            next(state) = noncritical : j;
    1  : k;
  esac;
```

Fig. 1: SMV code for Dekker's algorithm

figure

Then process 2 resets b_2 and c_2 to $true$. This scenario can repeat indefinely. Notice, that process 1 set at the beginning b_1 to $false$, thus indicate that it wish to enter critical section. Similar scenario can be constructed to show that non-blocking property is violated. Safety and no strict sequencing properties are satisfied.

C. Hyman's Algorithm

Hyman's algorithm is an adoption of Dijkstra's algorithm in the case of two processes. There are two variables b_1 and b_2 whose initial value is $false$, and a variable k which can take values 1 or 2 arbitrary.

```
-- Safety (refutation)
SPEC
  EF ((pr1.state=critical)&(pr2.state=critical))
-- Safety
SPEC
  AG !((pr1.state=critical)&(pr2.state=critical))
-- Liveness property
SPEC
  AG ((pr1.state=test_bj)->AF(pr1.state=critical))
SPEC
  AG ((pr2.state=test_bj)->AF(pr2.state=critical))
-- Non-blocking
SPEC
  AG ((pr1.state=noncritical)->EX(pr1.state=test_bj))
SPEC
  AG ((pr1.state=noncritical)->AF(pr1.state=test_bj))

-- No strict sequencing - no alternation
SPEC
  EF ( (pr1.state=critical)
  & E[ pr1.state=critical U (!(pr1.state=critical) &
  E[!(pr2.state=critical) U (pr1.state=critical)])])
```

Fig. 2: System specification in CTL using SMV

figure

```
FAIRNESS
  running
FAIRNESS
  !(state = critical)
FAIRNESS
  !(state = noncritical)
```

Fig. 3: Fairness specification in SMV

figure

Algorithm 2 Dijkstra's algorithm

var j: integer;
while true **do**
  <noncritical section>;
  $b[i] := false$;
  L_i:
  if $k \neq i$ **then**
    $c[i] := true$;
    if $b[k]$ **then**
      $k := i$;
    end if
    goto L_i;
  else
    $c[i] := false$;
    for $j := 1$ to n **do**
      if $j \neq i$ and $c[j]$ **then**
        goto L_i;
      end if
    end for
  end if
  <critical section>;
  $c[i] := true$;
  $b[i] := true$;
end while

Algorithm 3 Hyman's algorithm

while true **do**
  <noncritical section>;
  $b_i := true$;
  while $k \neq j$ **do**
    while b_j **do**
      skip;
    end while
    $k := i$
  end while
  <critical section>;
  $b_i := false$;
end while

Evaluation of the Hyman's algorithm demonstrate that it doesn't preserve mutual exclusion. Based on a counterexample generated using SMV we can explain why this algorithm doesn't satisfy safety property. We consider the case when k is set to 2 and process 1 on starts by setting $b1$ to $true$, thus indicating its wish to enter the critical section. Process 2 still hasn't started, $k \neq i$, so process 1 now tests b_2, which is $false$, so it stays in a loop. Suppose that now process 2 starts by setting b_2 to $false$. Process 1 can now set k to 1, but process 2 determines that $k = i$ so it enters the critical section. Next, process 1 sets variable k to 1, and enters its critical section.

D. Knuth's Algorithm

Knuth's algorithm represents improvement of Hyman's and Dijkstra's algorithm. Hyman's algorithm as we have shown doesn't satisfy mutual exclusion property, while Dijkstra's algorithm doesn't satisfy liveness property. We have verified that our model of Knuth's algorithm satisfies all properties. As in [8] we present only the case when there are two processes. In that case there are two variables c_1 and c_2 which can take values 0, 1, 2 (initially set to 0). The variable k may take values between 1 and 2.

Algorithm 4 Knuth's algorithm

```
while true do
  <noncritical section>;
  L_0 : c_i := 1;
  L_1 :
  if k = i then
    goto L_2
  end if
  if c_j ≠ 0 then
    goto L_1
  end if
  L_2 : c_i := 2;
  if c_j = 2 then
    goto L_0;
  end if
  k := i;
  <critical section>;
  k := j;
  c_i := 0;
end while
```

E. Peterson's Algorithm

Peterson's algorithm considers only two processes. There are two boolean variables b_1 and b_2 which are initially set to $false$, and a variable k which can initially be 1 or 2. Similarly, as models of Knuth's and Dekker's algorithm, the model of Peterson algorithm satisfies all specified CTL formulas.

Algorithm 5 Peterson's algorithm

```
while true do
  <noncritical section>;
  b_i := true;
  k := j;
  while b_j and k = j do
    skip;
  end while
  <critical section>;
  b_i := false;
end while
```

	Safety	Liveness	Non-blocking	No strict ordering
Dekker	+	+	+	+
Dijkstra	+	-	-	+
Hyman	-	-	+	+
Knuth	+	+	+	+
Peterson	+	+	+	+

TABLE I: Evaluation of properties for mutual exclusion algorithms
table

In Table I we summarize results obtained by model checking mutual exclusion algorithms against CTL specifications for safety, liveness, non-blocking and no strict ordering properties.

VI. Conclusions

We have presented formal verification of five mutual exclusion algorithms. The mutual exclusion algorithms represent simple concurrent systems with subtle behavior. We have used the SMV, tool for automatic formal verification, which provides suitable language to design a model of the concurrent system. It also enables property specification in CTL, and definition of fairness constraints. Future work will consider automatic formal verification of the algorithms for n processes using abstractions. Also, we will study granularity of transitions to gain better understanding when constructing a model of concurrent systems. Insights obtained when specifying and verifiying simple concurrent systems are essential for complex software and protocol verification.

References

[1] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.

[2] L. Kasteloot, "Survey of mutual-exclusion algorithms for multiprocessor operating systems." [Online]. Available: http://tofu.alt.net/ lk/242.paper/242.paper.html

[3] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, 1965.

[4] D. E. Knuth, "Additional comments on a problem in concurrent programming control," *Communications of the ACM*, vol. 9, no. 5, pp. 321–322, May 1966, letter to the editor.

[5] G. L. Peterson, "Myths about the mutual exclusion problem," *Information Processing Letters*, vol. 12, no. 3, pp. 115–116, June 1981.

[6] E. A. Emerson, "Temporal and modal logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. New York, N.Y.: Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990, vol. B: Formal Models and Semantics, ch. 14, pp. 996–1072.

[7] K. McMillan, *The SMV system*, 1992 (2001). [Online]. Available: http://http://www-2.cs.cmu.edu/ modelcheck/smv.html

[8] D. J. Walker, "Automated analysis of mutual exclusion algorithms using CCS," *Formal Aspects of Computing*, vol. 1, no. 3, pp. 273–292, 1989.

[9] M. R. A. Huth and M. D. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge, England: Cambridge University Press, 2000.

[10] R. Meolic, T. Kapus, E. Gungl, and Z. Brezočnik, "Verifi cation of mutual exclusion algorithms with est," in *Proceedings of the Tenth Electrotechnical and Computer Science Conference ERK'2001 Portorož, Slovenia*, B. Zajc, Ed., vol. B. Ljubljana, Slovenia: Slovenia Section IEEE, Sept. 2001, pp. 15–18.