Homework: Model-checking

Reynald Affeldt → Pr. Masami Hagiya

January 17, 2000

The introduction is just a reminder about PROMELA and SPIN. The use of SPIN is pictured in the second part where Dekker's algorithm is checked for safety and liveness. We then compute the formula $E\ F^{\infty}$ by three different methods.

Contents

1 Reminder

SPIN is a tool for verifying asynchronous process systems described in a modeling language called PROMELA. It can perform random simulations or generate a C program that performs efficient verification of correctness properties.

SPIN checks by default for the absence of deadlocks, unspecified receptions, and unexecutable code. It can verify the correctness of system invariants, find non-progress execution cycles, and verify correctness properties expressed in linear-time temporal logic formulas.

1.1 PROMELA

In PROMELA there is no difference between conditions and statements: they are either executable or blocked. Declarations and assignments are always executable. Conditions are only executable when they hold.

The behavior of a process is defined in a `proctype` declaration. Initially, just one process is executed: the one declared in the `init` construct.

Arrow `->` and semicolon `;` are both equivalent statement separators.

The selection structure (`if...fi`) contains execution sequences together with guards checking for their executability. Only one sequence is selected nondeterministically.

The repetition structure (`do...od`) is repeated until the control flow runs into a `break` statement.

The `timeout` models a special condition that allows a process to abort the waiting for a condition that may never become true.

`assert`'s are statements that are always executable and that are intended to produce errors only during verifications.

`end` labels are used to indicate normal end states that otherwise may constitute a deadlock.

`progress` labels mark states that must be executed for the protocol to make progress. An infinite cycle execution that does not pass through at least one progress state is a possible starvation loop.

`accept` labels can also denote acceptance states, and therefore acceptance cycles.

1.2 Spin

1.2.1 Simulation

A simulation is run through a sequence of time steps:

- `-p` shows the state changes of `p`rocesses at every time step.
- `-g` shows the current value of `g`lobal variables at every time step.
- `-l` shows the current value of `l`ocal variables at every time step.

1.2.2 Analyzer

If SPIN is invoked with the `-a` option, it generates a C analyzer that will perform verification through an exhaustive state space search: It tests all possible event sequences in all possible orders.

By default only a search for safety properties is performed (e.g., `assert` abortions, invalid `end`-states (deadlocks)).

- `-a` option performs acceptance cycles detection.
- `-l` option performs non-progress loops search. (The directive `-DNP` must beused at compile-time.)
- `-f` option enforces a weak-fairness constraint.

Any PROMELA system can also be checked for executions satisfying linear-time temporal formulas defined through `never`-claims.

2 Dekker's Algorithm for Mutual Exclusion

Let $i \in \{0, 1\}$ be a process:

```
for (;;) {
        flags[i] = true;
        while (flags[1-i] == true) {
                if (turn != i) {
                        flags[i] = false;
                        while (turn != i);
                        flags[i] = true;
                }
        }

        /* CRITICAL SECTION */

        turn = 1-i;
        flags[i] = false;

        /* FALSE */
}
```

The above algorithm grants two processes mutually exclusion access to a critical section by manipulating three global variables:

`flags[0]`	process 0 sets `flags[0]` to `true` to enter the critical section
`flags[1]`	process 1 sets `flags[1]` to `true` to enter the critical section
`turn`	indicates who's turn it is to try next

All the variables are initially set to `0` values. First, a process claims the critical section by setting its flag to `true`. Then it checks the flag of the other process.

- While this flag is `true`, the other process is in the critical section. The first process must therefore wait for the critical section to be released in a while-loop.
- If this flag turns out to be `false`, the other process is not in the critical section. The first process can thus enter the critical section.

2.1 Verify the correctness of Dekker's algorithm for mutual exclusion by exploring its state space

SPIN translates each process template written in PROMELA into a finite automaton. The behavior of the concurrent system is obtained by computing an asynchronous interleaving product of those automata that are involved in it and is itself represented by an automaton one may call its *state space*.

In the file `mutex`, we model Dekker's algorithm in PROMELA. We then ask SPIN to produce a stand-alone program analyzer we will compile and run. By default, it will check for the absence of deadlocks and for unexecutable code. These are general safety properties that ensure the correctness of the algorithm.

```
> more mutex
#define true    1
#define false   0

int flags[2];
bool turn;

proctype P(int i) {
        do
        :: (true) ->
                flags[i] = true;
                do
                :: (flags[1-i] == true) ->
                        if
                        :: (turn != i) ->
                                flags[i] = false;
                                (turn != i);
                                flags[i] = true
                        fi
                od;
                turn = 1-i;
                flags[i] = false
        od
}

init {
        turn = 0; flags[0] = 0; flags[1] = 0;
        run P(0); run P(1)
}
> spin -a mutex
> gcc -o pan pan.c
> ./pan
(Spin Version 3.3.7 -- 6 December 1999)
        + Partial Order Reduction

Full statespace search for:
        never-claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid endstates       +

State-vector 40 byte, depth reached 16, errors: 0
      20 states, stored
       5 states, matched
      25 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493   memory usage (Mbyte)

unreached in proctype P
        (0 of 18 states)
unreached in proctype :init:
        (0 of 6 states)
```

Let us comment the output yielded by the program analyzer. By default, a full statespace search has been carried out for assertion violations and invalid endstates (i.e., for absence of deadlocks as seen in the reminder). It is clearly stated that no error has been found during the verification whose state space used 40 bytes of memory per state and whose longest search path contained 16 transitions. As indicated, all the states are reachable, which means that there is no dead code.

We conclude by saying that Dekker's algorithm is correct. The two following safety properties are ensured: no deadlock is possible and there is no unexecutable code.

2.2 In addition to safety, verify liveness in some respect (You may have to introduce some kind of fairness.)

A *liveness property* expresses that something good eventually happens. It is opposed to *safety properties* which express that something bad never happens. *Fairness properties* are very similar to liveness properties, except that they express the reachability by considering only those execution sequences that are fair, that is, informally, that ensure that actions that are infinitely enabled should be infinitely chosen.

To perform such verifications, a correctness claim is first specified as a temporal logic formula, then it is converted into a Büchi automaton (that accepts only those infinite system executions that satisfy the corresponding formula), and a synchronous product with the state space is finally computed. The result is again a Büchi automaton whose accepted language is checked for emptiness. If so, the original claim is not satisfied. In SPIN, we use correctness claims to specify behaviors that are undesirable.

For instance, we may want to ensure that it is impossible for a process to keep the lock of the critical section indefinitely. In other words, the following linear-time logic property is undesirable:

$$\Diamond((\Box\texttt{turn}) \vee (\Box\neg\texttt{turn}))$$

By means of the `never`-claim construct, one may check for this requirement. SPIN first yields the Büchi automaton corresponding to the formula and we will modify our PROMELA program accordingly to take this correctness claim into account.

In fact, the program analyzer finds an acceptance cycle and yields a `.trail` file for further debugging. One may now use the simulator to track the cause of the problem and fix it. See page 6.

Progress Cycles One may also use `progress` labels to exhibit the unfairness of Dekker's algorithm. By affecting a `progress` label to the statement `turn = 1-i`, we indicate that it is desirable for both processes to be fairly granted access to the critical section.

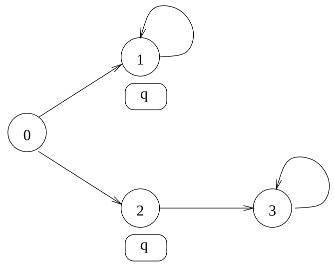
In this case, the program analyzer finds a non-progress cycle, which is considered as a violation of our liveness requirement. Similarly, a `.trail` file is printed out. See page 7.

```
> spin -f "<> (([] turn) || ([] ! turn))"
never {    /* <> (([] turn) || ([] ! turn)) */
T0_init:
        if
        :: ((turn)) -> goto accept_S3
        :: (! ((turn))) -> goto accept_S6
        :: (1) -> goto T0_init
        fi;
accept_S3:
        if
        :: ((turn)) -> goto T0_S3
        fi;
accept_S6:
        if
        :: (! ((turn))) -> goto T0_S6
        fi;
T0_S3:
        if
        :: ((turn)) -> goto accept_S3
        fi;
T0_S6:
        if
        :: (! ((turn))) -> goto accept_S6
        fi;
accept_all:
        skip
}
> vi mutex-modified
> spin -a mutex-modified
> gcc -o pan pan.c
> ./pan -a
pan: acceptance cycle (at depth 18)
pan: wrote mutex-modified.trail
(Spin Version 3.3.7 -- 6 December 1999)
Warning: Search not completed
        + Partial Order Reduction

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

State-vector 44 byte, depth reached 37, errors: 1
      18 states, stored (21 visited)
       1 states, matched
      22 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493   memory usage (Mbyte)
```

```
> more mutex-progress
#define true    1
#define false   0

int flags[2];
bool turn;

proctype P(int i) {
        do
        :: (true) ->
                flags[i] = true;
                do
                :: (flags[1-i] == true) ->
                        if
                        :: (turn != i) ->
                                flags[i] = false;
                                (turn != i);
                                flags[i] = true
                        fi
                od;
progress:               turn = 1-i;
                flags[i] = false
        od
}

init {
        turn = 0; flags[0] = 0; flags[1] = 0;
        run P(0); run P(1)
}
> spin -a mutex-progress
> gcc -DNP -o pan pan.c
> ./pan -l
pan: non-progress cycle (at depth 28)
pan: wrote mutex-progress.trail
(Spin Version 3.3.7 -- 6 December 1999)
Warning: Search not completed
        + Partial Order Reduction

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

State-vector 44 byte, depth reached 37, errors: 1
      18 states, stored (20 visited)
       2 states, matched
      22 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493   memory usage (Mbyte)
```

3 Compute $E\ F^{\infty} q$

Compare the methods for computing the formula $E\ F^{\infty}\ q$. As an example, take the following state transition system.



3.1 By the semantics of the following formula in μ-calculus

$$\nu X.\mu Y.(q \wedge \langle\rangle X) \vee \langle\rangle Y$$

The above system is considered as a Kripke transition system $T = (S, \mathsf{Act}, \rightarrow, I)$, where $S = \{0, 1, 2, 3\}$ is a set of states, $\mathsf{Act} = \emptyset$ is a set of actions, $\rightarrow \subseteq S \times \mathsf{Act} \times S$ is a transition relation, and I is an interpretation with $I(0) = \{\mathsf{true}\}, I(1) = \{\mathsf{true}, q\}, I(2) = \{\mathsf{true}, q\}, I(3) = \{\mathsf{true}\}$ (see [MSS]).

Based on the iterative characterization of fixpoints, the semantics of modal μ-calculus formulas can be effectively evaluated on finite-state Kripke transition systems.

$$\phi_X = \mu Y.(q \wedge \langle\rangle X) \vee \langle\rangle Y$$

and

$$\phi_Y = (q \wedge \langle\rangle X) \vee \langle\rangle Y$$

The iteration process can be summarized in the following table:

Iteration for ϕ_X	1	2	3
Assumption for X Iterations for Y	$\{0,1,2,3\}$ $\{0,1\}, \{0,1,2\}, \{0,1,2\}$ (See 1. below)	$\{0,1,2\}$ $\{1\}, \{0,1\}, \{0,1\}$ (See 2. below)	$\{0,1\}$ $\{1\}, \{0,1\}, \{0,1\}$ (See 3. below)

We conclude by saying that $\{0, 1\}$ satisfies $\nu X.\phi_X$.

1. $X = \{0, 1, 2, 3\}$:

$$\begin{aligned}
F^0_{\phi Y,\rho}(\emptyset) &= \emptyset \\
F^1_{\phi Y,\rho}(\emptyset) &= [\mathcal{M}(q)(\rho[Y \mapsto \emptyset]) \cap \mathcal{M}(\langle\rangle X)(\rho[Y \mapsto \emptyset])] \cup \mathcal{M}(\langle\rangle Y)(\rho[Y \mapsto \emptyset]) \\
&= [\{1,2\} \cap \{0,1,2,3\}] \cup \emptyset \\
&= \{1,2\} \\
F^2_{\phi Y,\rho}(\{1,2\}) &= [\mathcal{M}(q)(\rho[Y \mapsto \{1,2\}]) \cap \mathcal{M}(\langle\rangle X)(\rho[Y \mapsto \{1,2\}])] \cup \mathcal{M}(\langle\rangle Y)(\rho[Y \mapsto \{1,2\}]) \\
&= [\{1,2\} \cap \{0,1,2,3\}] \cup \{0,1\} \\
&= \{0,1,2\} \\
F^3_{\phi Y,\rho}(\{0,1,2\}) &= [\mathcal{M}(q)(\rho[Y \mapsto \{0,1,2\}]) \cap \mathcal{M}(\langle\rangle X)(\rho[Y \mapsto \{0,1,2\}])] \cup \mathcal{M}(\langle\rangle Y)(\rho[Y \mapsto \{0,1,2\}]) \\
&= [\{1,2\} \cap \{0,1,2,3\}] \cup \{0,1\} \\
&= \{0,1,2\}
\end{aligned}$$

2. $X = \{0, 1, 2\}$:

$$
\begin{array}{ll}
F^0_{\phi Y,\rho}(\emptyset) & = \emptyset \\
F^1_{\phi Y,\rho}(\emptyset) & = [\mathcal{M}(q)(\rho[Y \mapsto \emptyset]) \cap \mathcal{M}(\langle\rangle X)(\rho[Y \mapsto \emptyset])] \cup \mathcal{M}(\langle\rangle Y)(\rho[Y \mapsto \emptyset]) \\
 & = [\{1,2\} \cap \{0,1\}] \cup \emptyset \\
 & = \{1\} \\
F^2_{\phi Y,\rho}(\{1\}) & = [\mathcal{M}(q)(\rho[Y \mapsto \{1\}]) \cap \mathcal{M}(\langle\rangle X)(\rho[Y \mapsto \{1\}])] \cup \mathcal{M}(\langle\rangle Y)(\rho[Y \mapsto \{1\}]) \\
 & = [\{1,2\} \cap \{0,1\}] \cup \{0,1\} \\
 & = \{0,1\} \\
F^3_{\phi Y,\rho}(\{0,1\}) & = [\mathcal{M}(q)(\rho[Y \mapsto \{0,1\}]) \cap \mathcal{M}(\langle\rangle X)(\rho[Y \mapsto \{0,1\}])] \cup \mathcal{M}(\langle\rangle Y)(\rho[Y \mapsto \{0,1\}]) \\
 & = [\{1,2\} \cap \{0,1\}] \cup \{0,1\} \\
 & \{0,1\}
\end{array}
$$

3. $X = \{0,1\}$:

$$
\begin{array}{ll}
F^0_{\phi_Y,\rho}(\emptyset) & = \emptyset \\
F^1_{\phi_Y,\rho}(\emptyset) & = [\mathcal{M}(q)(\rho[Y \mapsto \emptyset]) \cap \mathcal{M}(\langle\rangle X)(\rho[Y \mapsto \emptyset])] \cup \mathcal{M}(\langle\rangle Y)(\rho[Y \mapsto \emptyset]) \\
 & = [\{1,2\} \cap \{0,1\}] \cup \emptyset \\
 & = \{1\} \\
F^2_{\phi_Y,\rho}(\{1\}) & = [\mathcal{M}(q)(\rho[Y \mapsto \{1\}]) \cap \mathcal{M}(\langle\rangle X)(\rho[Y \mapsto \{1\}])] \cup \mathcal{M}(\langle\rangle Y)(\rho[Y \mapsto \{1\}]) \\
 & = [\{1,2\} \cap \{0,1\}] \cup \{0,1\} \\
 & = \{0,1\} \\
F^3_{\phi_Y,\rho}(\{0,1\}) & = [\mathcal{M}(q)(\rho[Y \mapsto \{0,1\}]) \cap \mathcal{M}(\langle\rangle X)(\rho[Y \mapsto \{0,1\}])] \cup \mathcal{M}(\langle\rangle Y)(\rho[Y \mapsto \{0,1\}]) \\
 & = [\{1,2\} \cap \{0,1\}] \cup \{0,1\} \\
 & = \{0,1\}
\end{array}
$$

3.2 By inductive predicates on states

$\rightarrow$ and l are two functions such that:

$$\rightarrow: \text{state} \times \text{state} \rightarrow \text{bool} \quad l : \mathbb{N} \rightarrow S$$

The predicate *path* is defined as follows:

$$\mathit{path}(l) = \forall t.l(t) \rightarrow l(t+1)$$

The predicate $\mathit{accept}(s)$, where $s \in S$ holds if $q \in \mathsf{AP}$ holds at state s.
The predicates $\mathit{next}(s_1, s_2)$, where $s_1, s_2 \in S$, and $\mathit{ok}(s)$, where $s \in S$, are defined as follows:

$$\frac{s_0 \rightarrow s \quad \mathit{accept}(s)}{\mathit{next}(s_0, s)}$$

$$\frac{s_0 \rightarrow s \quad \neg\mathit{accept}(s) \quad \mathit{next}(s, s')}{\mathit{next}(s_0, s')}$$

$$\frac{\forall s'.\mathit{next}(s, s') \rightarrow \mathit{ok}(s')}{\mathit{ok}(s)}$$

The formula $E\ F^{\infty}\ q$ can be expressed in higher-order logic in the following way:

$$s \models E\ F^{\infty}\ q \quad \Leftrightarrow \quad \exists l.\mathit{path}(l) \wedge l(0) = s \wedge \forall t_1.\exists t_2.\mathit{accept}(l(t_1 + t_2))$$

Intuitively, it means that the formula is true at s iff there exists a path from s such that q is always true at a later state written here $l(t_1 + t_2)$.

We can show the following equivalence:

$$s \models \neg E\ F^{\infty}\ q \quad \Leftrightarrow \quad \text{ok}(s)$$

as done in [H99].

As far as our state transition system is concerned, we have $\text{next}(0,1)$, $\text{next}(1,1)$, and $\text{next}(0,2)$. If $\neg\text{ok}(2)$, then $\exists s'.\text{next}(2, s') \wedge \neg\text{ok}(s')$. Such a next does not exist, therefore $\text{ok}(2)$. Similarly, we show $\text{ok}(3)$.

It is now possible to take $\text{ok}(0)$ and $\text{ok}(1)$ since there is no contradiction, but the predicates are inductively defined, i.e., they are the minimal predicates satisfying the specified inference rules. In other words, we should try to have as many $\neg\mathrm{ok}(i)$ as possible. If we take $\neg\mathrm{ok}(0)$ and $\neg\mathrm{ok}(1)$, then no contradiction appears. $\neg\mathrm{ok}(0)$ implies $\exists s'.\mathrm{next}(0,s') \wedge \neg\mathrm{ok}(s')$, which is true if $\neg\mathrm{ok}(1)$.

To conclude, we can say that the predicated ok is true only at states 2 and 3. That is:

$$\begin{array}{l} 0 \models E\ F^{\infty}q \\ 1 \models E\ F^{\infty}q \\ 2 \models \neg E\ F^{\infty}q \\ 3 \models \neg F\ F^{\infty}q \end{array}$$

which is the expected result.

3.3 By the nested (depth-first) search

In Spin, we express liveness properties as linear-time logic formulas which are turned into Büchi automata. Spin computes the synchronous product of the latter automaton with the one that constitutes the state space of the given concurrent system. The result is again a Büchi automaton whose language is checked for emptiness: If it is empty, then no execution sequence matches the correctness claim that models an undesirable behavior, this is what is expected.

A Bčhi automaton accepts a system execution iff that execution forces it to pass through one or more of its accepting states infinitely often. We call such behavior *acceptance cycles.* To prove that no execution sequence of the system matches the correctness claim, it suffices to prove the absence of acceptance cycles in the combined execution of the system and the Büchi automaton representing the claim. In Spin, a nested depth-first search is used to detect those cycles.

The principle of the nested depth-first search algorithm is as follows. For an accepting cycle to exist in the state space, at least one accepting state must be both reachable from the initial system state (the root of the graph) and it must be reachable from itself. The first depth-first search establishes which accepting states are reachable from the initial system state. The second (nested) search starts at each accepting state thus detected, and it checks whether or not that state is reachable from itself. If it is, a complete execution sequence that include the acceptance cycle has also been constructed: It is the concatenation of all the steps that are on both the first and the second depth-first search stack. See the algorithm below.

```
proc dfs(s)
        add {s,0} to state space
        for each successor t of s
        do
                if {t,0} not in state space -> dfs(t) fi
        od
        if
                accept(s) -> seed := s; ndfs(s)
        fi
end
proc ndfs(s)
        add {s,1} to state space
        for each successor t of s
        do
                if {t,1} not in state space -> ndfs(t) fi
                else
                if t == seed -> report cycle fi
        od
end
```

In the case of our transition state system, `accept(s)` is true for states 1 and 2. We can thus sketched the following execution:

```
dfs(0)

add {0,0} to state space
1 is the first successor of 0
{1,0} not in state space -> dfs(1)

  add {1,0} to state space
  1 is the only successor of 1
  {1,0} is in state space and accept(1) -> (seed := 1; ndfs(1))

    add {1,1} to state space
    1 is the only successor of 1
    {1,1} is in state space and 1 == seed -> report cycle

2 is the second successor of 0
{2,0} not in state space -> dfs(2)

  add{2,0} to state space
  3 is the only successor of 2
  {3,0} not in state space -> dfs(3)

    add{3,0} to state space
    3 is the only successor of 3
    {3,0} in state space and non accept(3)

  accept(2) -> (seed := 2; ndfs(2))

    add{2,1} to state space
    3 is the only successor of 2
    {3,1} not in state space -> ndfs(3)

      add{3,1} to state space
      3 is the only successor of 3
      {3,1} in state space and t != seed
```

When running into the cycle, one may look at the contents of the first depth-first search stack and the second depth-first search stack onto which the states we are looking for have been pushed: they are indeed the states 0 and 1 from which one may reach a cycle during which q is infinitely accepted. That is the expected result.

References

[H97] Gerard J. Holzmann, *The Model-Checker Spin.*

[MSS] Markus Müller-Olm, David Schmidt, and Bernard Steffen, *Model-Checking: A Tutorial Introduction.*

[H99] Masami Hagiya, *embedding.txt.*

[HPY] Gerard J. Holzmann, Doron Peled, and Mihalis Yannakakis, *On Nested Depth First Search.*