

The Model Checker Spin, Part 2

Verifying LTL-formulas

Wim H. Hesselink, August 25, 2008

1 Stuttering for Accept Cycles

For the purpose of searching for accept cycles, the automaton of the model is extended by Spin (implicitly) with stutter steps when the automaton has no other steps.

Example.

```
bit y = 0 ;
active proctype A () { accept0:    y == 0 ; }
active proctype B () {           y = 1 ; }
```

This model has an accept cycle: when process *B* acts first, the automaton has no steps anymore in an acceptance state. The stutter steps now create an accept cycle.

This is currently not done at the search of non-progress cycles.

Example.

```
bit y ;
byte x ;

active proctype A () {
do
:: ! y -> progress0: x = (x+2) % 5 ;
:: y -> break ;
od
}

active proctype B () { y = 1 ; }
```

According to Spin (version 4.3.0 of 2007), this model has no non-progress cycle, although process *B* can force termination in a non-progress state. This is in conflict with the assertion in [1] (p. 460) that the standard stutter extension is applied. Holzmann has promised me to resolve this.

2 Acceptance and Progress as LTL-formulas

What does it mean to test for the absence of acceptance cycles?

Let X be the state space of the model, i.e. the set of the possible states. Let A be the set of the acceptance states. Let $InfEx$ be the set of the infinite executions of the model. An acceptance cycle is a sequence $xs \in InfEx$ for which there are infinitely many indices i such that $xs_i \in A$. Therefore, testing for the absence of acceptance cycles means verification of

$$\begin{aligned} & \neg(\exists xs \in InfEx : \#\{i \mid xs_i \in A\} = \infty) \\ \equiv & \{ \text{infinite means "always more"} \} \\ & \neg(\exists xs \in InfEx : (\forall n : \exists i : n \leq i \wedge xs_i \in A)) \\ \equiv & \{ \text{predicate calculus, negation through quantifier} \} \\ & \forall xs \in InfEx : (\exists n : \forall i : n \leq i \Rightarrow xs_i \notin A) \\ \equiv & \{ \text{definitions of } \Box \text{ and } \Diamond \} \\ & \Diamond \Box \neg A . \end{aligned}$$

Similarly, let P be the set of progress states. A non-progress cycle is a sequence $xs \in \text{InfEx}$ for which the number of indices i with $xs_i \in P$ is finite. Testing for absence of non-progress cycles means verification of

$$\begin{aligned}
& \neg(\exists xs \in \text{InfEx} : \#\{i \mid xs_i \in P\} < \infty) \\
\equiv & \{ \text{finite means "there is an upper bound"} \} \\
& \neg(\exists xs \in \text{InfEx} : (\exists n : \forall i : n \leq i \Rightarrow xs_i \notin P)) \\
\equiv & \{ \text{predicate calculus, negation through quantifier} \} \\
& \forall xs \in \text{InfEx} : (\forall n : \exists i : n \leq i \wedge xs_i \in P) \\
\equiv & \{ \text{definitions of } \Box \text{ and } \Diamond \} \\
& \Box \Diamond P .
\end{aligned}$$

This shows that testing for absence of acceptance cycles is verification of temporal formulas of the form $\Diamond \Box \neg A$, and that testing for absence of non-progress cycles is verification of temporal formulas of the form $\Box \Diamond P$.

Note that an assumption of weak fairness just influences the set of executions. So the above argument is equally valid with or without weak fairness.

Let us call a property like $\Diamond \Box Q$ a *simple acceptance property* and $\Box \Diamond Q$ a *simple progress property*. We then have that acceptance and progress labels can only serve to verify simple acceptance and simple progress properties. Other temporal formulas must be tested by other means than acceptance and non-progress cycles. This can be done with Spin's LTL manager.

3 Leads-to Properties and Strong Until

An example of a leads-to property is: "If I ever get a bachelor degree, I will eventually get a master degree". Let us not discuss the validity of this property.

If P and Q are state predicates, P leads to Q is defined to mean $\Box(P \Rightarrow \Diamond Q)$. A simple example with a valid leads-to property:

```

byte x = 10 ;

active proctype stepper () { do :: x = (x < 19 -> x+1 : 10) od }

active proctype demon () {
  do
    :: x > 9 -> x = 19
    :: x > 9 -> x = 0
  od
}

```

The demon is enabled if and only if $x > 9$. Whenever enabled, it can nondeterministically choose to make $x = 0$ or 19. The stepper increments x whenever x is less than 19, and otherwise it resets x to 10. Note that the system never terminates. Since the demon can always choose 19, there is no guarantee that x ever equals 9 (even under Spin's weak fairness!).

We have, however, the leads-to property that $x = 1$ leads to $x = 9$. This is expressed by $\Box(x = 1 \Rightarrow \Diamond(x = 9))$, or in ascii $\Box(x == 1 -> \Diamond(x == 9))$. Unfortunately, Spin does not recognize such formulas. So, how to verify it with Spin? We click at verification, set parameters, verify LTL formula, and then submit the formula

```
\Box (p -> \Diamond q)
```

with the symbol definitions

```

#define p (x == 1)
#define q (x == 9)

```

Running the verification, we get the answer *valid*. To see that Spin is really testing, omit one of the guards of the demon and check that the LTL formula is violated. Why?

Apart from the propositional operators and the temporal operators \square and $\langle \rangle$, Spin also has the *strong until* operator U . If P and Q are state predicates, then an infinite execution xs satisfies $P U Q$ if and only if there exists an index k such that $xs_k \in Q$ and that $xs_i \in P$ for all $i < k$.

In order to try it out, we submit the LTL formula $p U q$ with $p = (x < 19)$ and $q = (x = 19)$. We then generate the so-called never claim. Run the verification and obtain the result: not valid. This is due to the demon that can reset x to 0. So, we disable this and replace the corresponding guard by $x > 19$. Then Spin, indeed, accepts $p U q$ as valid.

4 From LTL-formulae to Büchi Automata

In order to verify an LTL formula, Spin's LTL-manager constructs an Büchi automaton. The LTL formula needs to satisfy the syntax

```

f      := p | true | false | f binop f | (f) | unop f .
binop := U | && | || | -> | <-> .
unop  := ! | [] | <> .

```

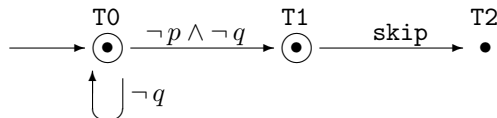
where p stands for an identifier, to be defined in the symbol definitions. The automaton (the *never* claim) is described in the LTL-manager by means of a Promela script. The verifier complains when the conjunction of the never-automaton with your model (extended with the stutter steps of section 1) has an execution that properly terminates or infinitely often visits an acceptance state. Here, the *conjunction* of two models is defined as the model with as state space the cartesian product of the state spaces, with steps that take simultaneous steps in both models. Proper termination means that the never-automaton reaches its final state.

Example. The LTL manager transforms the formula $p U q$ into the automaton:

```

never { /* !(p U q) */
accept_init:
T0_init:
    if
    :: (! ((q))) -> goto T0_init
    :: (! ((p)) && ! ((q))) -> goto accept_all
    fi;
accept_all:
    skip
}

```



Try this out. Note that the LTL manager asks you to state that the property should hold for *all* or for *none* of the executions.

The automaton here consists of three states, say $T0$, $T1$, and $T2$. The first two states are accepting. The initial state $T0$ has a self-loop guarded by predicate $\neg q$. There is a step $T0 \rightarrow T1$ guarded by $\neg p \wedge \neg q$ and a step $T1 \rightarrow T2$ labelled with *skip*. $T2$ is the final state. Why does this test $p U q$?

Consider an infinite execution of your model. If its first state satisfies $\neg p \wedge \neg q$, the automaton can take a first step to state $T1$. The next step of the model can be joined by the automaton since $T1$ has a *skip* step to $T2$. Since $T2$ is the final state, the verifier complains. This complaint is correct, since the execution does not satisfy $p \text{ U } q$. Now assume the first state satisfies q . Then the automaton has no step and there is no joint execution to complain about. Again, this is correct, since the execution indeed satisfies $p \text{ U } q$. Finally, assume that the first state of the execution satisfies $p \wedge \neg q$. Then the automaton can (only) do a step from $T0$ to itself, and we can repeat the arguments. If all states of the execution satisfy $p \wedge \neg q$, the joint execution remains in $T0$. Since state $T0$ is accepting, the verifier complains. Again correctly, for $p \text{ U } q$ is not satisfied.

Due to the stutter extension of section 1, this even works for terminating executions:

```
byte x ;
#define p (x < 4)
#define q (x == 4)

active proctype A () { do :: atomic{ x < 4 -> x ++ ; } od }
active proctype B () { atomic { x == 2 -> x = 5; } }
```

Here $p \text{ U } q$ is not valid because process B can set x to 5 after which both processes are blocked. Indeed, Spin reports an acceptance cycle.

Note that you can make a smaller automaton that does the job: you can remove state $T2$ and the transition towards $T2$.

Example. The LTL-manager transforms the leads-to formula $\Box(p \rightarrow \Diamond q)$ into the automaton:

```
never { /* !([] (p -> <> q)) */
T0_init:
    if
    :: (! ((q)) && (p)) -> goto accept_S4
    :: (1) -> goto T0_init
    fi;
accept_S4:
    if
    :: (! ((q))) -> goto accept_S4
    fi;
}
```

This automaton has two states: the initial state $T0$ and the accepting state $T1$, but no final state. There is a skip step from $T0$ to itself, because of the operator \Box . There is a step from $T0$ to the accepting state $T1$ guarded by $p \wedge \neg q$. State $T1$ has a self-loop guarded by $\neg q$. Whenever p holds and q does not, $T1$ gives the obligation to generate a state where q holds, to avoid the infinite acceptance cycle in $T1$.

References

- [1] G.J. Holzmann. *The SPIN Model Checker, primer and reference manual*. Addison-Wesley, 2004.