Automated Reasoning

Model Checking with SPIN (II)

Alan Bundy
Verifying Global Properties

• Assertions can be used to verify a property **locally**
  – For example, place `assert(MemReturned)` at the end of the process to verify that the requested memory has been returned before the process ends

• Some properties depend on **non-local behaviour** of the model
  – For example, that the system never enters illegal states
  – or that it eventually enters the desired state

• Assertions alone are **not** sufficient for verifying these properties

• A more powerful method is to use “**never claims**”
Never Claims

- Roughly, *never claims* are used to specify system behaviour which should *never* occur.

- In Promela, a never claim can be declared by

  ```promela
  never {
  ...
  <Promela code>
  ...
  }
  ```

- When generating a verifier, *Spin* creates one (active) process for the never claim declared.

- Verifier executes the never-claim process *between every execution* of other processes.

- Verifier reports *error* if the never-claim process ends.
Promela's goto Statement

- We might need to write loops in never claims. The simplest way is to use goto statement

- **Labels** mark locations in the code (Note: a location can have more than one label)

- **skip** means “do nothing” (usually needed because code cannot end with a label)

- goto statements redirect program execution to a labelled location in the code.

- So this code loops through label_1 as long as \( p \) is false
  - Where \( p \) is the property that is never meant to become true.

```
label_1:
  if
    :: p -> goto label_2
    :: else -> goto label_1
  fi;
label_2:
  skip
```
Promela's `do` Loop

The `do` command is another way to write loops in Promela:

```promela
do :: guard -> statement;
    :: guard -> statement;
od;
```

- The `guards` function as in an `if` statement: one statement whose guard is true will be executed.
- But unlike `if` statements, the `do` body repeats.
- The `do` loop ends only if a `break` statement is reached (or if a `goto` statement is used).
Fred's Drinks Revisited

mtype = { milk, lemonade, water };  
bool bought_milk = false,  
   bought_lemonade = false,  
   bought_water = false;  
int num_drinks = 0;

inline buy_item(a_drink, bought_a_drink) {
   (item==a_drink) -> bought_a_drink = true;
}

proctype perhaps_buy(mtype item) {
   atomic {
      bool bought_something = true;  
      if
         :: buy_item(milk, bought_milk);  
         :: buy_item(lemonade, bought_lemonade);  
         :: buy_item(water, bought_water);  
         :: true -> bought_something = false;  
      fi;
      if /* output what Fred bought, if anything */
         :: bought_something -> printf("FRED BOUGHT "); num_drinks++;  
         :: else -> printf("FRED DID NOT BUY ");  
      fi;
      printm(item);printf("\n");
   } }

Note:
num_drinks is now only incremented when Fred buys something.
(This differs from previously)
Fred's Drinks Revisited II

- As an example, in Fred's Drinks model, never claims can be used to verify that Fred can never buy more than two kinds of drinks

- **Desired behaviour:** in all runs, $\text{num\_drinks} \leq 2$ at all times, i.e. in all runs

  \[ \square (\text{num\_drinks} \leq 2) \]

- **Forbidden behaviour:** in some run, $\text{num\_drinks} > 2$ at some time, i.e. in some run

  \[ \Diamond (\text{num\_drinks} > 2) \]
Fred's Drinks Revisited III

- Never claim for this property can be declared as follows:

```solidity
never {  /* <> (num_drinks>2) */
  do
    :: ((num_drinks>2)) -> break;
    :: else -> skip
  od
} FredsNeverClaim
```

- `do`-loop exits and error is flagged whenever `num_drinks` becomes greater than 2
Verifying a Never Claim

- Run `spin -a -N <NeverClaimFile> <ModelFile>` to generate a verifier

% spin -a -N FredsNeverClaim FredsDrinks  # build a verifier
% cc -o pan pan.c  # compile the verifier
% ./pan -e  # run, looking for errors

pan: claim violated! (at depth 20)
pan: wrote FredsDrinks1.trail

% ./pan -rl  # examine the error

... FRED BOUGHT milk ...
... FRED BOUGHT lemonade ...
... FRED BOUGHT water ...
20: proc 0 (:never:) line 71 (state 8)
   -end- ...

- As expected, there's an error trail!
- If the condition is changed to `num_drinks>3`, the verifier will produce no errors
Using LTL with Spin

- Spin can generate a **never** claim from the given LTL formula automatically.

- Run `spin -f '<LTLFormula>'`.

- Spin's LTL syntax:

  - `[]` always (□)
  - `<>` eventually (◇)
  - `X` next (X)
  - `!` logical negation (¬)
  - `U` strong until (U)
  - `&&` logical and (∧)
  - `||` logical or (∨)
  - `->` logical implication (⇒)
  - `<->` logical equivalence (⇔)

- Can use Promela variables or `#defined` symbols as propositions.
Using LTL with Spin (cont)

- Example:

```c
% spin -f '<>p'
never { /* <>p */
T0_init:
    if
        :: ((p)) -> goto accept_all
        :: (1) -> goto T0_init
    fi;
accept_all:
    skip
}
```

- Need to add definition for p separately, e.g.

```c
#define p (num_drinks>2)
```
The Bank Machine Revisited

Model checking is often used to model dynamic systems, such as the bank machine from lecture 9:

We will represent this in Promela, using labels and `goto`, and then we will explore more complicated LTL claims.
Bank Machine Promela Model

Labels correspond to states in our Bank Machine automaton.

The description of current state is recorded in a global variable state (for display).

Notice the non-deterministic choices in if-block.

mtype = \{S_Welcome, S_Enter_Pin, S_Try_Again, S_Amount, S_Thanks_Goodbye, S_Sorry\}

mtype state

inline setState(x) {
  atomic {
    state = x; ... }
}

active proctype BankMachineModel() {
  s_welcome: setState(S_Welcome);
  printf("BANK [card inserted]\n");
  s_enter_pin: setState(S_Enter_Pin);
    if
    :: true -> {
      printf("BANK [wrong pin]\n");
      goto s_try_again;
    }
    :: true -> {
      printf("BANK [correct pin]\n");
      goto s_amount;
    }
    :: true -> {
      printf("BANK [cancel]\n");
      goto s_thanks_goodbye;
    }
  fi;
  s_try_again: setState(S_Try_Again);
  goto s_enter_pin;
  s_amount:
  ....
end: skip;
Bank Machine Runs

% spin BankMachine
BANK state is: S_Welcome
BANK [card inserted]
BANK state is: S_Enter_Pin
BANK [correct pin]
BANK state is: S_Amount
BANK [sufficient funds]
BANK state is: S_Thanks_Goodbye

% spin BankMachine
BANK state is: S_Welcome
BANK [card inserted]
BANK state is: S_Enter_Pin
BANK [correct pin]
BANK state is: S_Amount
BANK [sufficient funds]
BANK state is: S_Thanks_Goodbye

% spin BankMachine
BANK state is: S_Thanks_Goodbye

BankMachine
promela code

% spin BankMachine
BANK state is: S_Welcome
BANK [card inserted]
BANK state is: S_Enter_Pin
BANK [correct pin]
BANK state is: S_Amount
BANK [sufficient funds]
BANK state is: S_Thanks_Goodbye

% spin BankMachine
BANK state is: S_Welcome
BANK [card inserted]
BANK state is: S_Enter_Pin
BANK [correct pin]
BANK state is: S_Amount
BANK [sufficient funds]
BANK state is: S_Thanks_Goodbye

% spin BankMachine
BANK state is: S_Welcome
BANK [card inserted]
BANK state is: S_Enter_Pin
BANK [correct pin]
BANK state is: S_Amount
BANK [sufficient funds]
BANK state is: S_Thanks_Goodbye
Verifying Bank Machine

- **Desired behaviour:** all runs eventually reach $S_{Thanks\_Goodbye}$ or $S_{Sorry}$, i.e. all runs satisfies

  \[ \Diamond (\text{state} = S_{Thanks\_Goodbye} \lor \text{state} = S_{Sorry}) \]

- **Forbidden behaviour:** some run remains in other states, i.e. some run satisfies

  \[ \Box (\text{state} \neq S_{Thanks\_Goodbye} \land \text{state} \neq S_{Sorry}) \]

- We cannot tell that a run is forbidden from any (initial) finite portion. (Something unexpected might happen after that!)

Accept Cycles

- Can we write a never claim for this behaviour? Yes, but needs new error detection technique: accept cycles

- **Accept labels**: labels that begin with accept.
  - e.g. `accept`, `accept1`, `accept_all`, `accepting`, ...

- Verifier flags errors if there is a run containing a cycle which goes through a location with an accept label

- Such a run goes through an accept label infinitely often

- **Accept labels** are normally used in never-claim blocks
Verifying Bank Machine (II)

- Define p and q and run
  \[
  \text{spin } -f '[] (\neg p) \&\& (\neg q)\]
  to generate never claim

\[
\begin{align*}
  \text{#define p state==S_Thanks_Goodbye} \\
  \text{#define q state==S_Sorry} \\
  \text{never} \{ \text{ /* [] (\neg p \&\& \neg q) */} \\
  \text{accept_init:} \\
  \text{ T0_init:} \\
  \text{ if} \\
  \text{ :: (\neg ((p)) \&\& ! ((q))) } -> \\
  \text{ goto T0_init} \\
  \text{ fi;} \\
\}
\end{align*}
\]

- Run verifier using \texttt{pan -e -a} to detect \texttt{accept cycles}

- Verifier finds a problematic infinite run where the wrong pin is entered over and over again!
Verifying Bank Machine (III)

- At this stage, we can use the error trail(s) produced to identify and correct bugs in the model
- Let's consider a more complicated behaviour
- **Desired behaviour:** If a run eventually never goes to $S_{Try\_Again}$, then it will either eventually always be in $S_{Thanks\_Goodbye}$ or it will eventually always be in $S_{Sorry}$.

\[
(\Diamond \Box \neg s_{Try\_Again}) \Rightarrow
(\Diamond \Box s_{Thanks\_Goodbye})
\lor
(\Diamond \Box s_{Sorry})
\]
Verifying Bank Machine (IV)

- Negate the LTL claim from the previous slide
- Use `spin -f` to generate our never claim (above)
- Create and run the verifier (at right).
  No error trails found!

```c
#define s_Try_Again (state==S_Try_Again) ...
never { /*!((<>[])!s_Try_Again)->(<>[])s_Thanks_Goodbye)||
    (<>[])s_Sorry) */
    ... 
}
```

```
% spin -a -N BankMachineNever2
 BankMachine
% cc -o pan pan.c
% ./pan -e -a
... errors: 0 ...
```
Spin accept-cycle detection

- A few questions arise:

- If a model does not have the desired property, is it always the case that there will be an error trail which is either finite or eventually repeats in a cycle?
  - Yes, provided that the desired property is specifiable in LTL or using never claim (+ accept labels)
  - Such an error trail is said to be regular (i.e. it is a path in a finite graph)
  - Examples:
    - Finite run: \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_n \)
    - Regular run: \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \)
    - Irregular run: \( s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1 \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \)
Spin accept-cycle detection II

- Can Spin always find such an error trail? How?
  - **Yes.** This is where automata theory comes in. Spin constructs a Büchi automaton from the Promela model and the never claim. Then determines if it has an accepting run.
  - Use an algorithm that detects “accept cycles” (aka “bad cycles”) in the automaton.
  - Details in next lecture.

- Follows from well-known results in automata theory:
  1) If a Büchi automaton has an accepting run, then it must have a regular accepting run
  2) There is an effective procedure to find such a regular accepting run
Safety and Liveness Properties

Desired properties are often split into two categories:

- **Safety**
  - “Nothing bad ever happens”
  - Model checker searches for any possible execution that leads to a violation of a safety property
  - Example: **system invariance** (x never equals zero, or Fred has never bought both milk and water)

- **Liveness**
  - “Something good eventually happens”
  - Model checker searches for any possible execution in which the “good thing” can be postponed indefinitely
  - Example: **responsiveness** (requests get responses), closure (Fred eventually buys 3 drinks, or every bank machine session ends with Sorry or Thanks)
Notes on **Never Claims** and **LTL**

- Never claims (+ accept labels) are strictly more powerful than LTL formulae
  - Properties specifiable in LTL can be specified using never claims
  - but never claims can specify properties not specifiable in LTL

- Code in never claims must be side-effect free; e.g. they are not allowed to modify variables.

- The LTL operator next (\(X\)) is confusing – it is usually better to avoid it. (This is because if there are multiple processes, the next state will be a single step by any process, and \(X(p)\) means that \(p\) holds for all such immediately next states.)
Correctness in Spin

We have seen several ways to specify correctness in Spin:

- **An assert statement** specifies that a property holds at a certain point in the execution of a process
- **Never** claims describe behaviour that should never occur
- **Accept** labels specify that certain cycles should never occur

Spin supports other techniques (see the Spin manual on-line):

- **End** labels specify that all processes end in certain states
- **Progress** labels specify that a model will never run forever without passing through certain states infinitely often
- **Trace** assertions specify that channels send or receive certain types of data
Summary

- Loops in Promela (do and goto)
- Verifying using never claims
- Generating never claims from LTL formulae
- Accept cycles
- Safety and liveness properties