Formal Methods in Software Development
Resume of the 11/11/2020 lesson

Igor Melatti and Ivano Salvo

1 The SPIN model checker

- Each statement may have a label (e.g. `again` in Figure 1)
  - if the label begins with "end", then it is a valid end-state
  - an end-state is valid if it has an "end" label or if it consists of the closing brackets of a process
  - any other state from which it is not possible to execute a transition triggers a verification error, claiming a *deadlock* has been found
- Examples in Figures 1 and 2
- SPIN execution model
  - processes statements are executed in interleaving as in modern operating systems
  - it is possible to specify a statement block not to be interrupted by other processes: `atomic` and `d_step`
  - see Figure 3, which contains some simplifications
  - e.g., it could be possible to have non-determinism in atomic blocks too
  - compare with Murphi execution model
- SPIN state: values of both global and local variables and channels, plus program counters of all running processes
- Again, we define the Kripke structure $\mathcal{M} = \langle S, S_0, R, L \rangle$ corresponding to a given Promela model
  - $S = D_1 \times \ldots \times D_n \times \{1, \ldots, M_1\} \times \ldots \times \{1, \ldots, M_k\}$
    * here we are assuming n (flattened) local and global variables, including channels

```
/* Peterson's solution to the mutual exclusion problem - 1981 */

/* global vars (initialized to 0) */
bool turn, flag[2]; /* was Q in Murphi */
byte ncrit;
/* note that the P array in Murphi is not needed: program counters
 are already automatically handled... */

active [2] proctype user()
{
  assert(_pid == 0 || _pid == 1);
again:
  flag[_pid] = 1; /* process communication via shared memory */
  turn = _pid;
  (flag[1 - _pid] == 0 || turn == 1 - _pid);

  ncrit++;
  assert(ncrit == 1);  /* critical section */
  ncrit--;

  flag[_pid] = 0;
  goto again
}
```

Figure 1: Peterson protocol

```
#define p  0
#define v  1

/* zero dimension channel: rendez-vous */
chan sema = [0] of { bit };

proctype dijkstra()
{   byte count = 1; /* initialized local variable */

    do
    :: (count == 1) ->
        sema!p; /* send 0 and blocks, unless some other proc is
                   already blocked in reception */
        count = 0
    :: (count == 0) ->
        sema?v; /* receive 1, same as above */
        count = 1
    od
}

proctype user()
{   do
    :: sema?p; /* wait for dijkstra process to send 0, unless
                  it was already sent */
       /*     critical section */
       sema!v; /* send 1 to dijkstra ("I finished") */
       /* non-critical section */
    od
}

init
{   run dijkstra();
    run user();
    run user();
    run user()
}
```

Figure 2: Dijkstra protocol

```
/* Make a random walk in the NFSS described by SD */
void Make_a_run(SpinDescription SD)
{
  /* only one initial state */
  s := all non-initialized global variables are 0, all channels are
   empty;
  foreach active proctype p in SD
    add p as a running process in s with p.pc=1;
  if (SD contains the init process)
    add init as a running process in s with init.pc=1;
  s_current := s;
  while (1) { /* loop forever (unless an error occurs) */
    if (∃ running process p in s_current s.t. p.pc is in an atomic
     block)
      may_be_exec := statement istr at p.pc;
    else {
      may_be_exec := ∅;
      /* we do not deal with the rendez-vous communications */
      foreach running process p in s_current {
        foreach statement istr at p.pc {
        /* "pc" is the process program counter */
          if (istr is executable in s_current)
            may_be_exec = may_be_exec ∪ istr;
    } } }
    if (may_be_exec = ∅)
      error "Deadlock"; /* other errors may be checked */
    istr := pick at random a statement in may_be_exec;
    s_next := execute(s_current, istr);
    s_current := s_next;
  } /* while */
} /* Make_a_run() */
```

Figure 3: SPIN execution model

    * we also assume there are k running processes, with process i having M_i statements inside it
    * if a D_i corresponds to `short` or `int`, then it has 2^{16} or 2^{32} values on a typical 64-bit architecture, as it is in C
    * a channel is essentially an array of structures
    * SPIN does not have a special value for "undefined" (as Murphi has), but $\perp$ is needed for the local variables still not reached by the program counter
    * indeed, this state space is *dynamic*, as it contains the *currently running* processes
    * new processes may be added at any time by a `run` statement
    * thus, the state space cannot be defined *in advance* as it is with Murphi; this is only possible when only `active` proctypes are used, without `run` commands
    * even in this case, it is possible to some local variables definition is still not reached by the process program counter, and thus they actually don't exist...
  - $|I| = 1$, see Figure 3
  - R is intuitively defined as follows (also check Figure 3): $R(s, s')$ holds iff there is a running process p in s and an executable statement t at the current program counter of p (recall that the program counter for all processes is stored in s) s.t. t, when executed, leads from s to s'
    * if t is the beginning of an atomic sequence, then the whole atomic sequence must be executed
    * till the first blocking statement of the sequence
    * if t is a send on an empty channel c, and there is another current statement t' in another process p' (i.e., the value of the program counter of p' in s identifies t' as the next statement to be executed for p' in s) s.t. t' is a receive on c, both t and t' have to be executed when leading from s to s'
  - L is similar to Murphi, i.e., equations between (global and local) variables and values; however, also program counters must be considered

2 SPIN Verification Algorithm

- Able to answer to the following questions: is there a deadlock (invalid end state)? are there reachable assertions which fail (safety)? is a given LTL formula (safety or liveness) ok in the current system?
- Similar to Murphi:

1. the SPIN compiler (`SrcXXX/spin -a`) is invoked on `model.prm` and outputs 5 files, `pan.c`, `pan.h`, `pan.m`, `pan.b`, `pan.t` (unless there are errors...)
2. the 5 files given above are compiled with a C compiler; in this way, an executable file `model` is obtained; it is sufficient to compile `pan.c`, which includes all other files;
3. just execute `model` (option `-h` gives an overview of all possible options)

- PAN: Protocol ANalyzer
  - `pan.[ch]` is the fixed part of the verifier, it implements a DFS (also BFS starting from some later version, but less efficient), it also includes the other files
  - `pan.m` is the part of the verifier which depends on the Promela model: it contains a C `switch` statement implementing the transition relation
    - very similar to Murphi `Code` implementing a rule body
    - given the current state, saved in a memory buffer called `now` and very similar to the Murphi's `workingstate`, given a running process index i and the program counter p inside that process, it performs on `now` the modifications demanded by the Promela statement at line i of process p, so obtaining the next state
    - of cours, it takes into account special cases such that atomic sequences and synchronuous communications
  - `pan.b`: the same of `pan.m`, but *backwards*!
    - actually, `pan.m` does not surprise and it is not conceptually difficult to understand and implement
    - implementing the same backwards is not straightforward, but SPIN does it!
    - essentially, all Promela instruction may be reversed, and the code to reverse them is in `pan.b`
    - essentially, PAN maintains old values for all variables in the state (i.e., values are saved before overwriting due to new assignments)
    - thanks to the fact that the visit is a DFS (SPIN is optimized for DFS), it is only needed to maintain the *last* values, thus a stack for each variable is used for this purpose
  - `pan.t` creates a table with an entry for each statement in the source Promela model; for each statement, the corresponding values to execute the forward and backward in `pan.[bm]` are stored (needed for simulations and counterexamples)
- On-the-fly exploration: as in Murphi, the RAM contains only the part of the graph which has been explored till now

  - only the states, no transitions between them

- Hash table for the visited states
  - Murphi uses open addressing, here the hash table is handled with collision lists
  - in order to speed up visited states check, such lists are ordered (i.e., each new state is inserted in order)
- We already said that SPIN uses a DFS instead of Murphi BFS; so one could think to something such as Figure 4, i.e., a recursive implementation
- This is not what it is done by SPIN, as it is meant to be implemented in the most efficient way
- Thus, instead of using the standard implicit (and not efficient) call stack as in Figure 4, we have an ultra-light explicit stack
  - recall that Murphi had a queue, since a BFS is performed
- Moreover, recursion is simulated with C `goto` statements! Also global variables are widely used
- This leads us to the DFS in Figure 5, which is closer to what SPIN actually does

```
DFS(graph $G = (V, E)$, node $v$)
{
  Visited := Visited $\cup$ $v$;
  foreach $v' \in V$ t.c. $(v, v') \in E$ {
    if ($v'$ $\notin$ Visited)
      DFS($G$, $v'$);
  }
}
```

Figure 4: Standard recursive DFS

- However, we still need one more element to be added to Figure 5: namely, the stack does *not* store states
- Instead, each stack entry only stores a pair $\langle p, o \rangle$ of indices (integers)
  - p is a process pid
  - o identifies a statement at the current program counter of p
  - (recall that there may be non-determinism inside each process...)
- The rational behind this is the following

```
DFS(graph $G = (V, E)$)
{
  $s$ := init; i := 1; depth := 0;
  push($s$, 1);
Down:
  **if** ($s \in$ Visited)
    **goto** Up;
  Visited := Visited $\cup$ $s$;
  let $V' = \{v' \mid (v, v') \in E\}$;
  **if** ($|V'|$ >= i) {
    $s$ := i-th element in $V'$;
    increment i on the top of the stack;
    push($s$, 1);
    depth := depth + 1;
    **goto** Down;
  }
Up:
  ($s$, i) := pop();
  depth := depth - 1;
  **if** (depth > 0)
    **goto** Down;
}
```

Figure 5: DFS with `goto`s and explicit stack

- there is just one initial state
- let $\langle p_0, o_0 \rangle$ be the first (from the bottom) pair on the stack; it univocally identifies a statement $istr_0$ to be executed
- by applying $istr_0$ to s_0 we obtain a state s_1 (formally, s_1 =apply(s_0, p_0, o_0))
- analoguously, s_2 =apply(s_1, p_1, o_1) if $\langle p_1, o_1 \rangle$ is the second pair on the stack
- thus, a stack $\langle \langle p_0, o_0 \rangle, \ldots, \langle p_d, o_d \rangle \rangle$ univocally identifies a state s_d, obtained by chaining the executions due to pairs $\langle p_i, o_i \rangle$
- formally, $\forall 1 \leq i \leq d$ s_i =apply($s_{i-1}, p_{i-1}, o_{i-1}$)
- moreover, SPIN is able to define the *undo* function, with the same parameters of the apply function
  * of course, apply is defined in `pan.m`, undo in `pan.b`
  * undo needs a stack of values for each variable, as explained above
  * however, it tries to minimise such stacks usage; e.g., if a `c = c + 2` statement must be undone, then it is sufficient to execute `c = c - 2`
  * for direct assigments (e.g., `c = 4`), the apply function puts the preceding values of `v` in the stack of `v` before overwriting it with 4
  * undo will pop the value from the stack of `v` and put it back in `v`
  * this works because the whole visit is a DFS
- finally, recall we have a global fixed structure `now` implementing the current state (same as Murphi's `workingstate`)
- summing up, given what we said (Figure 6):
  * no need of pushing a whole state s in the DFS stack: SPIN pushes the pair $\langle p, o \rangle$ which generates s if applied to the current state
  * no need of popping a state s: SPIN pops the pair $\langle p, o \rangle$ which generates s if undone on the current state

• Finally, `ch13.pdf` adds some more details

- atomic sequences handling:
  * if we are inside an atomic sequence, SPIN must take care that only the current process can execute
  * this is done by setting `From = To = II` (line 44), which forces the `for` loop in line 24 to oly select the current process
  * normal behaviour is reprised at line 46
  * a state may be searched and possibly inserted in the hash table (line 13) only if we are not in an atomic sequence

```
DFS(NFSS $\mathcal{N}$)
{
  let $\mathcal{N} = (S, \{q\}, \mathcal{A}, \texttt{next}, L)$;
  now := init; depth := 0;
Down:
  if (now $\in$ Visited)
    goto Up;
  Visited := Visited $\cup$ now;
  foreach p s.t. p is a running process in now {
    foreach opt s.t. opt is enabled at p.pc {
      now := apply(now, p, opt);
      /* no need of incrementing opt on the top of the
         stack: when popping, it will be done by the
         foreach on opt... */
      push(p, opt);
      depth := depth + 1;
      goto Down;
Up:
      (p, opt) := pop();
      depth := depth - 1;
      now := undo(now, p, opt);
  } }
  if (depth > 0)
    goto Down;
}
```

Figure 6: SPIN DFS

- `timeout` handling:
  * it is a Promela boolean expression, which is true iff the whole system deadlocks (all processes must execute non-executable statements)
  * thus, when the double `for` at lines 24 and 28 is finished without any statement being executable (thus, `n` is still 0) and this is not a valid end state, PAN tries to perform the whole computation again with `timeout` set to 1
  * linea 46 reprises the normal non-`timeout` behaviour
- apply ed undo are implemented in `pan.m` (included at line 30) and `pan.b` (line 54)
  * if a statement cannot be executed, `pan.m` performs a C `continue` statement, which forces `for` in line 28 to go on with next iteration
  * otherwise, a `goto P999` is executed
  * instead, `pan.b` executes `goto R999`