Formal Methods in Software Development
Resume of the 30/10/2019 lesson

Igor Melatti and Ivano Salvo

# 1 SPIN Verification Algorithm (PAN): Optimizations

- States compression
- *Byte masking*
  - similar to Murphi bit compression
  - in PAN, the current state vector `now` is essentially a concatenation of C structures, each representing a processes
  - byte masking works by aligning each of such structures to each byte, instead of each 4 bytes (word) as it would be by default with C compiler
  - this is really simple, PAN does this by default (to disable it, you have to compile PAN with `-DNOCOMP`)
  - not very effective
- Collapse compression
  - not present in Murphi, as it is closely related to processes; requires compilation of PAN with `-DCOLLAPSE`
  - it exploits the Promela models structure
  - the idea is to separately storing:
    * processes state (program counter + local variables)
      · each process separated from the others, but if you compile PAN with `-DJOINPROCS` then they will be put together
    * channels state
      · all together, but you could store them separately by compling PAN with `-DSEPQS`
    * global variables values
  - for each of such fragments, an index is generated

- finally, a PAN (complete) state is stored as a vector of indices, which tells how the fragments above must be combined to obtain the complete state
- of course, this works well if there are many combinations of few fragments
  * e.g., this may happen if there are $n$ instances of the same proctype
- in order to check if a (complete) state is already visited or not, PAN does the following
  1. split $s$ in fragments
     * there will be $p + q + g$ fragments
     * note that $p = 1$ with `-DJOINPROCS`, $q = 1$ by default unless `-DSEPQS`, and $g = 1$ (global variables are always together)
  2. for each fragment $f$, PAN checks if $f$ is in the hash table
  3. if not, the state is of course not already visited; a new unique identifier for $f$ is generated and stored together with $f$
     * simply a counter: the $i$-th generated fragment (within the same fragment category) has identifier $i - 1$
  4. otherwise, the unique identifier is returned
  5. finally, $s$ is stored as the list of unique identifiers collected previously

- Hash compaction
  - as in Murphi
  - compile PAN with `-DHC`$n$ for $n$-bytes signatures; default is 2 bytes

- Minimized Automaton
  - kind of hybrid technique between explicit and implicit model checking
  - that is, it is explicit model checking with some ideas from implicit one
  - with this technique, *no hash table is required*
  - it is replaced by a minimized automaton which recognizes visited states
  - of course, states are viewed as sequences of bits
  - in fact, you can always write the set of visited states as a regular expression on their single bits
    * at the worst, as an OR of visited states, each of whom is the AND of its bits
    * this would probably result in a memory occupation which is higher that the standard hash table

    * however, usually this worst case does not occur, and a reduction in the RAM requirements is achieved by simplifying the regular expression with the recognizing automaton, using standard formal language techniques

  - hence, if the regular expression is "regular" enough, the minimized automaton requires less RAM than the hash table
  - generally speaking, in order to perform explicit model checking, the following operations must be allowed:
    1. return 1 if a given state $s$ has already been visited, and 0 otherwise
    2. insert a new state in the old set of visited states, and return the new set of visited states
  - this was straightforward with the hash table
  - with the automaton, operation 1 is still straightforward, operation 2 is not
    * it is necessary to modify the current automaton, by adding and/or deleting nodes and/or edges
  - to this aim, SPIN uses an ad-hoc structure representing a *limited* regular expression (recall that states are finite) and implementing sufficiently well operations 1 and 2
  - that is, a deterministic automaton with $k$ levels is used, being $k$ the maximum length of a state representation
    * such an automaton does not have cycles
  - see `spin_minaut.pdf`
  - the minimized automaton may be well combined with collapse compression
  - in this case, an hash table is brought back, but only to contain states fragments
  - identifiers vectors are stored with the minimized automaton

- PAN also efficiently implement the DFS stack through the *stack cycling* technique
  - the DFS stack is only accessed *sequentially*; no random access
  - thus, it is ok to store the stack on disk
  - a finite-length $M$ portion is kept in RAM, holding the currently needed stack
  - that is, once push and pop operations require to access to a stack portion which is outside RAM, that part is fetched from the disk
  - the block taken from the disk has size $\frac{M}{2}$, in order to avoid going back and forth on the disk due to sequences pop-push-pop-push...

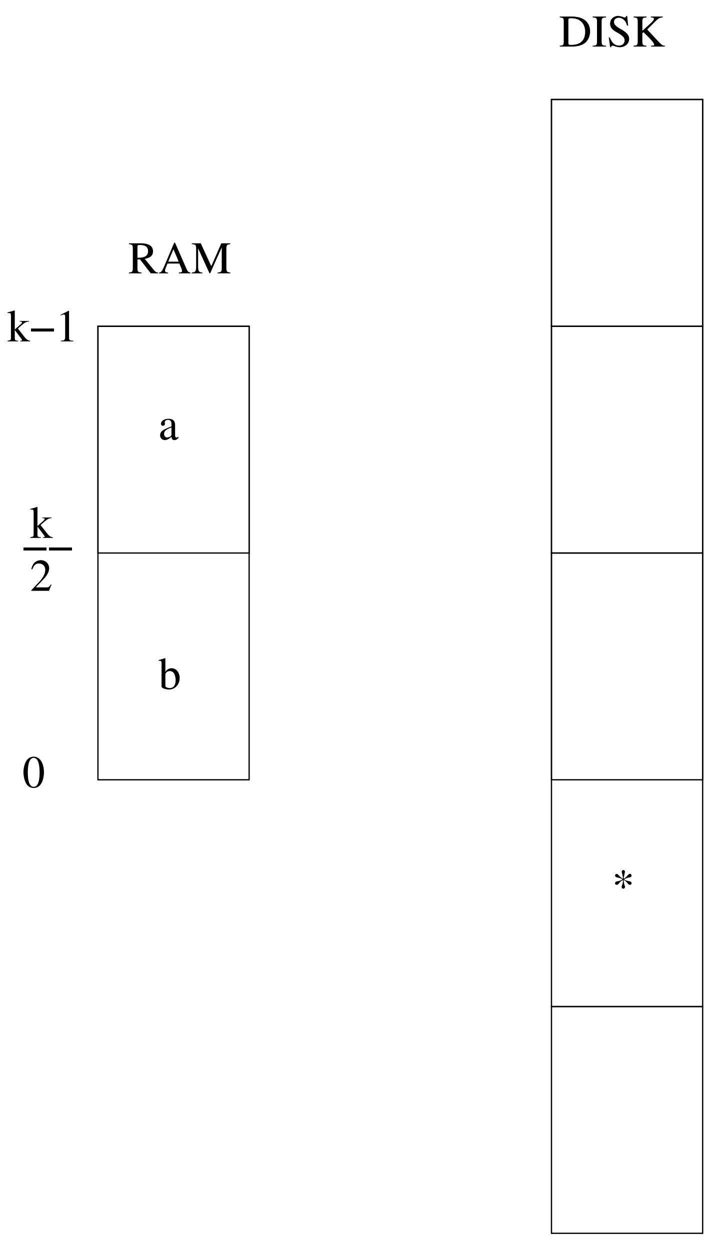


Figure 1: Situazione per lo stack cycling

- see Figure 1, and suppose pushes are towards the top (from 0 to $M - 1$), whilest pops are towards the bottom
- if a push over $k - 1$ is made, more memory is required, and such (clean) memory is fetched from the disk
- in order to do this, the part labelled $b$ is stored in some disk zone (e.g., that highlighted with an asterisk in Figure 1)
- then, $b$ may be overwritten by copying $a$ in it
- on the other end, now $a$ is free, and push may be executed starting from $\frac{k}{2}$
- for pops, the idea is symmetric; this time, fetching a disk zone does not bring a cleared memory buffer, but a part of stack which was stored in the disk previously (as a consequence of former too many pushes)

- All this techniques allow to save memory, when storing the same set of visited states
- It is difficult to tell which method is good for a given Promela model; you can only go for trial and errors
  - i.e., if a method exhaust all available RAM, you try with the following one
- SPIN and PAN also implement a strategy which reduces the number of visited states themselves: the *partial order reduction* (POR)
  - similar to Murphi symmetry reduction, in the sense that the goal is the same
  - however, in Murphi symmetry reduction the modeler is aware of such technique (some variables types such as `multiset` have to be used)
  - in SPIN, POR is applied to nearly all Promela models automatically, with very few execptions
  - the idea for POR is that not all possible interleavings of currently running processes in Promela have to be considered in order to verify the given property
  - this allows to lower down the number of states to be visited
  - from Figure 2 to Figure 3
  - not always applicable: the "diamond" case must be present (see states with X in Figure 2)
    - that is, the order of executing two given instructions must be irrelevant
  - some conditions which guarantee actions independence are in `spin_por.pdf`, pages 2 and 3

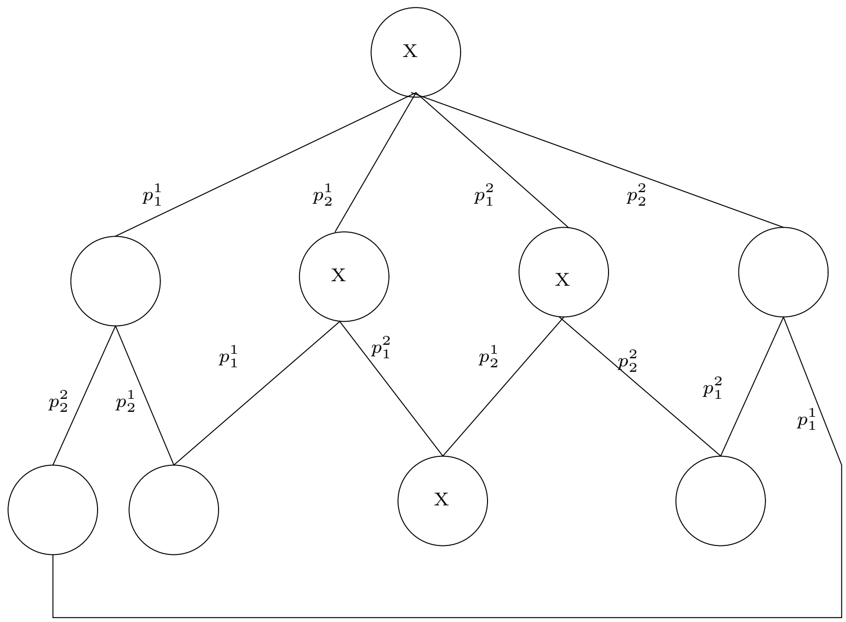


Figure 2: Typical interleaving case

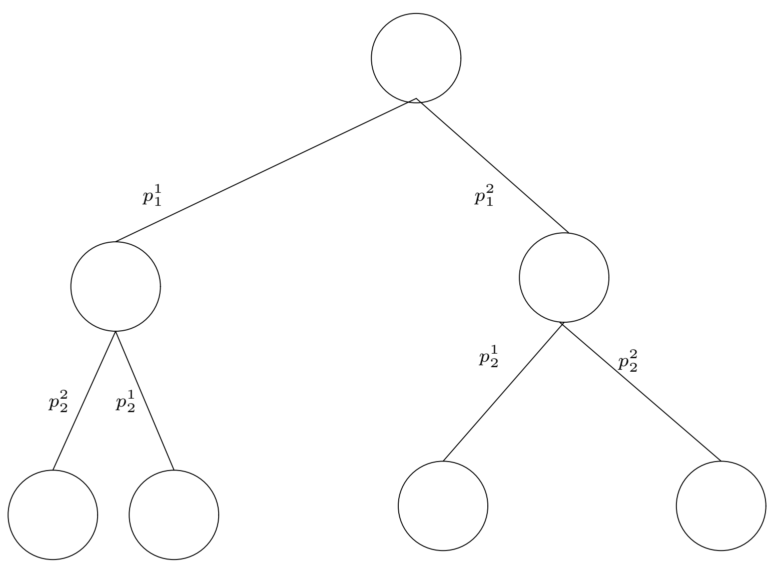


Figure 3: Typical interleaving reduction

- POR is always active in PAN, unless you compile with `-DNOREDUCE`
    * in some cases it is not applicable, e.g., when both fairness and synchronous channels are used

# 2 SPIN and LTL

- How to use SPIN to verify LTL formulas
  - not very user-friendly, not even with the graphical user interface
  - it is necessary to first generate the Büchi automaton (as a *never claim*) for the desired LTL formula and then *manually* attach to the Promela file
  - a never claim is a special proctype containing the Promela description of the Büchi automaton corresponding to the *negation* of the desired LTL formula
    * SPIN will try to find a path satisfying such negation, and such a path, if it exists, will be the counterexample...
  - moreover, atomic propositions in LTL formulas must be defined using `define` macros beginning with a capital letter
  - may be generated also from the command line with option `-f` (requiring the actual formula, enclosed in single apexes) or `-F` (requiring the name of a file containing the actual formula, in one line only)
    * see `exp.script`; both log files contain an error!
    * this notwithstanding I am verifying a formula $\varphi$ first, and then $\neg\varphi$
    * this may be happen in LTL!
    * in fact, as LTL model checking problem requires, PAN checks that *all* paths satisfy the given formula
    * among all possible paths in a Kriepke structure, there may be two paths s.t. $\pi_1 \neq \pi_2$ and $\pi_1 \models \varphi$ and $\pi_2 \not\models \varphi \equiv \pi_2 \models \neg\varphi$
    * thus:
      · $\exists \pi\ \pi \not\models \varphi$, hence $\mathcal{M} \not\models \varphi$
      · $\exists \pi\ \pi \not\models \neg\varphi$, hence $\mathcal{M} \not\models \neg\varphi$
    * of course, if $\mathcal{M} \models \varphi$, then $\mathcal{M} \not\models \neg\varphi$
    * for a visual representation see slide 3 of `timo5.pdf`
  - in order to verify $\varphi$ from the command line, it is necessary to generate $\neg\varphi$ and append it to the Promela description
  - it is sufficient to prefix a `!` enclosing the whole $\varphi$
  - using the GUI, the formula may be created with buttons, and `define`s may be not put in the file

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

Figure 4: Neverclaim generated by SPIN for LTL formula $\varphi \equiv \mathbf{G}(p\ \mathbf{U}\ q)$

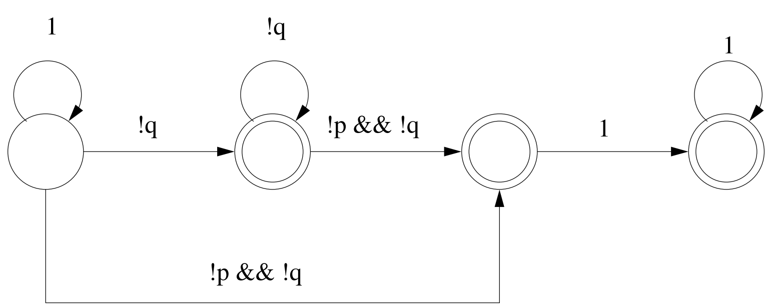


Figure 5: Automa di Büchi ricavato dalla figura 4

- it is also possible to specify either the desired or the undesired behavior
- in the first case, the negation of the given formula will be generated
- in order to check the generated never claim also writes as a comment the formula used
- example: $\varphi \equiv \mathbf{G}(p\ \mathbf{U}\ q)$
- with `spin -f '!([] (p U q))'` Figure 4 is obtained
- the corresponding Büchi automaton is in Figure 5
  * the last transition in the rightmost state is automatically inserted, as it is not present in the neverclaim
  * automaton in Figure 5 encodes all possible *counterexamples* to given $\varphi$
  * in fact, if the verification finds a path satisfying a neverclaim, it returns it as a counterexample

    - in particular, all paths that eventually satisfy $\neg p \wedge \neg q$ are sent in accepting states

- SPIN and PAN are good in combining POR and LTL, see `spin_por.pdf`
  - $a$ is an "observable action", if $a$ is executed, then same edge in the Büchi automaton will change validity
  - e.g., if an edge is labelled with $p = (v == 0)$ and $a := (v = 1)$...
  - if an edge is labelled with 1 (i.e., true) all actions are non-observable
  - thus, a "safe" action oes not change the value of variables in the Büchi automaton
  - Figures from 1a to 1e of `spin_por.pdf`; note that `dfs` $\neq$ `Dfs`
  - one step each: Büchi automaton and Kripke structure
  - `dfs` is for Kriepke, `Dfs` for neverclaim
  - they call each other
  - POR onyl applied to Kriepke
  - neverclaim is just one process, POR is useless
  - Figure 1d: acceptance cycles
  - after having reached an accepting state in the neverclaim...
    - labelled with something beginning with `accept`
    - an accepting state is actually a final state for the Büchi automaton
  - ... a cycle has been closed starting a new DFS from the accepting state
  - i.e., the current path $\pi$ (leading to the accepting state) has a cycle with an accepting state
  - thus, $\pi$ is a word in the language accepted by the automaton