Software Testing and Validation A.A. 2024/2025 Corso di Laurea in Informatica

The SPIN Model Checker

Igor Melatti

#### Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica



### Acronyms

- Murphi stands for nothing, though it is probable that it reminds Murphi's Laws
  - "if something may fail, it will fail", i.e.,  $\textbf{EF}p \rightarrow \textbf{AF}p$
- SPIN stands for Simple Promela INterpreter
- Promela is the SPIN input language
  - Murphi input language does not have a proper name
- Promela stands for PROcess MEta LAnguage
  - as we will see, it is actually based on Operating Systems-like processes
- Also see slides at https://spinroot.com/spin/Doc/SpinTutorial.pdf
  - some as reused here



- We recall that Murphi input language is based on:
  - global variables with finite types
    - base types are integer subranges and enumerations
    - higher types are arrays and structures
  - function and procedures
  - guarded rules and starting states (dynamics)
    - may call functions and procedures, in an atomic way
    - Pascal-like syntax: := for assignments, = for equality checks...
  - invariants



# Structure of a Promela Model

Promela instead has:

- global variables with finite types
  - base types are integer types of the C language
  - enumerations are very limited
  - arrays and records
  - channels!
- processes behaviour (dynamics)
  - possibly with arguments and local variables
- properties to be checked:
  - assertions
  - deadlocks
  - "neverclaim" describing a BA
  - a separate tool may translate an LTL formula in the corresponding BA



# Structure of a Promela Model

### Variables and Types (1)

- Five different (integer) basic types.
- Arrays
- Records (structs)
- Type conflicts are detected at runtime.
- Default initial value of basic variables (local and global) is 0.

Thursday 11-Apr-2002



 Mainly C-like syntax: = for assignments, == for equality checks...

• with some exceptions: if, while, message exchange

- No start states: there is only one starting state
  - an "empty" state, we will see how it is defined
- Thus, if you need multiple starting states, you will have to explicitly model this in Promela
  - having the "empty" state non-deterministically going in the desired starting states
- Assertions are conceptually the same as invariants



#### Processes in Promela

- Dynamics in Promela is defined through *processes*
- You may define many different codes for your processes: proctype
- You may instantiate many times each proctype
  - each instantiation of a proctype is a process
- Each process is either active (i.e., running) from the starting state, or it is explicitly started by some other process
- Though Promela proctypes may seem procedures, they are not!
  - running a process is not like calling a function
  - it is rather like forking a new process, which executes the given code concurrently with the "calling" one



### Peterson Protocol in Operating Systems

```
boolean flag [2];
int turn:
                                  Peterson's Algorithm
void PO()
     while (true) {
           flag [0] = true;
           turn = 1;
           while (flag [1] && turn == 1) /* do nothing */;
           /* critical section */:
           flag [0] = false;
           /* remainder */;
void P1()
     while (true) {
           flag [1] = true;
           turn = 0;
           while (flag [0] && turn == 0) /* do nothing */;
           /* critical section */;
           flag [1] = false;
           /* remainder */
void main()
     flag [0] = false;
     flag [1] = false;
     parbegin (P0, P1);
```



```
bool turn, flag[2];
byte ncrit;
active [2] proctype user()
Ł
 assert(_pid == 0 || _pid == 1);
again:
 flag[_pid] = 1;
 turn = _pid;
 (flag[1 - _pid] == 0 || turn == 1 - _pid);
 ncrit++;
 assert(ncrit == 1); /* critical section */
 ncrit --;
 flag[_pid] = 0;
goto again
}
```

### Peterson Protocol in Promela

- In this case, the starting state has:
  - two running processes, both ready to execute their first statement (i.e., the assert)
  - turn, flag, ncrit are all set to zero
    - both entries for flag
- A special local variable \_pid is available for all processes
  - similar to Operating Systems PID
  - n instantiations of a proctype will have \_pid going from 0 to n-1
  - read-only
- The often-deprecated goto statement is heavily used in Promela models
  - C-like labels also may have special meanings
  - the same holds for the break statement

#### Non-Determinism in Promela: Part I

- In the Peterson model above, there does not seem to be any non-determinism
- Instead, in each state there are two possible successors
  - one obtained executing the (current statement in) the first process
  - the other one obtained executing the second process
- Generally speaking, if in a state there are *n* active processes, then there are *n* successors
  - actually, they may be less, because of blocked statements
  - or more, because of the other source of non-determinism
- SPIN checks that properties hold for *all possible interleavings* between processes
  - using OS-like parlance, the model must be correct regardless of the scheduler

# Non-Determinism and Interleaving: Murphi vs. SPIN

- In Murphi, non-determinism is given by the fact that multiple rules may be fired
  - i.e., their guard is true in the same state
- In SPIN, for now, non-determinism is given by the fact that multiple processes may execute their next statement
- Statements interleaving is not possible in Murphi
  - statements are in rules (or startstates) bodies
  - possibly enclosed in functions/procedures
  - if two rules are fired together, first execute *all* statements in one body, so as to obtain one successor state...
  - ... then execute *all* statements in the other body, so as to obtain another successor state
  - it may be also the case that the two successor states turn out to be equal

DISIM

- Proctypes are sequences of *statements*
- Statements may be either *blocked* or *executable* 
  - again, it resembles OS blocking primitives, such as those on semaphores or on message exchange
- If a statement is blocked, then the corresponding process cannot be selected for execution
- For each type of statement, we will say when it is blocked and when it is executable



- The following statements are always executable: assignments, goto, break, skip, assert, printf
  - return does not exist
  - break must be inside a cycle
  - skip does "nothing", useful in some cases
  - assert takes an expression *e* as argument: if *e* is false, the verification is aborted and the error is reported
  - printf is only used in debugging the model itself (simulation mode)



- Expressions may be used as statements
  - in Peterson protocol:

(flag[1 - \_pid] == 0 || turn == 1 - \_pid);

- An expression *e* used as a statement is executable iff *e* is evaluated to true in the current state
  - e is always of some integer type
  - as usual in C, e is false if e = 0, and true otherwise
  - boolean is a particular type of integer with 0, 1 as only available values
  - boolean connectors are the same as in C and Java (&&, ||)
  - in OS parlance, we are implementing busy waiting
  - e is equivalent to while (e=0) /\* do nothing \*/
- Once it becomes executable and it is actually executed, SPIN simply goes to the next statement

```
bool turn, flag[2];
byte ncrit;
active [2] proctype user()
Ł
 assert(_pid == 0 || _pid == 1);
again:
 flag[_pid] = 1;
 turn = _pid;
 (flag[1 - _pid] == 0 || turn == 1 - _pid);
 ncrit++;
 assert(ncrit == 1); /* critical section */
 ncrit --;
 flag[_pid] = 0;
goto again
}
```

#### Blocked and Executable Statements

- The run statement may be used to create a new process
  - there is a limit to the number of active processes N
  - $\bullet\,$  run is executable iff the number of currently active processes is less than N
- Other ways to have processes:
  - declare a proctype as active [n]
    - active since the start state
    - *n* is the number of instances to be run, may be skipped if n = 1
    - if proctype has arguments, initialized to 0
  - name a proctype as init
    - again, active since the start state
- There must be either active proctypes or the init proctype in every Promela model

# Structure of a Promela Model



SQR

```
bool turn, flag[2];
byte ncrit;
active [2] proctype user()
Ł
 assert(_pid == 0 || _pid == 1);
again:
 flag[_pid] = 1;
 turn = _pid;
 (flag[1 - _pid] == 0 || turn == 1 - _pid);
 ncrit++;
 assert(ncrit == 1); /* critical section */
 ncrit --;
 flag[_pid] = 0;
goto again
}
```

```
bool turn, flag[2];
byte ncrit;
proctype user()
{
/* ... as before */
init {
  run user();
  run user();
}
```



- Each single statement is atomic
  - other processes must wait for an executable single statement completion
  - this differs from OS-like processes: if n is shared and n++ is executed, race conditions may arise
  - because n++ must be viewed as a sequence of assembly statements
  - not in Promela
- It is sometimes desirable to declare a sequence of statements  $s_1, \ldots, s_n$  as atomic: atomic  $\{s_1, \ldots, s_n\}$



```
bool turn, flag[2];
byte ncrit;
proctype user()
{
/* ... as before */
}
init {
  atomic{
    run user();
    run user();
  }
}
```



- An atomic block like atomic  $\{s_1, \ldots, s_n\}$  may be executable or blocked as well
- The rule is simple: atomic  $\{s_1, \ldots, s_n\}$  is executable iff  $s_1$  is executable
- What happens if  $s_i$  is blocked for some i > 1?
- The process loses the atomicity, it becomes blocked and other active processes will have to be executed
- This is the only case in which a statement is initially executable and then becomes blocked
- When *s<sub>i</sub>* is executable again, and the "scheduler" selects the process, the rest of the atomic section is executed atomically again
  - unless a new  $s_j$  is blocked with j > i...



- Another way of specifying atomic blocks is d\_step  $\{s_1,\ldots,s_n\}$
- Again, executable iff  $s_1$  is executable, but:
  - it is a (runtime) error if  $s_i$  is blocked with i > 1
  - each *s<sub>i</sub>* must be deterministic
  - all statements seen till now are deterministic, we will see non-deterministic ones later
- Thanks to these restrictions, d\_step is more efficient than atomic
  - intermediate states need not to be generated, as they cannot block and then resume







DISIM Dipatimento di Ingegneri e Scienze dell'Informazion e Manematica





3 DISIM Dipartimento di Ingegner e Scienze dell'Informazio e Mattematica

- The timeout statement may be used to avoid deadlocks
  - that is, states where all processes only have blocked statements to be executed next
- In fact, timeout is an expression
  - it becomes true (and thus, as a statement, executable) iff we are in a deadlock, in the sense described above
- Used as an escape in some cases



### Blocked and Executable Statements

- The if statement has a somewhat surprising syntax if
  - ::  $e_1 \rightarrow s_{11}; \ldots; s_{1n_1}$
  - :: *e<sub>m</sub>* → *s<sub>m1</sub>*;...;*s<sub>mnm</sub>* fi
    - inspired by Dijkstra guarded command language
    - it is executable if there exists i s.t.  $e_i$  is executable
      - typically  $e_i$  are expressions, thus when some  $e_i$  is true
    - as a special expression, else is true (executable) iff all e<sub>i</sub> are false (blocked)
      - thus, an if with an else is always executable
    - if all e; are blocked, then the if statement is blocked
      - note that this is very different from "normal" imperative languages ifs...
    - ; may be used instead of ->, which is actually syntactic sugar

DISIM

### Blocked and Executable Statements

- The while statement does not exist in Promela
- Instead, we have

```
do
:: e_1 \rightarrow s_{11}; ...; s_{1n_1}
:
:: e_m \rightarrow s_{m1}; ...; s_{mn_m}
od
```

- as for the if, it is executable if there exists *i* s.t. *e<sub>i</sub>* is executable
- if all e<sub>i</sub> are blocked, then the do statement is blocked
- of course, if is executed only once, while do is executed forever
- more precisely: once, for some i, s<sub>i1</sub>;...; s<sub>ini</sub> is executed, the whole do is evaluated again
- to exit from a do, a break is necessary
- or some other escape, such as goto or unless: see later

#### Non-Determinism in Promela: Part II

- There are two sources of non-determinism in Promela:
  - inter-process, as a process may non-deterministically be chosen among all the currently active non-blocked processes
    - a non-blocked process is a process which current statement is executable
  - intra-process: using if or do
- In fact, if  $E = \{e_{i_1}, \ldots, e_{i_k} \mid e_{i_j} \text{ is executable}\}\$  is such that |E| > 1, there will non-deterministically be |E| successors
  - of course, for the current process only
  - other processes may have a current if or do as well



#### Non-Determinism in Promela: Part II



DISIM Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

<u>।</u> १९९७

### Other Promela



 $\mathcal{O} \mathcal{O} \mathcal{O}$ 

# Other Promela





DISIM Dipartimerco di Ingegneri e Scienza dell'Informazion e Matematica

# Other Promela





Disim Dipartimento di Ingegneri e Scienze dell'informazion e Matematica

- Two processes may communicate using shared memory
  - that is, using global variables
  - one writes and the other reads
- If synchronization is required, busy waiting must be used
  - that is, read only after writing



```
byte x;
active [2] proctype user()
{
   byte y;
   if
   :: _pid == 0 -> x = 1
   :: _pid == 1 -> y = x;
   fi;
   ...
}
```

What if I want y = x to happen only after x = 1?


```
byte x;
bit b = 0;
active [2] proctype user()
{
    byte y;
    if
    :: _pid == 0 -> atomic{b = 1; x = 1}
    :: _pid == 1 -> atomic{b == 1; y = x;}
fi;
    ...
}
```



- Fortunately, Promela offers a simple way to handle communication: FIFO channels
  - similar to OS message exchange via mailbox
- To declare a channel, the chan data type can be used
  - the modeler must specify both the size of the channel and the type of the messages to be exchanged
- Messages may be tuples
  - their types must be enclosed in brackets





Sar

- To send a message in a channel: channel!value1,...valuen
  - executable iff channel has size m > 0 and contains at most m-1 messages
  - each message has *n* components
- To receive a message in a channel: channel?x1,...,xn
  - if all xi are variables, the first still undelivered message in channel is stored in each xi, breaking down the tuple
  - executable iff the channel is not empty
  - if all xi are constant values, the first still undelivered message in channel is compared to the values xi, breaking down the tuple
  - executable iff the first message in the channel matches the given values
  - in this case, the message is removed from their annel
  - variables and constants may be mixed



# Inter-Process Communication: Rendez-Vous Channels

- It is sometimes desirable to also have blocking send
  - that is, if there is not some other process receiving on the channel, the send must block
  - reading is always blocking, if there is not something to be received
- This may be achieved using rendez-vous channel
- Defined using 0 as the channel size
- Both the sending and the reading process will block, till when some other process perform the dual operation
- Then, both of them go on to the following statement
  - only case in which two separate statement of two different process are executed at the same time



```
#define p 0
#define v 1
chan sema = [0] of { bit }; /* rendez-vous */
proctype dijkstra()
ſ
    byte count = 1; /* local variable */
    do
    :: (count == 1) \rightarrow sema!p; count = 0
    /* send 0 and blocks, unless some other
       proc is already blocked in reception */
    :: (count == 0) -> sema?v; count = 1
    /* receive 1, same as above */
    od
}
```



```
proctype user()
ſ
    do
    :: sema?p;
            critical section */
       /*
       sema!v;
       /* non-critical section */
    od
}
init
{
    run dijkstra();
    run user(); run user(); run user()
}
```



# Channels Example: Alternating Bit Protocol

https:

//en.wikipedia.org/wiki/Alternating\_bit\_protocol

- Data link layer protocol, used in the first Internet
- Process A wants to send a multi-part message to process B
  - order of message parts are important, so first trunk first, then second...
- A sends current part together a bit b, and waits for B answer
- If B sends back ACKb, A proceed with the next part with flipped bit 1 b
- Otherwise, send the current part again, with the same b
- Try to simulate the Promela model with the graphical SPIN







DISIM Dipartimento di Ingegnes e Sciente dell'Informazio e Matternatica

# Promela: Other

- Each statement may have a label (e.g. again in Peterson's protocol)
- 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 bracket } 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
- If the label begins with "accept", then it is an accepting state
  - typically inside some neverclaim representing a BA of some LTL formula



- We define the Kripke structure S = (S, I, R, L) 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
    - we also assume there are k running processes, with process i having M<sub>i</sub> statements inside it
    - this is due to the fact that *program counters* must be stored for each running process
    - so we can single out the exact statement to be executed in each process, being on some current state
    - if a D<sub>i</sub> corresponds to short or int, then it has 2<sup>16</sup> or 2<sup>32</sup> values on a typical 64-bit architecture, as it is in C
    - a channel is essentially an array of structures



### From Promela to Kripke Structures

- 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 could only be possible when only active proctypes are used, without run commands
  - however, local variables may be defined at any point inside proctypes, thus it is not possible even in that case
- Thus, state space grows: as new processes run and new local variables are reached
- ... and shrinks: as some process terminate



### From Promela to Kripke Structures

- I = {s<sub>0</sub>} where s<sub>0</sub> contains only processes defined as active and all global variables are zero
  - all program counters are at the beginning, local variables still does not exist
- Intuitively, 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 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 a rendez-vous channel c, and there is another current statement t' in another process p' 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 counter must be considered

# **SPIN** Simulation

```
Almost equal to Murphi one
void Make_a_run(NFSS \mathcal{N})
ſ
 let \mathcal{N} = \langle S, \{s_0\}, \text{Post} \rangle;
 s_curr = s_0;
 if (some assertion fail in s_curr))
  return with error message;
 while (1) { /* loop forever */
  if (Post(s_curr) = \emptyset)
   return with deadlock message;
  s_next = pick_a_state(Post(s_curr));
  if (some assertion fail in s_curr))
   return with error message;
  s_curr = s_next;
```



# **SPIN** Verification

- 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?
  - is a given neverclaim (safety or liveness) ok in the current system?
- It is possible to specify some side behaviours:
  - is sending to a full channel blocking, or the message is dropped without blocking?
- It may report unreachable code
  - Promela statements in the model which are never executed



# **SPIN** Verification

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



# SPIN Verification of LTL Formulas

- The former is ok for assertion or deadlock checks
- If you also have an LTL formula
  - the SPIN compiler (SrcXXX/spin -F) is invoked on model.ltl and outputs a neverclaim on the standard output
    - model.ltl must be a text file with only 1 line
    - file extensions does not matter
    - syntax for the formula: G is [], F is <>, U is U
    - atomic propositions must be identifiers
  - 2 append the neverclaim to the promela file
  - O define the identifiers used as atomic proposition by #defines in the promela file
  - go on as before
- If you use the graphical GUI, it is much easier: such steps are automatically performed

- 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.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
  - this is needed for simulations and counterexamples



- 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
  - the current state is saved in a memory buffer called now which is very similar to the Murphi's workingstate
  - given the current state, 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
  - actually, a second index *j* is needed in the case the current statement is non-deterministic



### PAN: Protocol ANalyzer

#### • pan.b: the same of pan.m, but backwards!

- 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
- 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), each time an action overwriting a variable is undone, we need the *last* value, thus a stack for each variable is used



- 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)
- Iterative DFS (recursive one is inefficient)
  - with gotos and global variables!
  - DFS stack is explicitly handled in a lighter and more efficient way



### Standard Recursive DFS

```
HashTable Visited = \emptyset;

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');

}
```



### Iterative DFS Easy Version

```
DFS(graph G = (V, E))
{
  s := init;
  push(s, 1);
  while (stack \neq \emptyset) {
     (s, i) := top();
     increment i on the top of the stack;
     if (s \notin Visited) {
       Visited := Visited \cup s;
       let S' = \{s' \mid (s, s') \in E\};
       if (|S'| \ge i) {
         s := i-th element in S';
         push(s, 1);
       }
       else pop();
    }
     else pop();
} }
```

### Iterative DFS

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



### Iterative DFS

```
Up:
  (s, i) := pop();
  depth := depth - 1;
  if (depth > 0)
    goto Down;
}
```



# DFS in PAN

```
DFS(NFSS \mathcal{N})
ſ
  let \mathcal{N} = (S, I, \text{Post});
  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;
```

# DFS in PAN

```
Up:
    (p, opt) := pop();
    depth := depth - 1;
    now := undo(now, p, opt);
  } }
  if (depth > 0)
    goto Down;
}
```



- The stack does not store states
- Instead, each stack entry stores a pair (p, o) 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...)
  - so it is 8 bytes, whilst the current state may easily require some kB
- We now detail the rational behind this choice



- There is just one initial state
- Let (p<sub>0</sub>, o<sub>0</sub>) be the first (from the bottom) pair on the stack; it univocally identifies a statement *istr*<sub>0</sub> 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 p\_0, o\_0 \rangle, ..., \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,  $\forall 1 \leq i \leq d \ s_i = \operatorname{apply}(s_{i-1}, p_{i-1}, o_{i-1})$



- 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 assignments (e.g., c = 4), the apply function puts the preceding values of v in the stack of v before overwriting it with 4
  - $\bullet\,$  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:
  - 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 (p, o) which generates s if undone on the current state



# PAN: Details

- 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



# PAN: Details

- ch13.pdf adds some more details
- 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



# PAN: Details

- ch13.pdf adds some more details
- 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
- Finally, recall that, for LTL verification, a nested DFS is used



# PAN: Counteracting State Space Explosion

- PAN has the same bit compression (called *byte masking*) and hash compaction techniques we described for Murphi
  - to enable hash compaction, compile pan.c with -DHC
  - byte masking is always enabled, compile with -DNOCOMP to disable it
    - simply align to bytes instead of 4-bytes words
  - also bitstate hashing, a precursor of hash compaction
  - stack cycling, i.e., efficiently use disk for DFS stack
- Other interesting techniques: collapse compression, minimized automaton (may be combined), partial order reduction
- First two techniques try to use less memory to represent the set of visited states so far
  - same goal of hash compaction et similia
- Last technique directly prunes the state space
  - same goal of symmetry reduction in Murphi

# Collapse Compression

- Less effective than hash compaction, but exhaustive as bit compression
  - to enable it, compile pan.c with -DCOLLAPSE
- Recall the main components of a Promela model: N processes, global variables, channels
- The idea is to store in the hashtable N + 2 state fragments, instead of a single state
  - this is the default, but you can put all processes together (-DJOINPROCS)
  - or separate channels with DSEPQS
- A further special "order fragment" is used to say which is the first fragment, the second, ... till the (N + 2)-th fragment
## Collapse Compression

- Thus, to decide if the current state is visited, first split it as described above
- If at least one fragment is not in the hashtable, the state is new
  - of course, the missing fragment(s) must be placed inside the hash table
  - for each of them, a unique identifier is generated and stored together with the fragment
  - the unique identifier is an integer with value *i*, if this is the *i*-th fragment to be generated
    - of course, only considering the current fragment typology...
  - the special order fragment contains the sequence of such identifiers



## Collapse Compression

- Otherwise, also the order fragment must be checked
  - if it is found, then the state is already visited
  - otherwise, insert the new fragment order and return the state as not visited
- Very good if there are many combinations of a few state fragments
  - the order fragment is much shorter than fragments concatenation



- Explicit model checking, borrowing ideas from symbolic model checking
- We still have the DFS as above, but as for visited states check there is not any hash table!
- It is replaced by a "minimized automaton" representing the visited states
  - here, a minimized automaton is essentially similar to those recognizing regular expressions
  - but they are limited: no cycles (it is a DAG), as there is a maximum length to the words



- Finite State Automaton (FSA) for regular expressions:
  - $\mathcal{F} = \langle Q, \Sigma, \delta, q_0, F \rangle$ 
    - Q is the finite set of states
    - ullet being  $q_0\in Q$  the initial state and  $F\subseteq Q$  the final states
    - $\Sigma$  is the alphabet (input symbols) of the regular expression
    - $\delta \subseteq Q \times \Sigma \times Q$  is the transition relation
- A word  $w \in \Sigma^*$  is recognized if, starting from  $q_0$ , it ends up in a final state in F
  - $w = \sigma_1 \dots \sigma_n$ ,  $\langle q_0, \dots, q_n \rangle$  is such that  $(q_{i-1}, w_i, q_i) \in \delta$  for  $1 \le i \le n$

• w is recognized iff  $q_n \in F$ 

•  $\mathcal{L}(\mathcal{F})$  is the set of recognized words



## Minimized Automaton

- $\bullet$  A minimized automaton  ${\cal F}$  is a special case of a FSA where:
  - $\Sigma = \{0,1\}^8$  (input symbols are bytes)
  - |F| = 1
  - $\delta$  is deterministic, thus  $\delta: Q \times \Sigma \to Q$
  - $\mathcal{L}(\mathcal{F})$  is the set of bit sequences representing visited states, which implies  $|\mathcal{L}(\mathcal{F})| < \infty$
  - as a consequence, there are no cycles induced by  $\delta$  (it is a DAG)
    - "diamonds", i.e., circuits, are still possible
    - the original definition of minimized automaton also has layers of states
    - s.t.  $\delta$  goes from a state in level i to i+1
- $\bullet\,$  PAN incrementally constructs  ${\cal F}$  for each unvisited state
  - keeping it minimal w.r.t. the number of states
  - several heuristics are also used, not covered there university

#### Minimized Automaton: Why Effective?

- Suppose you have a k-bytes state vector, and that the visited states are exactly those having 8 zeros in the last byte
  thus, a visited state is represented by [0, 1]<sup>8(k-1)</sup>0
- Using an hash table, we have to store  $2^{k-1}$  states

Instead, using the minimized automaton:



## Minimized Automaton: Why Effective?

- As usual in Model Checking: impossible to a priori state that a given KS will be "well" represented by a minimized automaton, or collapse compression, or whatever
  - all such techniques may be seen as "heuristics" in some sense
- For the minimized automaton, some "regularity" is needed inside the bit representation of the set of visited state
- Also note that sometimes adding a state may improve regularity, making the minimized automaton smaller
  - and of course, in some other cases, adding a state may decrease regularity and make the automaton bigger



## PAN Saving Memory Recap: Normal





## PAN Saving Memory Recap: Hash Compaction





## PAN Saving Memory Recap: Collapse





### PAN Saving Memory Recap: Minimized Automaton







# PAN Saving Memory Recap: Collapse + Minimized Automaton



- POR does not try to use less memory to save the same states: it tries to save less states
  - while retaining correctness, of course
  - some states are "useless" and need not to be explored (and saved)
  - also saves in computation time, of course
- Similar to Murphi symmetry for the goal, but different in use and algorithm
  - use: Murphi modeler must specify which parts of the model are symmetric
  - in SPIN, POR is directly applied without the modeler being aware of it
  - though it is possible to disable it



- There are many ways to perform POR; here, we focus on *ample sets*
- The main idea is that not all interleavings of processes must actually be expanded
  - if we have, e.g., 2 processes, for some actions it is not important if we execute P1 and then P2 or viceversa
- We need an algorithm to decide when only one interleaving can be considered, retaining verification correctness
  - such algorithm must have a low overhead
  - must also work locally (we cannot first expand all reachable states and then decide which ones can be removed...)



- Let  $\mathcal{P} = \langle Q, q_0, T \rangle$  be a *finite state program* (FSP) where:
  - Q is a finite set of states,  $q_0 \in Q$  is the start state
  - T is a finite set of operations
    - also called actions or transitions
    - each action  $t \in T$  is a partial function  $t : Q \to Q \cup \{\bot\}$
    - i.e., executing t from a state q generates a new state q' = t(q)
    - we also define, for each action  $t \in T$ , the set  $en_t = \{q \in Q \mid t(q) \neq \bot\}$
    - furthermore, the function en : Q → 2<sup>T</sup> returns all actions enabled in a state q, i.e., en(q) = {t ∈ T | q ∈ en<sub>t</sub>}
  - paths are sequences  $\pi = r_0 \alpha_0 r_1 \dots$ 
    - notation:  $\pi^{(q)}(i) = r_i, \pi^{(a)}(i) = \alpha_i$
    - of course,  $r_{i+1} = \alpha_i(r_i)$ ,  $\alpha_{i+1} \in en(r_i)$



- From an FSP  $\mathcal{P} = \langle Q, I, T \rangle$  it is easy to generate a KS  $\mathcal{S} = \langle S, J, R, L \rangle$ 
  - Q = S, J = I
  - $(s,s') \in R$  iff  $\exists t \in en(s) : s' = t(s)$
  - L may be defined as needed
- Note that actions are deterministic, but the resulting KS may be non-deterministic
  - there may exists  $t, t' \in T, q \in S$  s.t.  $t \neq t', q \in en_t \cap en_{t'}$  and  $t(q) \neq t'(q)$
- It is easy to see that a Promela model is close to an FSP: each action is a statement
  - thus, an action is identified by a PID and a statement inside that PID
  - of course, states are defined as above from Promela to KSs
  - possible  $\perp$ : if the process is not at the correct Conversion
  - less straightforward: if t is not executable

## Partial Order Reduction: FSP vs Promela

- Actually, we may see that, given an action t, we have that  $q \in en_t$  iff the following holds
  - let *i* inside process *p* be the Promela statement corresponding to *t*
  - must be a single statement, thus dos are replaced by ifs with gotos
  - if nondeterminism is present, *i* is one of the nondeterministic options
  - if more processes of the same proctype are present, *t* is related to *one* of these processes
  - thus *T* is defined so as to consider the possible maximum number of processes for each proctype
  - then, q must be such that PC of p corresponds to i and i is executable



Given an FSP P = ⟨Q, I, T⟩, an *ample selector* is a function amp : Q → 2<sup>T</sup> s.t. amp(q) ⊆ en(q)

• for a given  $q \in Q$ , amp(q) is an *ample set* 

- An ample selector defines a new KS  $S' = \langle S, I, R', L \rangle$ , where  $(s, s') \in R'$  iff  $\exists t \in amp(s) : s' = t(s)$ 
  - of course,  $R' \subseteq R$
  - from a DFS point of view, we normally expand actions in en(q); instead, here we expand only amp(q)
- We want to choose a POR-sound amp

•  $\mathcal{S} \models \varphi$  iff  $\mathcal{S}' \models \varphi$ 

- $\, \bullet \,$  we start by considering only invariants (assertions) as  $\varphi$
- We want to compute amp(q) (almost) only looking at current state q
  - must be simple, i.e., with little overhead
  - no need to be optimal

#### Partial Order Reduction: Independent Actions

- Two actions α, β ∈ T are independent iff
  ∀q ∈ en<sub>α</sub> ∩ en<sub>β</sub>. α(q) ∈ en<sub>β</sub> ∧ β(q) ∈ en<sub>α</sub> ∧ α(β(q)) = β(α(q))
  - i.e.,  $\alpha,\beta$  can be executed in any order, obtaining the same result
  - otherwise, α, β are dependent, which means that ∃q ∈
    en<sub>α</sub> ∩ en<sub>β</sub> : (α(q) ∈ en<sub>β</sub> ∧ β(q) ∈ en<sub>α</sub>) → α(β(q)) ≠ β(α(q))
  - $\, \bullet \,$  in this case, it is both  $\alpha$  dependent on  $\beta$  and viceversa
  - example 1: two actions modifying local variables only are always independent
  - example 2: two actions modifying the same global variable are nearly always dependent
    - unless  $\alpha = \beta$ , or the new value is however the same





#### Partial Order Reduction: Invisible Actions

• An action  $\alpha$  is *invisible* w.r.t. a labeling  $L : Q \to 2^{AP}$  iff  $\forall q \in en_{\alpha}$ .  $L(q) = L(\alpha(q))$ 





#### Recall:

- we are performing a DFS of the KS generated by an FSP
- we have a current state q
- we want to decide if we can consider amp(q) ⊂ en(q) instead of en(q)
- The first 2 conditions only look at q and its actions
  - $\forall q \in Q. \ \mathrm{en}(q) \neq \emptyset \rightarrow \mathrm{amp}(q) \neq \emptyset$ 
    - otherwise, we have introduced a deadlock...
  - $\forall q \in Q$ .  $\operatorname{amp}(q) \subset \operatorname{en}(q) \to (\forall \alpha \in \operatorname{amp}(q), \alpha \text{ is invisible})$ 
    - if we cut some actions, then this must not affect the labeling
    - this also means that only invisible actions can be cut



The remaining conditions also consider paths starting from q

- $\forall q \in Q, \forall \pi \in \text{Path}(\mathcal{P}, q). \ (\exists i > 0, \alpha \in \text{amp}(q) : \pi^{(a)}(i), \alpha$ are dependent)  $\rightarrow \exists j < i : \pi^{(a)}(j) \in \text{amp}(q)$
- if this is true, then either:
  - there exists an  $\alpha \in \operatorname{amp}(q)$  which is the first from  $\operatorname{amp}(q)$  in  $\pi$ 
    - then,  $\alpha$  is independent on all previous actions on  $\pi,$  and can be executed first
  - otherwise, there exists an  $\alpha \in amp(q)$  which is independent on all other actions in  $\pi$ 
    - $\bullet\,$  again, such  $\alpha$  can be executed first



• Example till now:  $\alpha_1, \beta_1$  and  $\alpha_2, \beta_2$  are independent





- Essentially, POR *defers* execution of some actions
  - not executing an action at all means that a meaningful portion of the state space is omitted
- With these 3 conditions only, it may happen that an action is never expanded, due to cycles
  - $\bullet\,$  in the example below,  $\beta$  is independent on both  $\alpha_1,\alpha_2$



The remaining condition rules out the problem with cycles

- Consider a DFS on the reduced KS, and suppose an expanded state q is detected as already visited
- We also check if it is on the DFS stack; this implies:
  - there is a cycle
  - some part of the q sub-tree has not be explored
- Then,  $\operatorname{amp}(q) = \operatorname{en}(q)$ 
  - i.e., q must be fully expanded



## Partial Order Reduction with LTL Formulas

- It seemed that POR with ample set was ok for any stutter-invariant LTL formula
  - recall that a formula φ may be viewed as the set (language) of words L(φ) in AP\* which are recognized by φ
  - $\varphi$  is stutter-invariant iff, for any sequence of integers  $i_j \in \mathbb{N}$ and  $w = p_0 p_1 \ldots \in \mathcal{L}(\varphi)$ ,  $p_0^{i_0} p_1^{i_1} \ldots \in \mathcal{L}(\varphi)$
  - essentially, by repeating any character in the word for any number of times you still obtain a word in the language
  - if  $\varphi$  does not contain **X**, then it is stutter-invariant
  - viceversa does not hold
- However an error was discovered (and corrected) in 2019

