Ciências ULisboa | Informática

Model Checking

Software Fiável

Mestrado em Engenharia Informática
Mestrado em Informática
Faculdade de Ciências da Universidade de Lisboa

2019/2020

Vasco T. Vasconcelos

Model Checking

- Model Checking is a verification technology that provides an algorithmic means of determining whether an abstract model—representing, for example, a hardware or software design—satisfies a formal specification expressed as a temporal logic (TL) formula
- Moreover, if the property does not hold, the method identifies a counterexample execution that shows the source of the problem

[Moshe Y. Vardi, preface to 2007 ACM Turing Award]

Used in software and hardware companies

- The progression of model checking to the point where it can be successfully used for complex systems has required the development of sophisticated means of coping with what is known as the state explosion problem
- Great strides have been made on this problem since the early 80's by what is now a very large international research community
- As a result many major hardware and software companies are beginning to use model checking in practice
- Examples of its use include the verification of VLSI circuits, communication protocols, software device drivers, real-time embedded systems, and security algorithms

[Moshe Y. Vardi, preface to 2007 ACM Turing Award]

Model checking tools

- Model checking tools, created by both academic and industrial teams, have resulted in an entirely novel approach to verification and test case generation
- This approach, for example, often enables engineers in the electronics industry to design complex systems with considerable assurance regarding the correctness of their initial designs
- Model checking promises to have an even greater impact on the hardware and software industries in the future

[Moshe Y. Vardi, preface to 2007 ACM Turing Award]

What factors contributed to model checking successful deployment?

- It provides a "push-button," i.e., automated, method for verification
- It permits bug detection as well as verification of correctness. Since most programs are wrong, this is enormously important in practice
- While a methodology of constructing a program hand-in-hand with its proof certainly has its merits, it is not readily automatable (cf. Hoare Logic, Dafny, and VeriFast in previous lectures)

[E. Allen Emerson, 2007 ACM Turing Award]

Development and verification

- The separation of system development from verification and debugging facilitates model checking's industrial acceptance
- The development team can go ahead and produce various aspects of the system under design. The team of verifiers or verification engineers can conduct verification independently. Hopefully, many subtle bugs will be detected and fixed
- As a practical matter, the system can go into production at whatever level of "acceptable correctness" prevails at deadline time

[E. Allen Emerson, 2007 ACM Turing Award]

Hardware and software verification

- The principal validation methods for complex systems:
  - Simulation
  - Testing (addressed in the Software Verification and Validation course)
  - Deductive verification (this course)
  - Model checking (this course)
- Simulation and model checking (usually) performed on an abstraction or a *model* of the system
- Testing and deductive verification are (usually) performed on the system itself

Deductive verification and model checking

- Deductive verification

  Pros Can be used for reasoning about infinite systems

  Cons A time consuming process; requires some expertise

- Model checking

  Pros Verification can be performed automatically

  Cons Verifies finite systems

- Model checking use exhaustive search of the state space to determine if some property is true. Given enough resources, the procedure always terminates with a yes or no.

On the restriction to finite state systems

- Model checking is applicable to several important classes of systems:
  - Hardware controllers
  - Many communication protocols
  - In many cases errors can be found by restricting unbounded data structures to specific finite instances, e.g., an unbounded queue can be debugged by restricting the size of the queue to two or three
  - Non finite state systems may be checked with model checking combined with other techniques, s.a., abstraction and induction principles

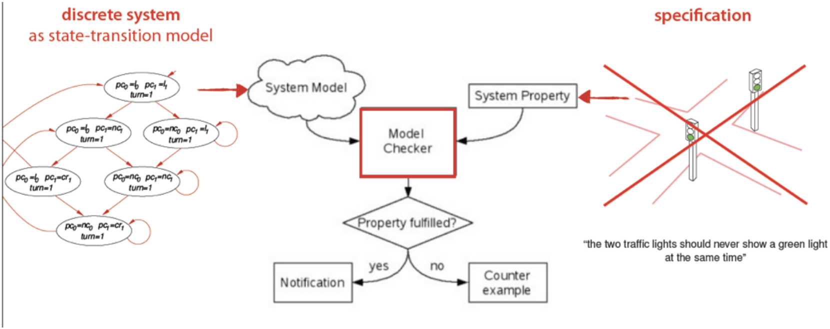
The process of model checking

Modelling Convert a design into a formalism accepted by a model checking tool. May need abstraction to eliminate irrelevant or unimportant details

Specification State the properties that the design must satisfy. Usually given in some logical formalism. *Temporal logic* allows asserting how the behaviour of a system evolves over time

Verification Automatic, ideally. In practice involves human assistance: for the analysis of the verification results, to use the counterexample to help the designer in tracking down the error

The process of model checking



[Slide by D. Bucur]

The rest of these slides

- The rest of these slides closely follow Ben Ari's book
- We will most cover of the book

Sequential Programming in Promela

Workflow

1. A model is written that describes the behavior of the system
2. Correctness properties that express requirements of the system's behavior are specified
3. The model checker is run to the check if the correctness properties hold for the model, and
4. if not, provide a counterexample: a computation that does not satisfy the correctness properties

Promela _ **Pro**tocol **Me**ta **La**nguage

- Types include integers only, of several sizes
- Assignment statements and expressions are written using the syntax of C-like languages
- A program in Promela is composed of a set of **processes**; we start with a single process declared by keywords

```
active proctype
```

Reversing digits

```
active proctype P() {
  int value = 123;
  int reversed =
    (value % 10) * 100 +
    ((value / 10) % 10) * 10 +
    (value / 100);
  printf("value = %d, reversed = %d\n",
         value, reversed)
}
```

- Semicolon is used as a separator (as in Pascal), rather than terminator (as in C, Java)

Random simulation

```
$ spin rev.pml
      value = 123, reversed = 321
1 process created
```

- There is no input to Promela models since it is intended for simulating closed systems. Still check the man pages for how to use the `stdin` channel
- Note: the meaning of “random” in **random simulation** will become apparent later

Ciências ULisboa | Informática

Numeric data types in Promela

Type	Values	Size (bits)
`bit`, `bool`	`0`, `1`, `false`, `true`	1
`byte`	`0..255`	8
`short`	`-32768..32767`	16
`int`	$-2^{31}..2^{31}-1$	32
`unsigned`	$0..2^{n}-1$	≤ 32

- `bool`, `false` and `true` are syntatic sugar
- Print `bit` and `bool` values with the `%d` specifier
- No strings or floating point numbers

Local and global variables

- Variables global to different processes may be declared outside the process

```
int reversed; /* global variable */
active proctype P() {
  int value = 123; /* local variable */
  reversed =
    (value % 10) * 100 +
    ((value / 10) % 10) * 10 +
    (value / 100);
  printf("value = %d, reversed = %d\n",
         value, reversed)
}
```

Initial values of variables

- All variables are initialized to zero; yet explicit initialization is strongly suggested
- Use

  ```
  byte n = 1;
  ```

- The pair of instructions

  ```
  byte n;
  n = 1;
  ```

  is equivalent but introduces an extra (unnecessary) state where `n` is zero

Expressions

- Operators are mostly as in C or Java
- Expressions must be **side-effect free** (expressions are used to determine whether statements are executed)
- An assignment is not an expression
- Symbolic names as in C macros

```
#define N 10
```

Control statements

- Control statements take the form of **guarded commands**, invented by E.W. Dijkstra
- Five control statements: sequence, selection, repetition, jump, `unless`

Selection statement

```
if
:: guard1 -> statement1
...
:: guard1n -> statementn
fi
```

- The execution of an `if`-statemente begins with the evaluation of the guards
- If at least one evaluates to `true`, the sequence of statements following the arrow is executed
- When the execution of these statements terminates, the `if`-statement terminates
- Guards are tried in **no particular order**
- Evaluation blocks until one of the guards becomes true

Example _ Discriminant of a quadratic equation

```
active proctype P() {
  int a = 1, b = -4, c = 4;
  int d = b * b - 4 * a * c;
  if
  :: d < 0  ->
      printf("d = %d: no real roots\n", d)
  :: d == 0 ->
      printf("d = %d: duplicate real roots\n", d
         )
  :: d > 0  ->
      printf("d = %d: two real roots\n", d)
  fi
}
```

Random selection

```
active proctype P() {
  if
  :: true -> printf("A\n")
  :: true -> printf("B\n")
  :: true -> printf("C\n")
  fi
}
```

```
$ spin random-selection.pml
      C
1 process created
$ spin random-selection.pml
      C
1 process created
$ spin random-selection.pml
      B
1 process created
```

else _ when all other guards evaluate to false

```
active proctype P() {
  byte days;
  byte month = 2;
  int year = 2000;
  if
  :: month==1 || month==3  || month==5 || month==7 ||
      month==8 || month==10 || month==12 ->
        days = 31
  :: month==4 || month==6  || month==9 || month==11 ->
        days = 30
  :: month==2 && year % 4==0 && (year % 100!=0 || year
       % 400==0) ->
        days = 28
  :: else ->
        days = 29
  fi;
  printf("month = %d, year = %d, days = %d\n",
         month, year, days)
}
```

Warning _ else and the true guard

- The `else` guard is not the same as a guard consisting of the constant `true`. The latter can always be selected even if there are other guards that evaluate to true, while the former is only selected if all other guards evaluate to false.
- The below code may print `success!`

```
int x = 5;
if
:: x == 5 -> skip;
:: true -> printf("success!\n");
fi;
```

- What if we replace `true` by `else`?

The `skip` expression and the empty statement

- The sequence of statements following a guard can be empty:

```
:: x == 5 -> ;
```

- Alternatively, one may use `skip`, an expression which always evaluate to true:

```
:: x == 5 -> skip;
```

Conditional expressions

- As in C, but use `->` rather than `?` (don't forget the parenthesis)

```
max = (a > b -> a : b)
```

- The expression above is **atomic**; whereas the program below can be interleaved with instructions from other processes

```
if
:: a > b -> max = a
:: else   -> max = b
fi
```

Repetitive statements

```
do
:: guard1 -> statement1
...
:: guard1n -> statementn
od
```

- Similar to the `if`-statement, except that after the evaluation of one of the branches, the `do`-statement is evaluated again
- Loop termination is accomplished by `break`

Greatest common denominator

```
active proctype P() {
  int x = 15, y = 20;
  int a = x, b = y;
  do
  :: a > b  -> a = a - b
  :: b > a  -> b = b - a
  :: a == b -> break
  od;
  printf("The GCD of %d and %d = %d\n", x, y, a)
      ;
}
```

Counting loops

- No such thing in Promela; use the general `do`-statement with a boolean guard and an `else` guard:

```
#define N 10
active proctype P() {
  int sum = 0;
  byte i = 1;
  do
  :: i > N -> break
  :: else ->
       sum = sum + i;
       i++
  od;
  printf("The sum of the first %d numbers = %d\n",
         N, sum);
}
```

- There is also a labelled `goto`-statement, as in C

Symbolic names

- As in C, e.g., `#define N 10`
- Use `mtype` to give mnemonic names to values; represented as positive `byte` values; use the `%e` specifier to print the type name (`%d` prints the integer, 1, 2 or 3)

```
mtype = { red, yellow, green };
active proctype P() {
  mtype light = green;
  do
  ::if
    :: light == red -> light = green
    :: light == yellow -> light = red
    :: light == green -> light = yellow
    fi;
    printf("The light is now %e\n", light)
  od
}
```

Exercise

- Write a program for integer division that works by repeatedly subtracting the divisor from the dividend until what remains is less than the divisor:

```
$ spin divide.pml
      15 divided by 4 = 3, remainder = 3
1 process created
```

Chapter 2

Verification of Sequential Programs

Program states and computations

```
1 active proctype P() {
2   int value = 123;
3   int reversed = (value % 10) * 100 + ((value /
      10) % 10) * 10 + (value / 100);
4   printf("value=%d, reversed=%d\n", value,
      reversed)
5 }
```

- A **state** of a program is a set of values for its variables and for the **location counter** of the form (value of `value`, value of `reversed`, location counter of `P`); e.g., `(123, 321, 4)`,
- A **computation** is a sequence of states beginning with the initial state and continuing with the states that occur as each statement is executed
- There is only one computation in the program above:

$$(0,0,2) \rightarrow (123,0,3) \rightarrow (123,321,4) \rightarrow (123,321,5)$$

State space of a program

- The state space of a program is the set of states that can **possibly** occur during a computation
- In model checking the state space of a program is generated in order to search for a counterexample—if one exists—to the correctness specifications
- For now, we express correctness specifications with assertions:

```
assert (divisor > 0);
```

Exercise: Introduce assertions in the integer division model to express its correctness

Integer division

```
active proctype P() {
  int dividend = 15;
  int divisor  = 4;
  int quotient, remainder;
  assert (dividend >= 0 && divisor > 0); /* pre */
  quotient = 0;
  remainder = dividend;
  do
  :: remainder >= divisor ->
       quotient++;
       remainder = remainder - divisor
  :: else ->
       break
  od;
  assert (0 <= remainder && remainder < divisor); /*
      post */
  assert (dividend == quotient * divisor + remainder);
}
```

Assertion statement

- When an assert statement is executed during a simulation, the expression is evaluated
- If true, execution proceeds normally to the next statement; if false the program terminates with an error message
- Use `spin -l` to obtain the state (local vars) of the program

```
$ spin -l divide1.pml
      15 divided by 4 = 4, remainder = 3
spin: line  23 "divide.pml", Error: assertion violated
spin: text of failed assertion:
  assert((dividend==((quotient*divisor)+remainder)))
#processes: 1
 26:  proc  0 (P) line  23 "divide.pml" (state 14)
1 process created
    P(0):remainder = 3
    P(0):quotient = 4
    P(0):divisor = 4
    P(0):dividend = 15
```

Verifying a program in Spin

```
active proctype P() {
  int a = 5, b = 5, max;
  if
  :: a >= b -> max = a;
  :: b >= a -> max = b + 1;
  fi;
  assert (max == (a >= b -> a : b))
}
```

- Warning: expression `(max == a >= b -> a : b)` parses as `((max == a >= b)-> a : b)`, which in this case is always `true`. Why?
- Remember: no boolean primitive values in Promela

Running the simulation repeatedly

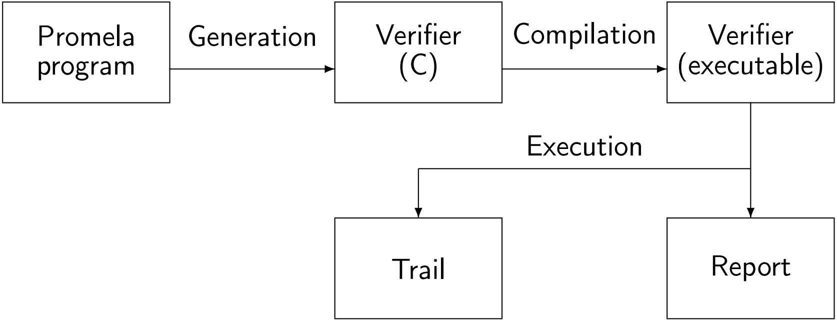
- If we run the simulation repeatedly, it is possible—although unlikely— that the same alternative will always be chosen, hence that the assertion is never violated:

```
$ spin max.pml
1 process created
$ spin max.pml
1 process created
$ spin max.pml
1 process created
$ spin max.pml
1 process created
```

Checking all possible computations

- No amount of simulation can ever verify that the postcondition is true
- The only way to verify that a program is correct is to systematically check that the correctness specification hold in **all possible computations**, and that is what model checkers like Spin are designed to do
- In a determininistic program with no input, there is only one possible computation, hence a single random simulation suffices to demonstrate the correctness
- For a nondeterministic or a concurrent program, checking all possible computations involves executing the program and backtracking at each choice point

Building a verifier for a program



Ciências ULisboa | Informática

Building a verifier from the command line

1. Run Spin with argument `-a` to generate the verifier source code

   ```
   $ spin -a max.pml
   ```

   This generates some files `pan.*` including `pan.c` (`pan` stands for **p**rotocol **an**alyzer)

2. Compile the verifier

   ```
   $ gcc -o pan pan.c
   ```

3. Run the verifier

   ```
   $ ./pan
   ...
   pan: assertion violated (max==( ((a>=b)) ? (a) : (b) ))
   pan: wrote max.pml.trail
   ```

The trail

- Almost invariably it takes a long time to understand why verification failed
- Spin supports the analysis of failed verifications by mantaining internal data structures during the search of the state space; these are used to reconstruct a computation that leads to an error
- The data required for reconstructing a computation are written in a file called a **trail**
- The trail is not intended to be read; instead, it is used to reconstruct a computation in **guided simulation mode**

Examining computations

- Spin can print:
  - The statements executed by the process, option `-p`
  - The values of the global variables, option `-g`
  - The values of the local variables, option `-l`
  - The send instructions executed on a channel, option `-s`
  - The receive instructions executed on a channel, option `-r`

Guided simulation

- After running the verifier, run Spin again, this time with option `-t` (follow the simulation trail)

```
$ spin -t -p max.pml
using statement merging
  1: proc  0 (P:1) max.pml:5 (state 3) [((b>=a))]
  1: proc  0 (P:1) max.pml:5 (state 4) [max = (b+1)]
spin: max.pml:7, Error: assertion violated
spin: text of failed assertion: assert((max==( ((a>=b)) -> (a) : (b) ))
  1: proc  0 (P:1) max.pml:7 (state 7) [assert((max==( ((a>=b)) -> (a)
spin: trail ends after 1 steps
#processes: 1
...
```

- We can then read the path that lead to error:
  `b>=a` → `max = b+1` → `assert (max == ...)`
- Do not forget to add the relevant options: `-p`, `-l`, `-g`, ...

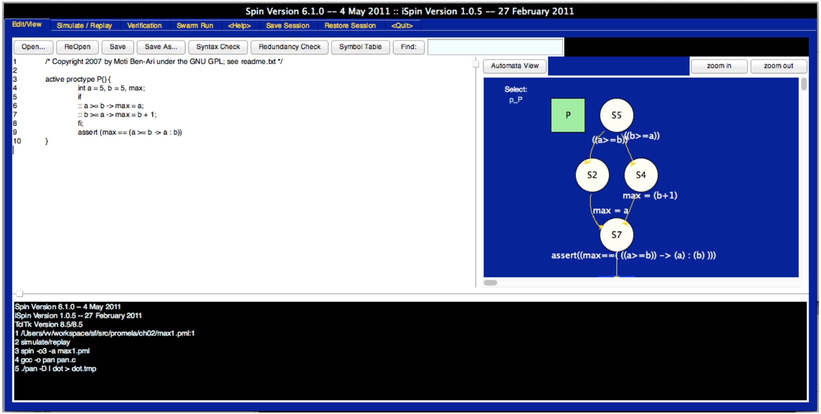
Using iSpin _ Starting

- `iSpin` is a graphical interface to `spin`
- `iSpin` runs `spin` in the background to obtain the desired output, and wherever possible it will attempt to generate a graphical representation of such output (this means that you must have Spin installed)
- Run

  ```
  $ ispin max.pml
  ```

- In the first view you can perform a basic syntax check (press `Syntax Check`) or view the automata corresponding to the program (press `Automata View`, you need `graphviz`)
- Select the only process: `p_P`

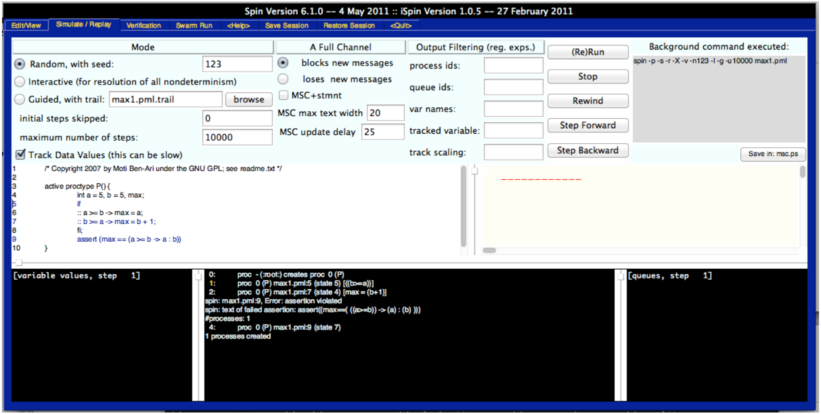
Automata view



Using iSpin _ Simulate

- Now chose the `Simulate/Replay` view
- Accept the default values and then press the `(Re)Run` button
- The lower left pane shows the `Data values`; the lower right maintains the values in the `Queues`
- The lower center pane shows the simulation output
- Once a run is completed you can use the `Rewind` button to go back to the start of the run, and step forward or backward: in our case two states only: guard (`a >= b` or `a <= b`) and statement (`max = a` or `max = b +1`)

Simulation view

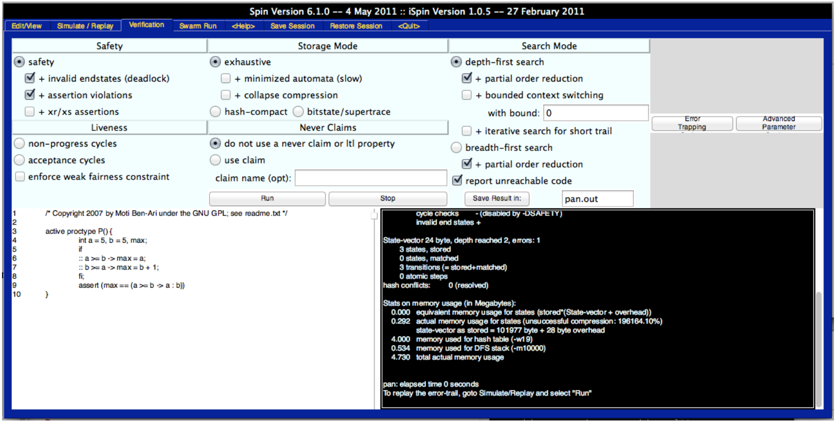


Using iSpin _ Verification

- Now chose the `Verification` view
- Accept the default values and then press the `Run` button
- The lower left pane shows the `Data values`; the lower right maintains the values in the `Queues`
- The lower right pane shows the verification output
- Notice the suggestion at the base of the output

```
To replay the error-trail,
  goto Simulate/Replay and select "Run"
```

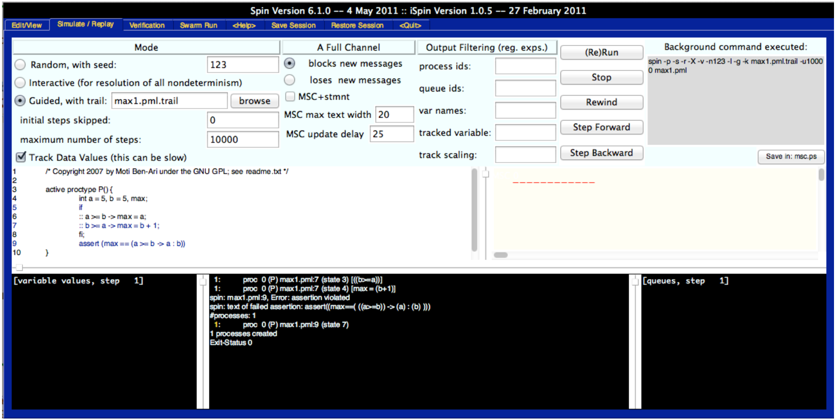
Verification view



Using iSpin _ Guided simulation

- Select the `Simulate/Replay` view
- Make sure the radio button `Guided, with trail:` is checked (this should be automatic)
- Press `(Re)Run`, followed by `Rewind`
- Then `Step Forward` (in this case there is not much to step through; try with a larger example).

Replay with trail



Chapter 3

Concurrency

Ciências ULisboa | Informática

Concurrency

- Spin supports modeling of both **concurrent** and **distributed** programming
- Concurrent $\leftarrow$ shared memory
- Distributed $\leftarrow$ message passing on channels
- We start with concurrency

Interleaving

```
byte n = 0;
active proctype P() {
  n = 1;
  printf("Process P, n = %d\n", n);
}
active proctype Q() {
  n = 2;
  printf("Process Q, n = %d\n", n);
}
```

- How many different outputs?
- How many different computations?

Computations

- One possible computation (detailed)

Process	Statement	`n`	Output
`P`	`n = 1`	0	
`P`	`printf(P)`	1	
`Q`	`n = 2`	1	`P, n = 1`
`Q`	`printf(Q)`	2	
			`Q, n = 2`

- The six possible computations (abbreviated)

1	2	3	4	5	6
n = 1	n = 1	n = 1	n = 2	n = 2	n = 2
printf (P)	n = 2	n = 2	**printf** (Q)	n = 1	n = 1
n = 2	**printf** (P)	**printf** (Q)	n = 1	**printf** (Q)	**printf** (P)
printf (Q)	**printf** (Q)	**printf** (P)	**printf** (P)	**printf** (P)	**printf** (Q)

Interleaving

- The computations of the program are obtained by **arbitrarily interleaving** of the statements of the processes
- Computation for process `P`
  (value of `n`, location counter for `P`, location counter for `Q`)
  `(0, 3, _)` $\rightarrow$ `(1, 4, _)` $\rightarrow$ `(1, 5, _)`
- Computation for process `Q`
  `(0, _, 7)` $\rightarrow$ `(2, _, 8)` $\rightarrow$ `(2, _, 9)`
- One interleaving leading to computation number 3 above
  `(0, 3, 7)` $\rightarrow$ `(1, 4, 7)` $\rightarrow$ `(2, 4, 8)` $\rightarrow$ `(2, 4, 9)` $\rightarrow$ `(2, 5, 9)`

Random simulation with more than one process

- Spin automatically indents `printf` statements so that it is easy to see which output comes from which process

```
$ spin interleave1.pml
          Process Q, n = 2
      Process P, n = 1
2 processes created
$ spin interleave1.pml
      Process P, n = 1
          Process Q, n = 2
2 processes created
```

Atomicity

- Assignment statements in Promela are **atomic**: `n = n + 1` is **not** composed of a read operation (the `n` on the right) followed by a write (the `n` on the left)
- Expressions are also atomic
- `if`- and `do`-statements are not atomic. This program may well divide by zero! (how?)

```
if
:: a != 0 -> c = b / a
if
```

Interactive simulation

- When there are two or more nontrivial processes, the number of computations become extremely large
- Random simulation tells us almost nothing about a program, except that it works for a few computations
- With interactive simulation a specific computation can be constructed. At each **choice point** you are presented with the various choices and can interactively choose one
- Choice points arise either because of nondeterminism within a single process (guarded commands) or because of a choice of the next statement of several concurrent processes

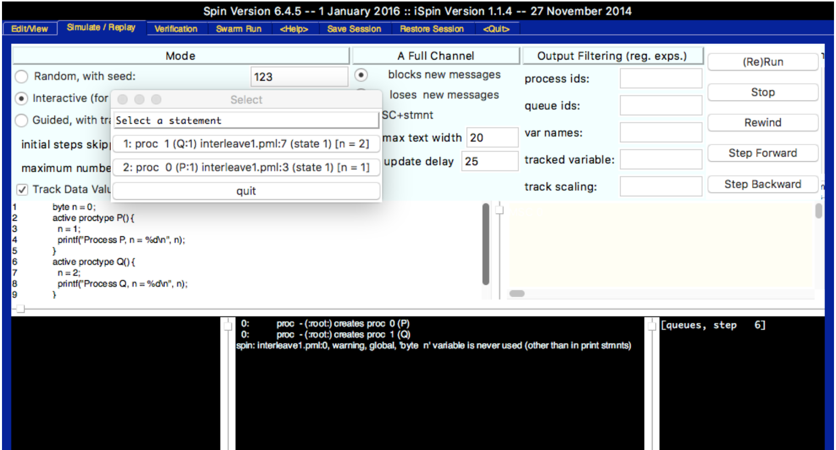
Interactive simulation at command line

- Execute Spin with argument `-i` and make your choices. Here we follow computation number 3 above

```
$ spin -i interleave1.pml
Select a statement
choice 1: proc  1 (Q) interleave1.pml:7 (state 1) [n = 2]
choice 2: proc  0 (P) interleave1.pml:3 (state 1) [n = 1]
Select [1-2]: 2
Select a statement
choice 1: proc  1 (Q) interleave1.pml:7 (state 1) [n = 2]
choice 2: proc  0 (P) interleave1.pml:4 (state 2) [printf('Process P, n = %d\\n',n)]
Select [1-2]: 1
Select a statement
choice 1: proc  1 (Q) interleave1.pml:8 (state 2) [printf('Process Q, n = %d\\n',n)]
choice 2: proc  0 (P) interleave1.pml:4 (state 2) [printf('Process P, n = %d\\n',n)]
Select [1-2]: 2
      Process P, n = 2
Select a statement
choice 1: proc  1 (Q) interleave1.pml:8 (state 2) [printf('Process Q, n = %d\\n',n)]
Select [1-2]: 1
          Process Q, n = 2
Select a statement
choice 1: proc  1 (Q) interleave1.pml:9 (state 3) <valid end state> [-end-]
Select [1-2]: 1
2 processes created
```

Interactive simulation with ispin

- Chose tab "Simulate/Reply", press the check box "Interactive (for resolution of all non-determinism)"
- Then press "(Re)Run"



Interference between processes

- Consider a CPU that performs computations in registers

```
load R1, n    /* n is a memory address */
add R1, 1
store R1, n
```

- The following program is a simple model of a shared memory multiprocessor

```
byte n = 0;
active proctype P() {
  byte R1 = n;
  R1 = R1 + 1;
  n = R1;
  printf("%d\n", n)
}

active proctype Q() {
  byte R1 = n;
  R1 = R1 + 1;
  n = R1;
  printf("%d\n", n)
}
```

- What are the possible outputs?

Perfect interleaving

Process	Statement	`n`	`P:R1`	`Q:R1`	Output
`P`	`R1 = n + 1`	0	0	0	
`Q`	`R1 = n + 1`	0	1	0	
`P`	`n = R1`	0	1	1	
`Q`	`n = R1`	1	1	1	
`P`	`printf(n)`	1	1	1	1
`Q`	`printf(n)`	1	1	1	1

Exercise: Create this computation by interactive simulation

Sets of processes

```
1 byte n = 0;
2 active [2] proctype P() {
3   byte R1;
4   R1 = n + 1;
5   n = R1;
6   printf("Process P%d, n = %d\n", _pid, n);
7 }
```

- The number in brackets (line 2) indicates the number of processes to instantiate
- Predefined variable `_pid` identifies the process number (starts at zero)

The run operator

```
byte n;
proctype P(byte id; byte incr) {
  byte R1;
  R1 = n + incr;
  n = R1;
  printf("Process P%d, n = %d\n", id, n)
}
init { /* gets pid 0 */
  n = 1;
  atomic {
    run P(1, 10); /* gets pid 1 */
    run P(2, 15)  /* gets pid 2 */
  }
}
```

- `run` statements are enclosed in an `atomic` sequence to ensure that all processes are instantiated before any of them begins execution

Controling the number of active processes

- Predefined variable `_nr_pr` contains the number of processes currently active
- Statement

```
(_nr_pr == 1) -> statement
```

blocks until the guard becomes true; it abbreviates

```
if
:: (_nr_pr == 1) -> statement
fi
```

Exercise _ More interference

```
#include "for.h"
byte n = 0;
proctype P() {
  byte temp;
  for (i, 1, 10)
    temp = n; n = temp + 1
  rof (i)
}
init {
  atomic { run P(); run P() }
  (_nr_pr == 1) -> printf("The value is %d\n", n
     );
}
```

- What is the max possible value of `n`? and the minimum?
- Which assertions do we need?

Verification with assertions

- I say: There is a computation whose output is 2!
- It can be verified via assertion

```
assert (n > 2)
```

- Counter intuitive? Remember that Spin looks for **counterexamples**
- We run the verification to obtain

```
pan: assertion violated n>2 (at depth 89)
```

- Now a guided simulation can be run with the trail in order to examine the computation that caused the assertion to be falsified

The critical section problem

- A system consists of two or more concurrently executing processes
- The statements of each process are divided into **critical** and **noncritical** sections that are repeatedly executed one after the other
- A process may halt in its noncritical section, but not in the critical section

Mutual exclusion
: At most one process is executing its critical section at any time

Absence of deadlock
: It is impossible to reach a state in which some processes are trying to enter their critical sections, but no process succeeds

Absence of starvation
: If any process is trying to execute its critical section, then eventually that process is successful

Incorrect solution for the critical section problem

```
bool wantP = false, wantQ = false;
active proctype P() {
  do
  :: printf("Non critical section P\n");
     wantP = true;  /* entering critical section */
     printf("Critical section P\n");
     wantP = false   /* leaving critical section */
  od
}
active proctype Q() {
  do
  :: printf("Non critical section Q\n");
     wantQ = true;  /* entering critical section */
     printf("Critical section Q\n");
     wantQ = false   /* leaving critical section */
  od
}
```

- Why incorrect?

Ghost variables

- Correctness specifications for concurrent programs must consider the global state of all the processes in the program
- To specify that two processes cannot be in their critical regions at the same time, the specification must talk about control points in both processes
- One solution: introduce a new variable (`critical`) that is not part of the algorithm but is only used for verification: a **ghost variable**

Verifying mutual exclusion

```
bool wantP = false, wantQ = false;
byte critical = 0;    /* ghost variable */
active proctype P() {
  do
  :: printf("Non critical section P\n");
     wantP = true;
     critical++;
     printf("Critical section P\n");
     assert (critical <= 1);
     critical--;
     wantP = false
  od
}
active proctype Q() {
  -- replace character 'P' with character 'Q'
}
```

Random simulation and verification

```
$ spin incorrectMutualExclusion.pml
      Non critical section P
          Non critical section Q
          Critical section Q
      Critical section P
spin: line  22 "incorrectMutualExclusion.pml", Error: assertion violated
spin: text of failed assertion: assert((critical<=1))
#processes: 2
wantP = 1
wantQ = 1
critical = 2
  8: proc  1 (Q) line  22 "incorrectMutualExclusion.pml" (state 5)
  8: proc  0 (P) line   9 "incorrectMutualExclusion.pml" (state 4)
2 processes created
```

```
$ spin -a incorrectMutualExclusion.pml; gcc -o pan pan.c; ./pan
...
pan: assertion violated (critical<=1) (at depth 22)
pan: wrote incorrectMutualExclusion.pml.trail
...
```

Chapter 4

Synchronization

Synchronization

- Promela does not have synchronization primitives such as semaphores, locks, and monitors that you may have encountered
- Instead, we model synchronization primitives by building on the concept of **executability** of statements

Synchronization via busy waiting

- The previous solution is trivially incorrect because no process reads the `want` variable of the other process
- Simple minded solution: write a loop before the entry of the critical section
- *Busy waiting* with a do-loop

```
do
:: !wantQ -> break
:: else -> skip
od
```

Synchronization via busy waiting _ The code

```
bool wantP = false, wantQ = false;
active proctype P() {
  do
  ::
    printf("Non critical section P\n");
    wantP = true;
    do
    :: !wantQ -> break
    :: else -> skip
    od;
    printf("Critical section P\n");
    wantP = false
  od
}
```

Synchronization via busy waiting and by blocking

- Loop in lines 7–10 performs no useful computation; it evaluates the guard repeatedly until it becomes true
- While busy waiting is an acceptable model for some systems (e.g., a multiprocessor with a large number of processors),
- Normally, computer systems are based upon **blocking** a process so that its processor can be assigned to another process process

Ciências ULisboa | Informática

Syncronization via blocking

- Replace the busy-waiting loop by blocking loop

```
do
:: !wantQ -> break
od
```

- *Simulation mode* in Spin will not choose the next statement to execute from that process
- In *verification mode* Spin will not continue the search for a counterexample for states that can be reached by executing statements from the process
- Hopefully statements from other processes will unblock the blocked process

Executability of statements

- There is no “looping” in the construct below. Why?

```
do
:: !wantQ -> break
od;
```

- The do-loop is superfluous; just use

```
!wantQ;
```

- In Promela it is possible to block on a single statement, not just on a compound statement
- An expression statement is **executable** if and only if it evaluates to true

The critical section problem written as it should be

```
bool wantP = false, wantQ = false;
active proctype P() {
  do
  :: printf("Non critical section P\n");
     wantP = true;
     !wantQ;
     printf("Critical section P\n");
     wantP = false
  od
}
active proctype Q() {
  -- replace character 'P' by 'Q'
}
```

Abbreviated solution for the critical section problem

```
bool wantP = false, wantQ = false;

active proctype P() {
  do :: wantP = true;
        !wantQ;
        wantP = false
  od
}

active proctype Q() {
  do :: wantQ = true;
        !wantP;
        wantQ = false
  od
}
```

State transition diagrams

- Recall that a state of a program is a set of values for the variables together with the location counters
- For the program above we have
  (value of `wantP`, value of `wantQ`, loc of `P`, loc of `Q`)
- The program has $2 \cdot 2 \cdot 3 \cdot 3 = 36$ possible states
- Not every state is **reachable** from the initial state
- A solution to the critical section problem is correct only if there are states that are **not** reachable. Which?

Building the set of reachable states

1. Let $\mathcal{S} = \{s_0\}$, where s_0 is the initial state; mark s_0 as *unexplored*
2. For each unexplored state $s \in \mathcal{S}$, let t be a state that results from executing an executable statement in s; it $t \notin \mathcal{S}$, add t to $\mathcal{S}$ and mark it an unexplored
3. Terminate when all states in $\mathcal{S}$ are marked explored

- The reachable states of a program can be viewed as a **state transition diagram**:

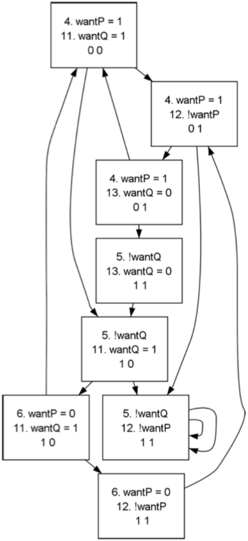
Nodes	are the reachable states
Edge	from s to t only when the execution of a statement in s leads to t

- Write the state transition diagram for the program above

State diagram

```
bool wantP = false,
     wantQ = false;
active proctype P() {
  do :: wantP = true;
        !wantQ;
        wantP = false
  od
}

active proctype Q() {
  do :: wantQ = true;
        !wantP;
        wantQ = false
  od
}
```



Mutual exclusion and absence of deadlock via state transition diagram analysis

Mutual exclusion holds if there is no state
(6. `wantP=false`, 13. `wantQ=false`, _ , _)
A quick glance at the diagram shows that no such state exists

Deadlock The program is **not free from deadlock**. State
(5. `!wantQ`, 12. `!wantP`, 1, 1)
is reachable and in that state both processes are trying to enter their critical regions, but neither can succeed

Detecting deadlocks in Spin

- The processes in the model consist of loops with no goto or break statements, so the program should never terminate
- Yet if you run several random simulations, you will see a `timeout` in the output:

```
$ spin third.pml
      Non critical section P
          Non critical section Q
...
      timeout
```

- An attempt at verification will discover an error called `invalid end state`:

```
pan: invalid end state (at depth 8)
```

- A process that does terminate must do so after executing its last instruction, otherwise it is said to be in an invalid end state. This error is always checked for regardless of any other correctness specifications

Summary _ Mutual exclusion and absence of deadlock

1. Instrument the code with a `critical` ghost variable
2. Use Spin in verification mode
3. Look for

   Mutual exclusion: `pan: assertion violated`

   Deadlock: `pan: invalid end state`

Avoiding deadlocks

- It is quite difficult to come up with a fully correct solution to the critical section problem just using expressions and assignment statements
- However, easy solutions to the problem can be given if the system can execute sequences of these statements atomically
- Idea: use a potentially blocking expression and the assignment statement as one atomic sequence of statements
- The atomic sequence may be blocked from execution, but once it starts executing, both statements are executed without interference from other processes

Atomic sequences of statements

```
bool wantP = false, wantQ = false;
active proctype P() {
  do
  :: printf("Noncritical section P\n");
     atomic {
       !wantQ;
       wantP = true
     }
     printf("Critical section P\n");
     wantP = false
  od
}
active proctype Q() {
  -- replace "P" with "Q"
}
```

Semaphores

- The most widely known construct for synchronizing concurrent programs is the **semaphore**
- There are two atomic operations defined for a semaphore:

  wait(sem) The operation is executable when the value of sem is positive; executing the operation decrements the value of sem

  signal(sem) The operation is always executable; executing the operation increments the value of sem

Critical section problem using semaphores

```
byte sem = 1;
active proctype P() {
  do ::
    printf("Non critical section P\n");
    atomic {
      sem > 0;
      sem--
    }
   printf("Critical section P\n");
    sem++
  od
}
active proctype Q() {
  -- replace character 'P' with 'Q'
}
```

A client-server program

```
byte request = 0;
active proctype Server1() {
  do
  :: request == 1 -> printf("Service 1\n"); request =
       0
  od
}
active proctype Server2() {
  do
  :: request == 2 -> printf("Service 2\n"); request =
       0
  od
}
active proctype Client() {
  request = 1;    /* invoke service 1 */
  request == 0;   /* wait for completion */
  request = 2;    /* invoke service 2 */
  request == 0    /* wait for completion */
}
```

Server processes and end states

- In a client-server system, clients are supposed to execute once
- Servers are there to serve clients and must execute an undefined number of times
- If we model a server with a `do-od` loop, we will get an error message: `invalid end state`
- Verify in Spin, and

```
$ ./pan
...
pan: invalid end state (at depth 10)
...
```

Valid end states

- The behaviour is nevertheless acceptable, for servers should wait indefinitely and be ready to supply a service whenever needed
- To mark a control point within a process that must be considered as a valid end point, prefix it with a label that begins with 'end'

```
active proctype Server1() {
endserver:
  do
  :: request == 1 -> printf("Service 1\n");
      request = 0
  od
}
```

Chapter 5

Verification with Temporal Logic

Beyond assertions

- Assertions are attached to specific control points
- Usually, it is necessary (at least more convenient) to express a correctness property as a **global property** of the system

Mutual exclusion In every state of every computation,
`critical <= 1`

Absence of deadlock In every state of every computation, if no statements are executable, the location counter of each process must be at the end of the process or at a statement labeled `end`

More global properties

Array index bounds If `i` is a variable used to index an array, then in every state of every computation, `0 <= i < LEN`

Quantity invariant In a token-passing algorithm, in every state of every computation, there is at most one token in existence

Properties that cannot be expressed using assertions

- There are properties that cannot be checked by evaluating an expression in a *single* state of a computation
- In a critical section problem:

Absence of deadlock In every state of every computation, if some processes are trying to enter their critical sections, eventually some process does so

Absence of starvation In every state of every computation, if a process tries to enter its critical section, eventually that process does so

Never claims

- The above specifications are expressed in Spin by a finite automaton called a **never claim**, executed together with the automaton that represents the model
- Specifying a correctness property directly as a never claim is difficult; instead
- A formula written in linear temporal logic is translated by Spin into a never claim, which is then used for verification

Propositional calculus

- Formulas of the propositional calculus are composed from atomic propositions (denoted by letters p, q, ...) and the operators:

Operator	Math	Spin	Spin (v. 6)
not	$\neg$	`!`	
and	$\wedge$	`&&`	
or	$\vee$	`||`	
implies	$\rightarrow$	`->`	`implies`
equivalent	$\leftrightarrow$	`<->`	`equivalent`

Linear Temporal Logic

- A formula of LTL is built from atomic propositions and from operators that include the operators of the propositional calculus as well as temporal operators:

Operator	Math	Spin	Spin (v. 6)
always	$\Box$	`[]`	`always`
eventually	$\Diamond$	`<>`	`eventually`
until	$\mathcal{U}$	`U`	`until`

- Example:

$\Box((p \wedge q) \rightarrow r\,\mathcal{U}\,(p \vee r))$

```
[]((p && q)-> r U (p || r))
always((p && q)implies r until (p || r))
```

It is always the case that (p and q) implies that r holds until (p or r) holds

The semantics of propositional calculus

- The **semantics**, the meaning, of a syntactically correct formula is defined by giving it an interpretation:
- an **assignment of truth values**, T (true) or F (false), to its atomic propositions, and
- the extension of the assignment to an interpretation of the entire formula according to the rules for the operators, familiar **truth tables**:

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

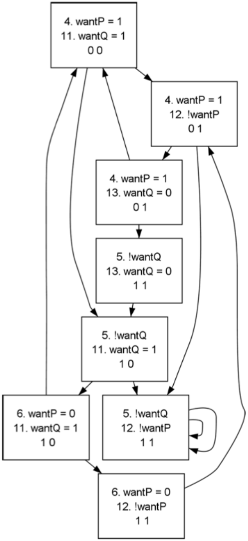
The semantics of linear temporal logic

- For temporal logic, the semantics of a formula is given in terms of computations and the states of a computation
- The atomic propositions of temporal logic are boolean expressions that can be evaluated in a single state independently of a computation
- E.g., the expression `critical <= 1` is an atomic proposition because it can be assigned a truth value in a state s just by checking the value of the variable critical in s

A solution for the critical section problem (revisited)

```
bool wantP = false,
     wantQ = false;
active proctype P() {
  do :: wantP = true;
        !wantQ;
        wantP = false
  od
}

active proctype Q() {
  do :: wantQ = true;
        !wantP;
        wantQ = false
  od
}
```



Finite representation of a computation

- A computation of the program is an **infinite** sequence of states that starts in the initial state
  (`4. wantP=1`, `11. wantQ=1`, 0, 0)
- An infinite computation may have a **finite** representation:

$$\begin{tikzcd}
4,11,0,0 \arrow[r] & 4,12,0,1 \arrow[r] & 4,13,0,1 \arrow[r] & 5,13,1,1 \arrow[r] & 5,11,1,0 \arrow[r] & 6,11,1,0 \arrow[lllll, bend right] \\
s_0 & s_1 & s_2 & s_3 & s_4 & s_5
\end{tikzcd}$$

Mutual exclusion in LTL

- Let *csp* be a proposition representing "Process P is in its critical section" (its location counter is at line 6)
- Let *csq* be a proposition representing "Process Q is in its critical section" (its location counter is at line 13)
- **For the computation shown above**, the formula

$$\neg(csp \wedge csq)$$

  expresses the correctness property of mutual exclusion is true in all its states
- Generalize: if we want to say that formula $\neg(csp \wedge csq)$ is true **in every state of every computation**, we write:

$$\Box\neg(csp \wedge csq)$$

Safety properties in LTL

- Let A be an LTL formula and let $\tau = (s_0, s_1, s_2, \dots)$ be a computation. Then $\Box A$, read **always** A, is true in state s_i if and only if A is true for all s_j in τ such that $j \geq i$
- The operator is reflexive so if $\Box A$ is true in a state s, then A must also be true in s
- The formula $\Box A$ is called a **safety property** because it specifies that the computation is safe in that nothing "bad" ever happens, or equivalently, that the only things that happen are "good"
- If the following diagram is extended indefinitely with all states labeled A, then $\Box A$ is true in s_0 (as well as in s_3, ...)

$$\begin{array}{ccccccccccc} \textcircled{A} & \longrightarrow & \textcircled{A} & \longrightarrow & \textcircled{A} & \longrightarrow & \textcircled{A} & \longrightarrow & \textcircled{A} & \longrightarrow & \cdots \\ s_0 & & s_1 & & s_2 & & s_3 & & s_4 & & \end{array}$$

Expressing safety properties in Promela

- Define the mutual exclusion problem with an `ltl` (Linear Temporal Logic) formula

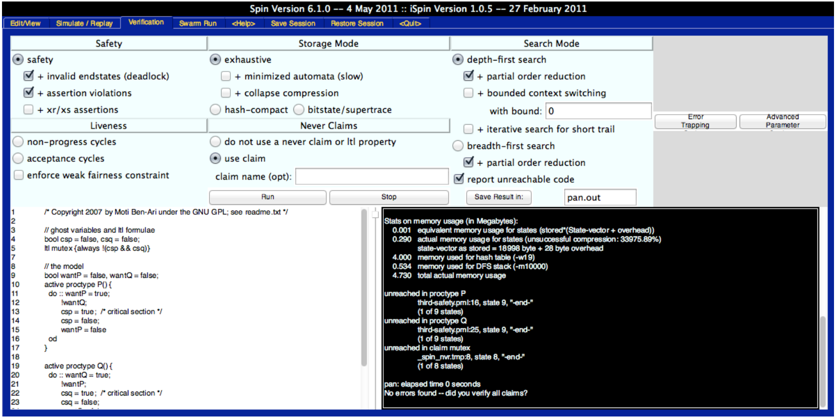
```
bool wantP = false, wantQ = false;
bool csp = false, csq = false;
ltl mutex {always !(csp && csq)}
active proctype P() {
  do
  :: wantP = true;
     !wantQ;
     csp = true;
     /* critical section here */
     csp = false;
     wantP = false;
  od
}
```

- `ltl` formulae in the source code (Spin ≥ 6); the name `mutex` is optional, but can be useful when specifying multiple formulae

Safety properties in iSpin

- Make sure your source code includes at least one `ltl` formula
- Select the `Verification` view
- Press the `safety` radio button
- Press the `use claim` radio button
- Press `Run`
- If the source code contains more than one `ltl` fill in the `claim name (opt)` the name of the claim you want to verify (the default is the first)

Verifying a claim



Ciências ULisboa | Informática

Safety properties from a command line

1. Generate the verifier (argument `-a`)

   ```
   spin -a third-safety.pml
   ```

2. Compile the verifier with the `-DSAFETY` argument so that it is optimized for checking safety properties

   ```
   gcc -DSAFETY -o pan pan.c
   ```

3. Run the thing!

   ```
   ./pan
   ```

4. Look for error messages

   ```
   pan:1: end state in claim reached (at depth 9)
   pan: wrote third-safety.pml.trail
   ```

5. Run a guided simulation with the trail to understand where the cycle is

   ```
   spin -t -p -g third-safety.pml
   ```

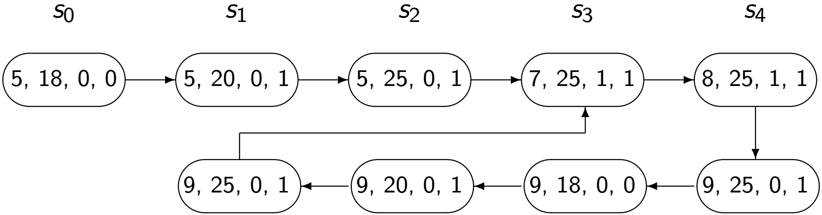
Liveness properties

- Let A be a formula of LTL and let $\tau = (s_0, s_1, s_2, \dots)$ be a computation. Then $\Diamond A$, read **eventually** A, is true in state s_i if and only if A is true for some s_j in τ such that $j \geq i$
- The operator is reflexive, so if A is true in a state s, then so is $\Diamond A$
- The formula $\Diamond A$ is called a liveness property because it specifies that something "good" eventually happens in the computation
- If *csp* is the atomic proposition that is true in a state if process P is in its critical section, then $\Diamond csp$ holds if and only if process P eventually enters its critical section
- It is essential that correctness specifications contain liveness properties because a safety property is vacuously satisfied by an empty program that does nothing!

Critical section with starvation

```
bool wantP = false, wantQ
     = false;
active proctype P() {
 do
 :: wantP = true;
   do
   :: wantQ ->
        wantP = false;
        wantP = true
   :: else -> break
   od;
   wantP = false /*c.s.*/
 od
}
```

```
active proctype Q() {
 do
 :: wantQ = true;
   do
   :: wantP ->
        wantQ = false;
        wantQ = true
   :: else -> break
   od;
   wantQ = false /*c.s.*/
 od
}
```



Verifying liveness properties

- Add the critical section marking statements and the `ltl` formula:

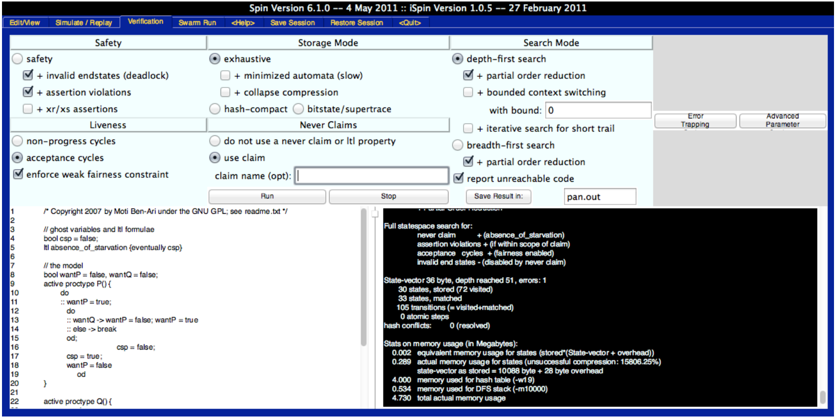
```
ltl absence_of_starvation {eventually csp}
...
do
  :: wantP = true;
     do
     :: wantQ -> wantP = false; wantP = true
     :: else -> break
     od;
     csp = true;
     /* critical section */
     csp = false;
     wantP = false
od
```

- Verification is checked as for a safety property; checker must be called in **acceptance cycles** and **weak fairness** mode

Liveness properties in iSpin

- Make sure your source code includes at least one `ltl` formula
- Select the `Verification` view
- Make sure the `acceptance cycles` radio button is on
- Press the radio button `use claim`
- Tick the `enforce weak fairness constraint` (Optional)
- Press `Run`
- If the source code contains more than one `ltl` fill in the `claim name (opt)` the name of the claim you want to verify (the default is the first)
- If an acceptance cycle is found, goto Simulate/Replay and select "Run"

Verifying a claim



Liveness properties from the command line

1. Generate and compile the verifier as before (do **not** use the `-DSAFETY` argument in the compilation)

   ```
   spin -a fourth-liveness.pml
   gcc -o pan pan.c
   ```

2. Run the verifier with the `-a` (acceptance) argument and the `-f` (weak fairness) argument

   ```
   pan -a -f
   ```

3. Look in the output for

   ```
   pan: acceptance cycle (at depth ...)
   ```

   meaning that liveness does not hold for the program

4. Run a guided simulation with the trail to understand where the cycle is

   ```
   spin -t -p -g fourth-liveness.pml
   ```

Counterexamples for safety and liveness properties

- Liveness does not hold for this program; the error message is

  `pan:1: acceptance cycle (at depth 14)`
- For safety properties, a counterexample consists of **one state** where the formula is false
- For liveness properties, a counterexample is an **infinite computation** in which something good (`csp` becomes true) never happens

Producing the loop

```
$ spin -t -p -g fourth-liveness.pml
[..]
pan:1: acceptance cycle (at depth 14)
pan: wrote fourth-liveness.pml.trail
[..]
$ spin -t -p -g fourth-liveness.pml
  2: proc 1 (Q) fourth-liveness.pml:21 (state 1) [wantQ = 1]
        wantQ = 1
[..]
 14: proc 0 (P) fourth-liveness.pml:10 (state 2) [(wantQ)]
  <<<<<START OF CYCLE>>>>>
[..]
 50: proc 0 (P) fourth-liveness.pml:10 (state 2) [(wantQ)]
spin: trail ends after 50 steps
```

- Spin outputs the first counterexample found
- Is there a shorter counterexample?

Finding the shortest counterexample

- The `-i` and `-I` arguments to `pan` can be used to perform an iterated search for shorter counterexamples.

```
$ ./pan -help
Spin Version 6.5.0 -- 1 July 2019
Valid Options are:
  ...
  -i  search for shortest path to error
  -I  like -i, but approximate and faster
  ...
```

Fairness

```
bool wantP, wantQ;
3  active proctype P() {
4   do
5   :: wantP = true;
6      do
7      :: wantQ ->
8           wantP = false;
9           wantP = true
10     :: else -> break
11     od;
12     wantP = false
13  od
14 }
```

```
15
16 active proctype Q() {
17  do
18  :: wantQ = true;
19     do
20     :: wantP ->
21          wantQ = false;
22          wantQ = true
23     :: else -> break
24     od;
25     wantQ = false
26  od
27 }
```

$$
\begin{array}{lllllll}
s_0 & = & (5.\ \texttt{wantP=1}, & 18.\ \texttt{wantQ=1}, & 0, & 0) & \longrightarrow \\
s_1 & = & (5.\ \texttt{wantP=1}, & 20.\ \texttt{wantP}, & 0, & 1) & \longrightarrow \\
s_2 & = & (5.\ \texttt{wantP=1}, & 25.\ \texttt{wantQ=0}, & 0, & 1) & \longrightarrow \\
s_3 & = & (5.\ \texttt{wantP=1}, & 18.\ \texttt{wantQ=1}, & 0, & 0) &
\end{array}
$$

- Is this computation a satisfactory counterexample to the claim that $\Diamond csp$ is true?

Weak fairness

- The above computation doesn't give process P a "fair" chance to try to enter its critical section: P remains in line 5 forever, while Q repeatedly enters its critical section
- A computation is **weakly fair** if and only if the following condition holds:
  If a statement is always executable, then it is eventually executed as part of the computation
- We require that only fair computations are considered as counterexamples
- Always add the argument `-f` (in addition to the argument `-a`) when executing the verifier `pan`
- Weak fairness takes a lot of memory; for more than two processes, use the compile-time directive `-DNFAIR=n`

Duality

- The operators $\Box$ and $\Diamond$ are dual in a manner similar to the duality expressed by deMorgan's laws

$$\neg(p \wedge q) \equiv \neg p \vee \neg q, \qquad \neg(p \vee q) \equiv \neg p \wedge \neg q$$

- Passing a negation through a unary temporal operator changes the operator to the other one

$$\neg\Box p \equiv \Diamond\neg p, \qquad \neg\Diamond p \equiv \Box\neg p$$

- We have the following equivalences

$$\neg\Box good \equiv \Diamond\neg good \equiv \Diamond\neg\neg bad \equiv \Diamond bad$$
$$\neg\Diamond good \equiv \Box\neg good \equiv \Box\neg\neg bad \equiv \Box bad$$

- "If it is false that something good is always true, then eventually something bad must happen"

Verifying correctness without ghost variables

- We have used ghost variables like `critical` and `csp`
- When modeling large systems you want to keep the number of variables as small as possible
- Promela supports **remote references** that can be used to refer to control points in correctness specifications
- Below, the expression `P@cs` (read "P at critical section") returns a nonzero value if and only if the location counter of process P is at the control point labeled by `cs`

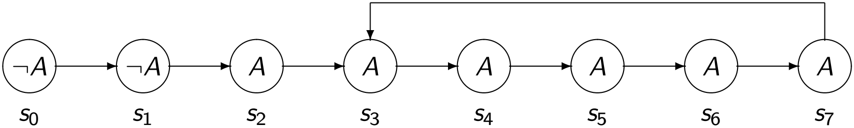
```
ltl mutex {always !(P@cs && Q@cs)}
active proctype P() {
  do
  :: wantP = true;
     !wantQ;
cs:  wantP = false;
  od
}
```

Advanced temporal specifications

- A formula with sequences of consecutive occurrences of the operators $\Diamond$ or $\Box$ is equivalent to one with a single occurrence. E.g., $\Diamond\Diamond\Box\Box A$ is equivalent to $\Diamond\Box A$
- A formula with any sequence of alternate occurrences of $\Diamond$ and $\Box$ is equivalent to one with a single sequence of $\Diamond\Box$ or $\Box\Diamond$ E.g., $\Diamond\Box\Diamond\Box A$ is equivalent to $\Box\Diamond A$
- Temporal logic formulas with more than two or three operators are difficult to understand
- Next we study some common patterns

Latching _ $\Diamond\Box A$

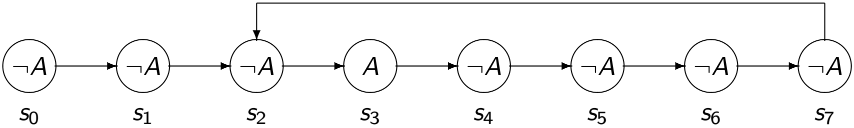
- The formula $\Diamond\Box A$ expresses a latching property: A may not be true initially in a computation, but eventually it becomes true and remains true



- It is unusual for a property to be true initially and always; rather, some statements must be executed to make the property true, although once it becomes true, the property remains true
- E.g., $\Diamond fails_Q \to \Diamond\Box\neg wantQ$

Infinitely often _ $\Box\Diamond A$

- The formula $\Box\Diamond A$ expresses the property that A is true infinitely often: A need not always be true, but at any state in the computation s, A will be true in s or in some state that comes after s



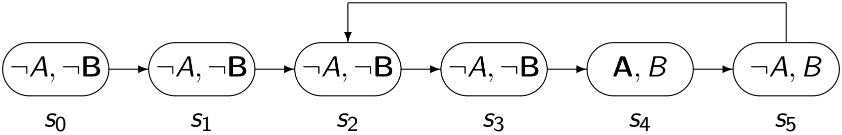
- For solutions to the critical section problem, liveness means not just that a process can enter its critical section, but that it can enter its critical section repeatedly
- E.g., $\Box\Diamond csp$

Precedence _ The until operator

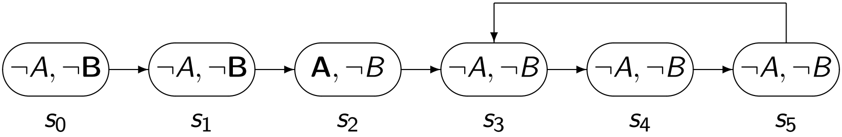
- The operators $\Box$ and $\Diamond$ are unary and cannot express properties that relate two points in time
- The **precedence** property requires that A becomes true before B becomes true
- Read $B\,\mathcal{U}\,A$ as "B remains true until A becomes true"
- $B\,\mathcal{U}\,A$ is true in state s_i of a computation τ if and only if there is some state s_k in τ with $k \geq i$, such that A is true in s_k, and forall s_j in τ such that $i \geq j < k$, B is true in s_j
- If A is already true in s_i, the second requirement is vacuous

Precedence _ $\neg B \, \mathcal{U} \, A$

- The formula $\neg B \, \mathcal{U} \, A$ is true in s_0 because B remains false as long as A does; only in s_4, when A becomes true, does B also become true:



- B can be false throughout the entire computation, and the truth of A beyond its first true occurrence is irrelevant. The formula $\neg B \, \mathcal{U} \, A$ is true in s_0



Chapter 6

Data and Program Structures

Self study

- Arrays
- Type definitions
- The preprocessor, `#include`, `#define`
- Inlining

Channels

Channels in distributed systems

- Distributed systems are computer systems consisting of a set of nodes connected by **communications channels**
- To model a distributed system we abstract away details of the network and its protocols, and model *nodes as concurrent processes* and *communications networks as channels* over which processes can send and receive messages
- One formalism for modeling distributed systems is called *Communicating Sequential Processes* (CSP), after a 1978 article by that name, written by C.A.R. Hoare (have we met him?)
- CSP was the inspiration for the communication constructs in several programming languages such as Occam and Ada, as well as for the channel construct in Promela

Channels in Promela

- Every channel has associated with it a message type; once a channel has been initialized, it can only send and receive messages of its message type
- The channel is declared with an initializer specifying the channel capacity and the message type:

```
chan ch = [capacity] of {typename, ..., typename
   }
```

- `typename` cannot be an array
- `capacity` denotes the length of the buffer to hold the messages.

  `capacity == 0` → *rendezvous* channels

  `capacity > 0` → *buffered* channels

Sending and receiving

- A channel in Promela is a data type with two operations, **send** and **receive**
- Send: exclamation is output

```
channel_variable ! expression, ...,
    expression
```

- Receive: question is input

```
channel_variable ? variable, ..., variable
```

- Receive statements cannot be executable unless a message is available in the channel. Receive statements will frequently appear as guards in an `if`- or `do`-statement

Client-server using channels

```
chan request = [0] of { byte };

active proctype Server() {
  byte client;
end: /* servers are not supposed to terminate */
  do
  :: request ? client -> printf("Client %d\n",
      client);
  od
}
active proctype Client0() {
  request ! 0;
}
active proctype Client1() {
  request ! 1;
}
```

A bit of syntactic sugar

- The send expression `ch!e1,e2,...` can be written

  ```
  ch!e1(e2,...)
  ```

- Particular useful when the first argument is an `mtype`

  ```
  mtype { open, close, reset };
  chan ch = [1] of {mtype, byte, byte};
  byte id, n;
  ```

- Rather than `ch ! open, id, n`, write:

  ```
  ch!open(id, n)
  ```

Channels and channel variables

- The type of all channel variables is `chan`; a channel variable holds a reference to the channel itself, which is created by an initializer
- Very little type-checking is performed on channels

```
chan c = [0] of {byte};
active proctype P() {
  c ! 5
}
active proctype Q() {
  byte x, y;
  c ? x, y -> printf("got %d and %d!\n", x, y);
}
```

- Compiles! only complains at runtime

```
$ spin -a channels-are-untyped.pml
$ gcc -DSAFETY -o pan pan.c
$ ./pan
pan:1: missing pars in receive (at depth 1)
```

Simple program with rendezvous

```
mtype { red, yellow, green };
chan ch = [0] of { mtype, byte, bool };

active proctype Sender() {
  ch ! red (20, false);
  printf("Sent message\n")
}

active proctype Receiver() {
  mtype color;
  byte time;
  bool flash;
  ch ? color, time, flash;
  printf("Received message %e, %d, %d\n",
         color, time, flash)
}
```

Rendezvous channels

- A channel declared with a capacity of zero is a **rendezvous channel**
- The transfer of the message from the sender to the receiver is **synchronous** and is executed as a **single atomic operation**

```
Sender                              Receiver
  ⋮                                    ⋮
(green,20,false)  ──────────→  (color,time,flash)
  ⋮                                    ⋮
```

- In line 5, the Sender *offers* to engage in a rendezvous
- In line 13, the rendezvous can be *accepted*
- Values are then copied from the sender to the receiver

State change

- State change:

```
5: ch ! ...,               6: printf...,
13: ch ? ...,    ------>   14: printf...,
0, 0, 0                    red, 20, false
```

- A send statement that offers to engage in a rendezvous for which there is no matching receive statement is itself not executable, and similarly for a receive statement with no matching executable send statement
- The process containing such a statement is blocked (unless there are alternatives with true guards in an `if`- or `do`-statement)

Client-server with reply channel

```
chan request = [0] of { byte };
chan reply = [0] of { bool };
active proctype Server() {
  byte client;
end:
  do
    :: request ? client ->
       printf("Client %d\n", client);
       reply ! true
  od
}
active proctype Client0() {
  request ! 0;
  reply ? _
}
active proctype Client1() {
  request ! 1;
  reply ? _
}
```

Ciências ULisboa | Informática

Identifying clients by `_pid`

- Notice the underscore in the reception `reply ? _`: an **anonymous variable**; we are interested only in the contents of the message (which is always `true`)
- The previous example works because the reply to the message was uniformly `true`
- If we want specific replies to be sent to specific clients, we can try to identify servers and clients via their `_pid`...

Multiple clients and servers

```
chan request = [0] of { byte };
chan reply = [0] of { byte };
active [2] proctype Server() {
  byte client;
end: do
     :: request ? client ->
        printf("Client %d processed by server %d\n",
               client, _pid);
        reply ! _pid
     od
}
active [2] proctype Client() {
  byte server;
  request ! _pid;
  reply ? server;
  printf("Reply received from server %d by client %d\n
     ",
         server, _pid)
}
```

Previous example not correct

- After a few random simulations…

```
$ spin rendezvous3.pml
      Client 2 processed by server 1
    Client 3 processed by server 0
          Reply received from server 1 by client 3
        Reply received from server 0 by client 2
```

- Exercise: How do we find the error by *verifying* the model?

Arrays of reply channels

- Use a separate reply channel for each client:

  ```
  chan reply[2] = [0] of { byte, byte };
  ```

- Senders reply on the appropriate `reply` channel
- Clients wait on the appropriate `reply` channel
- Exercise: adapt the code to use an array of reply channels

Local channels

- The version with arrays (suggested above) is quite fragile; we must get the channel number arithmetic right

```
request ! _pid, reply[_pid - 2]
```

- and we must not change the order by which `proctype Server` and `proctype Client` appear in the program
- and we must not change the number of servers
- A better solution uses local reply channels passed to servers together with the message contents

Client-server with local channels

```
chan request = [0] of { byte, chan };
active [2] proctype Server() {
  chan reply = [0] of { byte, byte };
  byte client;
end:
  do
  :: request ? client, reply ->
    reply ! _pid, client
  od
}
active [2] proctype Client() {
  chan reply = [0] of { byte, byte };
  byte server;
  byte whichClient;
  request ! _pid, reply;
  reply ? server, whichClient;
  assert _pid == whichClient
}
```

Limitations of rendezvous channels

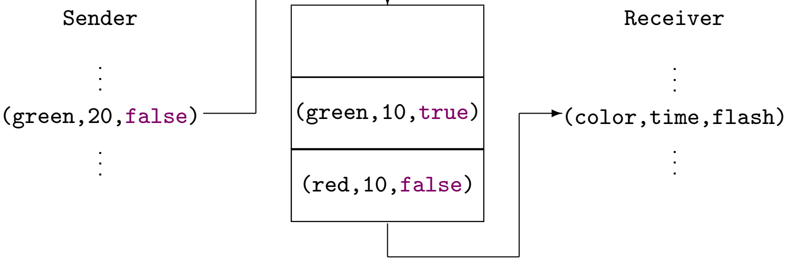
- Normally, there are many more clients than servers
- If rendezvous channels were used, the number of clients actually being served can be no larger than the number of servers, so the rest of the clients would be blocked
- Exercise: Prove this assertion with Spin. Suggestion: add a global variable for the number of unanswered requests.

Buffered channels

- A solution: queue the requests sent by the client in such a way that they do not block either the client or the server
- A channel declared with a positive capacity is called a **buffered channel**

```
chan ch = [3] of { mtype, byte, bool };
```

- The capacity is the number of messages that can be stored in the channel



Channel content is part of the state

- Send statements are executable if there is room in the channel queue
- Receive statements are executable if there are messages in the queue
- State diagram: channels as triples `[_,_,_]` of messages of the form `(_,_,_)`:

```
5: ch ! ...,
13: ch ? ...,
0, 0, 0,
[(red,10,false),
(green,10,true),
()]
```

```
6: printf,
13: ch ? ...,
0, 0, 0,
[(red,10,false),
(green,10,true),
(green,20,false)]
```

```
6: printf,
14: printf,
red, 10, false,
[(green,10,true),
(green,20,false),
()]
```

Checking the content of a buffered channel

- Systems may need to perform other tasks when channel operations are not executable
- There are four predefined boolean functions for checking a channel: `full` and `empty`, and their negations `nfull` and `nempty`
- The negations `!full` and `!empty` are not allowed in Promela; use `nfull` and `nempty` instead

Checking whether the channel is full or empty

```
chan request = [2] of { byte, chan};
active [2] proctype Server() {
 byte client; chan replyChannel;
  do
  :: empty(request) ->
      printf("No requests for server %d\n", _pid)
  :: request ? client, replyChannel ->
      printf("Client %d processed by server %d\n", client,
          _pid);
      replyChannel ! _pid
  od
}
active [2] proctype Client() {
  chan reply = [2] of {byte}; byte server;
  do
  :: full(request) ->
      printf("Client %d waiting for non-full channel\n",
          _pid)
  :: request ! _pid, reply ->
      reply ? server;
      printf("Reply received from server %d by client %d\n"
          , server, _pid)
  od
```

Random receive

- A different solution to the client-server problem: the array of *four* reply channels is replaced by a single channel of capacity *four*
- We need to ensure that it is possible for a client to receive only messages meant for it
- Problem 1: the client cannot receive the message until all messages ahead of it in the queue have been removed. Use **random receive**:

```
reply ?? server, ...
```

- Problem 2: the receive statement removes a message regardless of its content. Use **pattern matching**:

```
reply ?? server, 3
```

- When the constants in patterns are computed, use `eval`:

```
reply ?? server, eval(_pid)
```

Sorted send

- A send statement inserts the message at the tail of the message queue in the channel
- With the sorted send statement, the message is inserted ahead of the first message that is larger than it
- The first element in the queue is always the smallest
- Fields of the message are interpreted as integer values, and if there are multiple fields, lexicographic ordering is used
- Sorted send can be used to model a data structure such as a priority queue

Storing values in sorted order

```
chan ch = [3] of { byte };
inline getValue() {
  if
  :: n = 1
  :: n = 2
  :: n = 3
  fi
}
active proctype Sort() {
  byte n;
  getValue(); ch !! n;
  getValue(); ch !! n;
  getValue(); ch !! n;
  ch ? n; printf("%d\n", n);
  ch ? n; printf("%d\n", n);
  ch ? n; printf("%d\n", n)
}
```

Copying the value of a message

- Sometimes we are interested in copying the values in a message without removing the message from the channel
- Use angle brackets to enclose a list of variables

```
ch ? <color, time, flash>
ch ?? <color, time, flash>
```

Polling_ Checking whether there is data to be read

- Random receive

```
ch ?? <green, time, false>
```

  cannot be used in guards, since it creates side effects by reading values into variables
- A polling expression (written with square brackets) is side-effect free and can be used in a guard

```
do
:: ch ?? [green, _, false] ->
      ch ?? green, time, false
:: else -> /* Do something else */
od
```

- Since the evaluation of a guard and the execution of the first statement after the guard are two separate atomic operations, one may want to include the do statement within atomic

Comparing rendezvous and buffered channels

- Rendezvous channels are far more efficient. There is no "variable" associated with a rendezvous channel, so using one does not increase the size of a state
- Buffered channels greatly increase the potential size of the state space because every permutation of messages up to the capacity of the channel might occur in a computation
- The channel capacity must be carefully considered. A large capacity may be more realistic, but can cause an explosion in the size of the state space that can make verification impractical

Advanced Topics in Promela

Self study

- Specifiers for variables
- Predefined variables
- Priority
- Embedded C Code
- ...

Chapter 10

Advanced Topics in SPIN

Overview

- The success of Spin in industrial software development is primarily due to the efficiency with which it carries out verifications.
- Nevertheless, even the most efficient verifier will run up against limitations of time and memory, so that the task of the systems engineer is to find the appropriate tradeoffs between model complexity and resources.
- We will also see how correctness specifications in temporal logic are translated into never claims in Promela

How Spin searches the state space

- Spin does not actually "search the graph" in the sense that the graph is constructed and then searched; instead, Spin builds the target state "on-the-fly"
- Search becomes more efficient because Spin needs only construct states until the first counterexample is found
- If there are no errors, all the states in the state space will eventually be built, so the on-the-fly construction saves nothing, but in most cases it is efficient because we construct more models with errors than we do error-free models!
- For an efficient search it is important to maintain a data structure that stores all states that have been visited
- During verification Spin expends most of its resources (time and memory) storing states in this data structure and looking up newly created states to see if they have been visited before

Optimizing the performance of verifications

- Write efficient models
- Understand how Spin allocates memory for the hash table
- Compress the state vector
- Use a minimal automaton

Writing efficient models

- The data stored for each state, called the **state vector**, consist of the location counters of the processes and the values of the variables, for example, (4,11,0,0).
- Reducing the memory needed to store a space vector:
  - Do not declare unnecessary variables, and declare variables with as narrow a type as possible; so, `byte` is preferable to `int`, and `bit` or `bool` is preferable to `byte`
  - Avoid declaring channel capacities in excess of what is needed to verify the model
  - Use `atomic` and `d_step` where possible, but be sure that you are not "masking" possible error states by incorrectly restricting the interleaving
  - Use as few processes as possible (next slides)

Memory for mtype and arrays of bool

Memory requirements will not be reduced in the following cases:

- A value of an `mtype` is stored in a full byte, regardless of the number of symbols defined
- An array whose elements are of type `bit` or `bool` is stored as an array of type `byte`; section 11.7.2 shows how to encode sets of bits into bytes

Reducing the number of processes

- If a process serves only to generate data to be sent on a channel, you can remove the process and generate the data within the process receiving the data using a non-deterministic statement to select the message "received"
- In the next example, this reduces the size of the state vector from 20 bytes to 12 bytes and the number of distinct states from 123 to 64

Ciências ULisboa | Informática

Generating input in a separate process

```
#include "for.h"
chan ch = [0] of { byte };
active proctype Producer() {
  for (i, 1, 10)
    if
    :: ch ! 0
    :: ch ! 10
    :: ch ! 20
    fi
  rof (i)
}
active proctype Consumer() {
  byte n;
end:
  do
  :: ch ? n -> printf("%d\n", n)
  od
}
```

Generating input in the same process

```
#include "for.h"
active proctype Consumer() {
  byte n;
  for (i, 1, 10)
    if
    :: n = 0
    :: n = 10
    :: n = 20
    fi;
    printf("%d\n", n)
  rof (i)
}
```

Allocating memory for the hash table

- Spin uses a hash table to store the state vectors that have been previously encountered
- At the end of a verification, the verifier prints data on the use of memory:

```
State-vector 16 byte, depth reached 24, errors: 0
1.67772e+007 states, stored
8.38861e+006 states, matched
2.51658e+007 transitions (= stored+matched)
hash conflicts: 6.58266e+008 (resolved)
269.964 total actual memory usage
```

- About 16.8 million state vectors were stored in the hash table, while another 8.4 million were found to be in the table when they were generated by the search $\rightarrow$ number of states stored reduced by one third

Ciências ULisboa | Informática

Increasing the size of the hash table

- The number of hash conflicts was 658 million, and this indicates that most of the execution time went into searching the linked lists associated with conflicting hash table entries. Clearly, it is worthwhile increasing the number of elements in the table
- The default for the number entries in the table is $2^{18} = 256K$; use `-w` argument when executing the verifier:

  `pan -w20`

Hash table (2^w)	Memory (MB)	Conflicts (x10^6)	Time (sec)
18	270	658	86
20	273	151	33
22	286	52	22
24	336	29	18
26	537	0.1	17

Compressing the state vector

- Spin implements a sophisticated method for encoding the state vector called collapse compression. Use

  `gcc -DCOLLAPSE (other arguments) -o pan pan.c`
- Tradeoff:

Compression	Default	Collapse
Space	127 bytes	39 bytes
Time	6 sec	8 sec

Minimal automaton

- State vectors can be stored without a hash table using a representation called a **minimal automaton** that is similar to the binary decision diagrams used in other model checkers. Use

  ```
  gcc -DMA=10 (other arguments) -o pan pan.c
  ```

- The memory requirements can be reduced to a very small amount, but the execution time is likely to rise significantly

Exploring the search space in presence of an LTL formula

- We have seen how Spin explores the state transition diagram looking for error states where an assertion evaluates to false or for invalid end states that can indicate deadlock
- Checking correctness properties expressed as formulas of temporal logic is more difficult

Never claims

- Recall that `csp` is true if process `P` is in its critical section. The truth of `<>csp` cannot be evaluated just by looking at a single state
- It is true in a state s_0 if there is an accessible state s_k in which `csp` is true. Therefore, it can be falsified in s_0 only if there exists an infinite computation starting in s_0 in which `csp` is never true
- Spin transforms a formula in temporal logic into a Promela construct called a **never claim**
- A never claim specifies an automaton whose state space is searched in parallel with the one that is defined by the Promela program

A never claim for a safety property

- In an algorithm for solving the critical section problem:

  ```
  #define mutex (critical <= 1)
  ```

  and consider the specification `[]mutex`
- Spin translates the negation of `mutex` into

  ```
  never { /* !([]mutex) */
  T0_init:
    if
    :: (! ((mutex))) -> goto accept_all
    :: (1) -> goto T0_init
    fi;
  accept_all:
    skip
  }
  ```
- Looks similar to an ordinary Promela program

A competition

- Imagine a competition between you and the verifier generated by Spin. You claim `[]mutex`, while Spin aims to show that you are wrong because `![]mutex` holds
- You "win" if it is never true that `![]mutex` holds, while Spin "wins" if it can find a computation in which `![]mutex` holds
- Spin plays first, because it is possible that there is a counterexample in the initial state
- He who terminates first "wins" the game

The game when mutual exclusion does not hold

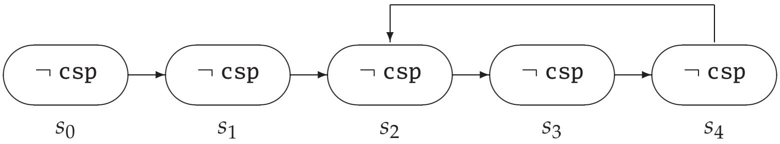
- In the initial state of an algorithm for the critical section problem, `mutex` is always true
- The only executable alternative is the one guarded by `(1)`
- Control returns to the start of the if-statement at the label `T0_init`
- You and spin take turns executing one (atomic) statement at a time. Your program will execute the steps of the algorithm, while Spin will remain in the loop defined by goto `T0_init` as long as `mutex` is true
- When your program finally enters a state in which `mutex` is false, the nondeterministic if-statement can choose the first alternative and jump to the label `accept_all`
- Spin has successfully terminated its program (the never claim). Spin wins!

The game when mutual exclusion holds

- In this case, Spin will never be able to complete the computation of the claim (which is why it was called a never claim)
- The search will terminate and Spin will not win because it cannot find a computation in which `![]mutex` is true
- If we unravel the double negation, `![]mutex` is not true so `[]mutex` is true, and the verifier reports that there are no errors
- You win!

A never claim for a liveness property

- We now consider the liveness property `<>csp`
- Spin wins the game if it can find a computation in which `!<>csp` holds
- Spin must find an infinite computation in which `!csp` is true in all states, such as this



The never claim for `!<>csp`

```
never { /* !(<>csp) */
accept_init:
T0_init:
  if
  :: (! ((csp))) -> goto T0_init
  fi;
}
```

First case: the property holds.

- If `csp` ever becomes true, Spin is blocked in its if-statement because there are no alternatives to `(!((csp)))`, and blocking is considered a loss for Spin
- A computation that contains a state in which `csp` is true has been found, so the computation cannot falsify `[]!csp`
- You win!

The liveness property does not hold

- If `csp` never becomes true, Spin will loop forever at the never claim; it repeatedly executes the if-statement labeled `accept_init`
- A computation of a never claim that infinitely often passes through a statement whose label begins with `accept` is called an **acceptance cycle**. If a verification finds an acceptance cycle, it is considered a win for Spin
- This acceptance cycle shows that there is an infinite computation in which `!csp` is always true, that is, `[]!csp` is true. By duality, this is equivalent to `!<>csp`, so `<>csp` is false
- Spin wins!
- The acceptance cycle is written to the trail so that you can examine the counterexample to find the error

Case studies

Case study 1 _ The eight-queens problem

- Problem: write an algorithm to place eight queens on an 8×8 chessboard so that no queen can capture any other

1	Q							
2							Q	
3					Q			
4								Q
5		Q						
6				Q				
7						Q		
8			Q					
	1	2	3	4	5	6	7	8

Floyd's algorithm

- A solution, due to Floyd (have we met him?), to the problem is an array of eight integer values stored in the variable `result`; for each column `i`, `result[i]` is the row in which the queen is placed
- The algorithm works by nondeterministically choosing a row for each column in sequence, and then checking that a queen placed on that square cannot capture a queen that has already been placed on the board
- To facilitate checking for captures, three auxiliary boolean arrays are used: `a[i]` is true if there is a queen in row `i`; `b[i]` is true if there is a queen on the positive diagonal `i`; `c[i]` is true if there is a queen on the negative diagonal `i` (positive diagonals go from the lower left to the upper right)

Eight queens in Promela

```
byte result[8];
bool a[8];  /* Queen in row i? */
bool b[15]; /* Queen in positive diagonal i? */
bool c[15]; /* Queen in negative diagonal i? */
active proctype Queens() {
  byte col = 1; byte row;
  do
  :: Choose(); // choose one [1..8] row arbitrarily
     !a[row-1];
     !b[row+col-2];
     !c[row-col+7];
     a[row-1]       = true;
     b[row+col-2]  = true;
     c[row-col+7]  = true;
     result[col-1]  = row;
     if
     :: col == 8 -> break
     :: else     -> col++
     fi
  od;
  Write();
}
```

Eight queens in Promela (cont)

```
inline Choose() {
  if
  :: row = 1
  :: row = 2
  :: row = 3
  :: row = 4
  :: row = 5
  :: row = 6
  :: row = 7
  :: row = 8
  fi
}

inline Write() {
  for (i, 1, 8)
    printf("%d, ", result[i-1])
  rof (i);
    printf("\n")
}
```

Checking the correctness of the solution

- There is not much point in running a random simulation of this algorithm. The vast majority of the computations of the program end with the process `Queens` blocked because one queen can capture another

```
$ spin queens.pml
      timeout
#processes: 1
    result[0] = 4
    result[1] = 8
    result[2] = 3
    result[3] = 0
    result[4] = 0
    result[5] = 0
    result[6] = 0
    result[7] = 0
```

- Exercise: How can we *verify* the model?

Verifying the solution

Two things must be done

1. Whenever the process is blocked because one queen can capture another, the process is at an end state, but it is an invalid one that is not the final statement of the process

   ```
   $ spin -a queens.pml; gcc -DSAFETY -o pan pan.c; pan
   pan: invalid end state (at depth 10)
   ```

   To enable the verifier to continue the search for a counterexample, add `end` labels at "dead ends":

   ```
   enda: !a[row-1];
   endb: !b[row+col-2];
   endc: !b[row-col+7];
   ```

2. Add `assert(false)` at the last line of the program. A counterexample will contain the solution!

Printing a solution to the problem

- Want to know the solution?
- Use the trail!

```
$ spin -a queens.pml; gcc -DSAFETY -o pan pan.c; ./pan
pan: assertion violated 0 (at depth 105)
pan: wrote queens.pml.trail
...
$ spin -t queens.pml
   1,    5,    8,    6,    3,    7,    2,    4,
spin: line  49 "queens.pml", Error: assertion violated
...
```

92 solutions

- It is well known that the eight-queens problem has 92 solutions
- Request that the verifier find them all

```
pan -E -c0 -e
pan: assertion violated 0 (at depth 104)
pan: wrote queens.pml1.trail
...
pan: wrote queens.pml86.trail
```

- Spin optimizes its search by ignoring write-only variables since they cannot affect the correctness of a correctness specification
- Solution: force a read to variable `result`

```
_ = result[0]
```

after the `do` and before the `assert(false)` statements

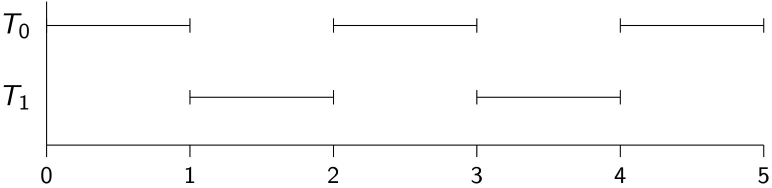
Case study 2 _ Modeling a real-time scheduling algorithm

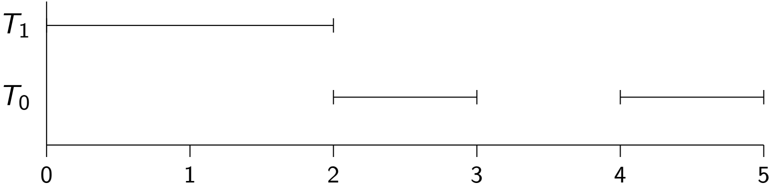
- Tasks in a real-time system are generally defined to be periodic: with each task we associate a period p and an execution time e. The task is required to execute at least once every p units of time (microseconds or milliseconds or seconds), and it needs at most e units to complete its execution
- Each task is given one or more slots within a period of time
- We are interested in the case where tasks are given priorities and a preemptive scheduler ensures that a lower-priority task is not run if a higher-priority task is ready

The problem

- Consider, for example, two tasks T_0 and T_1, such that $p_0 = 2$, $e_0 = 1$, and $p_1 = 5$, $e_1 = 2$
- Problem: is there a feasible assignment of priorities, i.e., is there an assignment of priorities such that each task receives the execution time it requires when the tasks are scheduled by an asynchronous scheduler

Feasible and unfeasible priority assignment





- Assigning T_0 a higher priority than T_1 is feasible; the opposit not

Periodic execution of tasks

```
#define N 2 /* Number of tasks */
byte clock = 0;
bool done[N]
proctype Task(byte ID; byte period; byte exec) {
  byte next = 0;
  do
  ::  atomic {
      clock >= next ->
        clock = clock + exec;
        next = next + period;
        done[ID] = true;
        printf("Process=%d, clock=%d\n", ID,
            clock)
    }
  od
}
```

Watchdog

```
/* One per task */
proctype Watchdog(byte ID; byte period) {
  byte deadline = period;
  do
  ::  atomic {
      clock >= deadline ->
        assert done[ID];
        deadline = deadline + period;
        done[ID] = false
    }
  od
}
```

- Used to solely for the `assert`

Running the scheduler

```
Proctype Idle() {
  do
  :: atomic {
       timeout -> {
         clock++;
         printf("Idle, clock=%d\n", clock)
       }
     }
  od
}
init {
  atomic {
    run Idle();
    run Task(0, 2, 1); run Watchdog(0, 2);
    run Task(1, 5, 2); run Watchdog(1, 5)
  }
}
```

Timeout

- Occurrences of a timeout state can be detected
- The predefined variable `timeout` is true when there are no executable statements in *any* process
- Sort of a global `else`
- `else` is executable when there are no executable guards in an enclosing `do`/`if` statement; `atomic` is executable when there are no executable statements in the program

An incorrect solution

- T_1 may be scheduled before T_0. Then T_1 is executed first, followed by the `Watchdog` for process T_0. It detects that the deadline for T_0 has arrived but its `done` flag is not set
- Before proceeding, we combine both the `Task` and its `Watchdog` into one process, in order to minimize the number of processes running

Simplifying the model

```
proctype Task(byte ID; byte period; byte exec) {
  byte next = 0;
  byte deadline = period;
  bool done = false;
  do
  ::  atomic {
      (clock >= next) && (clock < deadline) ->
        clock = clock + exec;
        next = next + period;
        done = true;
        printf("Process=%d, clock=%d\n", ID, clock)
    }
  ::  atomic {
      clock >= deadline ->
        assert done;
        deadline = deadline + period;
        done = false
    }
  od
}
```

Modeling a scheduler with priorities

- The priorities are modeled by a *sorted* queue used to store the tasks. The messages in the channel are the IDs of the tasks. Tasks with lower IDs are assumed to have higher priority

```
chan queue = [MAX] of { byte }
```

- When a task must be executed (`clock>=next`), it places its `ID` on the queue:

```
queue !! ID
```

- A guard ensures that a task is not queued if it is already in the channel

```
!(queue ?? [eval(ID)]) -> queue !! ID
```

When a task must be executed

```
proctype Task(byte ID; byte period; byte exec) {
  byte next = 0;
  byte deadline = period;
  byte current = 0;
end:do
  ::  atomic { /* when time comes... */
      (clock >= next) && (clock < deadline) &&
      (clock < maxPeriod) &&
      !(queue ?? [eval(ID)]) ->
        queue !! ID; /* enqueue the task */
        printf("Process=%d put on queue\n", ID)
    }
  ...
```

The execution of a task

- Tasks are executable under the same conditions on `clock` as the previous alternative, together with the condition that the task's `ID` is at the head of the queue

```
queue ? [eval(ID)]
```

- To enable preemption, `clock` is incremented by one unit of time. Same for variable `current` which keeps track of how many units have been executed by this task
- If the execution time of the task has completed (`current==exec`), the variable `current` is reset and the time when the task must be executed next is computed

The execution of a task (cont)

```
:: atomic {
     (clock >= next) && (clock < deadline) &&
     (queue ? [eval(ID)]) ->
       current++;
       clock++;
       printf("Process=%d, clock=%d, current=%d\n",
                   ID, clock, current);
       if
       :: current == exec ->
          queue ? eval(ID);
          current = 0;
          next = next + period;
          printf("Process=%d taken from queue\n", ID)
       :: else
       fi
   }
```

Watchdog

```
:: atomic {
    (clock >= deadline) ->
      assert (!(queue ?? [eval(ID)]));
      deadline = deadline + period
  }
od
/* end of proctype Task */
```

Idle and initial processes

```
proctype Idle() {
end:
  do
  :: atomic {
       (clock < maxPeriod) && timeout -> clock++
     }
  od
}
init {
  atomic {
    run Idle();
    maxPeriod = 5;
    queue ! 0; queue ! 1;
    run T(0, 2, 1);
    run T(1, 5, 2);
  }
}
```

Counterexamples?

- Clearly if a counterexample exists, then one exists within the initial part of the computation defined by the first period of some task
- It is sufficient to verify the model for values of clock up to `maxPeriod`
- How do we check the correctness of the model?
- No need to assert or to use temporal logic; Spin does it for you via invalid end states:
- The alternative that adds a task to the queue (line 14 of `proctype Task`) and the alternative that increments clock (line 5 of `proctype Idle`) become unexecutable when the value of `maxPeriod` is reached
- That simple!

Modeling distributed systems

- The natural way to model a distributed system in Promela is to represent the nodes as processes and the communications channels as channels
- We need to synchronize between different processes within the same node
- This we do via a nondeterministic `do`-statement. Each time the `do`-statement is executed, one of the alternatives is chosen, and this models the interleaving of the concurrent processes
- Programs with channels require a lot of resources to verify, so models with fewer channels are to be preferred
- We associate a single incoming channel with each node, and the process sending a message will pass its identification to the process receiving the message in an additional field

Global snapshots

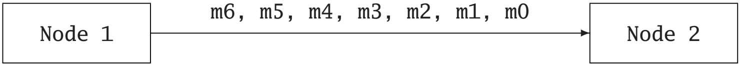
- **Global snapshot**: a snapshot a set of data giving a consistent state of a distributed system
- In a system with shared memory, obtaining a snapshot is easy: simply block all the processes and make a copy of the shared memory
- In a distributed system nodes can communicate only by messages, which are not transferred instantly and can take time to move through the channels $\rightarrow$ there is no global "bird's- eye" view of the system
- If Node 1 sends a message to Node 2, it is possible for the message to get temporarily "lost" because Node 1 sent the message, while Node 2 doesn't yet know of its existence
- A snapshot is **consistent** if it can unambiguously identify every message that has been sent as either **received** or as **still in the channel**

Case study 3 _ The Chandy-Lamport algorithm for global snapshots

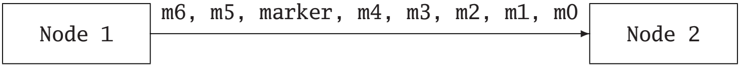
- The Chandy-Lamport algorithm adds a new type of message called a **marker**
- Markers are sent by each node over each outgoing channel as a signal that the node has recorded its state. Every message sent before the marker "belongs" to either the receiving node or the channel

Chandy-Lamport _ Example

- Seven messages in a channel between the two nodes:



- The marker has been sent after message `m4`



- Node 1 records its state as having sent message `m0` through `m4`. It is now the responsibility of Node 2 to record that it has received a subsequence of the messages, say `m0` through `m2`
- The rest of the messages, here `m3` and `m4`, are recorded as being in the channel

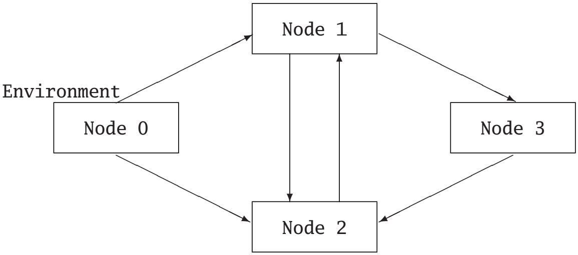
The algorithm

- **Send message** sends messages on an outgoing channel; records the last message sent (in a variable `lastSent`)
- **Receive message** receives messages on an incoming channel and records the last message received (in a variable `lastReceived`)
- **Receive marker** receives a marker on an incoming channel. First it records the last message that was received on this channel before the marker (in `messageAtMarker`). Then, if the state has not yet been recorded, the set of messages sent on each outgoing channel is recorded (in `stateAtRecord`) and the set of messages received on each incoming channel is recorded (in `messageAtRecord`). Finally, he markers are sent on all outgoing channels
- **Display** waits until markers have been received on all incoming channels and then displays the recorded state

The state of a node

- Consists of the set of messages sent on its outgoing channels before it recorded its state, and the set of messages received on its incoming channels before it recorded its state
- For each incoming channel on which a marker is received after the node has recorded its state, the messages between `messageAtRecord` and `messageAtMarker` are assigned by the node to the state of that channel

Example




```
Node 1, last sent to 2 = 2
     Node 1, last received from 2 = 7
     Messages in channel 2 -> 1 = 8 .. 12
     Node 1, last sent to 3 = 17
             Node 3, last received from 1 = 17
             Node 3, last sent to 2 = 6
         Node 2, last sent to 1 = 12
         Node 2, last received from 1 = 2
         Node 2, last received from 3 = 3
```

Structure of the program

- One process per node, plus one for the environment

```
proctype Env(byte outgoing) { ... }

proctype Node(byte me;
  byte numIncoming; byte incoming;
  byte numOutgoing; byte outgoing) { ... }

init {
  atomic {
    run Env(4+2);
    run Node(1, 2, 4+1, 2, 8+4);
    run Node(2, 3, 8+2+1, 1, 2);
    run Node(3, 1, 2, 1, 4)
}
```

Nodes

```
#define NODES 4
mtype = { message, marker };
chan ch[NODES] = [NODES] of { mtype,
  byte/*source node*/, byte/*message number*/ };

proctype Node(byte me;
  byte numIncoming; byte incoming;
  byte numOutgoing; byte outgoing) {
  do
  :: /* Send a message */
  :: /* Receive a message */
  :: /* Receive a marker */
  :: markerCount == numIncoming ->
    PrintState();
    break
  od
}
```

Encoding sets of ids

- For each node, messages and markers are sent and received only on the incoming and outgoing channels
- Three methods:
  - An array of boolean flags: a flag is true if the node corresponding to the array index is in the subset
  - A channel that contains the subset of the node IDs
  - An integer variable that encodes the subset: a bit is 1 if the position of the bit is in the subset.

Integers as sets

Incoming	0	0	0	0	1	0	1	1
Outgoing	0	0	0	0	0	0	1	0

- To check if a channel exists, we need to check if the corresponding bit is 1. This is done by shifting the byte by a number of places equal to the index of the channel and then masking the lowest-order bit

```
#define isOne(v,n) (v >> n & 1)
```

The environement node

```
proctype Env(byte outgoing) {
  startSnapshot; // for interesting simulations
  for (I, 1, NODES-1)
  if
  :: isOne(outgoing,I) ->
    ch[I] ! marker, 0, 0;
  :: else
  fi
  rof (I)
}
```

Local data for each node

- Message numbers sent on outgoing channels or received from incoming channels (a node may be connected to every other node)

```
byte lastSent[NODES];
byte lastReceived[NODES];
byte stateAtRecord[NODES];
byte messageAtRecord[NODES];
byte messageAtMarker[NODES];
byte markerCount;
bool recorded;
```

- `markerCount` counts the number of incoming markers; when markers have been received on all incoming edges, its value equals `numIncoming` and the process can print the state and terminate
- `recorded` is used to ensure that a state is recorded only once

Sending a message

```
:: numOutgoing != 0 ->
    GetOutgoing(); // choose arbitrary
        destination
    if
    :: full(ch[destination])
    :: nfull(ch[destination]) ->
        ch[destination]!message(me,messageNumber
            );
        lastSent[destination] = messageNumber;
        messageNumber++;
    if
    :: messageNumber > MESSAGES ->
        startSnapshot = true
    :: else
    fi
  fi
```

Ciências ULisboa | Informática

Receiving a message

```
:: ch[me] ? message(source, received) ->
    lastReceived[source] = received;
```

Receiving a marker

```
:: ch[me] ? marker(source, _) ->
    messageAtMarker[source] = lastReceived[source];
    markerCount++;
    if
    :: recorded -> skip
    :: else ->
        recorded = true;
        for (I, 0, NODES-1)
          stateAtRecord[I] = lastSent[I];
          messageAtRecord[I] = lastReceived[I]
        rof (I);
        for (J, 0, NODES-1)
          if
          :: isOne(outgoing,J) ->
                ch[J] ! marker(me, 0)
          :: else
          fi
        rof (J)
    fi
```

Nondeterministic choice of a channel _ Hard code

- If the topology of the network were coded within each node, a simple nondeterministic `if`-statement would suffice for choosing a destination for sending a message:

```
/* Choose destination in Node 1 */
if
:: destination = 2
:: destination = 3
fi
```

- Instead...

Nondeterministic choice of a channel _ Soft code

```
inline GetOutgoing() {
  atomic { // we are just computing a local variable
    byte num, out;
    num = numOutgoing; out = outgoing;
    destination = 0;
    do
    :: (out&1)==0 ->
        out = out >> 1;
        destination++
    :: (out&1)==1) ->
        break
    :: ((out&1)==1)&&(num>1) ->
        num--;
        out = out >> 1;
        destination++
    od
  }
}
```

Verification of the snapshot algorithm

- It is sufficient to check its behavior on a single channel
- In principle, a second channel is needed in order to model the case where a marker has been received on one channel (and the state recorded) before the marker is received on another channel. But it is not necessary to model the reception of the first marker using a channel!
- It is sufficient if the action of recording the state can occur at an arbitrary control point in the algorithm, and this is easily modeled by nondeterministic selection
- We use two processes, a `Sender` and a `Receiver` connected by a single channel

Global variables

```
mtype = { message, marker };
chan ch = [SIZE] of { mtype, byte };
byte lastSent, lastReceived,
  messageAtRecord, messageAtMarker;
bool recorded;
```

The sending process

```
active proctype Sender() {
  do
  :: lastSent < MESSAGES ->
       lastSent++;
       ch ! message(lastSent)
  :: ch ! marker(0) ->
       break
  od
}
```

The receiving process

```
active proctype Receiver() {
  byte received;
  do
  :: ch ? message(received) ->
        lastReceived = received
  :: ch ? marker(_) ->
        messageAtMarker = lastReceived;
        if
        :: !recorded ->
              messageAtRecord = lastReceived
        :: else
        fi;
        break
  :: !recorded ->
        messageAtRecord = lastReceived;
        recorded = true
  od
}
```

Specifying channel capacity and number of messages

- It is well known that bugs tend to occur at the limits of a data structure, for example, when it is empty or when it is full
- If we choose `SIZE` to be four, we can claim to have verified the algorithm if the marker is sent before or after the first or the last element, as well as "in the middle"
- Sending six messages seems to be reasonable. Clearly you wouldn't send fewer messages than the channel capacity, because then you would not check the case of a full channel; furthermore, there is no need to send more than one or two messages beyond those needed to fill the channel

What properties to prove?

- Verify the safety of the algorithm: that the snapshot is consistent
- Two assertions placed after the `do`-statement in the `Receiver`

```
assert (lastSent == messageAtMarker);
assert (messageAtRecord <= messageAtMarker)
```

- The first assertion states that all messages sent before the marker have been received
- It is of course possible that the state had already been recorded when the marker was received on the channel; in that case, the messages between `messageAtRecord` and `messageAtMarker` are attributed to the channel rather than to one of the nodes
- Since we are assured that messages are received in FIFO order, it is sufficient to check the expression in second assertion

Bibliography

- Mordechai Ben-Ari, *Principles of the Spin Model Checker*. Springer, 2008. Chapters 1 to 11.
- *Model Checking: Algorithmic Verification and Debugging*. Turing Lecture from the winners of the 2007 ACM Turing Award. In particular the contribution by Allen Emerson.

Antónia Lopes, Vasco T. Vasconcelos

December 10, 2019