Model Checking

Raul Barbosa

VALU3S Summer School
Genova, Italia, July 18–20, 2023

- Part I – Promela introduction
  We begin by studying PROMELA, which is a formal language and allows us to describe systems precisely and unambiguously. This language is firstly interpreted by a tool called SPIN.
- Part II – Model checking using Spin
  We study how to use SPIN to perform exhaustive verification of sequential programs.
- Part III – Verifying concurrent programs
  Concurrent programs are especially tricky when it comes to verification, so we examine how to look for deadlocks, livelocks, starvation, race conditions, and so on, using model checking.
- Part IV – Specification and verification using LTL
  Linear Temporal Logic for expressing liveness & safety properties.

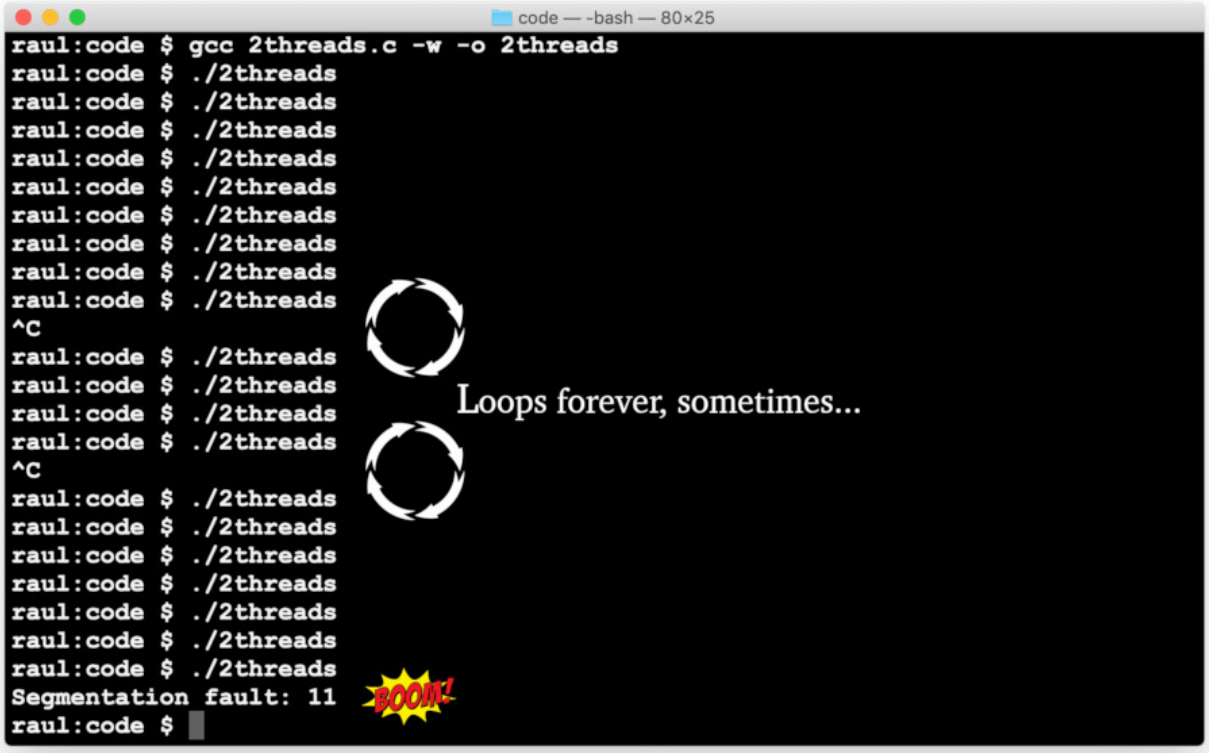
Motivation for (automated) formal verification

```
#include <pthread.h>

void *run(void *);

int a = 0, b = 0, c = 0;
int *pa = &a, *pb = &b, *pc = &c;

int main() {
  pthread_t t1, t2;
  pthread_create(&t1, NULL, run, (void*)pa);
  pthread_create(&t2, NULL, run, (void*)pb);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

```
void *run(void *ptr) {
  while(*pa == *pb) {
    pb = ptr;
    ptr = pa;
    pa = pb;
    pb = pc;
    pc = (int*)*pb;
    pc = ptr;
    (*pc)++;
  }
  return NULL;
}
```

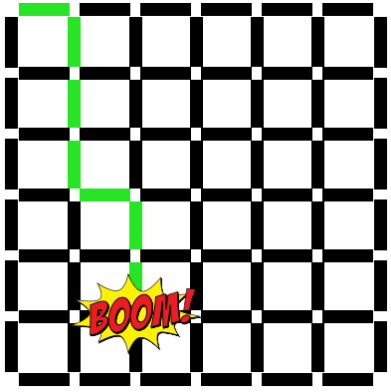
Does the program terminate? Is the program memory-safe?

Often it works, sometimes it fails



Even though there's no input (single initial state $a = b = c = 0$) thread scheduling is non-deterministic.

- Not all paths are equal...



Software complexity

There are two ways of constructing a software design: One way is to make it so simple that there are **obviously no deficiencies**, *and the other way is to make it so complicated that there are* **no obvious deficiencies**. — *Tony Hoare*

Promela Introduction

Raul Barbosa

VALU3S Summer School
Genova, Italia, July 18–20, 2023

The PROMELA language

- PROMELA stands for PROcess MEta LAnguage.
- It is a language for describing concurrent systems.
- PROMELA models (or programs) have multiple processes.
- Typical C-like control structures (`for`, `while`, `if`, etc.).
- All data structures have fixed bound.

PROMELA is not a programming language

It does inherit similar features to a programming language, but...

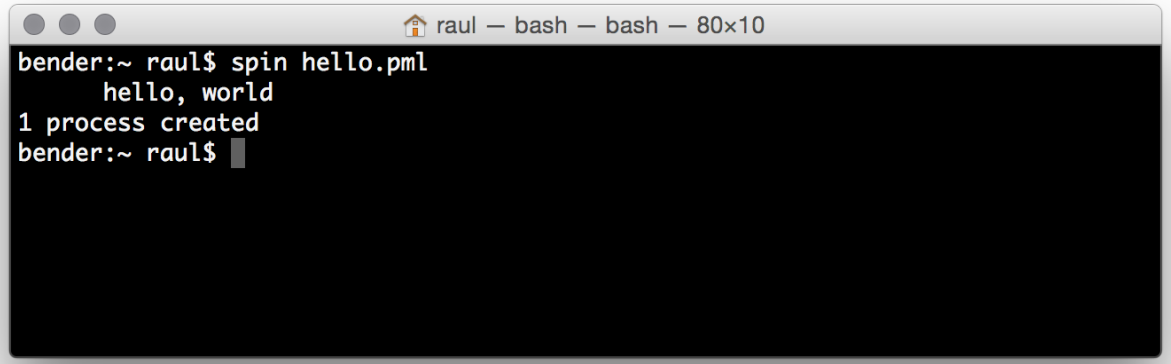
- No memory management or real I/O;
- No floats nor strings (string literals can be printed);
- No graphical user interfaces;
- No external libraries nor methods.

PROMELA is a modelling language

- It is non-deterministic.
- Concurrent processes (shared memory or message passing).
- Nevertheless, we may write simple programs in PROMELA.

A first PROMELA program

```
active proctype H() {
  printf("hello, world\n")
}
```



Remarks

- The `proctype` keyword declares a process.
- Visually similar to C (`printf` is one of many examples).
- SPIN, the Simple Promela INterpreter, computes the program.

Arithmetic and boolean data types

- All variables are implicitly initialized with value zero, by default.
- All local variables are declared at the top of the process block.
- Signed and unsigned integers with operations `+ - * / %`

Type	**Domain**
bool	0,1 (false,true)
byte	0..255
short	$-2^{15}..2^{15}-1$
int	$-2^{31}..2^{31}-1$

Arithmetic data types

```
active proctype Swap() {
  int a = 3, b = 31;
  a = a - b;
  b = a + b;
  a = b - a;
  printf("a = %d, b = %d\n", a, b)
}
```

Control structures

Three fundamental constructs

- Sequence $\rightsquigarrow$ commands separated by ; are executed in order.
- Selection $\rightsquigarrow$ conditions determine which branch gets executed.
- Repetition $\rightsquigarrow$ iterations controlled by some condition.

Control structures: selection

```
active proctype Q() {
  short a = 0, abs;
  if
  :: a >= 0 -> abs = a
  :: a <= 0 -> abs = -a
  fi
}
```

Alternatives are delimited by `if` and `fi`

- Each alternative is signalled by `::` followed by a guard.
- A guard is a condition that allows an alternative to execute.
- Between a guard and its commands, we place a `->`

Control structures: selection

```
active proctype Q() {
  short a = 0, abs;
  if
  :: a >= 0 -> abs = a
  :: a <= 0 -> abs = -a
  fi
}
```

Remarks

- Guards may overlap, leading to non-deterministic choices.
- Any guard which evaluates to `true` may be chosen.
- The keyword `else` is true iff all other guards are false.

Control structures: repetition

```
active proctype R() {
  byte i;
  for(i : 1 .. 10) {
    printf("%d\n", i)
  }
}
```

The `for` statement increments a variable and executes the corresponding block of commands a given number of iterations.

Control structures: repetition

```
active proctype R2() {
  byte i = 1;
  do
  :: i <= 10 -> printf("%d\n", i); i++
  :: else -> break
  od
}
```

Repetition may also consist of alternatives delimited by do and od

- Similarly to the `if` structure, alternatives are identified by `::`
- Iterates until a `break` command is executed.
- If no alternative is true the process blocks (like `if` statements).

Control structures: `for` statements $\Leftrightarrow$ do loops

```
#define N 13

active proctype R3() {
  int i;

  // the DO loop is equivalent to the FOR statement below
  i = 0;
  do
  :: i <= N-1 -> printf("%d\n", i); i++
  :: else -> break
  od;

  // FOR loops are more sensitive to spaces and new lines
  for (i : 0 .. N-1) {
    printf("%d\n", i)
  }
}
```

Arrays

```
#define N 5

byte a[N];
int b[10];
```

PROMELA supports unidimensional arrays (vectors).

- The size of an array is a fixed constant.
- Constants may be declared using the `#define` macro.
- We may use expressions like `b[9] = a[0]` (implicit type cast).

Enumerating elements

```
mtype = {red, green, blue}

active proctype M() {
  mtype color = red;
  if
  :: color != blue -> color = blue
  :: else -> skip
  fi;
  printf("%e\n", color)
}
```

Enumerations are allowed for PROMELA models.

- Symbolic names (easier to read) rather than magic numbers.
- Note the `if` statement, with an expression, an assignment, and an `else` with a `skip` command (does nothing).

Records

User-defined data types may be introduced by the programmer.

- A `typedef` declaration must be global.
- Processes may locally have data items of the new type.
- Typically used for records (e.g., dates) or for an ugly workaround:

```
typedef array {
  byte column[4]
};

active proctype A() {
  array line[5]; /* 5x4 matrix */
  line[1].column[2] = 3
}
```

Inline functions (the only kind supported)

PROMELA supports inline functions to improve code reuse.

- These are "macros" literally replaced wherever the call is made.
- Parameters are passed by reference, and there's no return value.

```
inline maximum(a, b, max) {
  if
  :: a > b -> max = a
  :: else -> max = b
  fi
}
```

This is not equivalent to a typical function, as it behaves mostly like

```
#define odd(x) (x & 1)
```

Deterministic execution

A single process with no overlapping guards is deterministic.

- Variables are always initialized (zero by default);
- There is strictly no user input;
- Each program state has (at most) a single successor state.
- There is one possible computation.

Non-deterministic programs

Overlapping guards

Selection statements in which multiple guards evaluate to true lead to "random" choices.

Process scheduling

Concurrent processes are scheduled "randomly" and instructions become interleaved.

PROMELA programs usually are non-deterministic. This feature allows us to specify behavior which is known to be unknown.

Non-deterministic values

```
int i;

if
:: i = 1
:: i = 2
:: i = 3
fi;
```

- All statements are executable (assignments always are).
- The value of `i` is non-deterministic, between 1 and 3.

Non-deterministic values

```
int i;

i = 1;
do
:: i < 30 -> i++
:: break
od
```

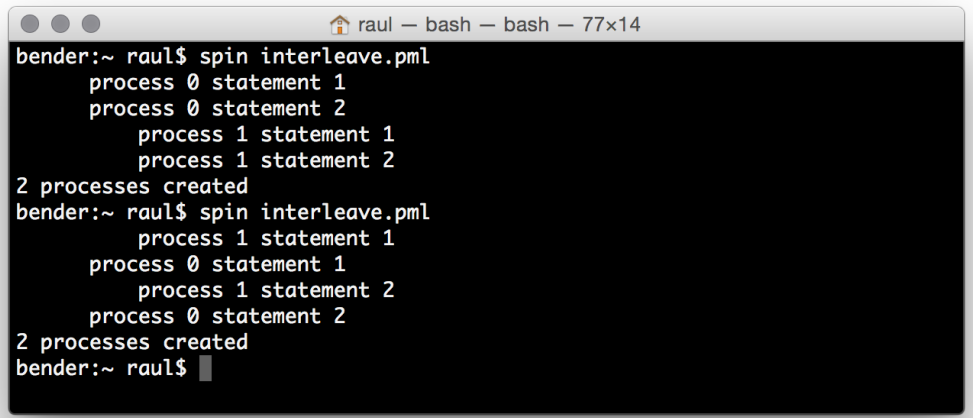
- The first guard is only executable when `i < 30`
- The second alternative is always executable.
- The loop may exit with any value of `i` between 1 and 30.
- Same effect as `select (i : 1..30);`

Non-deterministic scheduling

```
active [2] proctype P() {
  printf("process %d statement 1\n", _pid);
  printf("process %d statement 2\n", _pid);
}
```

The above fragment declares two active processes.

- Several identical processes that differ in the value of `_pid`
- How many possible computations?



Granularity of interleaving

Atomicity:

Every expression or statement is either fully executed or not at all.

- This includes assignments, expressions and jumps.
- Note that guarded statements are not atomic.
  - We may advance past a guard, then another process may be scheduled, before we resume the guarded statement.

Atomic sequences

PROMELA provides an `atomic` keyword.

- It prevents a block of instructions from being interleaved.
- When a statement blocks within an atomic sequence, other processes will execute...
- once control resumes, the sequence continues atomically.
- How many computations are possible in the code below?

```
active [2] proctype P() {
  atomic {
    printf("process %d statement 1\n", _pid);
    printf("process %d statement 2\n", _pid);
  }
}
```

Toward model checking

- System model – a formal description of a system.
- System properties – a formal specification of the system.

Model checking tool

Takes as input a system model and its properties.

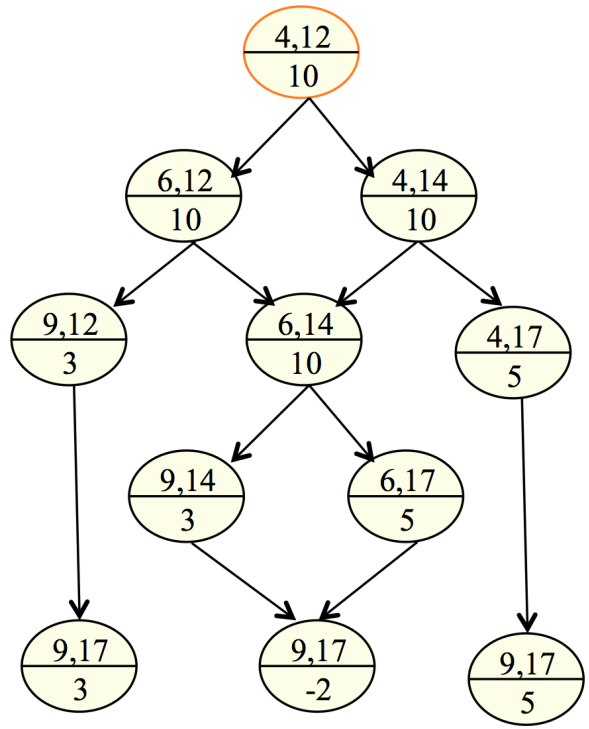
- Either it confirms that the system fulfils the properties.
- Or it returns a counter-example (a computation in which a property is violated).

Interleaving

```
 1  int account = 10;
 2
 3  active proctype W1() {
 4    if
 5    :: account >= 7 ->
 6       account = account-7
 7    :: else -> skip
 8    fi
 9  }
10
11  active proctype W2() {
12    if
13    :: account >= 5 ->
14       account = account-5
15    :: else -> skip
16    fi
17  }
```

Interleaving

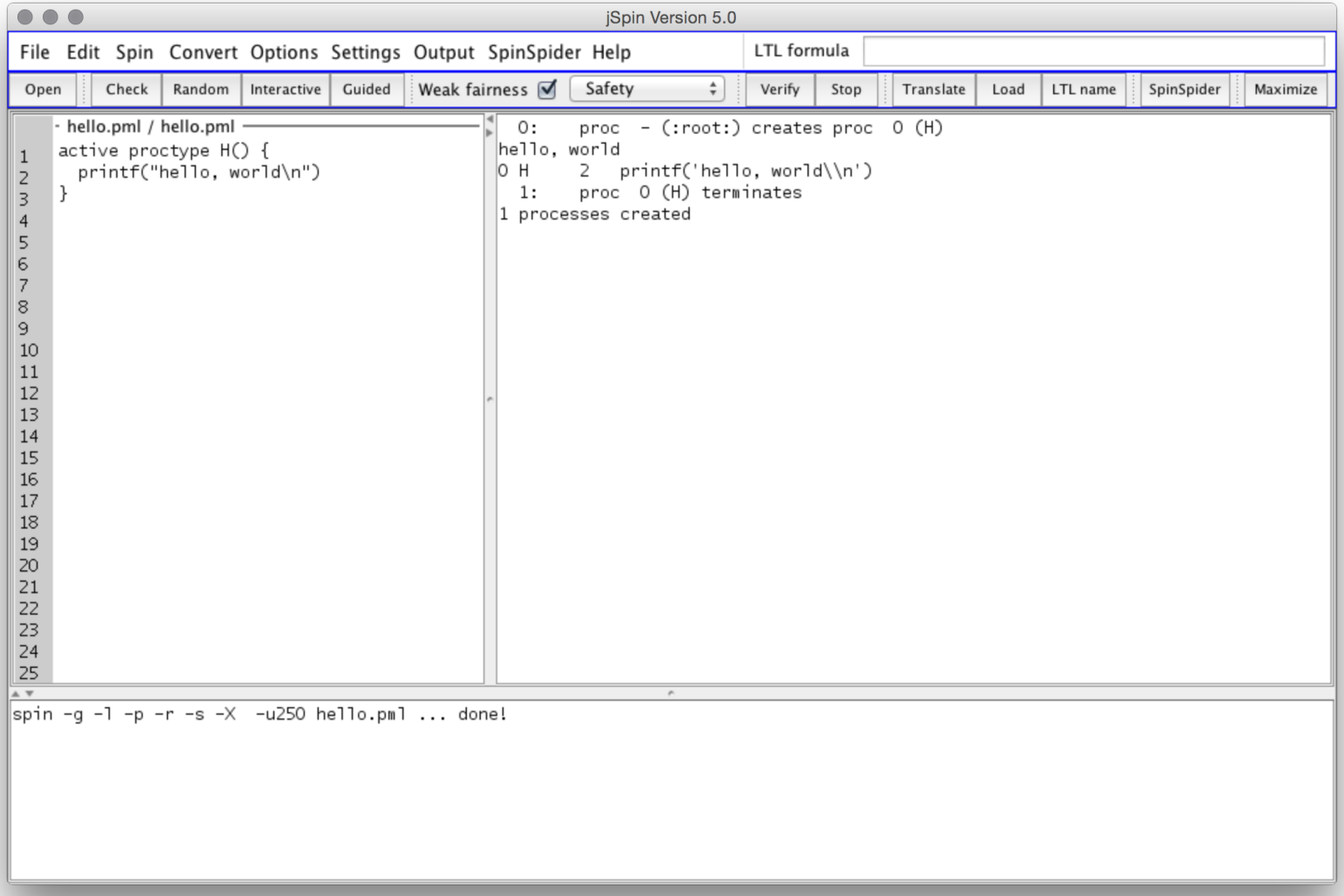
```
 1  int account = 10;
 2
 3  active proctype W1() {
 4    if
 5    :: account >= 7 ->
 6       account = account-7
 7    :: else -> skip
 8    fi
 9  }
10
11  active proctype W2() {
12    if
13    :: account >= 5 ->
14       account = account-5
15    :: else -> skip
16    fi
17  }
```



Directed graph of reachable states (program counters and variables).

Before next class...

Run the "hello world" example using jSpin.



Documentation supporting this lecture

- R. Barbosa, Verificação de modelos, Ch. 1
- Promela language reference
  http://spinroot.com/spin/Man/promela.html
- Downloading and installing Spin
  http://spinroot.com/spin/Man/README.html
- jSpin GUI for Spin
  http://code.google.com/p/jspin/
- jSpin user's guide
  docs/jspin-user.pdf distributed with the jSpin GUI

Supplemental bibliography

- Mordechai Ben-Ari, "Principles of the Spin Model Checker", ISBN 1846287693, Springer, 2008.
- Gerard Holzmann, The SPIN Model Checker: Primer and Reference Manual, ISBN 0321773713, Addison-Wesley, 2003.

Model Checking using Spin

Raul Barbosa

VALU3S Summer School
Genova, Italia, July 18–20, 2023

Formal verification

Program testing can be used to show the presence of bugs, but never to show their absence. — Edsger W. Dijkstra

Model checking...

...is a process for verifying if a model satisfies a given specification.

- Model and specification are formulated in a precise language.
- Model checking is appropriate for finite-state systems.
- The model is verified exhaustively and automatically.

ACM TURING AWARD HONORS FOUNDERS OF AUTOMATIC VERIFICATION TECHNOLOGY

Researchers Created Model Checking Technique for Hardware and Software Designers

NEW YORK, February 4, 2008 – ACM, the Association for Computing Machinery, has named Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis the winners of the 2007 A.M. Turing Award, widely considered the most prestigious award in computing, for their original and continuing research in a quality assurance process known as Model Checking. Their innovations transformed this approach from a theoretical technique to a highly effective verification technology that enables computer hardware and software engineers to find errors efficiently in complex system designs. This transformation has resulted in increased assurance that the systems perform as intended by the designers. The Turing Award, named for British mathematician Alan M. Turing, carries a $250,000 prize, with financial support provided by Intel Corporation and Google Inc. Clarke of Carnegie Mellon University, and Emerson of the University of Texas at Austin, working together, and Sifakis, working independently for the Centre National de la Recherche Scientifique at the University of Grenoble in France, developed this fully automated approach that is now the most widely used verification method in the hardware and software industries.

Model checking

- A model checker attempts to find computations in which at least one property is proven to be false.
- Such a computation would be a counter-example (a path from the initial state to an invalid state).
- The search is exhaustive, by visiting all reachable states.

Possible outcomes after verification

1. One or more counter-examples found.
2. Graph exhaustively searched and no counter-example found.

Model checking

- A model checker attempts to find computations in which at least one property is proven to be false.
- Such a computation would be a counter-example (a path from the initial state to an invalid state).
- The search is exhaustive, by visiting all reachable states.

Possible outcomes after verification

1. One or more counter-examples found.
2. Graph exhaustively searched and no counter-example found.

Exhaustive search without a counter-example proves correctness.

Implications of the word “exhaustive”

Exhaustive search $\Rightarrow$
Exploring all possibilities when there is non-determinism.

Implications of the word "exhaustive"

Exhaustive search $\Rightarrow$
Exploring all possibilities when there is non-determinism.

Explicit non-determinism

Selection statements in which multiple guards evaluate to true lead to non-deterministic choices.

Implications of the word "exhaustive"

Exhaustive search $\Rightarrow$
Exploring all possibilities when there is non-determinism.

Explicit non-determinism

Selection statements in which multiple guards evaluate to true lead to non-deterministic choices.

Implicit non-determinism

Concurrent processes are scheduled non-deterministically and instructions become interleaved.

The SPIN model checker

SPIN stands for Simple Promela INterpreter

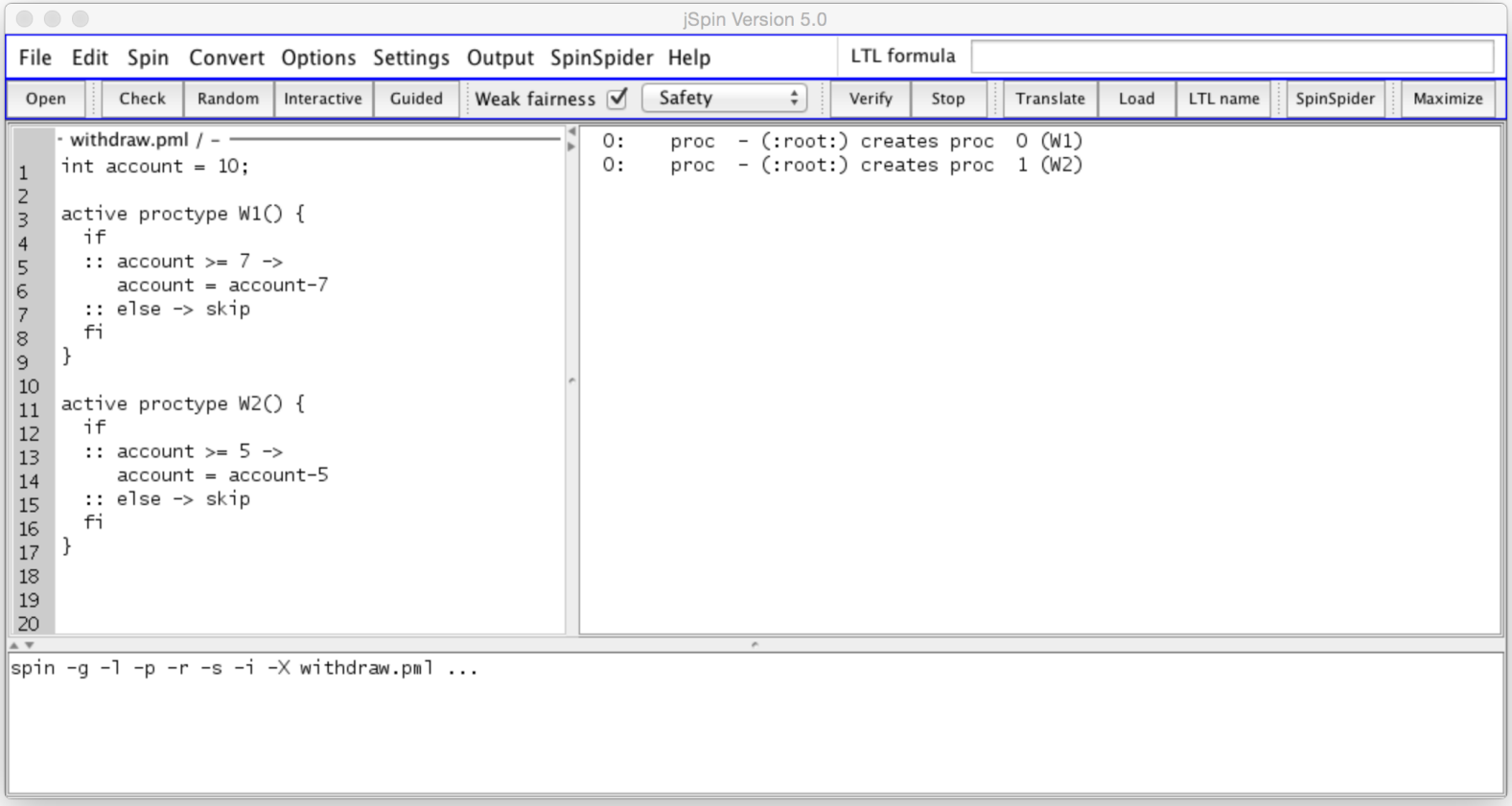
- Previously, SPIN was used to simulate PROMELA programs.
- Today, SPIN generates a verifier (it's not a simple interpreter).

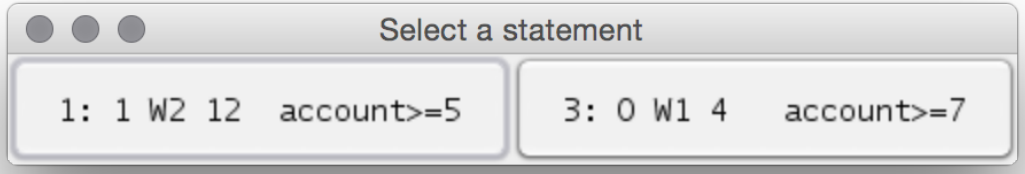
SPIN generates a verifier for each model

The verifier is a C program that checks the model exhaustively.

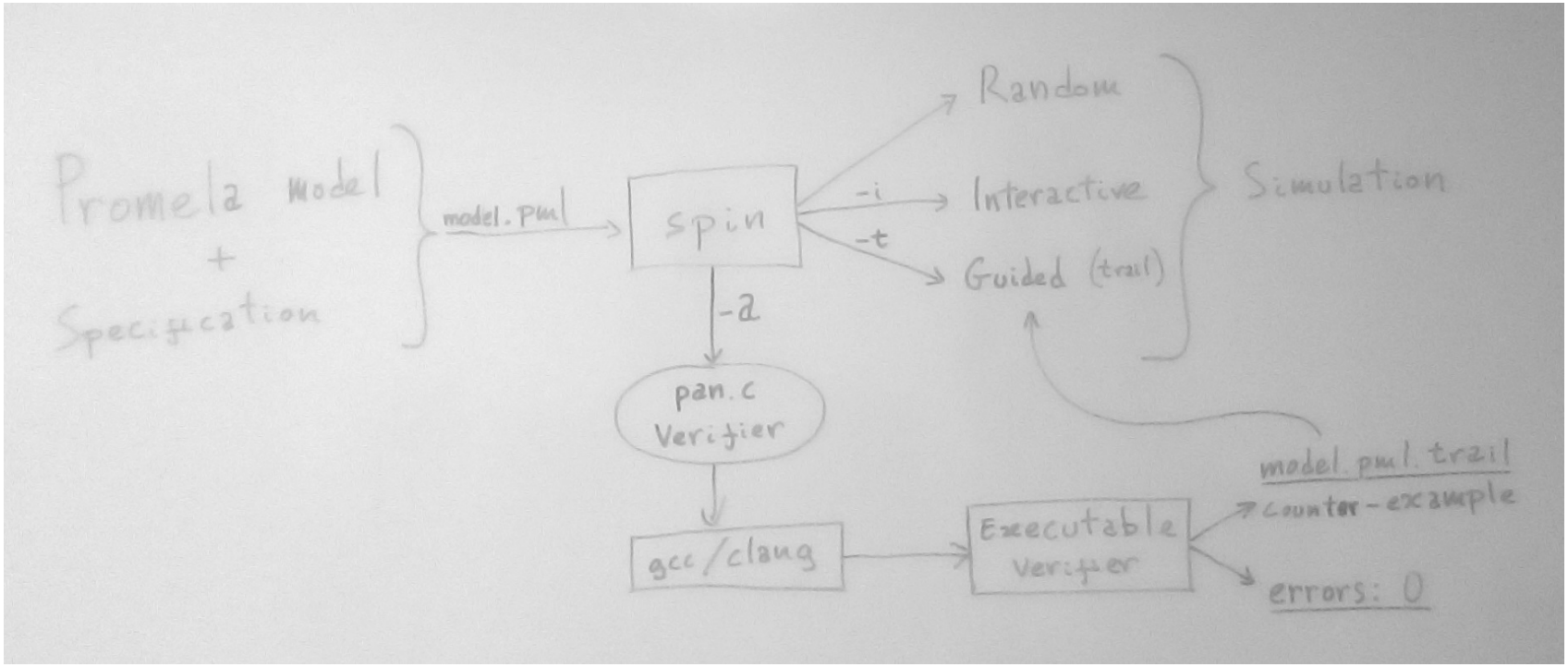
- If it finds an error, it saves a `.trail` file with the path.
- Otherwise, it returns `errors: 0`

Simulation: interactive // random // guided





Workflow: simulate (random), verify, simulate (guided)



Specifying correctness properties

Properties may be stated within or outside a PROMELA model.

Properties specified within a model

- assertions
- end labels
- accept labels
- progress labels

Properties specified outside a model

- temporal logic formulas
- never claims (from LTL formulas)

Assertion statements

An assertion is a command in the form `assert(exp)`

- `exp` is a boolean expression.
- Assertions may be placed anywhere a statement would be.

```
active proctype Swap() {
  int a = 3, b = 31;
  a = a - b;
  b = a + b;
  a = b - a;
  assert(a == 31 && b == 3)
}
```

Effects of assertions

`assert(exp);`

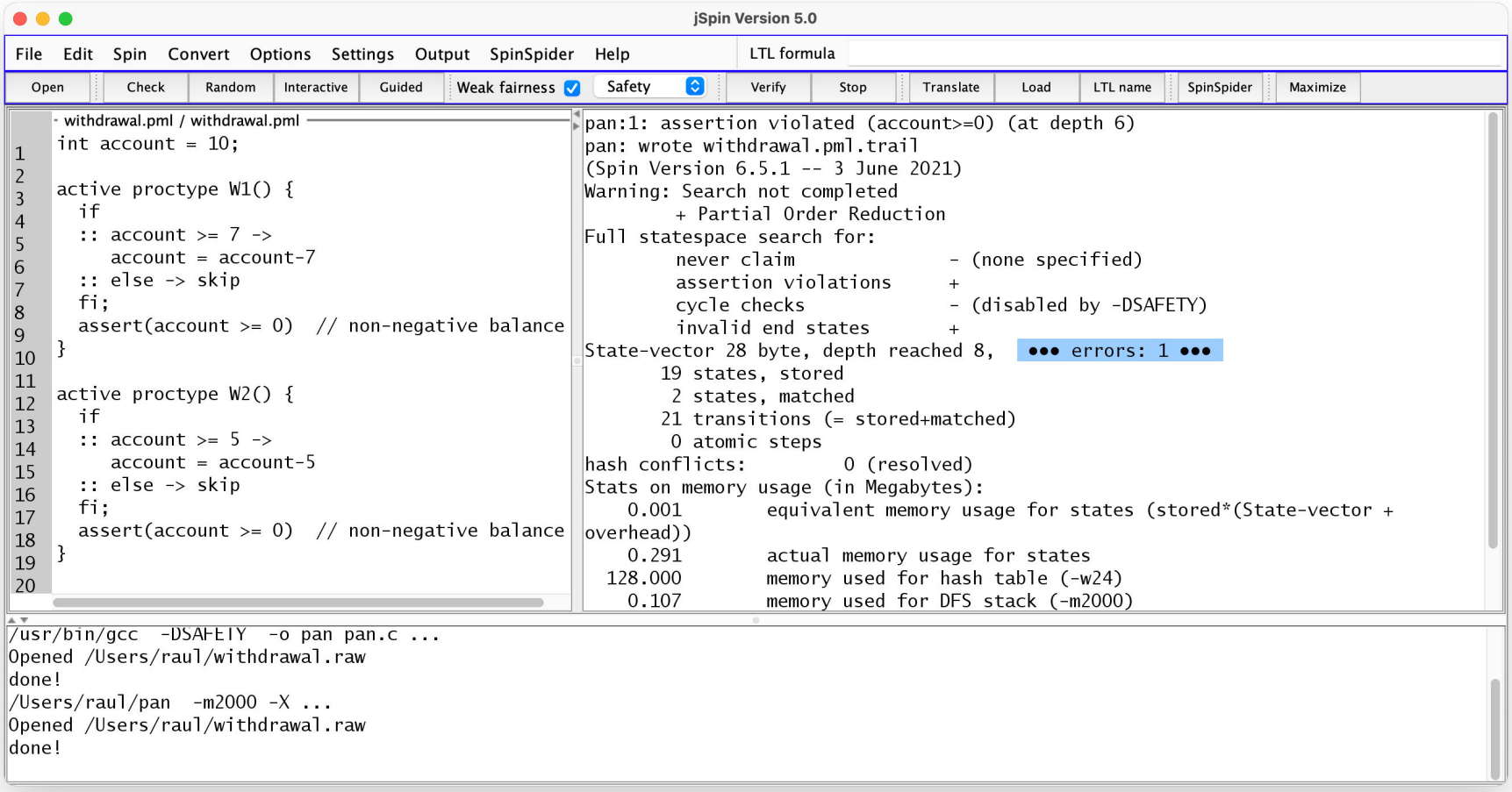
- If `exp` is true, the assertion has no effect.
- If `exp` is false, execution halts and an error message shows.
- Generally, `exp != 0` means true, and `exp == 0` means false.

This is the behavior of `assert(exp)` both in simulation and verification modes.

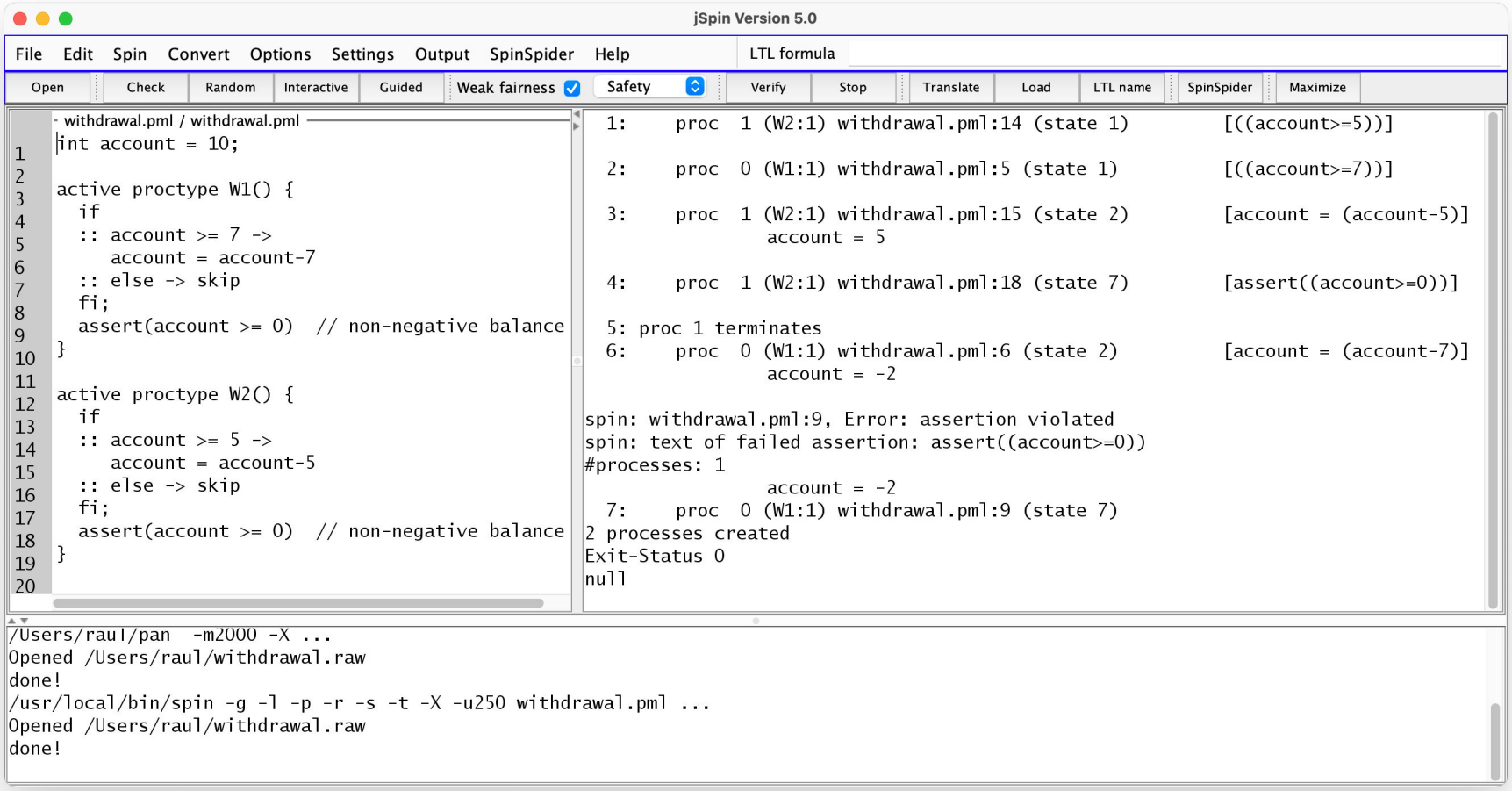
Examples

- Demo showing `abs(x)` or `min(a,b)` with an assertion
- Step by step: generate, compile, run, `errors: 0`
- Withdrawal example, same steps, `errors: 1`
- Use `spin -t` (simulation guided by a `.trail` file)

Coming back to our "withdrawal" example



Guided simulation shows the trail to an error



This error trail is present in the state space graph seen earlier in the lecture series.

What would it take to perform full verification?

```
// select a and b non-deterministically
a = a - b;
b = a + b;
a = b - a;
// appropriate assertion (maybe using temporary variables)
```

Most common commands

- Random simulation

  ```
  spin file.pml
  ```

- Interactive simulation

  ```
  spin -i file.pml
  ```

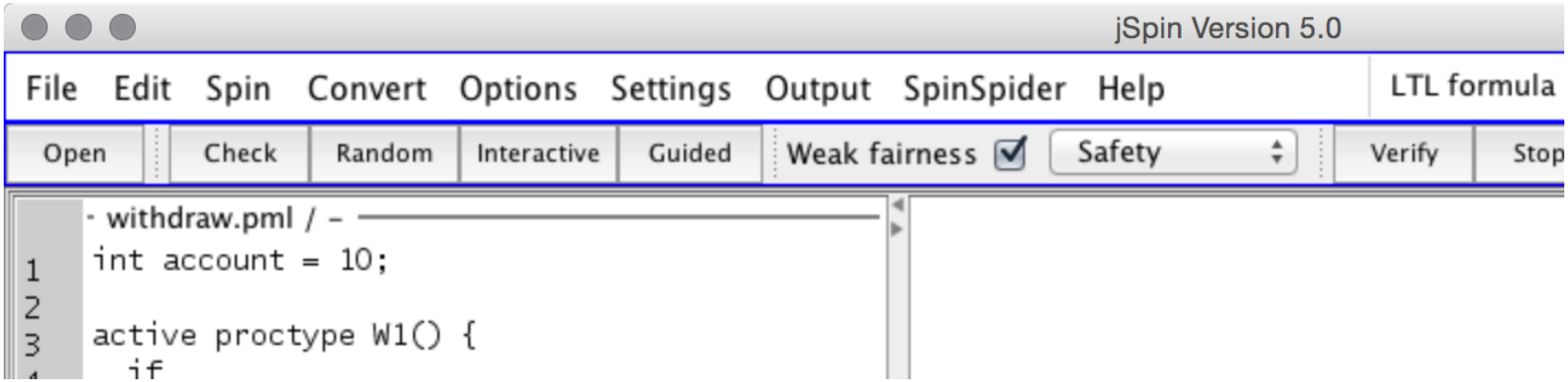
- Full verification

  ```
  spin -a file.pml
  cc pan.c -o pan
  ./pan or pan.exe
  ```

- Guided simulation (with `file.pml.trail`)

  ```
  spin -t -p file.pml
  ```

The jSpin IDE



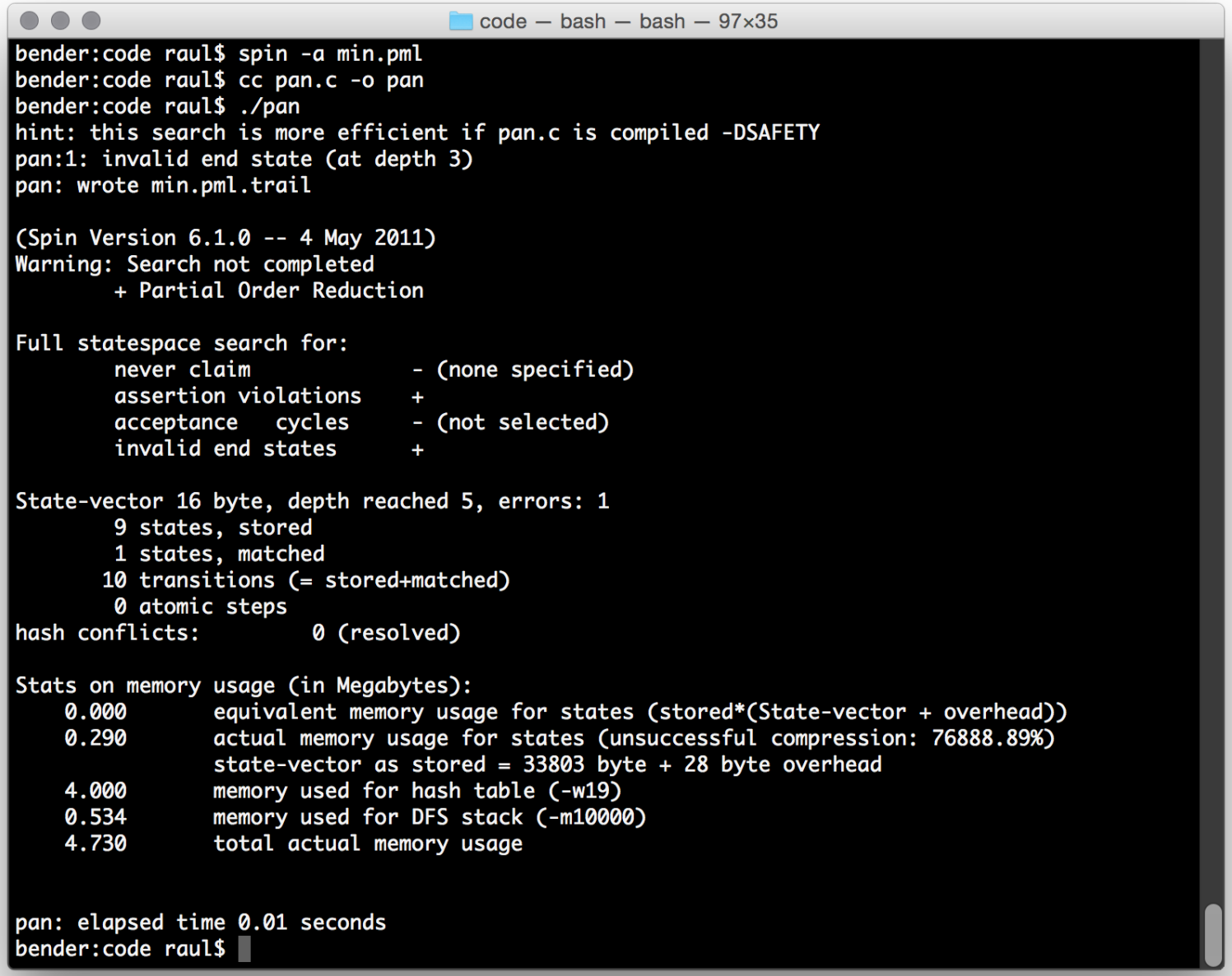
- jSpin is a GUI for SPIN.
- Just like a coding IDE, it automates many important tasks.
- It is written in Java, and is adequate for students.
- There is also a far more advanced IDE called ispin.

What about non-terminating processes?

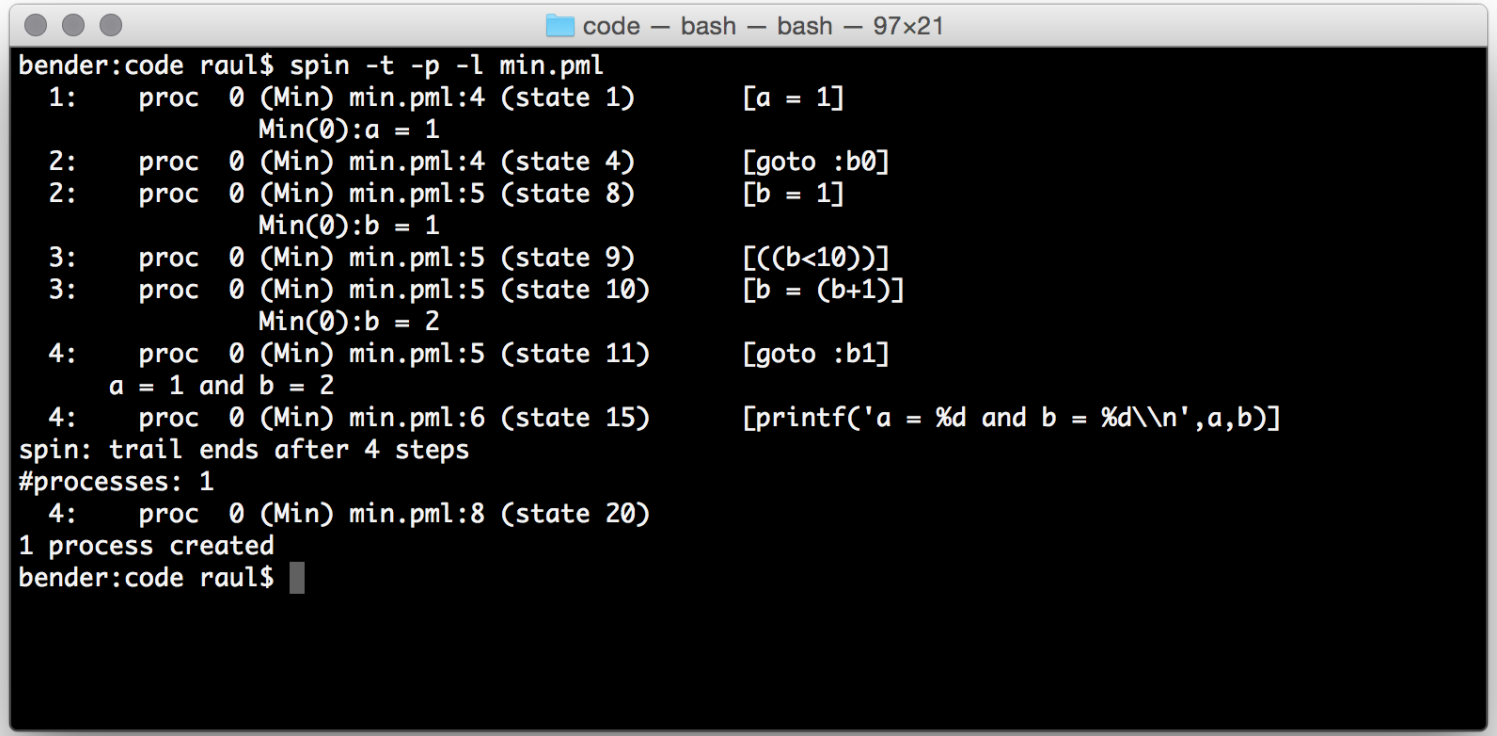
```
active proctype Min() {
  byte a, b, min;

  select (a : 1 .. 10);
  select (b : 1 .. 10);
  printf("a = %d and b = %d\n", a, b);

  if
  :: a >= b -> min = b
  :: b <= a -> min = a
  fi;
  printf("min = %d\n", min)
}
```

Simulate it a few times. Getting a strange timeout sometimes?

Inspecting the output of pan...



Guided simulation shows where it halts



Blocking

- A process blocks when no statements are executable.
- It is an error for all processes to block simultaneously.
- Some process must be able to execute.

"healthy" blocking

- A process blocks to let others execute in a concurrent system.
- Necessary very often for correctness (e.g., mutual exclusion).

"unhealthy" blocking

- One process blocks and all other processes are blocked.
- It is likely a sign of undesired deadlock.

Coming back to our example...

```
active proctype Min() {
  byte a, b, min;

  select (a : 1 .. 10);
  select (b : 1 .. 10);
  printf("a = %d and b = %d\n", a, b);

  if
  :: a >= b -> min = b
  :: b <= a -> min = a
  fi;
  printf("min = %d\n", min)
}
```

- There is an execution that ends with a blocked process.
- The only existing process is blocked at line 8.
- We know what it takes for an `if` statement to block...

Valid end states

We get "invalid end state", because not all end states are valid.

All executions must terminate with the program counters of all processes at an appropriate end location.

Valid end locations for a process are:

- The closing bracket of the process;
- Any statements labelled with an `end` label.

endxyz: statement

An end-state label marks a location as a valid termination point.

How may we get a "valid end state" even with a timeout?

```
byte ball = 1;

active proctype Player1() {
  do
  :: ball == 1 ->
     printf("ping\n");
     ball = 2
  :: true ->
     printf("thanks for the game\n");
     break
  od
}

active proctype Player2() {
  do
  :: ball == 2 ->
     printf("pong\n");
     ball = 1
  od
}
```

A frequent bug

Sometimes users forget to add an `else` guard...

It is now clear why such a lapse might lead to `timeout` and invalid end states. The code below is likely missing an `else` alternative.

```
if
:: x < y -> x = y
fi
```

Documentation supporting this lecture

- R. Barbosa, Verificação de modelos, Ch. 2
- Spin reference card
  docs/spin-card.pdf distributed with the jSpin GUI
- Pan – the verifier generated by Spin
  http://spinroot.com/spin/Man/Pan.html
- The `assert` statement
  http://spinroot.com/spin/Man/assert.html
- The `end` label prefix
  http://spinroot.com/spin/Man/end.html

Supplemental bibliography

- Mordechai Ben-Ari, "Principles of the Spin Model Checker", ISBN 1846287693, Springer, 2008.
- Gerard Holzmann, The SPIN Model Checker: Primer and Reference Manual, ISBN 0321773713, Addison-Wesley, 2003.

Verifying Concurrent Programs

Raul Barbosa

VALU3S Summer School
Genova, Italia, July 18–20, 2023

Modeling

All models are wrong, but some are useful. — George E. P. Box

Concurrent systems...

Concurrency

Multiple processes executing simultaneously, sharing resources.

- Coordination needed to access those resources.
- Need to avoid deadlocks, starvation, race conditions, etc.

- To avoid unsafe situations, we use mechanisms...
  ...that may create other problems (semaphores $\rightarrow$ deadlocks).

Spin

Model checking using SPIN:

- The main target is precisely concurrent and distributed systems.
- Our focus during this lecture is to model and analyze concurrent systems.

Concurrent systems are hard to verify

- State spread over multiple components (processes, OS, drivers).
- Combinatorial explosion of possible computations.
- Scheduling/interleaving may create tricky problems.
- Difficult to reproduce a execution to find an error.
- Debug information is usually partial.

Model checking

A model checker like SPIN helps to design systems:

- **before** formalizing a design imposes useful reasoning
- **during** exposes defects in order to improve systems
- **after** implementation takes advantage of correct design

Challenges

Modeling (system)

- Must express all relevant details of the original system.
- Must be simple enough to allow exhaustive verification.
- Aiming for the "smallest sufficient model".

Specifying (correctness)

- Specification must be complete (avoid underspecification).
- It must also be unambiguous.

Concurrency in Promela

In Promela, concurrency is the norm.

- The unit of execution is a process (`proctype`).
- The `active` keyword declares a process which is ready-to-run in the initial state.

```
active proctype S() {
  printf("Process %d started.\n", _pid)
}
```

Process initialization

we may declare active processes

```
active [2] proctype S() {
  printf("Process %d started.\n", _pid)
}
```

or start processes during the execution

```
proctype S() {
  printf("Process %d started.\n", _pid)
}

init {
  run S();
  run S()
}
```

the two programs are slightly different (note `_pid` is reusable)

Atomic initialization

In most cases the `init` process starts processes atomically.

- The `init` process performs some global initialization;
- and then starts the system (all processes).

```
proctype S() {
  printf("Process %d started.\n", _pid)
}

init {
  // perform some initialization
  atomic {
    run S();
    run S()
  }
}
```

Processes may receive parameters

```
proctype Parity(short number) {
  if
  :: number & 1 -> printf(" odd ")
  :: else       -> printf(" even ")
  fi
}

init {
  short c;
  if
  :: c = 0
  :: c = 1
  fi;
  run Parity(c)
}
```

Recursive programs are possible...

```
proctype Parenthesis(byte n) {
  printf("[");
  if
  :: n > 0 -> run Parenthesis(n-1)
  :: else  -> skip
  fi;
  printf("]")
}

init {
  run Parenthesis(5)
}
```

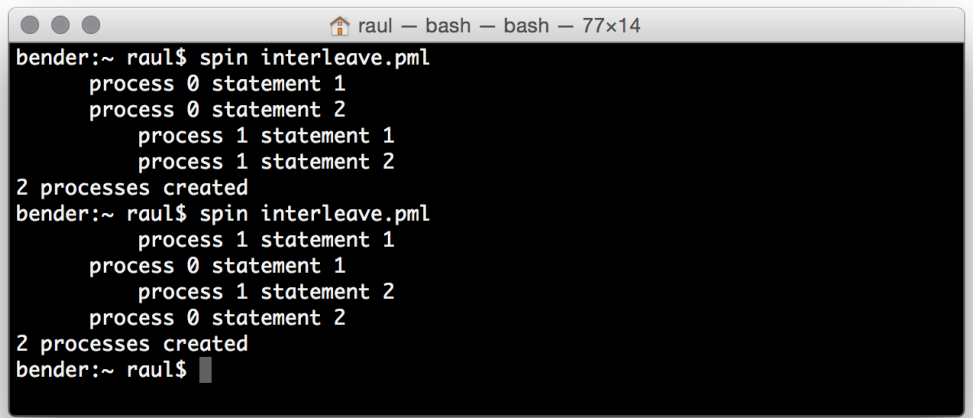
... but the result might be not be what one would expect.
Simulate using `spin -T parenthesis.pml` a few times.

Non-deterministic scheduling

```
active [2] proctype P() {
  printf("process %d statement 1\n", _pid);
  printf("process %d statement 2\n", _pid);
}
```

The above fragment declares two active processes.

- Several identical processes that differ in the value of `_pid`
- How many possible computations?



Global and local variables

```
int account;

proctype Withdraw(int amount) {
  int available = account;
  if
  :: available >= amount ->
     available = available - amount
  :: else -> skip
  fi;
  account = available;
  printf("final value = %d\n", account)
}

init {
  account = 10;
  run Withdraw(7);
  run Withdraw(5)
}
```

Global variables

Global variables are useful for modeling

- Shared memory.
- System state, semaphores, monitors.
- Message passing. A shared queue can represent a socket between two processes (datatype `chan` is better).

Interference when accessing a shared variable

```
byte b;

proctype T() {
  b = _pid
}

init {
  run T();
  run T();
  printf("b = %d\n", b)
}
```

How many possible results?

Process synchronization

- The cornerstone of process synchronization is the executability of statements.
- Semaphores, test and set, monitors can be implemented with executability.
- A statement may or may not be executable, in a given state.

Statement executability

- Assignments are always executable;
- Assertions are always executable;
- `printf` statements are always executable;
- Expressions are only executable when and only when evaluated to true;
  - Examples: `(false);` `(x == 0);` `(b);`
  - Usually surrounded by `(...)` for emphasis.

Statement executability

- Atomic sequences are executable if and only if the first statement is executable;
- `if` and `do` statements are executable if any alternative is executable;
- An alternative is executable if the guard is executable.

Blocking

- If a statement is not executable, then it is blocked.
- If a process is at a location with a blocked statement, it blocks.
- At any point in time, the "scheduler" chooses non-deterministically one process to execute which is not blocked.

Joining processes with a blocking statement

```
byte b;

proctype T() {
  b = _pid
}

init {
  run T();
  run T();
  (_nr_pr == 1);
  printf("b = %d\n", b)
}
```

Critical section problem

The critical section problem requires

- Mutual exclusion – there are never two processes executing the critical section at the same time.
- Absence of deadlock – there should be no situation in which two or more processes become permanently blocked.
- Absence of starvation – when one process wishes to enter the critical section, it eventually does so.

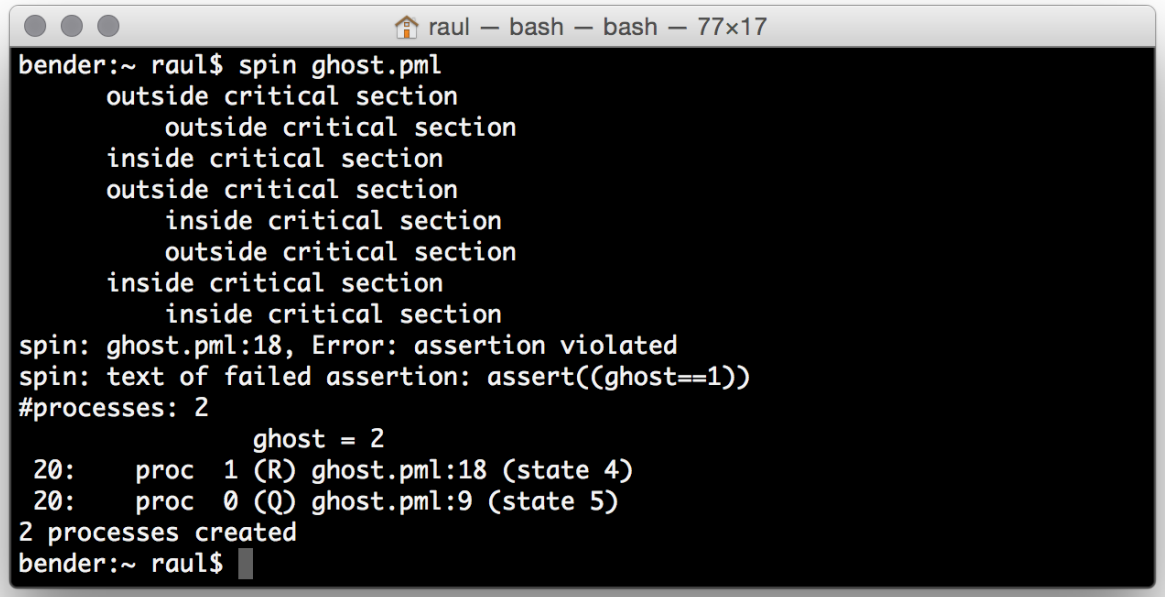
Mutual exclusion in the withdrawal example

- We may attempt to guarantee access to the critical section in numerous ways.
- One of them is to use `atomic` sequences.
- May we design and verify an algorithm using a monitor?
- What do invalid end states mean?
- How do we specify our properties?

Using ghost variables

```
byte ghost = 0;

active proctype Q() {
  do
  :: printf("outside critical section\n");
     ghost++;
     printf("inside critical section\n");
     assert(ghost == 1);
     ghost--
  od
}

active proctype R() {
  do
  :: printf("outside critical section\n");
     ghost++;
     printf("inside critical section\n");
     assert(ghost == 1);
     ghost--
  od
}
```

Using ghost variables



A naive solution that guarantees mutual exclusion

```
bool turn;
byte mutex = 0;

active [2] proctype P()
{
  if
  :: skip
  :: true -> end: (false)     /* may end blocked forever */
  fi;
non_cs:
  turn = 1 - _pid;
  (turn == _pid);
cs:
  mutex++;                    /* critical section */
  assert(mutex == 1);
  mutex--;
exit:
  goto non_cs
}
```

A naive solution that guarantees mutual exclusion

- The previous solution does guarantee mutual exclusion.
- However, it does not guarantee absence of deadlock nor absence of starvation.
- Invalid end states usually point to deadlocks (so they are a form of specification).
- We will need Linear Temporal Logic (LTL) for specifying more sophisticated properties.

Documentation supporting this lecture

- R. Barbosa, Verificação de modelos, Ch. 3
- The `init` process
  http://spinroot.com/spin/Man/init.html
- Creating new processes with the `run` statement
  http://spinroot.com/spin/Man/run.html
- The predefined local variable `_pid`
  http://spinroot.com/spin/Man/_pid.html
- The predefined global variable `_nr_pr`
  http://spinroot.com/spin/Man/_nr_pr.html

Supplemental bibliography

- Mordechai Ben-Ari, "Principles of the Spin Model Checker", ISBN 1846287693, Springer, 2008.
- Gerard Holzmann, The SPIN Model Checker: Primer and Reference Manual, ISBN 0321773713, Addison-Wesley, 2003.

Specification and verification with temporal logic

Raul Barbosa

VALU3S Summer School
Genova, Italia, July 18–20, 2023

Beyond assertions

Linear Temporal Logic (LTL) properties are globally specified:

```
ltl [optional_property_name] { formula }
```

LTL formulæ allow to specify global correctness requirements.

- Multiple inline LTL formulæ are allowed (selected during verification with the `-N` argument)
- An LTL property may include any propositional state formula (*e.g.*, , $a < b$)
- The model must satisfy the property exactly as specified

Motivation

Temporal logic qualifies propositions with respect to time.

- A proposition p such as *"the sun is shining"* refers only to the present state
- Often we need to refer to time:
  - "the sun is always shining"
  - "the sun will eventually shine"
  - "the sun is shining until it rains"

Propositions are valid LTL formulas

Operator	Symbol	Text mode notation
true	$\top$	`true`
false	$\bot$	`false`
negation	$\neg$	`!`
conjunction	$\wedge$	`&&`
disjunction	$\vee$	`||`
implication	$\rightarrow$	`->`
equivalence	$\leftrightarrow$	`<->`

Table 1: Operators from propositional calculus.

Linear temporal logic operators

$\Box\phi$ means "always ϕ". If ϕ is an LTL formula, then $\Box\phi$ means that ϕ is always true, globally, in all states.

$\Diamond\phi$ means "eventually ϕ". Formula $\Diamond\phi$ indicates that ϕ will necessarily become true in some future state of the computation.

$\phi\mathcal{U}\psi$ means "ϕ until ψ". If ϕ and ψ are LTL formulas, then $\phi\mathcal{U}\psi$ specifies that ϕ holds true at least until ψ is true, and it is guaranteed that ψ will eventually hold.

$\phi\mathcal{W}\psi$ means "ϕ unless ψ". If ϕ and ψ are LTL formulas, then $\phi\mathcal{W}\psi$ indicates that ϕ must hold at least until ψ is true, but ψ does not need to hold. The $\mathcal{W}$ operator is also called "weak until": $\phi\mathcal{W}\psi \leftrightarrow \phi\mathcal{U}\psi \vee \Box\phi$.

Frequently used properties

$\Box p$ A safety property specifying that p is **invariant**. Often negated: $\Box \neg q$ (bad things never happen).

$\Diamond p$ A liveness property: p is **guaranteed** at some point.

$\Box \Diamond p$ Specifies **recurrent** behaviour because, at any point, p must eventually (in a finite number of steps) become *true* again. Spin accepts both `ltl { []<>p }` and spelled out as `ltl { always eventually p }`

$\Diamond \Box p$ Eventually always p specifies **stable** behaviour because p shall become invariantly *true* after some point.

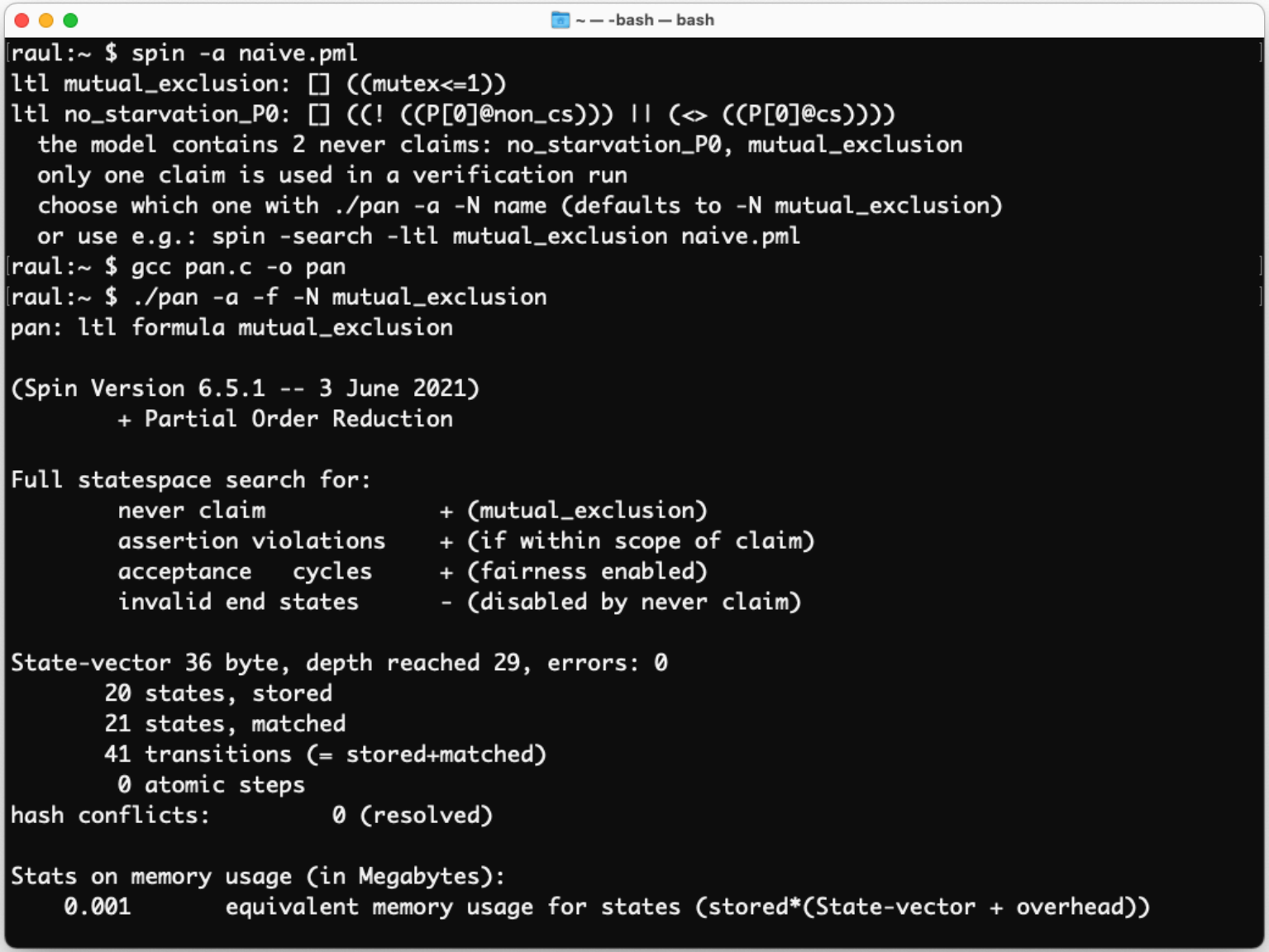
$\Diamond p \rightarrow \Diamond q$ Specifies that there is a logical **correlation** (eventually p implies eventually q).

Does the property $p \rightarrow q$ mean anything similar to $\Box(p \rightarrow \Diamond q)$?

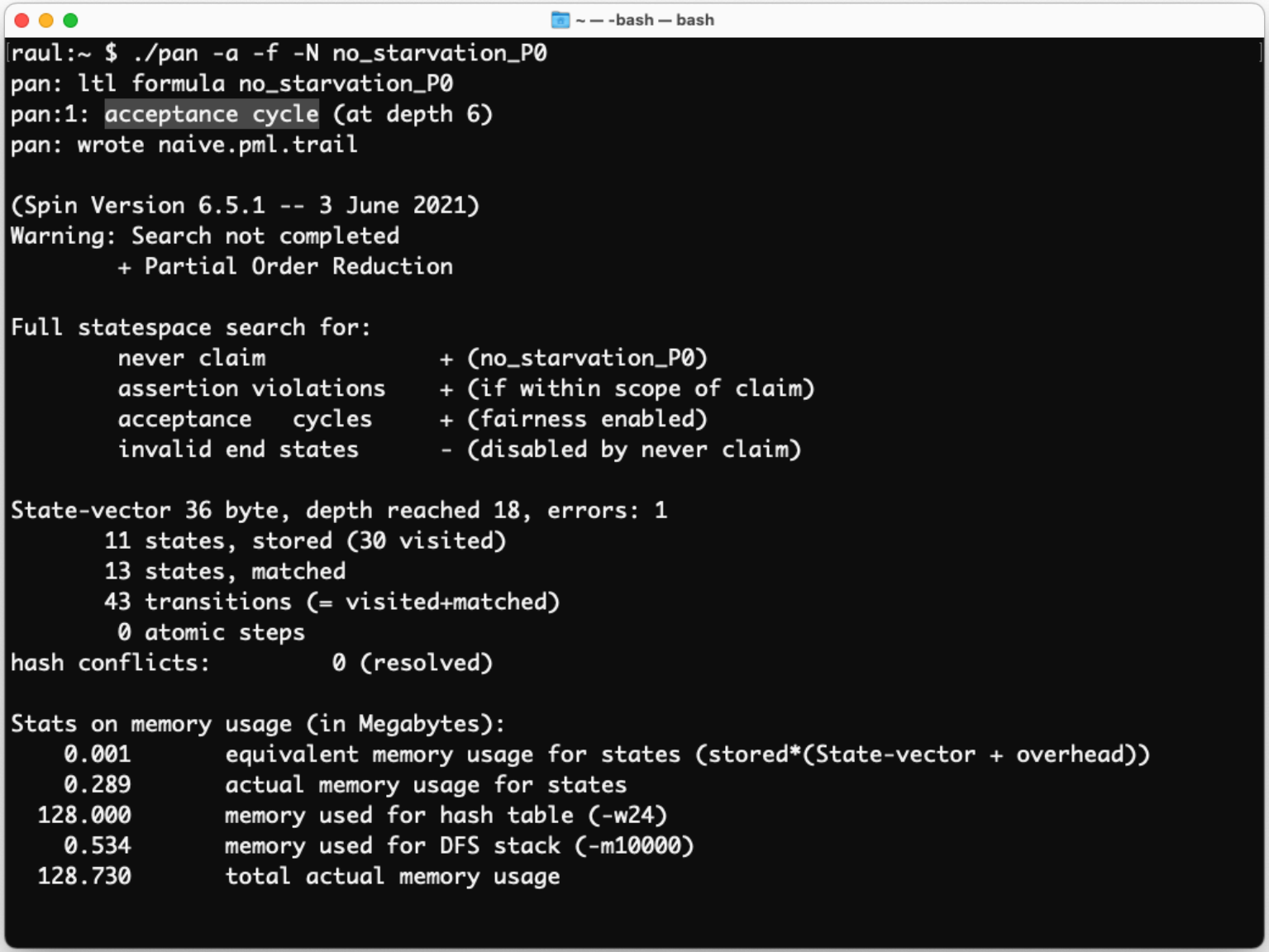
Two LTL formulas specifying behaviour of processes

```
bool turn;
byte mutex = 0;

ltl mutual_exclusion { [](mutex <= 1) }
ltl no_starvation_P0 { []((P[0]@non_cs) -> <>(P[0]@cs)) }

active [2] proctype P()
{
non_cs:
  turn = 1 - _pid;
  if
  :: (turn == _pid) -> skip
  :: else -> goto non_cs
  fi;
cs:
  mutex++;                    /* critical section */
  mutex--;
exit:
  goto non_cs
}
```

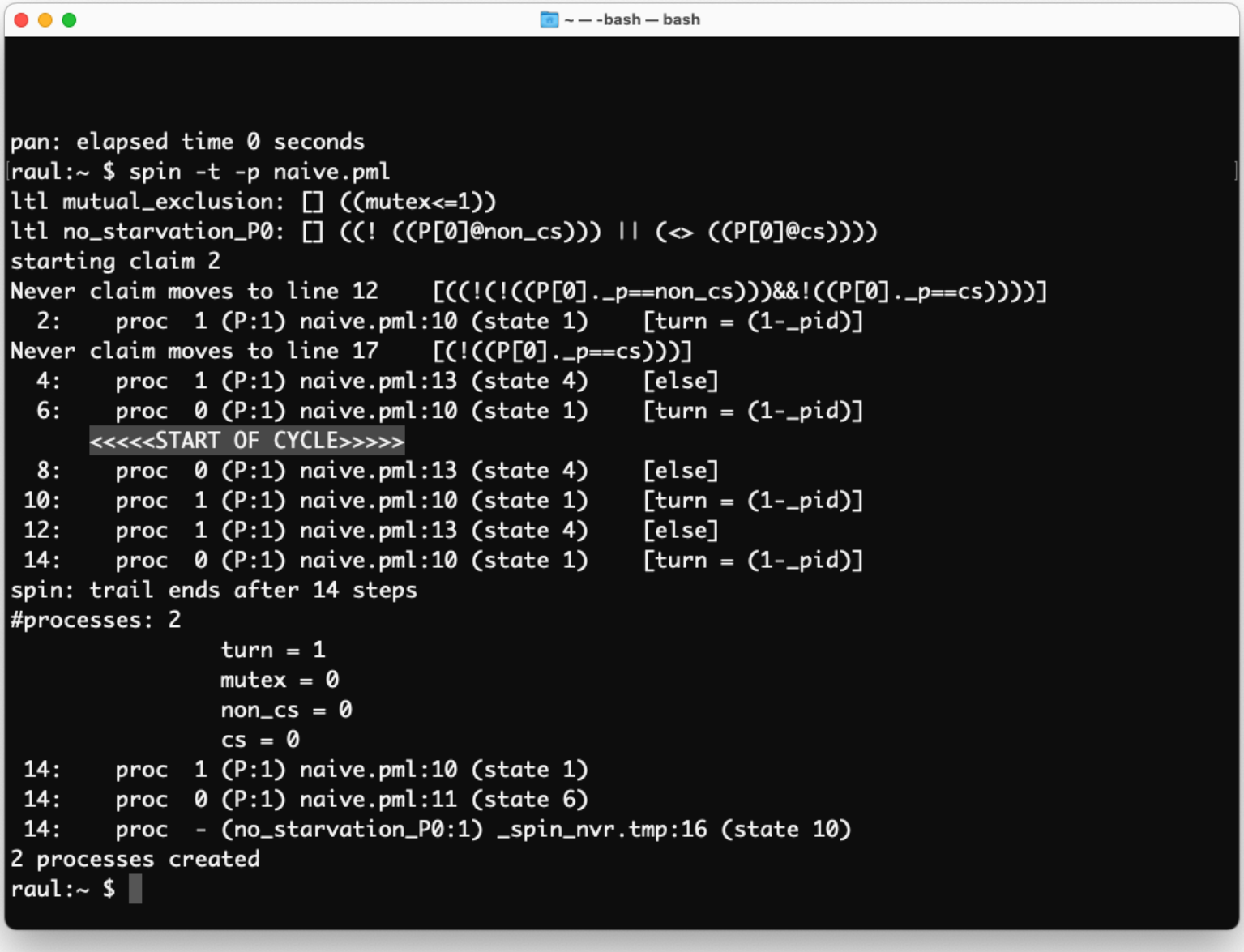
Mutual exclusion holds



Absence of starvation does not hold



A cycle is found violating the specified liveness property



Cat and mouse game on an square grid

```
#define n 3

ltl p1 { []!(mouse_x==cat_x && mouse_y==cat_y) }
ltl p2 { <>(mouse_x==cat_x && mouse_y==cat_y) }
ltl p3 { <>(cat_x>=1 || cat_y>=1) }
ltl p4 { (cat_x!=n) U (mouse_x==n) }
ltl p5 { (cat_x!=n) W (mouse_x==n) }

byte mouse_x = 1, mouse_y = 1, cat_x = 0, cat_y = 0;

active proctype Mouse() {
    do
    :: mouse_x < n -> mouse_x++
    :: mouse_x > 0 -> mouse_x--
    :: mouse_y < n -> mouse_y++
    :: mouse_y > 0 -> mouse_y--
    od
}
active proctype Cat() {
    do
    :: cat_x < mouse_x -> cat_x++
    :: cat_x > mouse_x -> cat_x--
    :: cat_y < mouse_y -> cat_y++
    :: cat_y > mouse_y -> cat_y--
    od
}
```

Cat and mouse: meaning and validity of each formula?

```
ltl p1 { []!(mouse_x==cat_x && mouse_y==cat_y) }
ltl p2 { <>(mouse_x==cat_x && mouse_y==cat_y) }
ltl p3 { <>(cat_x>=1 || cat_y>=1) }
ltl p4 { (cat_x!=n) U (mouse_x==n) }
ltl p5 { (cat_x!=n) W (mouse_x==n) }
```

Spin can prove that the only property that holds is p5...

```
spin -a cat_mouse.pml
cc pan.c -o pan
./pan -a -i -N p1
spin -t -p -g cat_mouse.pml
./pan -a -i -N p2
(...)
```

...p1, p2, p3, p4 are *false* even under the weak fairness assumption:

```
spin -a cat_mouse.pml
cc pan.c -o pan
./pan -a -i -N p3 -f
spin -t -p -g cat_mouse.pml          Should we be surprised about p3?
```

Termination of this model is guaranteed only under weak fairness

```
ltl { <>(_nr_pr == 1) }

bool exit;

active proctype P() {
  do
  :: exit -> break
  :: else -> skip
  od
}

active proctype Q() {
  exit = true
}
```

Weak Fairness (definition)

A run is said to be weakly fair if and only if:
any process that reaches a statement that is continuously executable will eventually execute that statement.

Documentation supporting this lecture

- R. Barbosa, Verificação de modelos, Ch. 4
- Specifying LTL formulæ in Spin
  http://spinroot.com/spin/Man/ltl.html
- Remote references
  http://spinroot.com/spin/Man/remoterefs.html

Modeling Distributed Systems

Raul Barbosa

VALU3S Summer School
Genova, Italia, July 18–20, 2023

Declaring communication channels

- Channels allow active processes to transfer messages.
- The keyword `chan` declares such channels.
- Much like any other variables, channels may be global or local.
- Channels function in first in first out (FIFO) order, although this can be configured.

Declaring channels

```
chan c, c1, cs[4] ...
```

Declaring and initializing channels

```
chan c = [8] of {byte}
```

A channel that can store up to 8 messages, each containing a single field of type byte.

```
chan a[4] = [0] of {mtype}
```

Four rendezvous channels (synchronous write-read) containing one message field of type mtype.

```
chan ch = [10] of {mtype, int, chan}
```

A channel that can exchange messages containing multiple fields of different types.

Buffered channels significantly increase verification complexity – start small.

Sending and receiving messages

Syntax for sending

`channel ! message_args`	(most common, FIFO order)
`channel !! message_args`	(less used, numerically ordered)

Syntax for receiving

`name ? receive_pattern`	(reads the first message)
`name ?? receive_pattern`	(reads one matching message)

Example 1 (run with the -c option)

```
chan ch = [4] of {byte};

active proctype P() {
  byte b;
  ch!1; ch!2; ch!3; ch!4;
  ch?b; ch?b; ch?b; ch?b;
}
```

Example 2 (run with the -c option)

```
init {
  chan ch = [3] of { short };
  byte x;

  ch!3; ch!2; ch!1;  // send 3 messages
  ch?x;              // read the first message
  if
  :: ch?<x>          // just copy the message
  :: empty(ch)
  :: ch?2            // match and read a 2, if found
  :: ch?_            // flush and discard a message
  fi
}
```

Example 3 (run with the -c option)

```
mtype = {ping, pong};

chan t = [1] of {mtype};

active proctype P1() {
  do
  :: t!ping; t?pong
  od
}

active proctype P2() {
  do
  :: t?ping; t!pong
  :: break
  od
}
```

Channels as queues

- Typically, models use FIFO order (single ! and ?).
- A proctype may receive an uninitialized chan as argument.
- Buffered channels *vs* rendezvous channels:
  - A channel initialized with capacity > 0 is called a *buffered channel*.
  - A channel with capacity $= 0$ is a *rendezvous channel* which passes messages through handshakes between sender/receiver (it does not store messages).

Documentation supporting this lecture

- Mordechai Ben-Ari, "Principles of the Spin Model Checker", ISBN 1846287693, Springer, 2008.
- Gerard Holzmann, The SPIN Model Checker: Primer and Reference Manual, ISBN 0321773713, Addison-Wesley, 2003.
- Message passing channels
  http://spinroot.com/spin/Man/chan.html
- Sending and receiving messages
  http://spinroot.com/spin/Man/send.html
  http://spinroot.com/spin/Man/receive.html

Model Checking

Raul Barbosa

VALU3S Summer School
Genova, Italia, July 18–20, 2023