Concurrent and Distributed Systems

Talk Five

Model Checking
with SPIN
and Linear Temporal Logic

Maxim Mozgovoy

Formal Verification Revisited

We've just used SPIN like an ordinary coding environment: to run Promela programs… but this is not its real purpose ☺

Reminder: model checking tools analyze scenarios of code execution to find deadlocks and race conditions. It is done by building and analyzing the program's state space diagram.

SPIN implements a number of different instruments suitable for finding different problems in your code.

Assertions in SPIN

The most basic and commonly used SPIN tool is assertions.

Use assert statement to state that a certain expression should be true at some point:

```
active proctype main() {
    int x = 1
    x = 5 + x
    x = 10 + x
    assert(x == 16)          // OK
    assert(x == 17)          // prints error message
}                            // and exits
```

A failed assertion will terminate your code with an error message!

Model Checking: The First Example! (1)

Consider a simple program:

```
int S = 0
active [10] proctype P() { S++ }
active proctype main() { assert(S != 3) }
```

Here we have 10 concurrent processes that increment S.

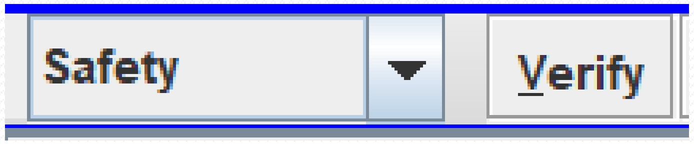
Another process checks at some point that S != 3.

This assertion may fail, but the chances are low.

Run this program using **Random** button in SPIN and see yourself!

Model Checking: The First Example! (2)

Now, instead of running the code, let's check it (use **Verify** button in **Safety** mode).



```
pan:1: assertion violated (S!=3) (at depth 3)
pan: wrote test.pml.trail
```

So, SPIN has found a scenario where our assertion can be violated!

To display the previously found scenario, press **Guided** button.

Model Checking: The First Example! (3)

Process ID

Line number

Statement

Process name

Variables

```
9 P     4   S = (S+1)
Process Statement          S
8 P     4   S = (S+1)      1
7 P     4   S = (S+1)      2
spin: test.pml:9, Error: assertion violated
spin: text of failed assertion: assert((S!=3))
#processes: 11
  4: proc 10 (main) test.pml:9 (state 1)
...
```

Assertion failed here

Assertions and Race Conditions

Assertions help to prove that your algorithm does not depend on race conditions.

In other words, use assertions to prove that all execution scenarios of your code provide the same results. For example:

- Check that the resulting array of an array sorting function is indeed a sorted array.
- Check that the resulting value of a maximum-finding function is indeed the maximum value.
- Check that a square root function indeed returns a square root of its argument, etc.

Another Simple Example

The following program swaps two variables:

```
int temp
active proctype P() {
     int a = _pid
     int b = _pid + 1
     temp = a; a = b; b = temp
     assert(a == _pid + 1 && b == _pid)
}    // (check that the variables are swapped)
```

Do:

1) Verify this code. 2) Run two processes instead of one (with **active [2] proctype**). 3) Check this new program. 4) Fix it.

Detecting Deadlocks (1)

A deadlock is an example of an invalid end state in SPIN.
Verify button finds possible invalid end states automatically!

```
int sticks = 2  // rice-eating example

active [2] proctype P1() {
  atomic { sticks > 0       // wait for a stick
           sticks--  }      // take a stick
  atomic { sticks > 0       // wait for a stick
           sticks--  }      // take a stick
  sticks++      // eat and put the sticks back
  sticks++
}
```

Detecting Deadlocks (2)

Verifier output:

```
pan:1: invalid end state (at depth 1)
pan: wrote test.pml.trail
```

Guided run output:

```
1 P1     5    sticks>0
1 P1     6    sticks = (stic
Process Statement             sticks
0 P1     5    sticks>0        1
0 P1     6    sticks = (stic 1
spin: trail ends after 2 steps
```

…

Safety and Liveness Properties

Sometimes we need to check more complex global properties of the algorithm, often classified into safety and liveness:

Safety: “something bad never happens”. For example:

- Two processes never execute the same critical section simultaneously.
- The number of chopsticks on the table in the “eating rice” problem is never negative.

Liveness: “something good eventually happens”. For example:

- The array sorting code will eventually produce a sorted array.
- A process that tries to enter a critical section will sooner or later enter it.

SPIN Algorithms: Technical Notes

Not all properties are equally easy to check in practice.

The least computationally expensive are safety properties.

Thus, if you are proving safety (e.g., checking assertions and detecting deadlocks), **Safety** mode is sufficient.

To check fairness properties, **Acceptance** or **Non-progress** modes should be used.

In principle, you can use Acceptance or Non-progress modes to check safety properties as well, but you may face slowdowns or limitations when checking larger programs.

Acceptance Mode (1)

Acceptance mode reports an error if your program can potentially execute an “acceptance” statement infinitely often.

“Acceptance” statements are marked with any labels starting with the prefix `accept`.

The idea of this mode is to find out whether the system can be constantly doing something undesirable.

Note: “accept” means “bad” in this context;
this term is quite counter-intuitive.

Acceptance Mode (2)

Suppose a player in a board game rolls a 1-3 die to move a pawn.

- If 1 is got, the player moves the pawn one cell forward.
- If 2 is got, the player moves the pawn two cells forward.
- If 3 is got, the player has to re-roll the die.

Acceptance Mode (3)

This process can be simulated in Promela as follows:

```
int move = 0

roll: if
:: true -> move = 1
:: true -> move = 2
:: true -> move = 0
fi

if
:: move == 0 -> goto roll
:: else -> printf("move: %d", move)
fi
```

Note: theoretically, the player can get stuck forever in "re-roll the die" loop

Acceptance Mode (4)

```
int move = 0

roll: if
:: true -> move = 1
:: true -> move = 2
:: true -> move = 0
fi


if
:: move == 0 -> accept_1: goto roll
:: else -> printf("move: %d", move)
fi
```

We can label die re-rolling as "acceptance" statement and run verification in "Acceptance" mode to produce an error.

Non-Progress Mode (1)

Non-progress mode reports an error if your program does not execute a “progress” statement infinitely often.

“Progress” statements are marked with any labels starting with the prefix `progress`.

The idea of this mode is to find out whether the system cannot achieve any results in the course of normal execution. This is a common result of a livelock.

Note: “Weak fairness” option in Non-progress and Acceptance modes ensures there is no starvation of processes during run.

Non-Progress Mode (2)

Let’s return to the “rice eating” example and consider an algorithm that leads to a livelock:

```
IF I can grab a free stick, I should do it
IF I have one stick and there are no free sticks,
   I should return my stick to the table
IF I have two sticks, I eat, then return them
```

In this algorithm, “I eat” step is a progress statement.

Non-Progress Mode (3)

Verification in "Non-progress" mode generates an error for this code.

```
int free = 2   // free sticks

active [2] proctype Eater() {
    int my = 0 // my sticks
    do
    :: free >= 1 -> atomic { free--; my++ }
    :: free == 0 && my == 1 -> atomic {
                                my--; free++ }
    :: else -> progress_1: // eat
               atomic { free = 2; my = 0 }
  od
}
```

Linear Temporal Logic

Global properties in SPIN can also be declared with linear temporal logic (LTL) statements that deal with *time*.

The most often used temporal (time-related) operators are:

always: `[]p` -- "`p` is always true" (safety)

eventually: `<>p` -- "sooner or later `p` becomes true" (liveness)

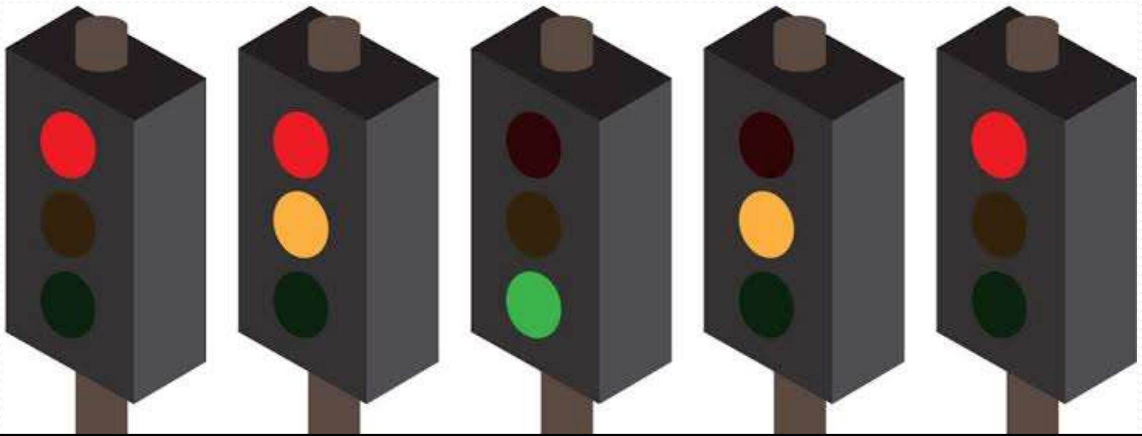
until: `p U q` -- "`p` is true at least until `q` becomes true"

Boolean algebra operations (AND, OR, NOT…) are also allowed in LTL (C/Java syntax). LTL formulas very often use *implication*: `p -> q` -- "if `p` is true, `q` is also true"

LTL: Example Statements (1)

```
mtype = { Red, Red_Yellow, Green, Yellow, Off }
mtype state = Red

active proctype main() { // traffic lights
do
:: state == Red -> state = Red_Yellow
:: state == Red_Yellow -> state = Green
:: state == Green -> state = Yellow
:: state == Yellow -> state = Red
od
}
```



(British sequence)

LTL: Example Statements (2)

In SPIN, LTL formulas are specified inside **`ltl`** `{ }` clauses:

```
// the lamp is always Red/Red_Yellow/Green/Yellow
ltl { [](state == Red || state == Red_Yellow ||
         state == Green || state == Yellow) }
```

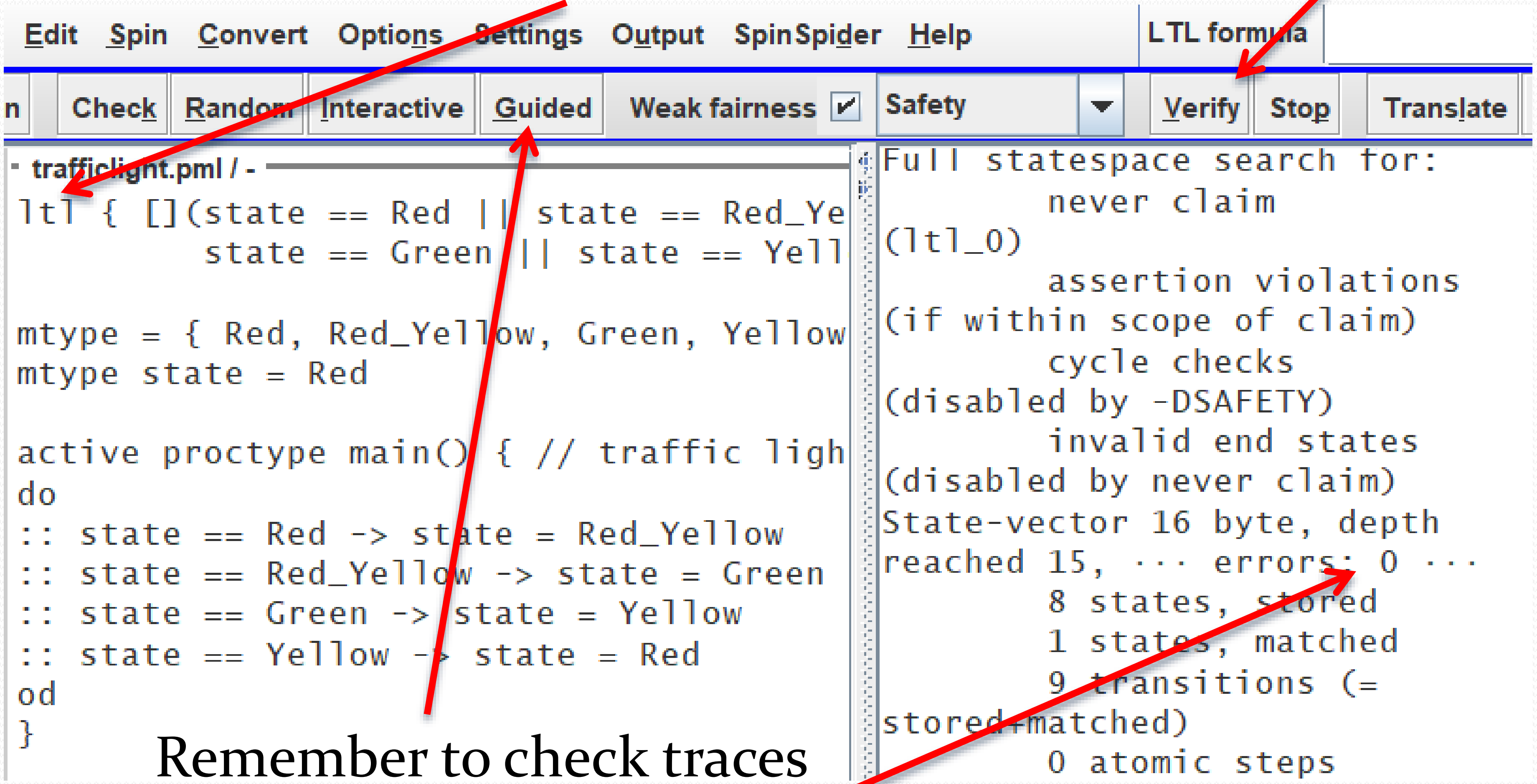
This formula describes a safety property.
Thus, we can use “Safety” mode in SPIN to check it.

However, as a rule, you should use “Acceptance” mode with LTL formulas.

Checking LTL in SPIN

Copy an LTL formula into your Promela code and push **Verify**.

Note: you can check only one LTL statement at time.



```
ltl { [](state == Red || state == Red_Ye
         state == Green || state == Yell

mtype = { Red, Red_Yellow, Green, Yellow
mtype state = Red

active proctype main() { // traffic ligh
do
:: state == Red -> state = Red_Yellow
:: state == Red_Yellow -> state = Green
:: state == Green -> state = Yellow
:: state == Yellow -> state = Red
od
}
```

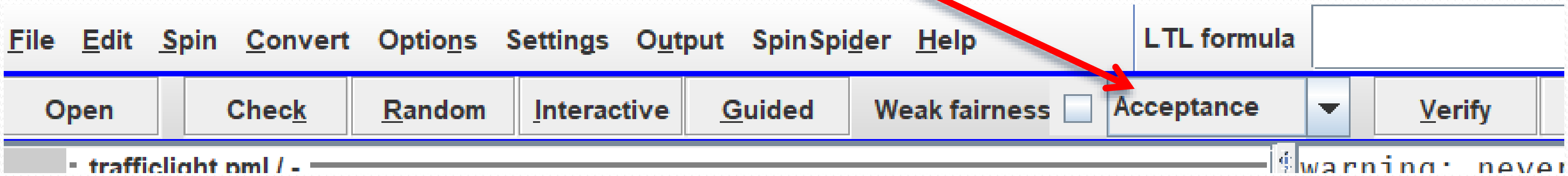
```
Full statespace search for:
        never claim
(ltl_0)
        assertion violations
(if within scope of claim)
        cycle checks
(disabled by -DSAFETY)
        invalid end states
(disabled by never claim)
State-vector 16 byte, depth
reached 15, ··· errors: 0 ···
        8 states, stored
        1 states, matched
        9 transitions (=
stored+matched)
        0 atomic steps
```

Remember to check traces if you got an error!

LTL: Example Statements (3)

```
// if there is a red light,
// eventually there will be a green light
ltl { (state == Red -> <>(state == Green))}
```

This formula describes a liveness property.
Thus, we must use the “Acceptance” mode in SPIN to check it.



More on Safety vs Liveness (1)

It might be tricky to recognize whether the given property belongs to the “safety” or “liveness” type.

SPIN works by searching a state space diagram of the given program for counterexamples that violate the stated LTL properties. The choice of operating mode depends on the required type of work.

In “Safety” mode, SPIN searches for a single situation violating the given LTL formula.

In “Acceptance” (liveness) mode, SPIN searches for a path violating the given LTL formula.

More on Safety vs Liveness (2)

For example, a property like

```
<>(state == Off)
```

cannot be checked in “Safety” mode: by analyzing any single situation we cannot prove that each code path eventually turns `state` into `Off`.

In contrast, a property like

```
[](state == Red)
```

can be checked in “Safety” mode, because we can report an error if we find any situation where `state` is not `Red`.

For simple programs (like our exercises)
you can always use Acceptance mode with LTL formulas.

Latching Property (1)

Let's modify our traffic light program a bit,
so it starts in a "turned off" state.

```
mtype state = Off

active proctype main() { // traffic lights
state = Red
do
:: state == Red -> state = Red_Yellow
:: state == Red_Yellow -> state = Green
:: state == Green -> state = Yellow
:: state == Yellow -> state = Red
od
}
```

Latching Property (2)

Now the program does not satisfy the "always" formula:

```
[](state == Red || state == Red_Yellow ||
   state == Green || state == Yellow)
```

Instead, it satisfies another important property called *latching*:

```
<>[]A
```

("`A` might be false in the beginning,
but eventually it becomes true and stays true").

In our case,

```
<>[](state == Red || state == Red_Yellow ||
   state == Green || state == Yellow)
```

“Infinitely Often” Property

Another important property called *“infinitely often”* is expressed with a formula `[]<>A`

(“for any situation there is a later situation where `A` is true”)

For our traffic lights system, we can check the property “the green signal appears *indefinitely often*” (rather than just once or twice):

```
[]<>(state == Green)
```

Detecting Livelocks (1)

"Infinitely often" property can be used to detect livelocks.

Let's again consider our livelocked "rice eating" algorithm:

```
IF I can grab a free stick, I should do it
IF I have one stick and there are no free sticks,
   I should return my stick to the table
IF I have two sticks, I eat, then return them
```

It does not satisfy the following desired properties:

1) *Eventually someone can eat, and*
2) *Eating repeats infinitely often.*

Detecting Livelocks (2)

```
int free = 2    // free sticks
int eat = 0     // how many processes eat

active [2] proctype Eater() {
    int my = 0 // my sticks
    do
    :: free >= 1 -> atomic { free--; my++ }
    :: free == 0 && my == 1 -> atomic {
                                 my--; free++ }
    :: else -> eat++; eat--;
               atomic { free = 2; my = 0 }
  od
}
```

Check
`[]<>(eat > 0)`

Expressing Precedence (1)

```
int Semaphore = 2
bool sorted1 = false
bool sorted2 = false
bool merging = false

active proctype SortFirstHalf()
    { sorted1 = true; Semaphore-- }

active proctype SortSecondHalf()
    { sorted2 = true; Semaphore-- }

active proctype Merging()
    { Semaphore == 0; merging = true }
```

Let’s revisit the “array sorting” code from the previous lecture.

Expressing Precedence (2)

The “until” operation `U` is most commonly used in expression

`!p U q` (`p` remains false until `q` becomes true)

In case of array sorting,
the merging phase is not active until both halves are sorted:

```
ltl { !merging U (sorted1 && sorted2) }
```

And that sooner or later both phases will be completed:

```
ltl { <>(merging && sorted1 && sorted2) }
```

Conclusions

- SPIN provides a number of tools for program verification.
- Typically, check for safety (bad things never happen) and liveness (good things eventually happen) properties.
- Typical safety properties: no race conditions, no deadlocks.
- Typical liveness properties: latching, precedence and "infinitely often" (no livelocks guarantee).
- SPIN provides tools for checking both safety and liveness properties (assertions; search for invalid end states, acceptance and non-progress cycles).
- Alternatively, linear temporal logic (LTL) can be used to declare certain global properties to be checked.