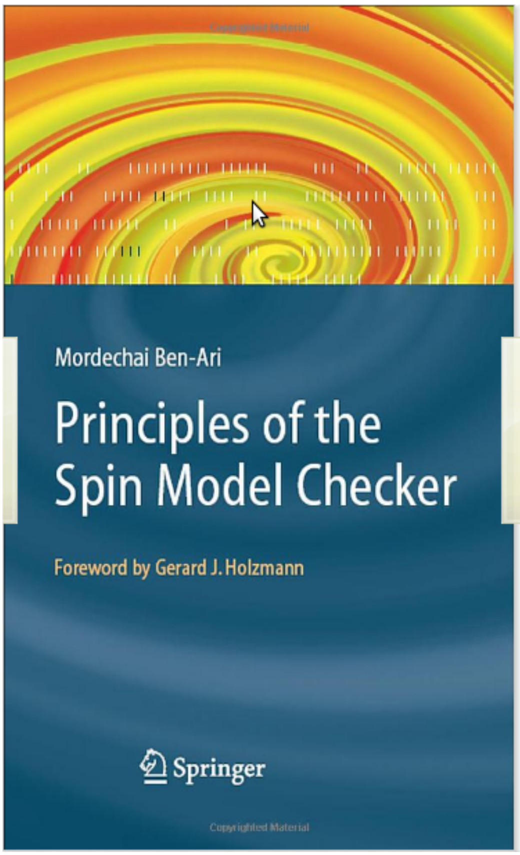
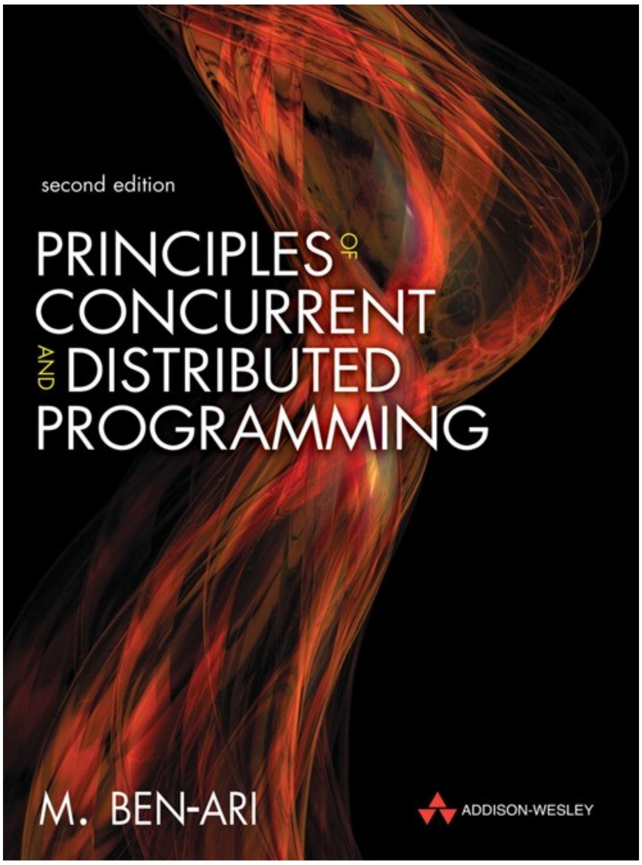

Mordechai Ben-Ari
Principles of the Spin Model Checker
Foreword by Gerard J. Holzmann
Springer

Mordechai Ben-Ari

Principles of the Spin Model Checker

Springer, 2008

ISBN: 978-1-84628-769-5



Mordechai Ben-Ari

Principles of Concurrent and Distributed Programming

Second Edition
Addison-Wesley, 2008

ISBN: 978-0-32131-283-9

http://www.springer.com/computer/swe/book/978-1-84628-769-5

Supplementary material (zip, 38 kB)

978-1-84628-769-5-additional material.zip

Chapter 5

Verification with Temporal Logic

Assertions are not sufficient to specify and verify most correctness properties of models.

This chapter presnts ***linear temporal logic*** (*LTL*), which is the formal logic used for verification in Spin.

Section 5.1

Beyond assertions

Assertions are limited in the properties that they can specify because they are attached to specific control points in the processes.

For example, in order to verify that mutual exclusion holds for a solution to the critical section problem, we inserted the following code at the control points representing the critical section in *each* process:

```
critical++;
assert (critical <= 1);
critical--;
```

Usually, however, it is necessary or at least more convenient to express a correctness property as a global property of the system that is not associated with specific control points.

Here are several examples of such properties:

- **Mutual exclusion**
  Mutual exclusion can be expressed as a global invariant:

  In every state of every computation, `critical <= 1`.

- **Absence of deadlock (invalid end states)**
  A Promela program is said to deadlock if it enters an invalid end state; this can be expressed as a global invariant:

  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***.*

  This correctness property is checked automatically by Spin.

- **Array index bounds**

Let `a` be an array, let `LEN` be the length of the array, and let `i` be a variable used to index the array. An important global invariant is:

In every state of every computation, `0 <= i <= LEN-1`.

This formula could be added as an assertion after every statement that assigns a new value to `i`, but it is easier to specify that it holds in every state. This avoids errors caused if you forget to attach an assertion to one of the relevant statements.

- **Quantity invariant**

In distributed algorithms called *token-passing algorithms*, mutual exclusion is achieved by passing a *token* – an explicit representation of the permission to enter the critical section – among the processes. A global invariant that must hold in such algorithms is:

In every state of every computation, there is at most one token in existence.

There are some correctness properties that simply cannot be expressed using assertions, because the properties cannot be checked be evaluating an expression in a *single state* of a computation. For example, in the critical section problem the following two properties are expressed as relations between two states of the computation: a state ***s*** in which processes are trying to enter their critical sections, and a state ***t*** in which a process does enter it. ***t*** may occur thousands of states later in the computation than ***s***:

- **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.*

A correctness specification like the ones given in this section is expressed in Spin by a finite automaton called a ***never claim*** that is executed together with the finite automaton that represent the Promela program.

A never claim is used to specify behavior that should ***never*** happen.

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.

Section 5.2

Introduction to linear temporal logic

In the usual study of mathematical logic, an assignment of truth values to atomic propositions is used to give an interpretation to a formula; in the interpretation, the formula will be either *true* or *false*. But within the context of the execution of a program, the assignment may change from state to state.

For example, if *A* is the proposition **`turn = 1`**, then the proposition is true in the initial state of Dekker's algorithm, but it will become false in any state that immediately follows the execution of the statement **`turn ← 2`**.

In **`turn = 1`** is sometimes true and sometimes false, and a new system of logic is needed to deal with such situations.

Temporal logic is a system of formal logic obtained by adding temporal operators to propositional or predicate logic.

Subsection 5.2.1

The syntax of LTL

LTL is based upon the propositional calculus; formulas of the propositional calculus are composed from atomic propositions (denoted by letters *p*, *q*, ...) and the operators:

Operator	Math	Spin
not	$\neg$	!
and	$\wedge$	&&
or	$\vee$	||
implies	$\rightarrow$	->
equivalent	$\leftrightarrow$	<->

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. The atomic propositions of LTL are described in the next subsection. The temporal operators are:

Operator	Math	Spin
always	□	`[]`
eventually	◇	`<>`
until	U	`U`

The □ and ◇ operators are unary and the U operator is binary. Temporal and propositional operators combine freely, so the following formula (given in both mathematical and Promela notation) is syntactically correct:

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

```
[]((p && q) -> r U (p || r)).
```

Read this as:

Always, (*p* and *q*) implies that *r* holds until (*p* or *r*) holds.

Subsection 5.2.2

The semantics of LTL

The semantics, the meaning, of a syntactically correct formula is defined by a 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. For the propositional calculus these are given by the familiar *truth tables*, where *A* and *B* are any formulas:

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*

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.

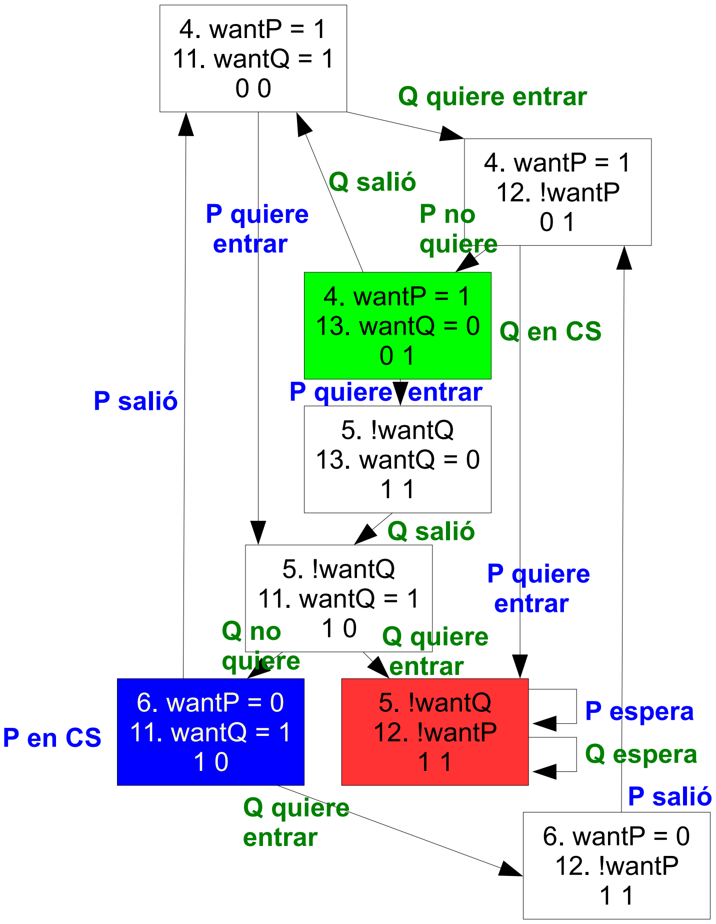
For example, let *critical* be the value of the variable **`critical`** in a program for the critical section problem; the expression *critical* ≤ 1 is an atomic proposition because it can be assigned a truth value in a state *s* just by checking the value of the variable **`critical`** in *s*.

Similarly, if *csp* is a boolean expression that is true if and only if the location counter of process P is at the control point corresponding to the critical section of the process, then *csp* is an atomic proposition because it can be evaluated in any single state.

Atomic propositions can be combined using the operators of the propositional calculus; such formulas can also be evaluated just by checking values in a single state.

For example, if *csq* is similar to *csp* but for process Q, the expresion ¬(*csp* ∧ *csq*) specifies that mutual exclusion holds *in the state in which it is evaluated*.

Fig. 5.1. State diagram for the third attempt



Consider again the program for the critical section problem in Listing 4.3 and its state diagram in Figure 4.1 (repeated in Figure 5.1). 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)`, and continues by taking legal transitions, which are the ones shown in the diagram. For example, here are the first few states of a computation:

s_0 = `(4. wantP=1, 11. wantQ=1, 0, 0)` →

s_1 = `(4. wantP=1, 12. !wantP,  0, 1)` →

s_2 = `(4. wantP=1, 13. wantQ=0, 0, 1)` →

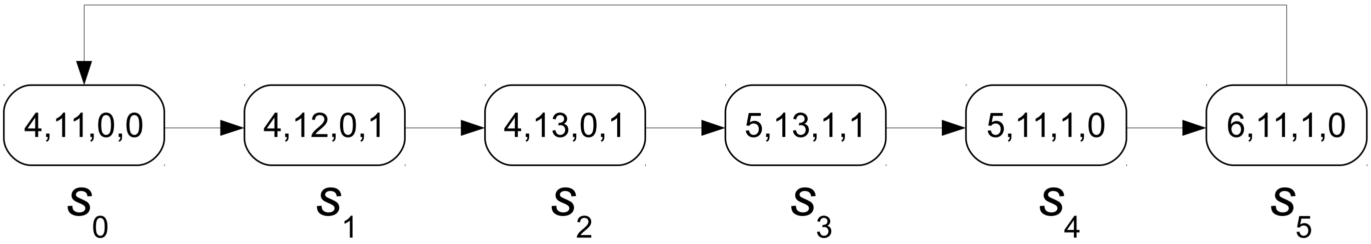
s_3 = `(5. !wantQ,  13. wantQ=0, 1, 1)` →

s_4 = `(5. !wantQ,  11. wantQ=1, 1, 0)` →

s_5 = `(6. wantP=0, 11. wantQ=1, 1, 0)` →

s_6 = `(4. wantP=1, 11. wantQ=1, 0, 0)`

A computation, an infinite sequence of states, is obtained by repeating the same transitions indefinitely. Since the last state is the same as the first, the infinite sequence of states can be *finitely presented* by identifying the first and last states; that is, instead of creating a new state s_6, we create a transition from s_5 to s_0:



Process P is in its critical section if its location counter is at line 6 and process Q is in its critical section if its location counter is at line 13. Let *csp* and *csq* be atomic propositions representing these properties. Clearly, *for the computation shown above*, the formula $\neg(csp \wedge csq)$ that expresses the correctness property of mutual exclusion is true in *all* its states.

We have shown that this formula is true for one specific computation, but since there are no states in which process P is at line 6 and process Q is at line 13, we can generalize and claim that the following statement is true:

> The formula $\neg(csp \wedge csq)$ is true in every state of every computation.

Section 5.3

Safety properties

Subsection 5.3.1

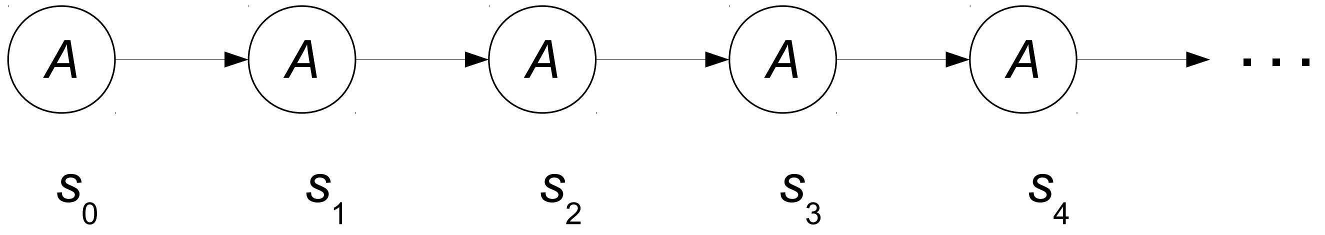
Expressing safety properties in LTL

Let *A* be an LTL formula and let $\tau = (s_0, s_1, s_2, ...)$ be a computation. Then □*A*, read *box* or *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 □*A* is true in state *s*, then *A* must also be true in *s*.

The formula □*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."

We can draw a diagram of a computation, labeling each state s_j with *A* if *A* is true in s_j and with ¬*A* if *A* is false in s_j. If the following diagram is extended indefinitely with all states labeled *A*, then □*A* is true in s_0:



The correctness property of mutual exclusion can be expressed by the LTL formula $\Box\neg(csp \wedge csq)$.

This is a safety property because it is true if something "bad" – $csp \wedge csq$, meaning the two processes in their critical section – never happens.

Equivalently, the only states the computation enters are "good" ones in which $\neg(csp \wedge csq)$ is true.

Subsection 5.3.2

Expressing safety properties in Promela

For the program in Listing 4.3 we verified that mutual exclusion holds by writing **assert** statements in each critical section. It is also possible to express this property in LTL. As before, we declare a variable **critical**, incrementing it at the beginning of each critical section and decrementing it at the end:

```
active proctype P() {
  do
  :: wantP = true
     !wantQ
     critical++
     critical--
     wantP = false
  od
}
/* Similarly for process Q */
```

Since the critical section in the abbreviated program is not explicitly written, the statement **critical--** immediately follows the statement **critical++**, and the critical section is the control point between them.

The symbol **mutex** is defined to represent an expression that is true if and only if mutual exclusion holds:

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

Mutual exclusion can now be specified in Promela by LTL formula:

```
[]mutex
```

Alternatively, we could define two variables `csp` and `csq` of boolean type, and set these variables to indicate when a process is in its critical section:

```
active proctype P() {
  do
  :: wantP = true
     !wantQ
     csp = true
     csp = false
     wantP = false
  od
}
/* Similarly for process Q */
```

The LTL formula expressing mutual exclusion is now

```
[]!(csp && csq)
```

Subsection 5.3.3

Verifying safety properties in Spin

NAME

ltl - linear time temporal logic formulae for specifying correctness requirements.

SYNTAX

Grammar:

    ltl ::= opd | (ltl) | ltl binop ltl | unop ltl

Operands (**opd**):

    true, false, user-defined names starting with a lower-case letter,
    or embedded expressions inside curly braces, e.g.,: { a+b>n }.

Unary Operators (**unop**):

```
        []      (the temporal operator always)
        <>      (the temporal operator eventually)
        !       (the boolean operator for negation)
```

Binary Operators (**binop**):

```
        U       (the temporal operator strong until)
        W       (the temporal operator weak until)
        V       (the dual of U): (p V q) means !(!p U !q))
        &&      (the boolean operator for logical and)
        ||      (the boolean operator for logical or)
        /\      (alternative form of &&)
        \/      (alternative form of ||)
        ->      (the boolean operator for logical implication)
        <->     (the boolean operator for logical equivalence)
...
```

DESCRIPTION

Starting with Spin version 6 there are two different formalisms for specifying LTL formulae. The most convenient mechanism is to specify LTL formulae inline, as part of a verification model. The older mechanism of using Spin to first convert a formula into a never claim and to include that never claim into the model is still supported as well. We first describe the new mechanism.

INLINE SPECIFICATION

The easiest way to specify an LTL property is to specify it inline. The formula is specified globally (i.e., outside all **proctype** or **init** declarations) with the following syntax:

```
        ltl [ name ] '{'  formula '}'
```

The name is optional, but can be useful when specifying multiple formulae. (Each such formula follows the same basic format.) The formula has the grammar outlined above, with some extensions. First, white space (newlines, spaces, tabs) can be used anywhere to separate operands and operators. Second, the names of operators can either be abbreviated with the symbols shown above, or spelled out in full (as **always**, **eventually**, **until**, **implies**, and equivalent. The alternative operators **weakuntil**, **stronguntil**, and **release** (for the V operator, see above), are also supported.

This means that the following two are equivalent:

```
        ltl p1 { []<> p }
        ltl p2 { always eventually p }
...
```

The properties stated in this way are taken as positive properties that must be satisfied by the model. The model checker will perform an automatic negation of the formula to find counter-examples. (The negation is not automatic when you use the alternative, and older, method described below).

Another improvement in the specification of ltl formula in Spin version 6 and later is that the inline ltl formula can contain any propositional state formula, i.e., it is not restricted to the lower-case propositional symbols from before, where each propositional symbol has to be defined in macro definitions. This means that we can write:

```
        ltl p3 { eventually (a > b) implies len(q) == 0 }
```

There is just one restriction: you cannot use the predefined operators **empty**, **nempty**, **full**, **nfull** in inline ltl formula (and should not use them in the alternative method either, although it is not enforced there). The reason is technical: under the specific partial order reduction rules used in the model checker the four channel operators listed cannot be negated. In the conversion process from LTL formula into never claims (Buchi automata) negations will take place and can end up in the final automaton though, which would risk making partial order reductions invalid (logically unsound).

When multiple inline LTL formulae are specified, you get to choose which one will be applied during a verification run with a new runtime parameter -N for the pan verifiers. By default this will be the first LTL formula that appears in the specification. To disable all LTL formula during a verification you can compile **pan.c** with the directive **-DNOCLAIM**.
...

ALTERNATIVE METHOD
Spin can translate LTL formulae into *Promela* never claims with command line option **-f** . The never claim that is generated encodes the Buchi acceptance conditions from the LTL formula. Formally, any omega-run that satisfies the LTL formula is guaranteed to correspond to an accepting run of the never claim.

The operands of an LTL formula are often one-character symbols, such as p , q , r , but they can also be symbolic names, provided that they start with a lowercase character, to avoid confusion with some of the temporal operators which are in uppercase. The names or symbols must be defined to represent boolean expressions on global variables from the model. The names or symbols are normally defined with macro definitions.
Starting with Spin version 5.2.4 operands can also be defined as embedded free-form expressions enclosed in curly braces, for instance, something like: [] ({a>b} -> {a>100}). This form avoids the need for the use of macros to define predicate symbols. (The inline method described earlier also avoids the need to use the curly braces.)

All binary operators are left-associative. Parentheses (round braces) can be used to override this default. Note that implication and equivalence are not temporal but logical operators.

EXAMPLES
Some examples of valid LTL formulae follow, as they would be passed in command-line arguments to *Spin* for translation into never claims. Each formula passed to *Spin* has to be quoted. We use single quotes in all examples given here, which will work correctly on most systems (including Unix systems and Windows systems with the cygwin toolset). On some systems double quotes can also be used.

```
    spin -f '[] p'
    spin -f '!( <> !q )'
    spin -f 'p U q'
    spin -f 'p U ([] (q U r))'
```

The conditions p , q , and r can be defined with macros, for instance as:

```
    #define p       (a > b)
    #define q       (len(q) < 5)
    #define r       (root@Label)
```

elsewhere in the *Promela* model. It is prudent to always enclose these macro definitions in round braces to avoid misinterpretation of the precedence rules on any operators in the context where the names end up being used in the final never claim. The variables a and b , the channel name q , and the proctype name root from the preceding example, must be globally declared.

Prepare el siguiente programa:

```
$ cat -n second_v6.pml
     1  bool csP = false, csQ = false
     2
     3  ltl p1 { []!(csP && csQ) }
     4
     5  active proctype P() {
     6    do ::  !csQ
     7           csP = true
     8           csP = false
     9    od
    10  }
    11
    12  active proctype Q() {
    13    do ::  !csP
    14           csQ = true
    15           csQ = false
    16    od
    17   }
```

```
$ spin -run second_v6.pml
ltl p1: [] (! ((csP) && (csQ)))
warning: never claim + accept labels requires -a flag to fully verify
pan:1: assertion violated  !( !( !((csP&&csQ)))) (at depth 8)
pan: wrote second_v6.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             + (p1)
        assertion violations    + (if within scope of claim)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      - (disabled by never claim)

State-vector 28 byte, depth reached 11, errors: 1
...

pan: elapsed time 0 seconds
ltl p1: [] (! ((csP) && (csQ)))
```

```
$ spin -run -a second_v6.pml
pan:1: assertion violated  !( !( !((csP&&csQ)))) (at depth 8)
pan: wrote second_v6.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             + (p1)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 28 byte, depth reached 11, errors: 1
...

pan: elapsed time 0 seconds
ltl p1: [] (! ((csP) && (csQ)))
```

Prepare el siguiente programa (sin LTL):

```
$ cat -n second.pml
     1  bool csP = false, csQ = false
     2
     3
     4
     5  active proctype P() {
     6    do ::  !csQ
     7           csP = true
     8           csP = false
     9    od
    10  }
    11
    12  active proctype Q() {
    13    do ::  !csP
    14           csQ = true
    15           csQ = false
    16    od
    17  }
```

```
$ spin -f '![]!(csP && csQ)' -run second.pml
warning: never claim + accept labels requires -a flag to fully verify
warning: for p.o. reduction to be valid the never claim must be
stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
pan:1: assertion violated  !((csP&&csQ)) (at depth 8)
pan: wrote second.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             + (never_0)
        assertion violations    + (if within scope of claim)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      - (disabled by never claim)

State-vector 28 byte, depth reached 11, errors: 1
...
```

Prepare el siguiente programa:

```
$ cat -n third-safety.pml
     1  #define mutex (critical <= 1)
     2
     3  bool wantP = false, wantQ = false
     4  byte critical = 0
     5
     6  active proctype P() {
     7      do
     8      :: wantP = true
     9         !wantQ
    10         critical++
    11         critical--
    12         wantP = false
    13      od
    14  }
    15
    16  active proctype Q() {
    17      do
    18      :: wantQ = true
    19         !wantP
    20         critical++
    21         critical--
    22         wantQ = false
    23      od
    24  }
```

```
$ spin -f '![]mutex' -run third-safety.pml
warning: for p.o. reduction to be valid the never claim must be
stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)

(Spin Version 6.4.5 -- 1 January 2016)
        + Partial Order reduction

Full statespace search for:
        never claim             + (never_0)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 28 byte, depth reached 0, errors: 0
...
```

```
...
unreached in proctype P
        third-safety.pml:9, state 2, "(!(wantQ))"
        third-safety.pml:10, state 3, "critical = (critical+1)"
        third-safety.pml:11, state 4, "critical = (critical-1)"
        third-safety.pml:12, state 5, "wantP = 0"
        third-safety.pml:14, state 9, "-end-"
        (5 of 9 states)
unreached in proctype Q
        third-safety.pml:19, state 2, "(!(wantP))"
        third-safety.pml:20, state 3, "critical = (critical+1)"
        third-safety.pml:21, state 4, "critical = (critical-1)"
        third-safety.pml:22, state 5, "wantQ = 0"
        third-safety.pml:24, state 9, "-end-"
        (5 of 9 states)
unreached in claim never_0
        third-safety.pml.nvr:10, state 8, "-end-"
        (1 of 8 states)

pan: elapsed time 0 seconds
```

No se detectan los errores porque el programa sí, garantiza la exclusión mutua, pero ...

Mientras tanto, se puede guardar la fórmula LTL en el archivo de una sola línea:

```
$ echo '![]mutex' > safety.prp
$ spin -F safety.prp -run third-safety.pml
...
```

Alternatively, the translation of the LTL formula to a never claim can be saved in a file and this file included in the generation of the verifier using the -N argument:

```
$ spin -f '![]mutex' > safety.ltl
$ spin -N safety.ltl -run third-safety.pml
...
```

Prepare el siguiente programa:

```
$ cat -n third-safety_v6.pml
     1  #define mutex (critical <= 1)
     2
     3  bool wantP = false, wantQ = false
     4  byte critical = 0
     5
     6  ltl { []mutex }
     7
     8  active proctype P() {
     9      do
    10      :: wantP = true
    11         !wantQ
    12         critical++
    13         critical--
    14         wantP = false
    15      od
    16  }
    17
    18  active proctype Q() {
    19      do
    20      :: wantQ = true
    21         !wantP
    22         critical++
    23         critical--
    24         wantQ = false
    25      od
    26  }
```

```
$ spin -run -noclaim third-safety_v6.pml
pan:1: invalid end state (at depth 6)
pan: wrote third-safety_v6.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 20 byte, depth reached 7, errors: 1
...


pan: elapsed time 0 seconds
ltl ltl_0: [] ((critical<=1))
```

```
$ spin -t -p third-safety_v6.pml
ltl ltl_0: [] ((critical<=1))
using statement merging
  1:    proc  1 (Q:1) third-safety_v6.pml:20 (state 1)  [wantQ = 1]
  2:    proc  1 (Q:1) third-safety_v6.pml:21 (state 2)  [(!(wantP))]
  3:    proc  1 (Q:1) third-safety_v6.pml:22 (state 3)  [critical = 
(critical+1)]
  4:    proc  1 (Q:1) third-safety_v6.pml:23 (state 4)  [critical = 
(critical-1)]
  5:    proc  0 (P:1) third-safety_v6.pml:10 (state 1)  [wantP = 1]
  6:    proc  1 (Q:1) third-safety_v6.pml:24 (state 5)  [wantQ = 0]
  7:    proc  1 (Q:1) third-safety_v6.pml:20 (state 1)  [wantQ = 1]
spin: trail ends after 7 steps
#processes: 2
                wantP = 1
                wantQ = 1
                critical = 0
  7:    proc  1 (Q:1) third-safety_v6.pml:21 (state 2)
  7:    proc  0 (P:1) third-safety_v6.pml:11 (state 2)
2 processes created
```

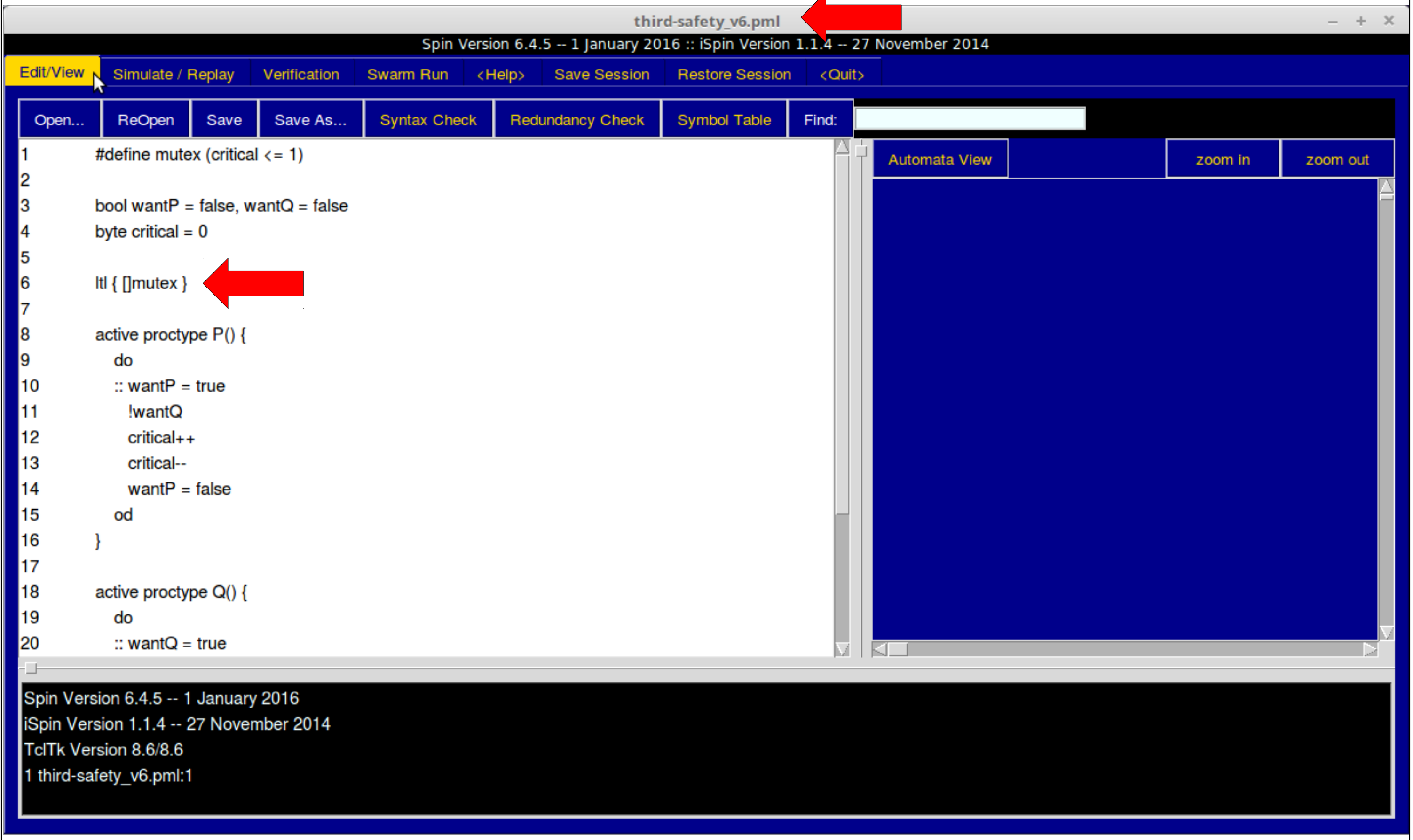
```
$ spin -run -noclaim -DBFS third-safety_v6.pml
pan:1: invalid end state (at depth 2)
pan: wrote third-safety_v6.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Breadth-First Search
        + Partial Order Reduction

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 20 byte, depth reached 2, errors: 1
...


pan: elapsed time 0 seconds
ltl ltl_0: [] ((critical<=1))
```

```
$ spin -t -p third-safety_v6.pml
ltl ltl_0: [] ((critical<=1))
using statement merging
  1:    proc  1 (Q:1) third-safety_v6.pml:20 (state 1)  [wantQ = 1]
  2:    proc  0 (P:1) third-safety_v6.pml:10 (state 1)  [wantP = 1]
  3:    proc  0 (P:1) third-safety_v6.pml:11 (state 2)  [(!(wantQ))]
        transition failed
spin: trail ends after 3 steps
#processes: 2
                wantP = 1
                wantQ = 1
                critical = 0
  3:    proc  1 (Q:1) third-safety_v6.pml:21 (state 2)
  3:    proc  0 (P:1) third-safety_v6.pml:11 (state 2)
2 processes created
```

```
$ spin -run -noclaim -i third-safety_v6.pml
pan:1: invalid end state (at depth 6)
pan: wrote third-safety_v6.pml.trail
pan: reducing search depth to 7
pan: wrote third-safety_v6.pml.trail
pan: reducing search depth to 2

(Spin Version 6.4.5 -- 1 January 2016)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 20 byte, depth reached 7, errors: 2

...
```

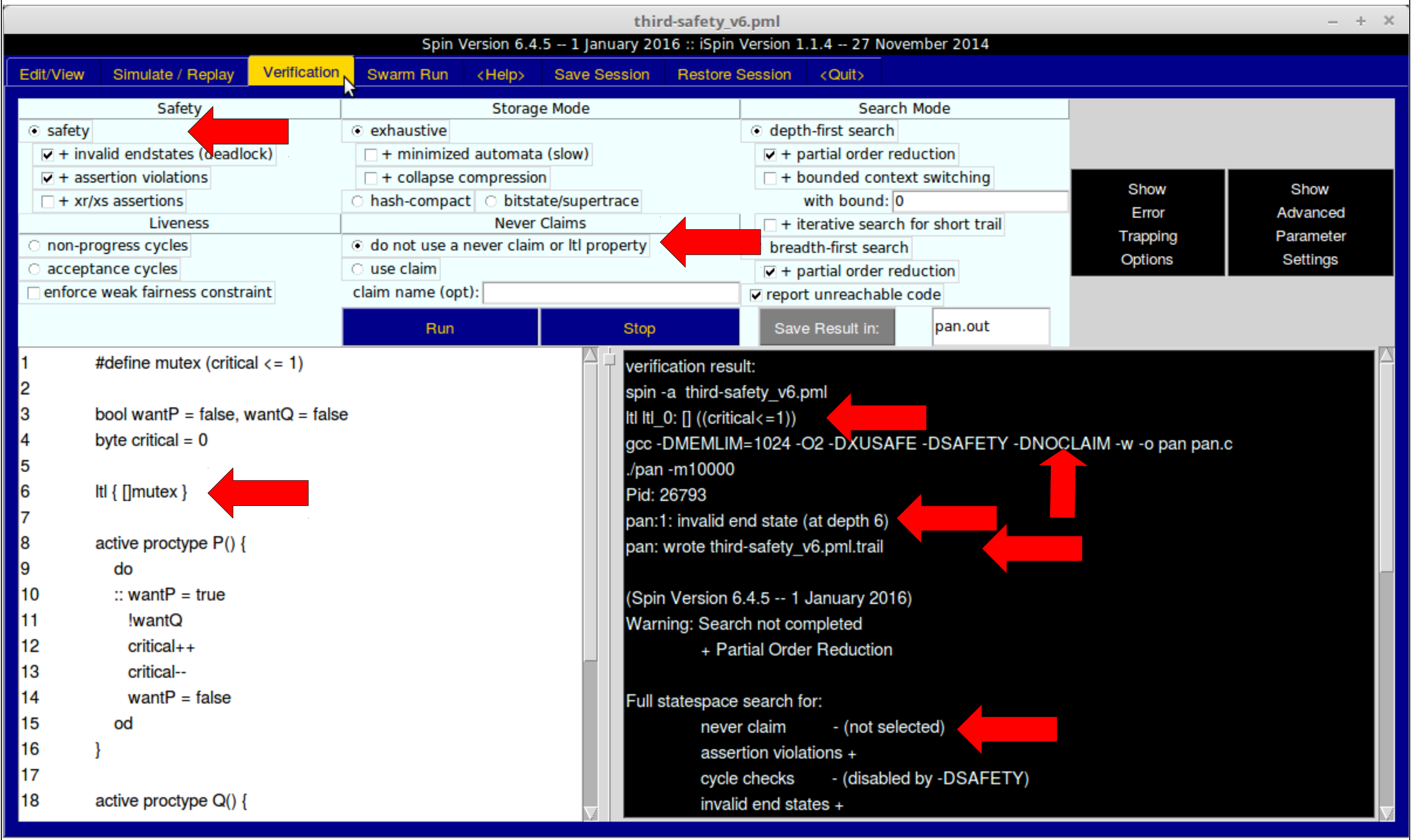
```
...

unreached in proctype P
        third-safety_v6.pml:13, state 4, "critical = (critical-1)"
        third-safety_v6.pml:14, state 5, "wantP = 0"
        third-safety_v6.pml:16, state 9, "-end-"
        (3 of 9 states)
unreached in proctype Q
        third-safety_v6.pml:26, state 9, "-end-"
        (1 of 9 states)
unreached in claim ltl_0
        _spin_nvr.tmp:3, state 6, "(!((critical<=1)))"
        _spin_nvr.tmp:3, state 6, "(1)"
        _spin_nvr.tmp:8, state 10, "-end-"
        (2 of 10 states)

pan: elapsed time 0 seconds
ltl ltl_0: [] ((critical<=1))
```

```
$ spin -t -p third-safety_v6.pml
ltl ltl_0: [] ((critical<=1))
using statement merging
  1:    proc  1 (Q:1) third-safety_v6.pml:20 (state 1)  [wantQ = 1]
  2:    proc  0 (P:1) third-safety_v6.pml:10 (state 1)  [wantP = 1]
spin: trail ends after 2 steps
#processes: 2
                wantP = 1
                wantQ = 1
                critical = 0
  2:    proc  1 (Q:1) third-safety_v6.pml:21 (state 2)
  2:    proc  0 (P:1) third-safety_v6.pml:11 (state 2)
2 processes created
```



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

bool wantP = false, wantQ = false
byte critical = 0

ltl { []mutex }

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

active proctype Q() {
   do
   :: wantQ = true
```

```
Spin Version 6.4.5 -- 1 January 2016
iSpin Version 1.1.4 -- 27 November 2014
TclTk Version 8.6/8.6
1 third-safety_v6.pml:1
```


third-safety_v6.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Show Error Trapping Options
Show Advanced Parameter Settings
Run
Stop

Safety
● safety
☑ + invalid endstates (deadlock)
☑ + assertion violations
□ + xr/xs assertions

Liveness
○ non-progress cycles
○ acceptance cycles
□ enforce weak fairness constraint

Storage Mode
● exhaustive
□ + minimized automata (slow)
□ + collapse compression
○ hash-compact ○ bitstate/supertrace

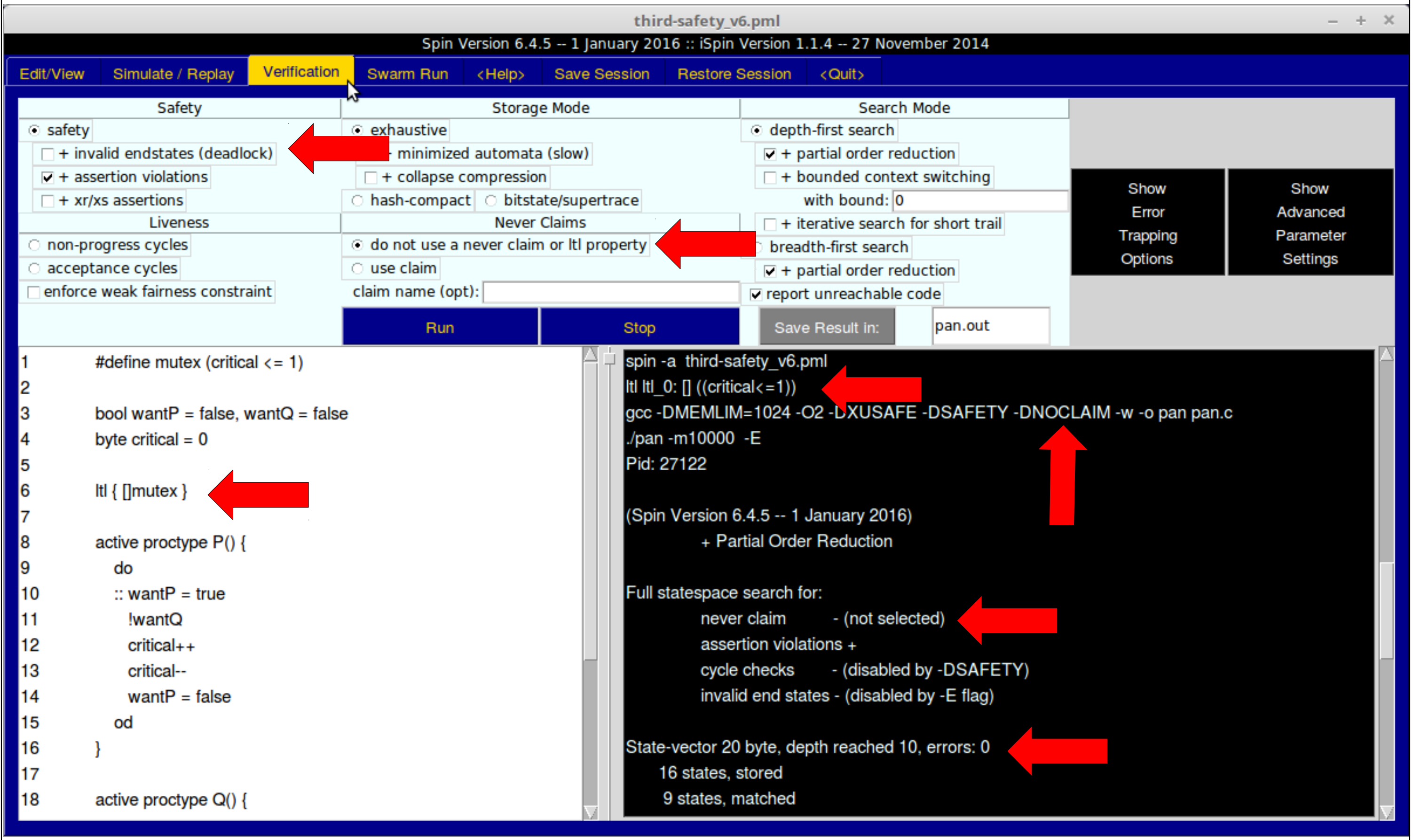
Never Claims
● do not use a never claim or ltl property
○ use claim
claim name (opt):

Search Mode
● depth-first search
☑ + partial order reduction
□ + bounded context switching
with bound: 0
□ + iterative search for short trail
breadth-first search
☑ + partial order reduction
☑ report unreachable code

Save Result in: pan.out

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

bool wantP = false, wantQ = false
byte critical = 0

ltl { []mutex }

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

active proctype Q() {
```

```
verification result:
spin -a  third-safety_v6.pml
ltl ltl_0: [] ((critical<=1))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 26793
pan:1: invalid end state (at depth 6)
pan: wrote third-safety_v6.pml.trail

(Spin Version 6.4.5 -- 1 January 2016)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (not selected)
        assertion violations +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states +
```


third-safety_v6.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Show Error Trapping Options
Show Advanced Parameter Settings
Run
Stop
Save Result in:

Safety
● safety
☐ + invalid endstates (deadlock)
☑ + assertion violations
☐ + xr/xs assertions

Liveness
○ non-progress cycles
○ acceptance cycles
☐ enforce weak fairness constraint

Storage Mode
● exhaustive
☐ + minimized automata (slow)
☐ + collapse compression
○ hash-compact ○ bitstate/supertrace

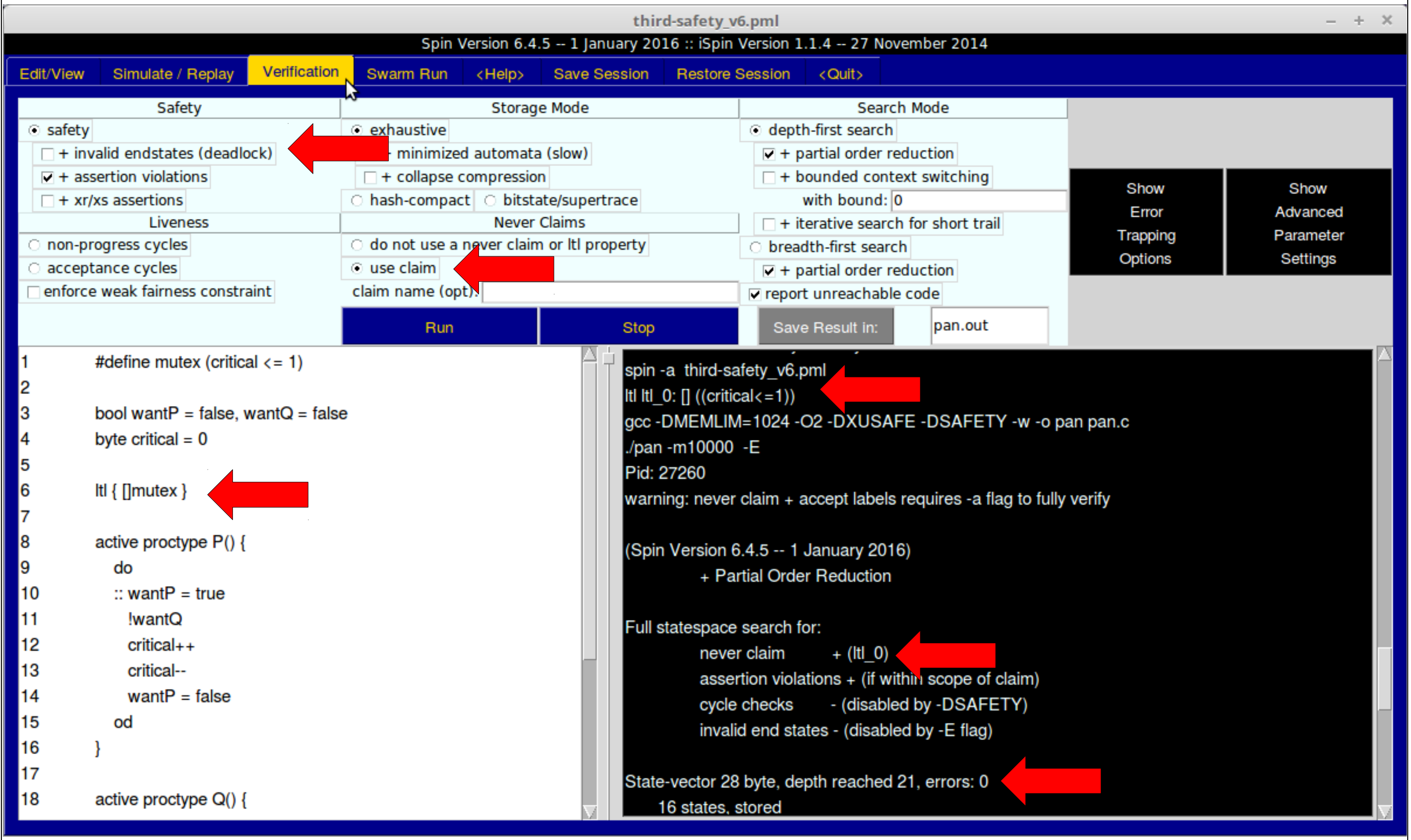
Never Claims
● do not use a never claim or ltl property
○ use claim
claim name (opt):

Search Mode
● depth-first search
☑ + partial order reduction
☐ + bounded context switching
with bound: 0
☐ + iterative search for short trail
○ breadth-first search
☑ + partial order reduction
☑ report unreachable code

pan.out

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

bool wantP = false, wantQ = false
byte critical = 0

ltl { []mutex }

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

active proctype Q() {
```

```
spin -a  third-safety_v6.pml
ltl ltl_0: [] ((critical<=1))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000  -E
Pid: 27122

(Spin Version 6.4.5 -- 1 January 2016)
        + Partial Order Reduction

Full statespace search for:
        never claim         - (not selected)
        assertion violations +
        cycle checks        - (disabled by -DSAFETY)
        invalid end states - (disabled by -E flag)

State-vector 20 byte, depth reached 10, errors: 0
      16 states, stored
       9 states, matched
```


third-safety_v6.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Show Error Trapping Options
Show Advanced Parameter Settings
Run
Stop
Save Result in:

Safety
● safety
☐ + invalid endstates (deadlock)
☑ + assertion violations
☐ + xr/xs assertions

Liveness
○ non-progress cycles
○ acceptance cycles
☐ enforce weak fairness constraint

Storage Mode
● exhaustive
☐ + minimized automata (slow)
☐ + collapse compression
○ hash-compact ○ bitstate/supertrace

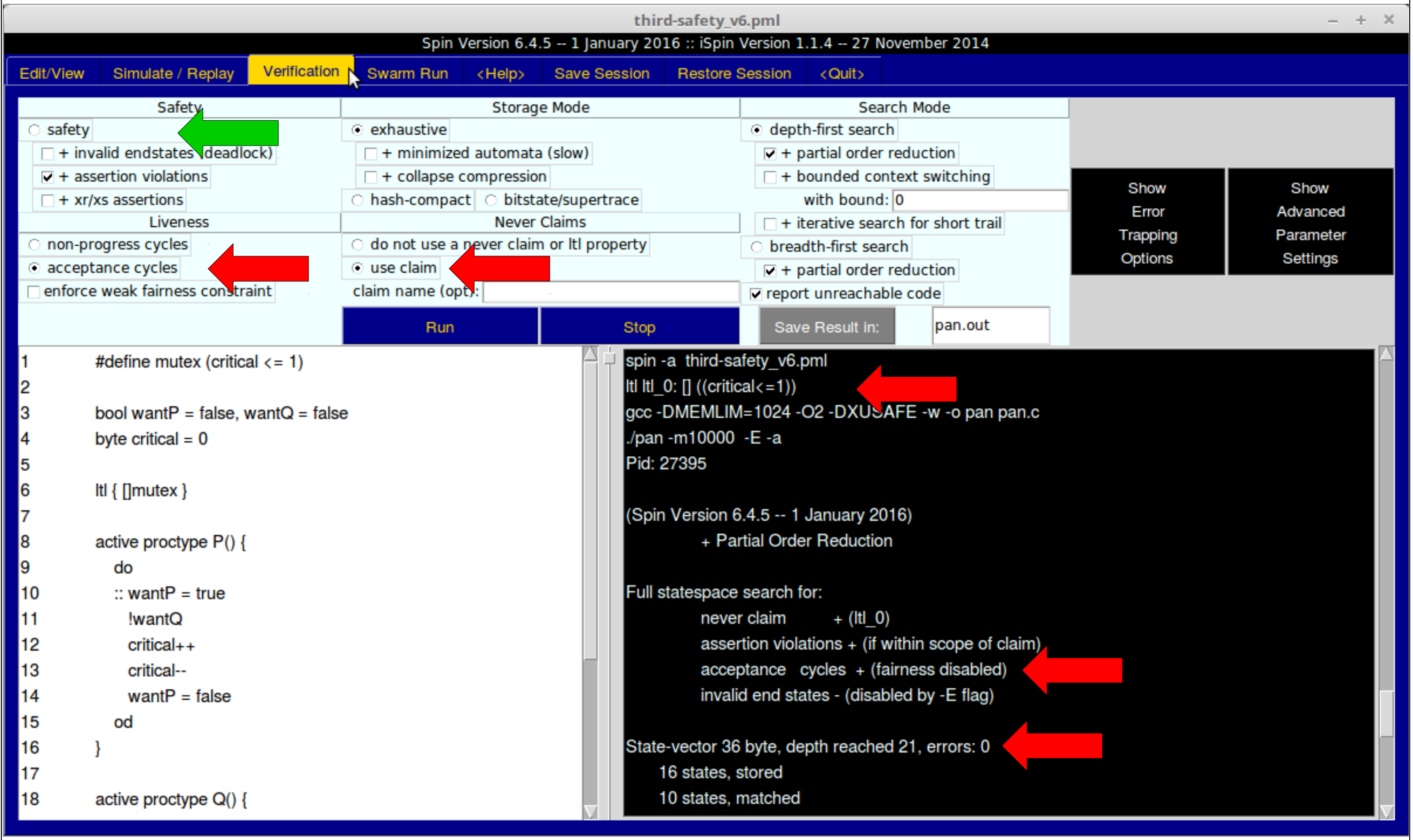
Never Claims
○ do not use a never claim or ltl property
● use claim
claim name (opt):

Search Mode
● depth-first search
☑ + partial order reduction
☐ + bounded context switching
with bound: 0
☐ + iterative search for short trail
○ breadth-first search
☑ + partial order reduction
☑ report unreachable code

pan.out

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

bool wantP = false, wantQ = false
byte critical = 0

ltl { []mutex }

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

active proctype Q() {
```

```
spin -a  third-safety_v6.pml
ltl ltl_0: [] ((critical<=1))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -w -o pan pan.c
./pan -m10000  -E
Pid: 27260
warning: never claim + accept labels requires -a flag to fully verify

(Spin Version 6.4.5 -- 1 January 2016)
        + Partial Order Reduction

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 -E flag)

State-vector 28 byte, depth reached 21, errors: 0
      16 states, stored
```


third-safety_v6.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Show Error Trapping Options
Show Advanced Parameter Settings
Run
Stop
Save Result in:

Safety
- ○ safety
- □ + invalid endstates (deadlock)
- ☑ + assertion violations
- □ + xr/xs assertions

Liveness
- ○ non-progress cycles
- ● acceptance cycles
- □ enforce weak fairness constraint

Storage Mode
- ● exhaustive
- □ + minimized automata (slow)
- □ + collapse compression
- ○ hash-compact ○ bitstate/supertrace

Never Claims
- ○ do not use a never claim or ltl property
- ● use claim
- claim name (opt):

Search Mode
- ● depth-first search
- ☑ + partial order reduction
- □ + bounded context switching
- with bound: 0
- □ + iterative search for short trail
- ○ breadth-first search
- ☑ + partial order reduction
- ☑ report unreachable code

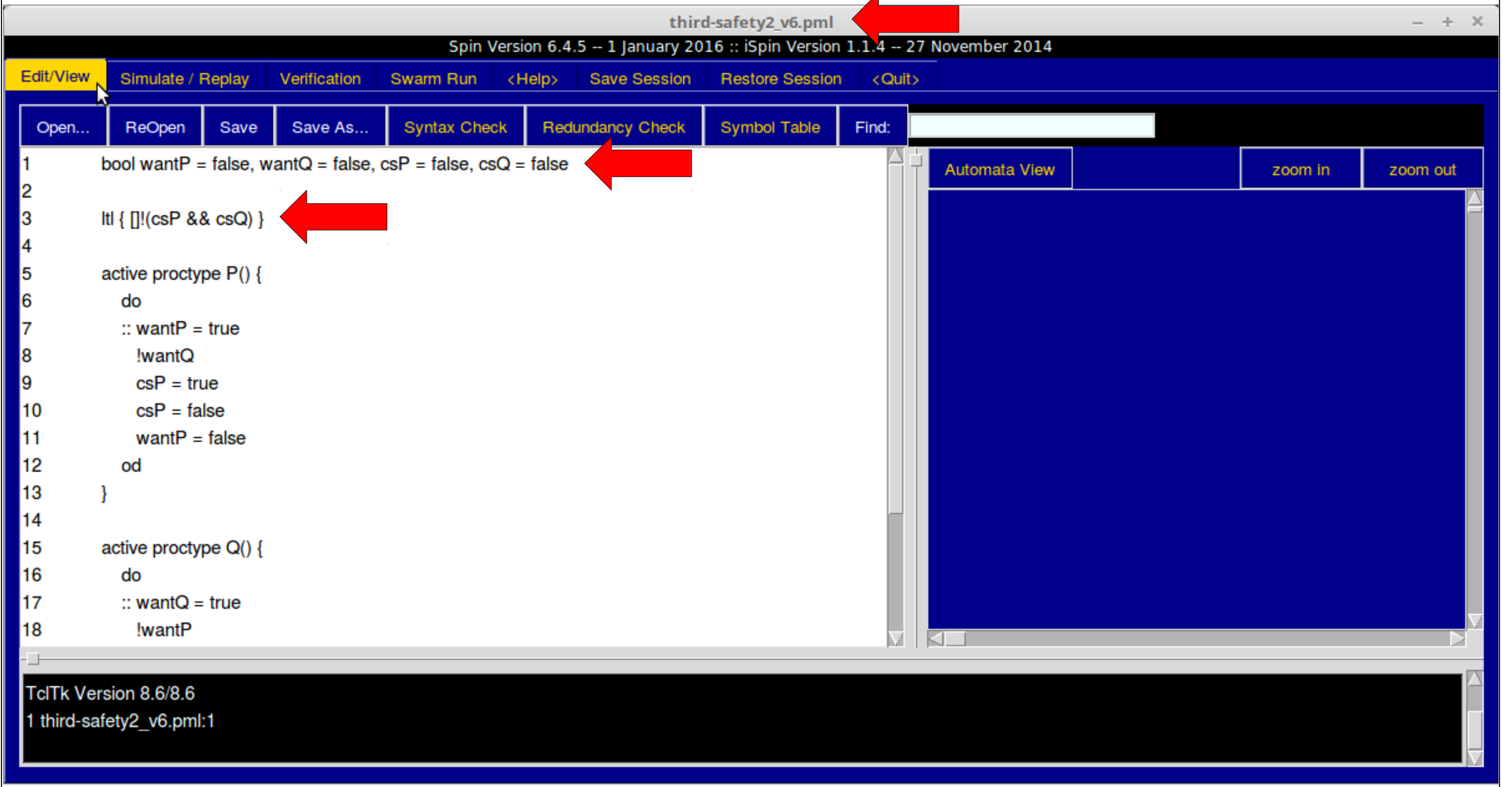
Save Result in: pan.out

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

bool wantP = false, wantQ = false
byte critical = 0

ltl { []mutex }

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

active proctype Q() {
```

```
spin -a  third-safety_v6.pml
ltl ltl_0: [] ((critical<=1))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -w -o pan pan.c
./pan -m10000  -E -a
Pid: 27395

(Spin Version 6.4.5 -- 1 January 2016)
        + Partial Order Reduction

Full statespace search for:
        never claim         + (ltl_0)
        assertion violations + (if within scope of claim)
        acceptance   cycles  + (fairness disabled)
        invalid end states - (disabled by -E flag)

State-vector 36 byte, depth reached 21, errors: 0
     16 states, stored
     10 states, matched
```

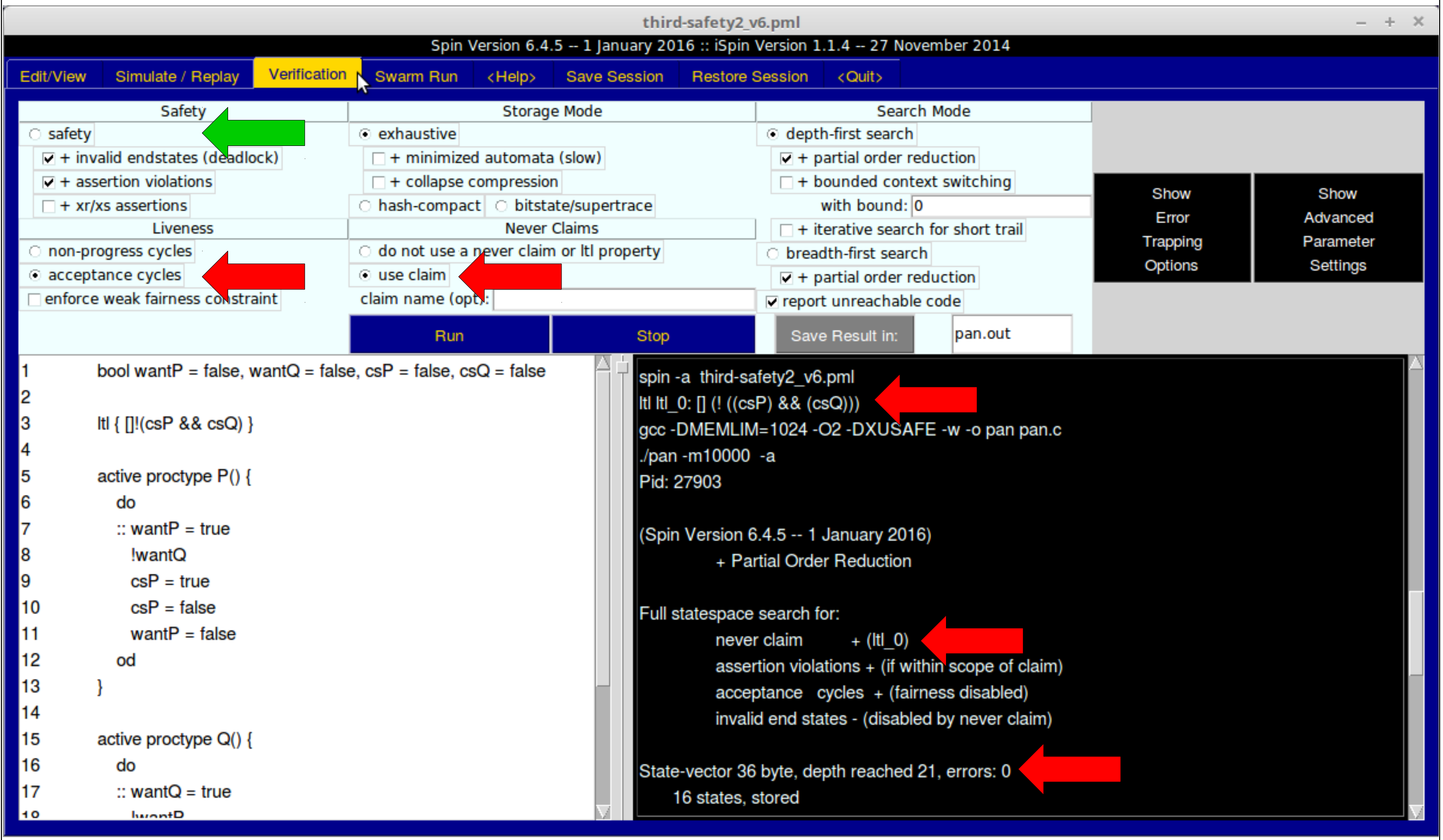
Prepare el siguiente programa:

```
$ cat -n third-safety2_v6.pml
     1  bool wantP = false, wantQ = false, csP = false, csQ = false
     2
     3  ltl { []!(csP && csQ) }
     4
     5  active proctype P() {
     6      do
     7      :: wantP = true
     8         !wantQ
     9         csP = true
    10         csP = false
    11         wantP = false
    12      od
    13  }
    14
    15  active proctype Q() {
    16      do
    17      :: wantQ = true
    18         !wantP
    19         csQ = true
    20         csQ = false
    21         wantQ = false
    22      od
    23  }
```



```
bool wantP = false, wantQ = false, csP = false, csQ = false

ltl { []!(csP && csQ) }

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

active proctype Q() {
   do
   :: wantQ = true
      !wantP
```

```
TclTk Version 8.6/8.6
1 third-safety2_v6.pml:1
```


third-safety2_v6.pml
Spin Version 6.4.5 -- 1 January 2016 :: iSpin Version 1.1.4 -- 27 November 2014
Edit/View
Simulate / Replay
Verification
Swarm Run
<Help>
Save Session
Restore Session
<Quit>
Show Error Trapping Options
Show Advanced Parameter Settings
Run
Stop

Safety
○ safety
☑ + invalid endstates (deadlock)
☑ + assertion violations
□ + xr/xs assertions

Liveness
○ non-progress cycles
● acceptance cycles
□ enforce weak fairness constraint

Storage Mode
● exhaustive
□ + minimized automata (slow)
□ + collapse compression
○ hash-compact ○ bitstate/supertrace

Never Claims
○ do not use a never claim or ltl property
● use claim
claim name (opt):

Search Mode
● depth-first search
☑ + partial order reduction
□ + bounded context switching
with bound: 0
□ + iterative search for short trail
○ breadth-first search
☑ + partial order reduction
☑ report unreachable code

Save Result in: pan.out

```
bool wantP = false, wantQ = false, csP = false, csQ = false

ltl { []!(csP && csQ) }

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

active proctype Q() {
   do
   :: wantQ = true
```

```
spin -a  third-safety2_v6.pml
ltl ltl_0: [] (! ((csP) && (csQ)))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -w -o pan pan.c
./pan -m10000  -a
Pid: 27903

(Spin Version 6.4.5 -- 1 January 2016)
        + Partial Order Reduction

Full statespace search for:
        never claim          + (ltl_0)
        assertion violations + (if within scope of claim)
        acceptance   cycles  + (fairness disabled)
        invalid end states - (disabled by never claim)

State-vector 36 byte, depth reached 21, errors: 0
     16 states, stored
```