

Topics in Distributed Systems

An Introduction to Distributed Systems Verification

Bernard Boigelot

E-mail : bernard.boigelot@uliege.be
WWW : <https://people.montefiore.uliege.be/boigelot/>
<https://people.montefiore.uliege.be/boigelot/courses/tds>

Introduction

Motivation: Distributed systems are often employed in **critical applications**.



(Source: Airbus)



(Source: TechNode)



(Source: RailEngineer)



(Source: Distrelec)

In such applications, **software defects** can have serious consequences.

Famous (and unfortunate) examples

- Therac-25 (1986-87): 4 deaths, 2 serious injuries.



(Source: medium.com)

- Ariane 5 maiden flight (1996): > 300 M€.



(Source: ESA)

- AT&T long-distance network collapse (9h, 1990): 60 M\$ + indirect costs.



(Source: AT&T)

- AT&T mobile network outage (11h, 2024): > 500 M\$ + indirect costs.

- A400M test flight crash (2015): 4 deaths.



(Source: Wikimedia Commons)

- Uber self-driving car collision with cyclist (2018): 1 death.



(Source: NTSB)

What can be done?

- Approach 1: Testing

Essential, but not sufficient for distributed systems because

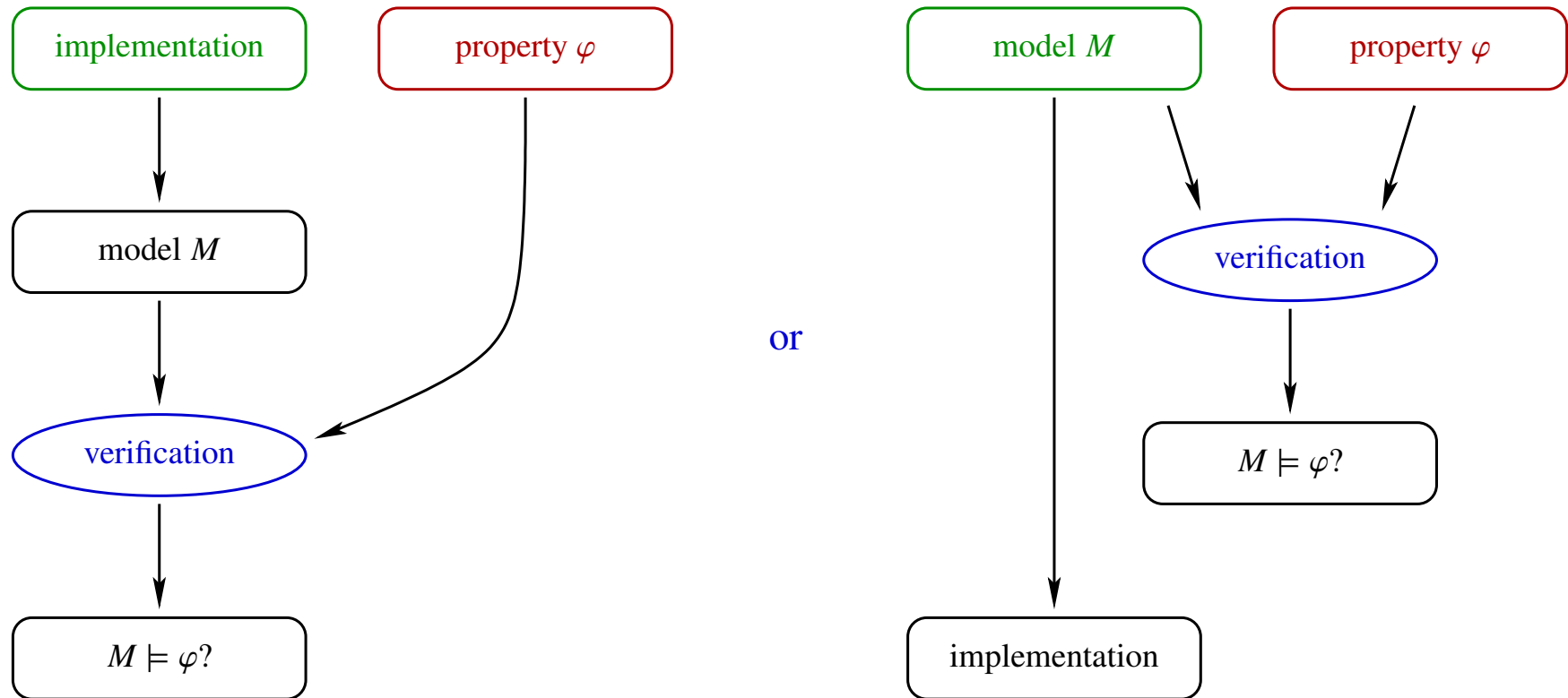
- concurrency, and
- their unpredictable environment

usually make them highly **non-deterministic**.

- Approach 2: Formal proof of correctness

- Tedious and difficult.
- No guarantee that the proofs hold for **actual implementations**.

Other approach: Automated verification



Principles

- The **model** M describes the system at some level of abstraction.
- The **property** φ is either
 - generic: absence of deadlocks, assertion violations, buffer overflows, arithmetic exceptions, . . . , or
 - specific to the application:
 - * **safety** properties: bad things never happen,
 - * **liveness** properties: good things eventually happen.
- The aim is to cover 100% of the model's behaviors.

Notes:

- This usually does not guarantee that **actual implementations** will be 100% error-free.
- The real goal is to **find elusive bugs** that are missed by testing.

Case study: LeLann-Chang-Roberts's leader election algorithm on a ring

(Source: Pascal Fontaine's lectures)

```
process i
1   is_leader ← ⊥
2   max_id ← idi
3   next!idi
4   while ⊤ do
5       prev?j
6       if j = idi then
7           is_leader ← ⊤
8           next!idi
9           prev?j
10          return
11      if j = max_id then
12          next!j
13          return
14      if j > max_id then
15          max_id ← j
16          next!j
```

Setting:

- At first, we look at a **safety** property: **Is the elected leader always unique?**

Note: We will later check whether a leader is **always elected**.

- The verification strategy is **exhaustive state-space exploration**.
- To obtain a **finite-state** model, we will fix
 - the number of processes participating in the election, and
 - the capacity of the communication channels.
- The ID of each node will be assigned **non-deterministically**.
- The goal is to cover **all possible interleavings** of process actions.

States

Definition: A **state** of the model is the collection of all data characterizing its possible **future behaviors**.

For our case study, a state is composed of

- a **control location** (i.e., line number in the code) for each process,
- a value for the **variables** id , max_id , is_leader and j of each process, and
- the content of the **communication channels**.

The **initial** states are such that

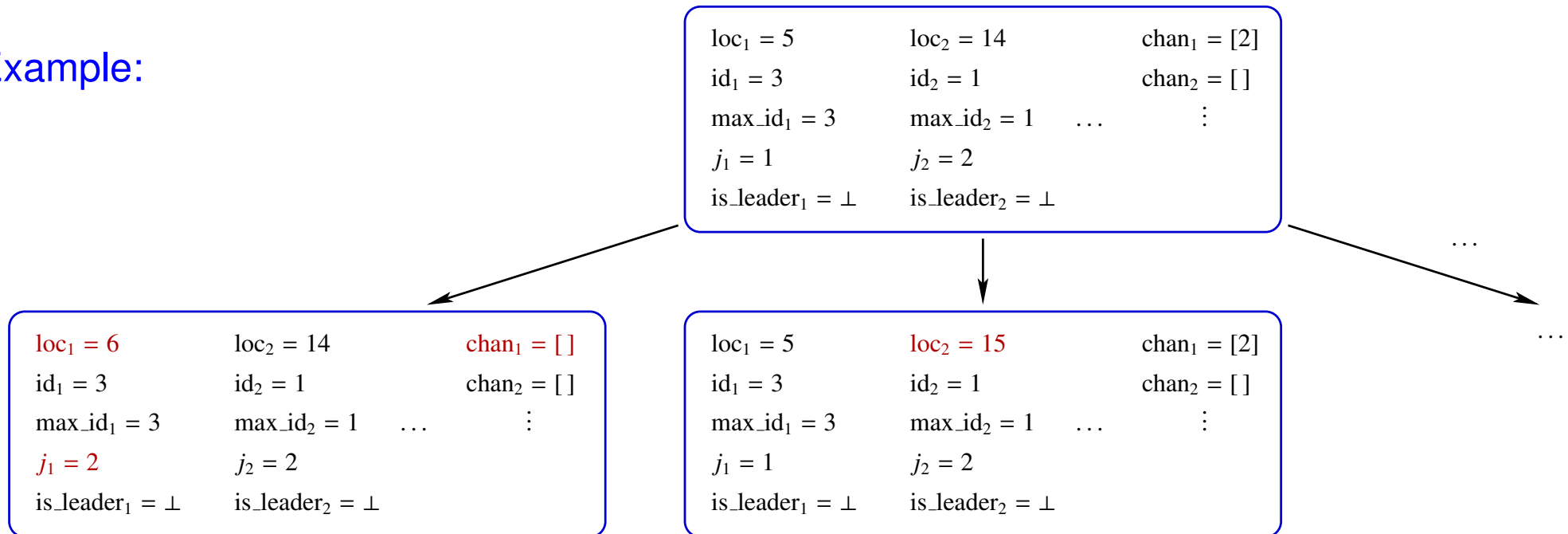
- each process is in its first control location,
- each variable has its specified initial value,
- each communication channel is empty.

Transitions

The current state of the model changes by following **transitions**.

- We consider the **interleaving** model of concurrency, in which each transition corresponds to performing one **atomic operation** in a single process.
- The transitions that can be followed from a state s are said to be **enabled** in that state. The set of such transitions is written $enabled(s)$.

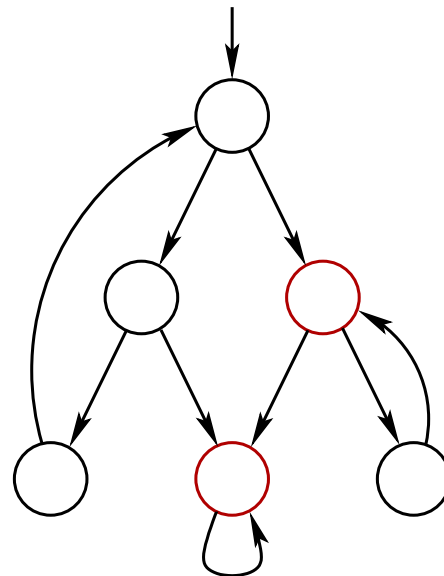
Example:



State-space exploration

Principles:

- One explores one by one the **reachable** states of the model (i.e., those that can be reached by following finitely many transitions from an initial state).
- For each reachable state, one checks whether the **safety properties** are satisfied.
- One remembers the **already visited** states, in order to consider them only once.
- If a **error state** is explored, one extracts an execution trace that allows to reproduce the corresponding bug.



State-space exploration algorithm: Depth-First Search (DFS)

```
procedure DFS():
```

```
   $St \leftarrow []$ 
```

```
   $H \leftarrow \emptyset$ 
```

```
  for all initial  $s_0$ :
```

```
    explore( $s_0$ )
```

```
procedure explore( $s$ ):
```

```
  St.push( $s$ )
```

```
   $H \leftarrow H \cup \{s\}$ 
```

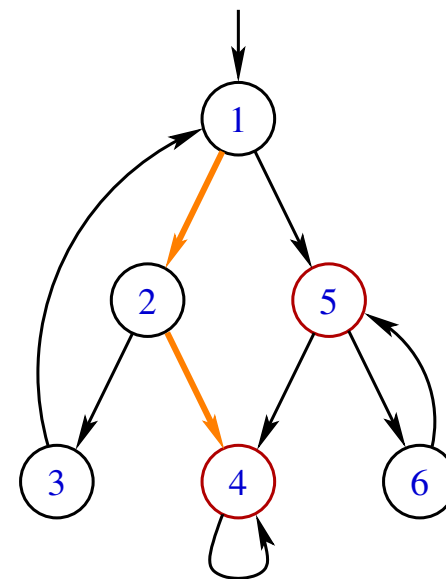
```
  for all  $t \in \text{enabled}(s)$ :
```

```
     $s' \leftarrow \text{succ}(s, t)$ 
```

```
    if  $s' \notin H$  then:
```

```
      explore( $s'$ )
```

```
  St.pop()
```

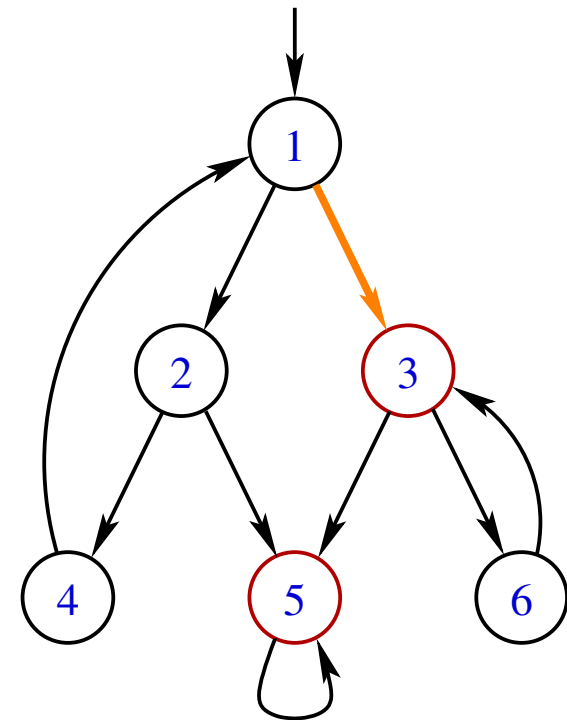


Notes:

- The function $succ(s, t)$ returns the state reached by following the transition t from s .
- The stack S_t does not affect the execution of the algorithm. Its purpose is to provide an **execution trace** that shows how an error state can be reached.
- The **transition relation** does not need to be stored: the state-space graph is generated **on-the-fly**.

Alternative exploration strategy: Breadth-First Search (BFS)

```
procedure BFS():  
   $H \leftarrow \emptyset$   
   $S \leftarrow \emptyset$   
  for all initial  $s_0$ :  
     $S \leftarrow S \cup \{s_0\}$   
  while  $S \neq \emptyset$ :  
     $H \leftarrow H \cup S$   
     $S' \leftarrow \emptyset$   
    for all  $s \in S$ :  
      for all  $t \in \text{enabled}(s)$ :  
         $s' \leftarrow \text{succ}(s, t)$   
        if  $s' \notin H$  then:  
           $S' \leftarrow S' \cup \{s'\}$   
     $S \leftarrow S'$ 
```



Notes:

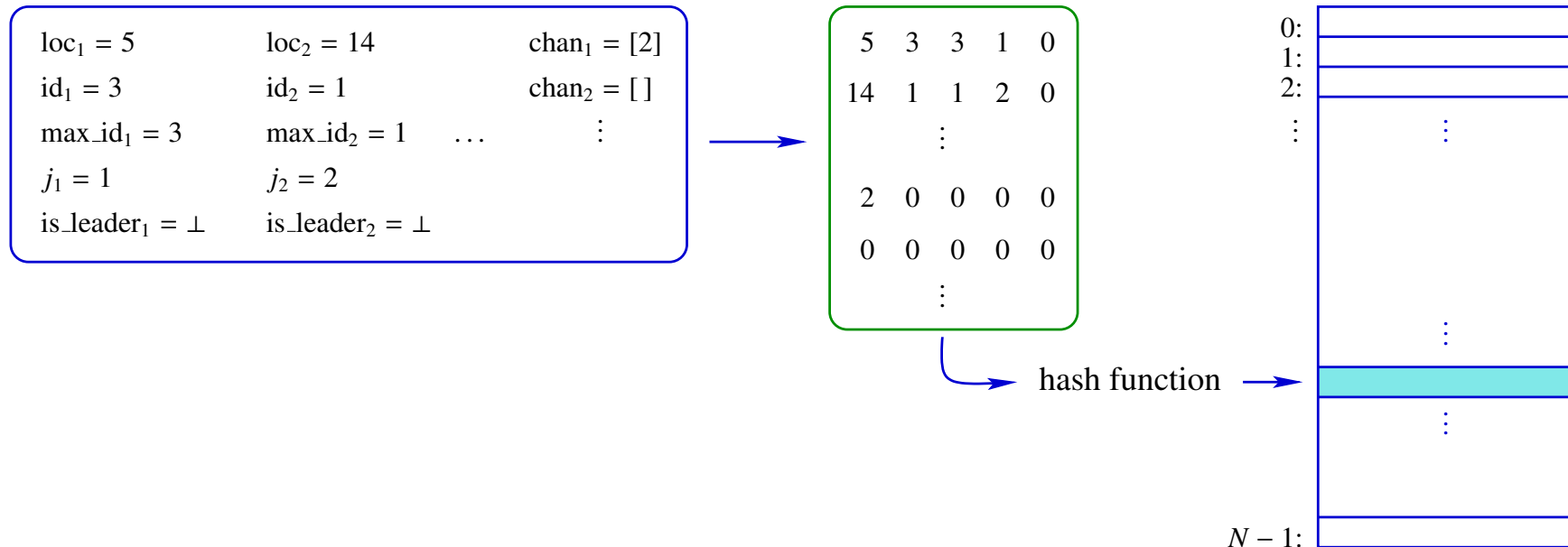
- This version is more **memory-consuming**, since one has to store the current state set S in addition to the set H of already visited states.
- It is however able to produce the **shortest trace** leading to an error state.

Storing the visited states

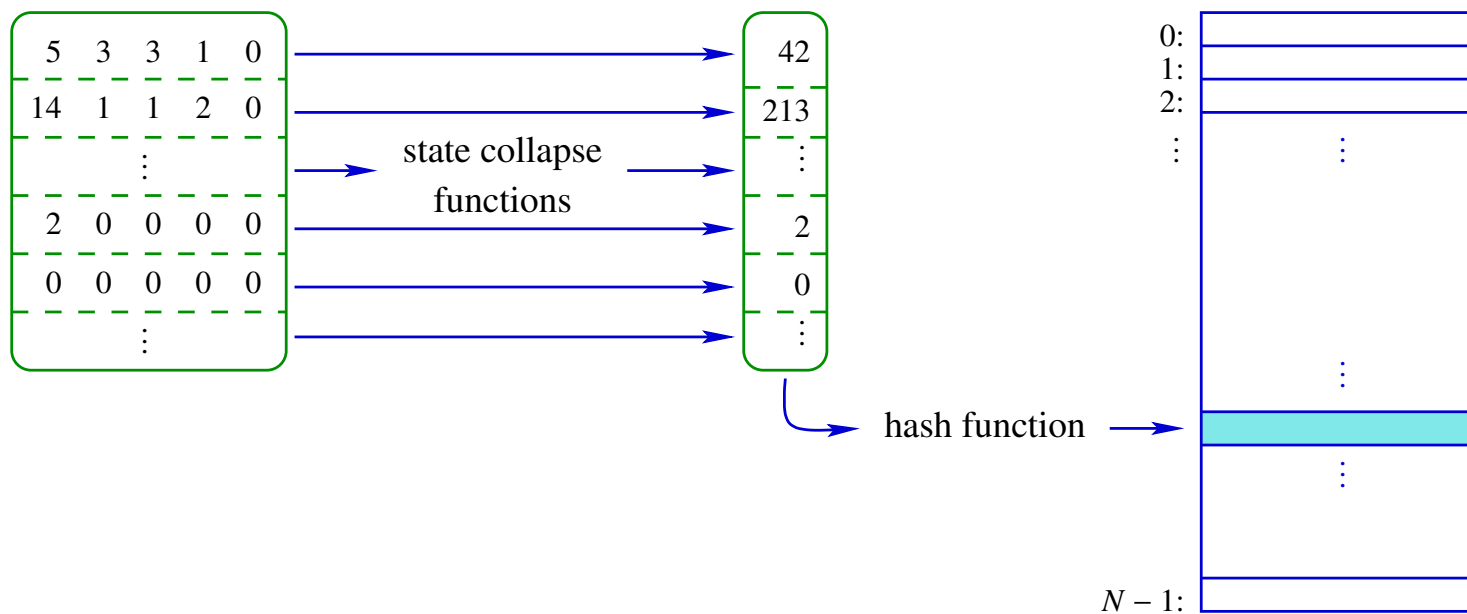
The main bottleneck is the **size of the data structure** representing the set H of already visited states. It should be such that

- **adding** a new state s to H , or **checking** whether $s \in H$, are both performed in average $O(1)$ time.
- this data structure requires **as little memory** as possible.

Option 1: Explicitly store the **state vector** in a hash table.



Option 2: Use an additional structure encoding **parts of the state vector** (for instance, the state of each process and each channel) with a reduced amount of information.



(**Motivation:** a process with a n -bit state vector very often has **much less than 2^n** reachable configurations.)

Option 3: Use a **probabilistic** approach.

Idea (naive):

- Store in the hash table only **one bit** per visited state.
- Make the table big enough for the probability of a **collision** to be negligible.
- The **guarantee of correctness** is lost, but the method is still useful for finding bugs.

Problem: The **birthday paradox** makes this solution unfeasible:

If m values are stored in a hash table of size N , the probability of a collision is

$$p_c(m, N) = 1 - \left(1 - \frac{1}{N}\right) \left(1 - \frac{2}{N}\right) \cdots \left(1 - \frac{m-1}{N}\right) = 1 - \frac{N!}{N^m (N-m)!}.$$

For large N , we have $1 - \frac{1}{N} \approx e^{-\frac{1}{N}}$, which yields

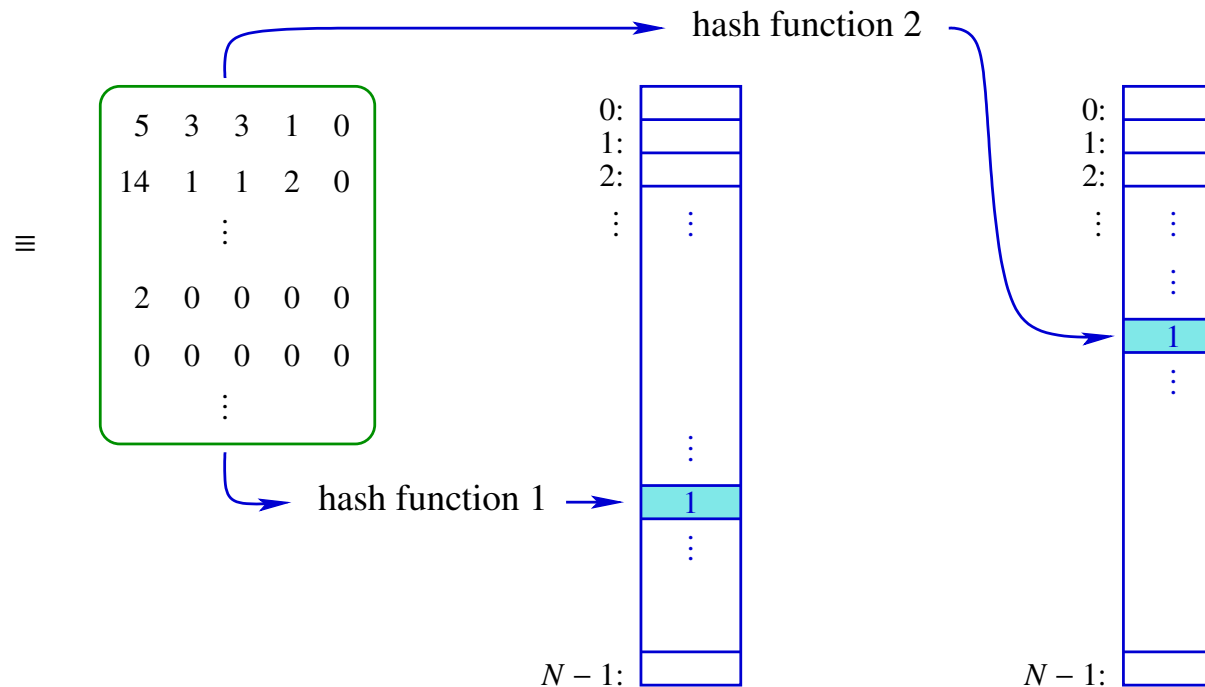
$$p_c(m, N) \approx 1 - e^{-\frac{m^2}{2N}}.$$

Examples:

- $N = 365, m = 40: p_c(m, N) \approx 89\%$.
- $N = 10^9, m = 10^6: p_c(m, N) \approx 99,999999 \dots \%$ (with 217 nines).
- For $m = 10^6$, the smallest value of N that gives $p_c(m, N) < 10^{-3}$ is $\approx 5 \cdot 10^{14}$, corresponding to a hash table of ≈ 57 TB.

Note: It is important to avoid collisions as much as possible, since the **successors** of states wrongly marked as being already visited are not explored.

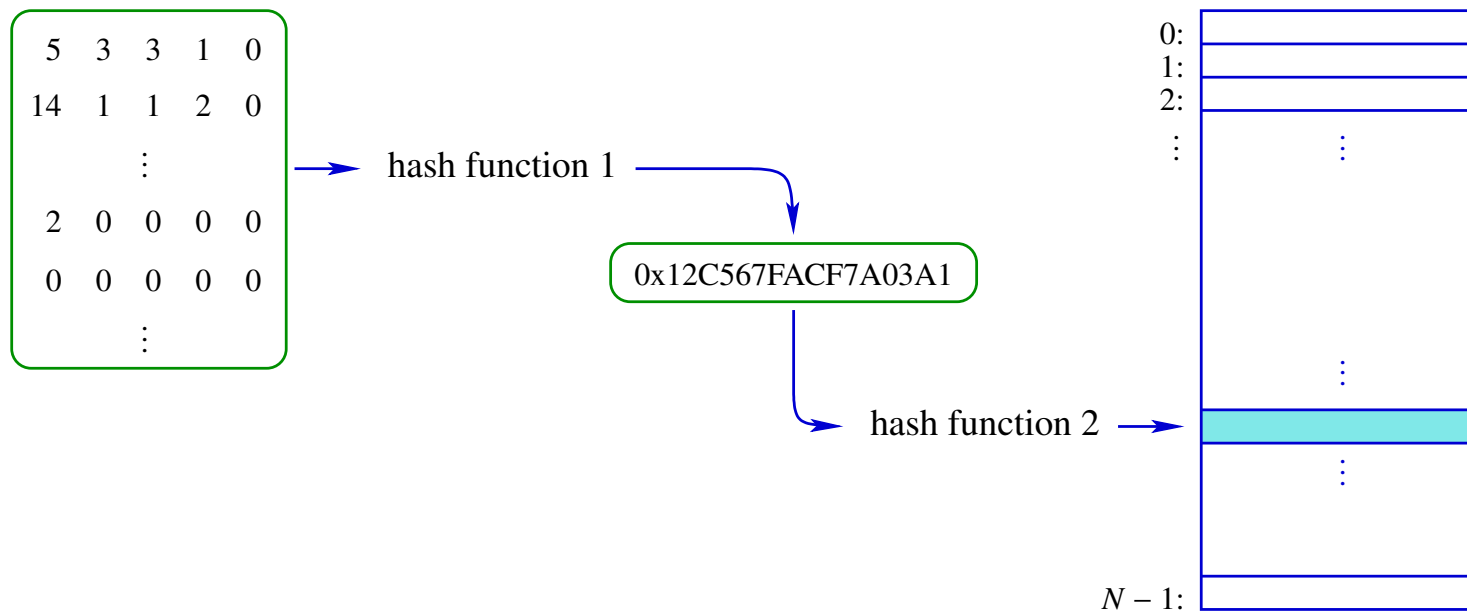
Solution 1: Use **multiple** independent hash functions (**bitstate** hashing).



Drawbacks:

- Computation time increases linearly with the number of hash functions.
- The **number of hash functions** needed to bring the probability of collision sufficiently low can be large.

Solution 2: **Compress** the state vector into a small-size descriptor, and store this descriptor into a hash table (**hashcompact**).



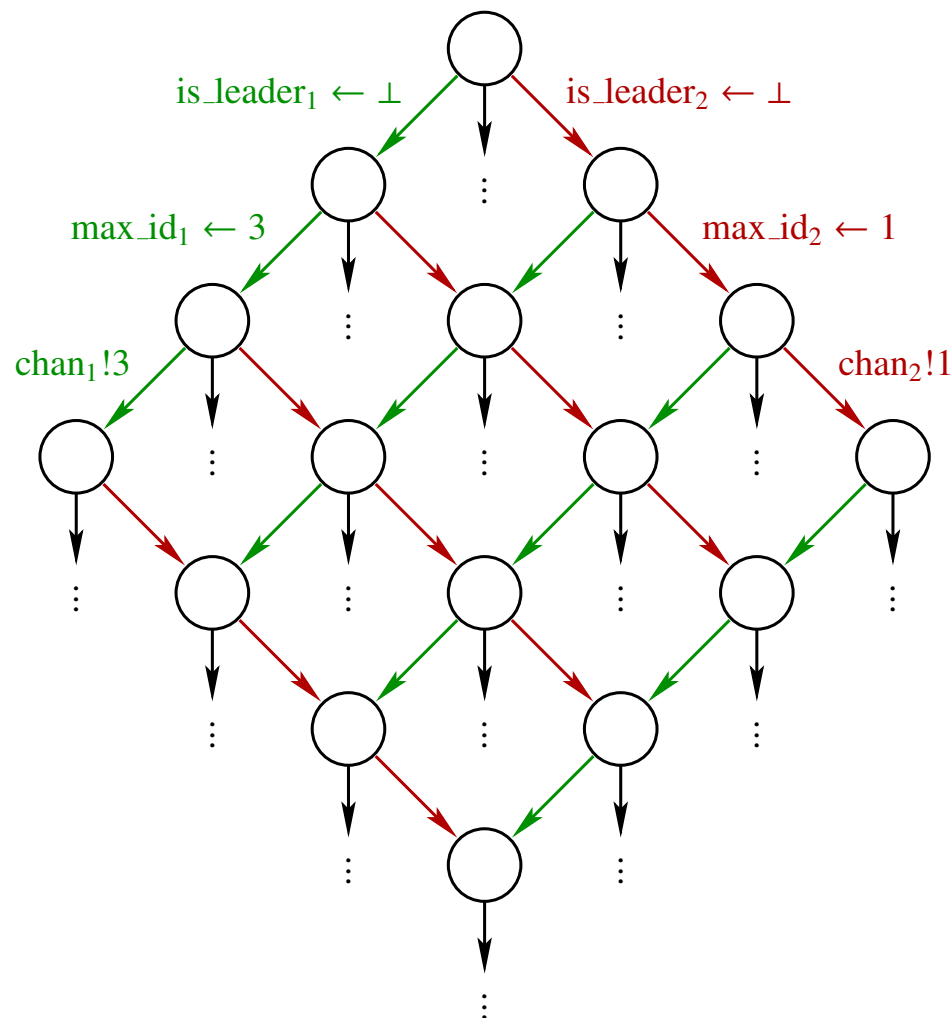
Example: With $m = 10^9$ and 64-bit descriptors ($N = 2^{64}$), one gets $p_c(m, N) \approx 2,7\%$. With 80-bit descriptors, $p_c(m, N) \approx 0,00004\%$.

Reducing the search space

Sometimes, exploring **all interleavings** of the operations performed by processes is highly redundant.

```

process  $i$ 
1    $\text{is\_leader} \leftarrow \perp$ 
2    $\text{max\_id} \leftarrow \text{id}_i$ 
3    $\text{next!id}_i$ 
4   while  $\top$  do
5      $\text{prev?}j$ 
6     if  $j = \text{id}_i$  then
7        $\text{is\_leader} \leftarrow \top$ 
8        $\text{next!id}_i$ 
9        $\text{prev?}j$ 
10      return
11     if  $j = \text{max\_id}$  then
12        $\text{next!}j$ 
13       return
14     if  $j > \text{max\_id}$  then
15        $\text{max\_id} \leftarrow j$ 
16        $\text{next!}j$ 
  
```



The Spin tool

Spin is a model checker for **finite-state** concurrent models.

Main features:

- Can check assertion violations, deadlocks, unreachable code, and verify liveness properties expressed in **Linear-Time temporal Logic (LTL)** (more on that later ...).
- Several options for **storing visited states**:
 - hash table.
 - collapsed representation.
 - bitstate hashing and hashcompact (possibly incomplete search).
 - minimized automaton (memory efficient and exact, but much slower).
- Choice of two **search strategies**:
 - depth-first search (DFS).
 - breadth-first search (BFS).

- **Partial-order reductions** for reducing the search space.
- Simple **graphical front-end**: iSpin.

For installation instructions, documentation, and examples: <https://spinroot.com>.

The Promela modeling language

- **Processes** are defined by proctype declarations, and are created dynamically using the run statement.

Note: Processes corresponding to active proctypes are automatically created from the beginning.

- Processes communicate via **shared variables** and **communication channels**.
- An init declaration defines an **initial process**, that typically
 - initializes global variables, and
 - creates other processes.

Example:

```
int x1, x2;
```

```
proctype p1()
```

```
{
```

```
    printf("p1, x1: %d\n", x1)
```

```
}
```

```
proctype p2()
```

```
{
```

```
    printf("p2, x2: %d\n", x2)
```

```
}
```

```
init
```

```
{
```

```
    x1 = 1;
```

```
    x2 = 1;
```

```
    run p1();
```

```
    run p2()
```

```
}
```

Data types and variable declarations

Primitive:

<code>bool</code> or <code>bit</code>	1 bit	$[0, 1]$
<code>byte</code>	8 bits	$[0, 255]$
<code>short</code>	16 bits	$[-2^{15}, 2^{15} - 1]$
<code>int</code>	32 bits	$[-2^{31}, 2^{31} - 1]$

```
byte c1, c2 = 2, c3;
```

Arrays

```
bool table[16];  
int v[8] = 1;
```

Symbolic constants

```
mtype = { MSG, ACK, NACK };
```

```
mtype message = MSG;
```

Communication channels:

```
chan xmit = [3] of { mtype, short };
```

```
xmit!MSG,10;
```

```
xmit?MSG,nb;
```

Structures:

```
typedef message
```

```
{
```

```
    bit b[8];
```

```
    int nb
```

```
};
```

```
message m;
```

Statements

- The body of a proctype takes the form of a **sequence of statements**. Statements are separated by semicolons (“;”).
- At any moment during execution, a statement is either
 - **executable**, or
 - **blocked**.
- An **assignment** (e.g., $x = a + b$) is always executable.
- An **expression** (e.g., $a + b$) can also be used as a statement. It is executable if its evaluation returns a non-zero value.

Special statements

- `skip` is always executable, and does **nothing**.
- `run` instantiates a **new process**. Such a statement is executable only if the maximum number of processes has not yet been reached.
- `printf` is always executable, and displays **debugging information**. This statement has no effect during verification.

Example:

```
int x1 = 0;
```

```
init
```

```
{
```

```
  run p1(10);
```

```
  x1 != 0; // to become executable, another process must modify x1
```

```
  skip
```

```
}
```

- `assert` is always executable. It evaluates an expression, and **generates an error** if the returned value is equal to 0.

Example: `assert(nb <= 10)`

The if statement

Syntax:

```
if
  :: guard1 -> instructions1;
  :: guard2 -> instructions2;
  :
fi
```

Notes:

- This statement selects **non-deterministically** one sequence

$\text{guard } i \rightarrow \text{instructions } i$

among those for which $\text{guard } i$ is executable.

- If no guard is executable, then the if statement itself is **not executable**.
- There exists a special guard `else` that becomes executable only if **none of the other guards** is executable.

- The “->” symbol is **equivalent to “;”**. It is used by convention for separating the guards from the instructions.
- There is no need for the guards to be **mutually exclusive**.
- There is also a special guard timeout that only becomes executable if **no other process** is executable in the current state.

The do statement

Syntax:

```
do
  :: guard1 -> instructions1;
  :: guard2 -> instructions2;
  :
od
```

Notes:

- The modalities are similar to those of the `if` statement. The difference is that the `do` statement **repeats the operation** after each execution of a sequence `guard i -> instructions i` .
- The instruction `break` makes it possible to **exit the loop**. This instruction is always executable.
- Another possibility of exiting the loop is to use the `goto` instruction.

Atomicity

In Promela, every individual statement is executed **atomically**. Sequences of operations, however, can be interleaved with operations performed by other processes. There are two ways to modify this default mode of execution:

- The statement

```
atomic { instructions }
```

attempts to execute instructions **without interleaving operations** from other processes.

Notes:

- This statement is executable if the **first statement** of instructions is executable.
- If a subsequent statement of instructions becomes blocked, **atomicity is lost**.

- The statement

```
d_step { instructions }
```

is similar to an atomic block, but

- executes instructions in a **single step**, without generating intermediate states,
- imposes that the block instructions is executed **deterministically**,
- does not allow to **jump** in or out of instructions,
- does not allow any statement inside instructions except the first one to become **blocked**.

Note: Very often, the use of atomic and d_step blocks makes it possible to **greatly reduce** the search space.

Example: Simple mutual exclusion

```
int s = 1;
int nb = 0;

proctype p()
{
    do
        :: skip ->
            atomic { s > 0; s-- }
            nb++;
            assert(nb == 1);
            nb--;
            s++;
    od
}

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

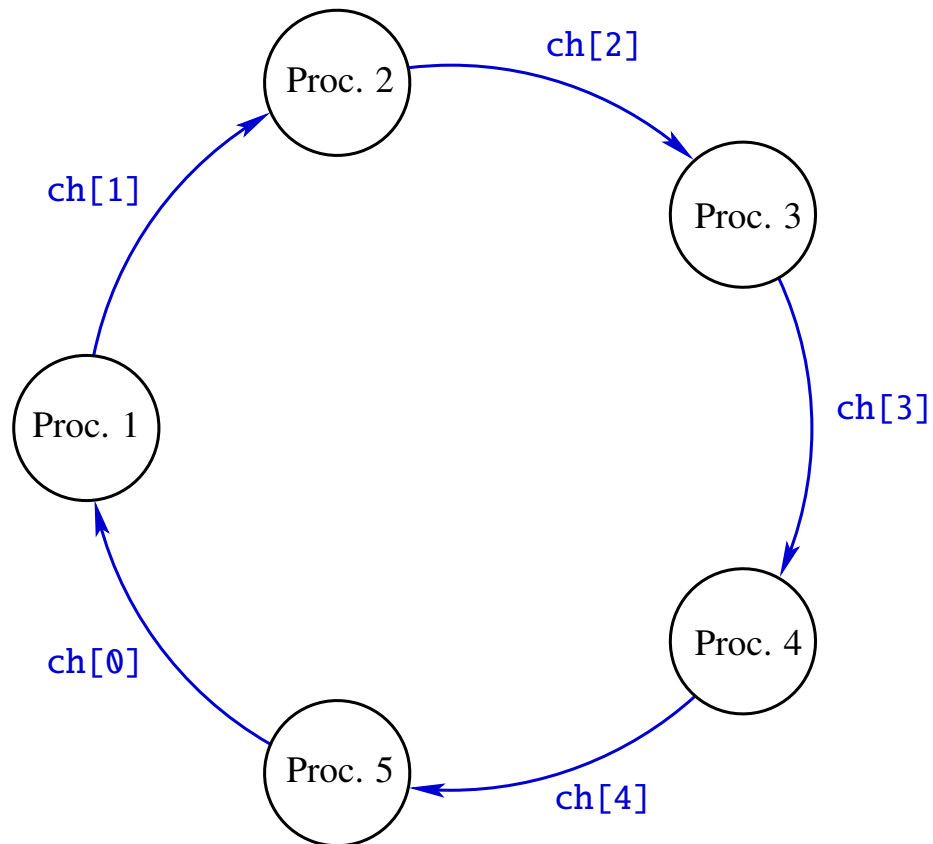
Modeling LeLann-Chang-Roberts's algorithm

Principles:

- Fixed number of processes, each running the **same code**.
- The communication channels are stored in a **global array**.

```
#define NB_NODES 5
#define CAPACITY 5

chan ch[NB_NODES] =
    [CAPACITY] of { byte };
```



Instantiating the processes

Problem: Assigning **sequential numbers** to the processes (which they need to know to access their input and output channels).

Solution:

```
init
{
  atomic
  {
    byte i = 0;

    do
      :: i < NB_NODES ->
        i++;
        run node(i)

      :: i == NB_NODES -> break
    od
  }
}
```

Assigning process IDs

Problem: How can we make sure that **all possible assignments** of process IDs are considered?

Solution:

- Let each process choose its own ID **non-deterministically**.
- Use a global bitfield for enforcing that all process IDs are **distinct**.

```
bit id_taken[NB_NODES] = 0;

proctype node(byte num)
{
    byte id = 1;

    do
        :: id < NB_NODES -> id++
        :: id > 1 -> id--
        :: atomic { !id_taken[id - 1] -> id_taken[id - 1] = 1 ; break }
    od

    ...
}
```

Specifying the property to verify

Problem: How can we check that the elected leader is always **unique**?

Solution:

- Define a global variable containing the **ID of the elected leader**.
- Before assigning a value to this variable, assert that this has **not yet been done**.

```
byte leader = 0;
```

```
proctype node(byte num)
```

```
{
```

```
...
```

```
    atomic
```

```
    {
```

```
        assert (leader == 0);
```

```
        leader = id
```

```
    }
```

```
...
```

```
}
```

LeLann-Chang-Roberts's: Full Promela model

```
#define NB_NODES 5
#define CAPACITY 5

chan ch[NB_NODES] = [CAPACITY] of { byte };
bit id_taken[NB_NODES] = 0;
byte leader = 0;

proctype node(byte num)
{
    byte id = 1;

    do
        :: id < NB_NODES -> id++
        :: id > 1 -> id--
        :: atomic { !id_taken[id - 1] -> id_taken[id - 1] = 1 ; break }
    od

    byte max_id = id;
    byte j;
```

```

ch[num % NB_NODES] ! id;

do
  :: ch[num - 1] ? j;
  if
    :: j == id ->
      atomic { assert (leader == 0); leader = id }
      ch[num % NB_NODES] ! id;
      ch[num - 1] ? j;
      goto end
    :: else -> skip
  fi;

  if
    :: j == max_id -> ch[num % NB_NODES] ! j; goto end
    :: else -> skip
  fi;

  if
    :: j > max_id ->
      atomic { max_id = j; ch[num % NB_NODES] ! j }
    :: else -> skip
  fi
od;

end: skip
}

```

```
init
{
  atomic
  {
    byte i = 0;

    do
      :: i < NB_NODES ->
        i++;
        run node(i)

      :: i == NB_NODES -> break
    od
  }
}
```

Experiments with Spin

NB_NODES = 5:

```
State-vector 112 byte, depth reached 4768, errors: 0
 2063539 states, stored
 2615680 states, matched
 4679219 transitions (= stored+matched)
 129362 atomic steps
hash conflicts:    151699 (resolved)
```

Stats on memory usage (in Megabytes):

```
275.512 equivalent memory usage for states (stored*(State-vector + overhead))
220.768 actual memory usage for states (compression: 80.13%)
      state-vector as stored = 84 byte + 28 byte overhead
128.000 memory used for hash table (-w24)
  0.534 memory used for DFS stack (-m10000)
349.140 total actual memory usage
```

pan: elapsed time 0.89 seconds

No errors found -- did you verify all claims?

NB_NODES = 6:

(Note: The amount of usable memory and the maximum search depth have to be increased from the default values.)

```
State-vector 128 byte, depth reached 66116, errors: 0
 95050831 states, stored
1.3719902e+08 states, matched
2.3224985e+08 transitions (= stored+matched)
 6260876 atomic steps
hash conflicts: 1.30046e+08 (resolved)
```

```
Stats on memory usage (in Megabytes):
14141.016 equivalent memory usage for states (stored*(State-vector + overhead))
11715.116 actual memory usage for states (compression: 82.84%)
      state-vector as stored = 101 byte + 28 byte overhead
 512.000 memory used for hash table (-w26)
  5.341 memory used for DFS stack (-m100000)
12232.352 total actual memory usage
```

```
pan: elapsed time 78.2 seconds
No errors found -- did you verify all claims?
```

With collapse compression:

```
State-vector 128 byte, depth reached 66116, errors: 0
 95050831 states, stored
1.3719902e+08 states, matched
2.3224985e+08 transitions (= stored+matched)
 6260876 atomic steps
hash conflicts: 1.0634934e+08 (resolved)
```

Stats on memory usage (in Megabytes):

```
14141.016 equivalent memory usage for states (stored*(State-vector + overhead))
 4493.540 actual memory usage for states (compression: 31.78%)
    state-vector as stored = 22 byte + 28 byte overhead
 512.000 memory used for hash table (-w26)
 5.341 memory used for DFS stack (-m100000)
5009.989 total actual memory usage
```

pan: elapsed time 103 seconds

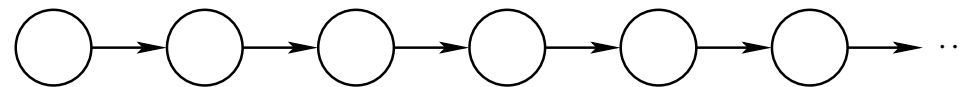
No errors found -- did you verify all claims?

Note: Our model is far from being optimal! For instance, the state space could be reduced by a factor equal to NB_NODES by exploiting **symmetry**: the ID of the first process could be set to 1 without loss of generality.

Beyond reachability properties

To specify **liveness properties**, one uses **temporal logics**, which express properties of infinite sequences of states.

- They extend **propositional** or **first-order** logic.
- In this introduction, we present **Linear-time Temporal Logic (LTL)** on discrete time.
 - A LTL formula is interpreted on states belonging to a **sequence**:

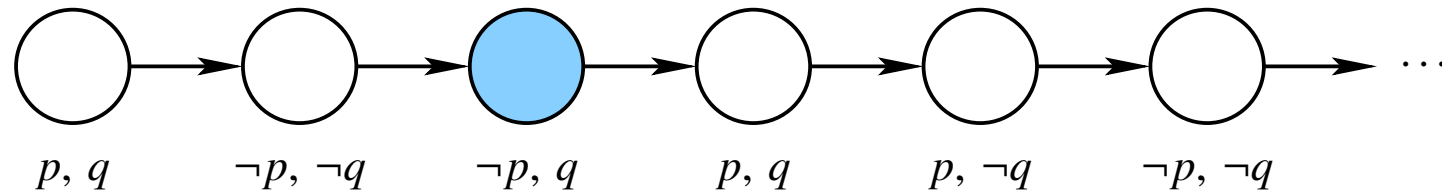


- Each state assigns a truth value to atomic **propositions**.
- **Temporal operators** indicate in which states the components of formulas should be interpreted.

Some temporal operators

- $\bigcirc \varphi$ (Next): φ is true in the next state of the sequence.
- $\square \varphi$ (Always): φ is true in the current and all future states of the sequence.
- $\diamond \varphi$ (Eventually): φ is true in the current or some future state of the sequence.
- $\varphi_1 U \varphi_2$ (Until): φ_1 is true in the current and all future states until φ_2 becomes true, which must occur.

Examples



In the colored state, the following formulas are **true**:

- $\neg p \wedge q$
- $\bigcirc(p \wedge q)$
- $\diamond \neg q$

In that state, the following formulas are **false**:

- $\square q$
- $\bigcirc \square p$

More examples

- $\Box(p \Rightarrow \bigcirc q)$ is satisfied in all states where, for this state and all future ones, each state in which p is true is immediately followed by a state in which q is true.
- $\Box \Diamond p$ is satisfied in all states for which p is true infinitely often in the future.
- $\Diamond \Box p$ is satisfied in all states for which, from some point on in the future, p becomes and remains true.

Verifying LTL properties with Spin

Goal: Checking that LeLann-Chang-Roberts's algorithm always manages to **elect a leader**.

Solution: Add the following LTL property to the model:

```
ltl { eventually (leader != 0) }
```

Notes:

- To be able to verify LTL properties, the options “**acceptance cycles**” and “**use claim**” of Spin must be selected.
- For technical reasons, the temporal operator \bigcirc cannot be used with partial-order reductions.

Result

Warning: Search not completed

State-vector 120 byte, depth reached 23, errors: 1
6 states, stored
0 states, matched
6 transitions (= stored+matched)
12 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):

0.001 equivalent memory usage for states (stored*(State-vector + overhead))
0.286 actual memory usage for states
128.000 memory used for hash table (-w24)
5.341 memory used for DFS stack (-m100000)
133.536 total actual memory usage

pan: elapsed time 0 seconds

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

→ Spin finds an execution that violates the LTL property.

Indeed, our mechanism for **selecting node IDs** allows infinite executions in which such IDs are never assigned:

```
bit id_taken[NB_NODES] = 0;

proctype node(byte num)
{
    byte id = 1;

    do
        :: id < NB_NODES -> id++
        :: id > 1 -> id--
        :: atomic { !id_taken[id - 1] -> id_taken[id - 1] = 1 ; break }
    od

    ...
}
```

Solution:

- Add a **counter** keeping track of how many IDs have been assigned:

```
bit id_taken[NB_NODES] = 0;
byte nb_ids = 0;

proctype node(byte num)
{
    byte id = 1;

    do
        :: id < NB_NODES -> id++
        :: id > 1 -> id--
        :: atomic { !id_taken[id - 1] -> id_taken[id - 1] = 1 ; nb_ids++; break }
    od

    ...
}
```

- Modify the LTL property accordingly:

```
ltl { always ((nb_ids == NB_NODES) -> eventually (leader != 0)) }
```

Result

```
State-vector 120 byte, depth reached 11282, errors: 0
 15528757 states, stored (2.16152e+07 visited)
1.0021168e+08 states, matched
1.2182692e+08 transitions (= visited+matched)
 7538332 atomic steps
hash conflicts: 25327747 (resolved)
```

Stats on memory usage (in Megabytes):

```
2191.788 equivalent memory usage for states (stored*(State-vector + overhead))
1659.886 actual memory usage for states (compression: 75.73%)
    state-vector as stored = 84 byte + 28 byte overhead
128.000 memory used for hash table (-w24)
 5.341 memory used for DFS stack (-m100000)
1792.618 total actual memory usage
```

pan: elapsed time 22.6 seconds

No errors found -- did you verify all claims?

→ Now the LTL property is **validated**.

Further exercises

- In his lectures, Pascal Fontaine asked a few questions about LeLann-Chang-Roberts's algorithm. **Can you answer them using Spin?**

```
process i
1   is_leader ← ⊥
2   max_id ← idi
3   next!idi
4   while ⊤ do
5       prev?j
6       if j = idi then
7           is_leader ←
            ⊤
8           next!idi
9           prev?j
10          return
11      if j = max_id then
12          next!j
13          return
14      if j > max_id then
15          max_id ← j
16          next!j
```

What happens if:

- lines 11-13 are swapped with 14-16?
- line 9 is removed?
- line 12 is removed?
- one (and only one) message is lost at some point?

- Can you harness the **search power** of Spin to solve the following puzzle (**Rush Hour**)?



(Source: Samuel Hiard)

Hints:

- Each car can be modeled by a **separate process**, trying non-deterministically and repeatedly all possible moves.
- The simplest approach is to define one proctype for each **car type**.
- **Two-dimensional arrays** cannot be directly defined in Promela. To represent the free cells on the grid, you can
 - define a typedef corresponding to a row (as an array of Booleans), and
 - represent the grid as an array of rows.
- You will probably have to modify the default parameters (such as the maximum exploration depth).

Other approaches to state-space exploration

- Instead of exploring states one at a time, **symbolic** state-space exploration techniques handle **sets of reachable states**, represented using dedicated data structures.

This makes it possible to explore very large, and even **infinite**, state spaces.

- **Bounded model checking** proceeds by setting a bound on the **exploration depth**, which makes the model finite-state, and increasing this bound until a bug is found.
- **Counter-Example-Guided Abstraction Refinement (CEGAR)** starts from a simple model that **over-approximates** the behaviors of the analyzed system. When a bug is found, the error trace is replayed against the actual system and, if needed, the abstraction is iteratively refined, until either it is validated, or a real bug is found.
- There exist modeling formalisms (and associated exploration algorithms) for dealing with **timed** and **hybrid** systems.
- ...