SPIN

A tool to ***verify*** them all

Matteo Brambilla - Paolo Cerutti - Carlo Chiodaroli, 5/18/2023

What are we going to see?

SPIN

Promela-based tool for analyzing the logical consistency of **concurrent systems**, specifically data communication protocols

- automatically verify whether `` `M ⊨ φ` `` holds, where `` `M` `` is a (finite-state) model of a system and property `` `φ` `` is stated in some formal notation.
- Can be used via command line or via iSpin

With SPIN one may check the following type of properties:

- deadlocks (invalid end-states)
- assertions
- unreachable code
- LTL formulae
- safety and liveness properties
  - non-progress cycles (live-locks)
  - acceptance cycles

Verification Algorithm

SPIN uses a **depth first search algorithm** (DFS) to generate and explore the **complete state space**.

In simple terms:

```
procedure dfs(s: state)
    if error(s) then report error
    add s to stateSpace                         //stateSpace is an hashtable of states
    foreach successor t of s do
        if t not in stateSpace then dfs(t)
end dfs
```

Construction and error checking happen at the same time, this allows SPIN to be an **on-the-fly** model checker.

SPIN then builds a Büchi automaton from the negated LTL formula and from the state space, and checks whether the intersection of the two is empty.

If so, the property holds.

SPIN also has the possibility to use breath first search (BFS) to explore the state space which

Reduction Algorithms

SPIN has several optimizations to make verification more *efficient* and more *effective*:

- **partial order reduction**
  - the validity of a property *does not depend* on the order in which *independent* executed events are *interleaved* so it is sufficient to check one of the possible interleaving
- **bitstate hashing**
  - instead of storing the whole state, only one bit of memory is used to store a reachable state
- **state vector compression**
  - instead of storing the whole state, a compressed version of the state is stored
- **minimization of the Büchi automaton**
  - states are stored in a deterministic automaton that changes dynamically during the verification process (very memory efficient but really slow)
- **dataflow analysis**
  - SPIN can detect when a variable is not used anymore and remove it from the state space

PROMELA

Promela language

C-like language for modeling concurrent systems

- **Identifiers** are strings of letters, digits and underscores, starting with a letter, `.` or `_`
- **Keywords** are reserved words (*only 45*)

```
active      assert      atomic      bit
bool        break       byte        chan
d_step      D_proctype  do      else
empty       enabled     fi      full
goto        hidden      if      init
int     len     mtype       nempty
never       nfull       od      of
pc_value    printf      priority    proctype
provided    run     short       skip
timeout     typedef     unless      unsigned
xr      xs
```

Promela language

C-like language for modeling concurrent systems

- **Identifiers** are strings of letters, digits and underscores, starting with a letter, `.` or `_`
- **Keywords** are reserved words (*only 45*)
- **Comments** are C-like with `/* */`
- **Operators** are C-like

```
+       -       *      /      %       >
>=      <       <=     ==     !=      !
&       ||      &&     |      ~       >>
<<      ^       ++     --
len()   empty() nempty()      nfull() full()
run     eval()  enabled()     pc_value()
```

Variable definition

- 5 basic types, default initialized to `0`

```
bit turn=1; /*[0..1]*/         bool flag=true; /*[0..1]*/
byte counter; /*[0..255]*/
short s; /*16-bit signed*/    int msg; /*32-bit signed*/
```

- Array

```
byte a[10]; /*array of 10 bytes*/
```

- Create enum with `mtype`

```
mtype = {A, B, C};
mtype d;
mtype:fruits = {apple, banana, pear};
```

- Variable size integer could be declared with `unsigned`

```
unsigned x : 5 = 10
```

Variable definition

- More complex type defined using `` `typedef` ``

```
typedef My_array {
    byte a[10];
    int b;
}
My_array a; /*declaration of new type*/
a.b = 10; /*access to field*/
```

- Variables could be used in **assignments** or **expressions**

```
a = 10;
a = a + 1;
a >= 10
```

Statements

A statement is a single instruction that can be executed by the program

- **Executable** if it can be executed immediately
  - Assignments are always executable
- **Blocked** if it cannot
  - Expressions are blocked if they evaluate to `` `0` ``

```
2 < 3   //always executable
x < 27  //only executable if value of x is smaller 27
3 + x   //executable if x is not equal to -3
```

- `` `skip` `` is always executable, it just cause the process counter to go one step forward
- `` `assert(<expr>)` `` is always executable, but if `` `<expr>` `` evaluates to `` `0` `` the program stops with an exception

Modelling behavior

Processes

Processes are the main component of a Promela program. They are the basic behavioral unit and are executed **concurrently**.

```
proctype Foo() {
    ...
}
```

- They are defined using `` `proctype` `` keyword
- They communicate with others using *global variables* and *channels*
- They are created using `` `run` ``, which returns the process id
  - They could be created at any moment by other processes
  - They start **immediately after** the run statement
- They could be started at creation using `` `active` ``, eventually with a quantity to spawn more process at the same time

  ```
  active[2] proctype Bar() {
      ...
  }
  ```

Hello world

```
/* A "Hello World" Promela model for SPIN. */
active proctype Hello() {
    printf("Hello process, my pid is: %d\n", _pid);
}
init {
    int lastpid;
    printf("init process, my pid is: %d\n", _pid);
    lastpid = run Hello();
    printf("last pid was: %d\n", lastpid);
}
```

- `init` is the first process to be executed
- `printf` is a built-in function to print on the console
- `_pid` is a built-in variable that contains the process id

Control flow

`if` statement

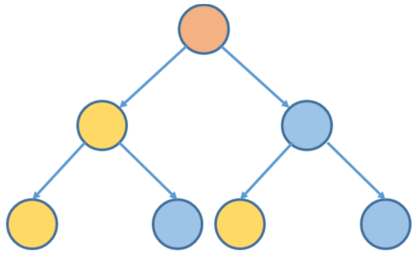
```
if
:: <expr> -> <statement>; ...
:: <expr> -> <statement>; ...
:: else -> <statement>; ...
fi;
```

- execute the statement of an executable `<expr>`
- the `else` branch is executed if all the other are blocked
- if no `<expr>` is executable and `else` branch not provided, the process is blocked

> 📖 The `->` operator is an alias for `;` used to separate the `<expr>` from the `<statement>`

- Useful to model non-deterministic branching

```
if
:: skip -> x = 0;
:: skip -> x = 1;
:: skip -> x = 2;
fi;
```



`do` statement

```
do
:: <expr> -> <statement>; ...
:: <expr> -> <statement>; ...
:: <expr> -> break ...
:: else -> <statement>; ...
od;
```

- works like `if`, but at the end of the statements list it restarts from the choice point
- if no `<expr>` is executable the process is blocked; `else` branch could be provided
- the `break` statement is used to exit the loop

Communication

Communication

The communication between processes is done using **channels**. A channel is a **FIFO** queue of messages.

```
chan c = [10] of {byte, int};
```

- The channel must have a **bounded size**
- The message structure supported by the channel is defined in the `of` clause
- A channel could be used for two-way communication
- Using the same channel for communication between multiple process could be tricky
  - use `xr` to assert exclusive reading
  - use `xs` to assert exclusive writing

Communication

Send

A message is sent to a channel using the `!` operator

```
channel! <expr>, <expr> ...;
```

The action is executable only if the channel is **not full**

Receive

A message is received from a channel using the `?` operator

```
channel? <var>, <var> ...;
```

The action is executable only if the channel is **not empty**

❗ The `<expr>` type must match the message type defined in the `of` clause

If one constant or enum value is provided, instead of a variable, the statement is executable *only if the message attributes match the provided constants*

Communication - handshaking

It is possible to use **channels** as synchronization mechanism between processes. The channel size must be `0`

- The sender is blocked until the receiver is ready to receive the message

```
chan c = [0] of {bit, byte};
proctype Sender() {
    c!1, 3+4;
}
proctype Receiver() {
    byte x;
    c?1, x;
}
```

- Only when both processes are ready these statements are considered executable

Synchronization concepts

Atomic blocks

All the single statements are atomic, but it is possible to group them in a block to make them atomic together

```
atomic {
    <statement>; ...
    <statement>; ...
}
```

No pure atomicity!

- It is executable if the first statement is executable
- If one of the following statements isn't executable, then process is temporarily suspended

Atomic blocks

Real total atomicity is obtained with the `` `d_step` `` statement

```
d_step {
    <statement>; ...
    <statement>; ...
}
```

- It is executable if the first statement is executable
- If one of the following statements is not executable, then generates a runtime error

Time model

Promela is a **functional language**, so it does not have a time model.

However, process often require clock or timeout to resend data

This is possible to model using the ``timeout`` statement

```
active proctype Receiver()
{
    bit recvbit;
    do
        :: toR ? MSG, recvbit -> toS ! ACK, recvbit;
        :: timeout -> toS ! ACK, recvbit;
    od
}
```

- ``timeout`` is executable if no other statements are executable
- ``timeout`` is used to escape from *deadlocks*

Goto

Goto is used as unconditional jump to a label

```
goto <label>;
```

- `label` is an identifier that precedes a statement
- `goto` jump to the label and executes the statement
- Useful to model *communication protocols*

```
wait_ack:
    if
    :: B?ACK -> ab=1-ab ; goto success
    :: ChunkTimeout?SHAKE ->
        if
        :: (rc < MAX) -> rc++;
                         F!(i==1),(i==n),ab,d[i];
                         goto wait_ack
        :: (rc >= MAX) -> goto error
        fi
    fi ;
```

Unless

```
{<statement> ...} unless {<guard>; <err_stat> ...};
```

- Execute `statement` until `guard` is true, then execute `err_stat`
- Useful to model *exception handling*

LTL logic

LTL formula

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

- `opd` is an operand
- `unop` is a unary operator
  - `[]` is the **globally** operator $\Box$
  - `<>` is the **eventually** operator $\Diamond$
  - `!` is the **negation** operator $\neg$

LTL formula

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

- `` `opd` `` is an operand
- `` `unop` `` is a unary operator
- `` `binop` `` is a binary operator
  - `` `U` `` is the temporal operator **strong until**
  - `` `W` `` is the temporal operator **weak until** (only when used in inline formula)
  - `` `V` `` is the dual of U: `` `(p V q)` `` means `` `!(!p U !q)` ``
  - `` `&&` `` or `` `/\` `` is the **logical and** operator
  - `` `||` `` or `` `\/` `` is the **logical or** operator
  - `` `->` `` is the logical **implication** operator
  - `` `<->` `` is the logical **equivalence** operator

LTL usage

A *ltl* property should be declared in global scope, like a global variable

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

```
ltl p { [] b }; // b always true

bool b = true;
active proctype main() {
    printf("hello world!\n");
    b = false;
}
```

C code

C Code

Starting from Spin `v4.0` it is possible to use C code in Promela with some limitations

It's primarily used by automatic verification tool, like **Modex** that we will see later, to model complex statement

- declaration of complex type or struct

```
c_decl {<typedef>};
c_code {<type> <identifier>};
c_track "&<identifier>" "sizeof(<type>)";
```

- atomic expression

```
c_code {<expr>};
```

- atomic expression executable only when evaluate to *non zero* value

```
c_expr{<expr>};
```

⚠ `c_expr` must contain only one statement without any side effect because could be executed multiple times

C & Promela

- C code could access *global* identifier declared in Promela using `` `now` `` followed by a dot because it refers to the **state vector**

```
c_code { now.<identifier> = <expr>; }
```

- C code could access *local* identifier declared inside process using `` `P` `` followed by the name of the `` `proctype` `` and the pointer arrow because it refers to the **state vector** of the process

```
c_code { P<proctypeName>-><identifier> = <expr>; }
```

- Promela code could access C identifiers declared with `` `c_decl` `` after they have been inserted into the state vector. This approach substitute the usage of `` `c_track` ``

```
c_state "<type> <identifier>" "Global|Local";
```

Execution

In order to execute a model that contains some C code instruction we have to use the `` `gcc` `` compiler to translate id

```
$spin -a model.pml
$gcc -o pan pan.c
$./pan
```

Questions?