Software Safety

Model Checking with SPIN!

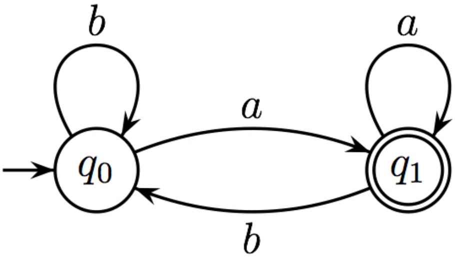
Prof. Dr.-Ing. Patrick Mäder, M.Sc. Martin Rabe

Contents

Background

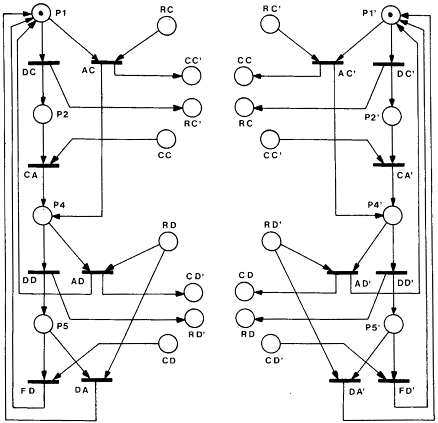
Automaton

- allows infinite inputs
- one start state
- some states are accepting states



[source: https://en.wikipedia.org/wiki/File:Automate_de_Buchi2.jpg]

- places (with token)
- transitions
- directed edges between places and transitions/transitions and places
- firing of transitions only possible if all places from incoming edges have at least one token, the token is consumed
- all places with an edge from a firing transition will get a token



Model Checking

Model Checking

Model checking is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for (a given initial state in) that model.

[Clarke & Emerson 1981]

Common Design Flaws

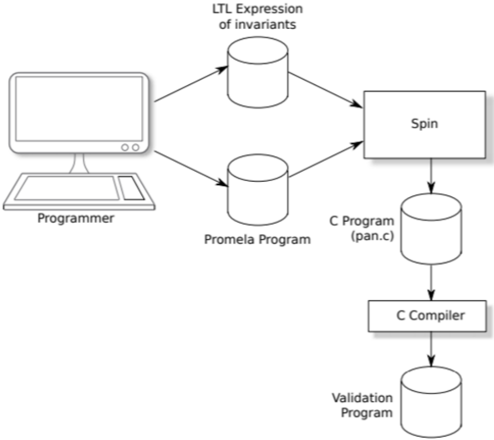
- Deadlocks
  - is a state in which each process is waiting for another [source: Wikipedia]
- Starvation
  - a state in which a process is perpetually denies a necessary resource [source: Wikipedia]
- Violation of constraints
  - e.g. protocol should stay in certain range
- Underspecification
- Overspecification
- Assumptions about speed

SPIN

SPIN: Introduction (I)

- **S**imple **P**romela **In**terpreter
- tool for analyzing logical consistency of concurrent systems
- written in C for Linux, MacOS and Windows
- originally designed for verifying communication protocols
- has become one of the most widely used verification tools both for teaching but also by professional software engineers
- particularly suited for modeling concurrent and distributed systems that are based upon interleaving of atomic instructions

SPIN: Structure



SPIN: Introduction (II)

- does not perform model checking itself
- achieves efficiency by generating optimized model checking program in C for each model and each correctness claim
- user does not need to look at the C source code but a C compiler is needed
- four modes:
  1. random simulation
  2. interactive simulation
  3. verification mode
  4. guided simulation

SPIN: Promela

- **Pro**tocol/**Pro**cess **Me**ta **La**nguage
- language with limited set of features
- syntax and semantics similar to C
- leverages Dijkstra's guarded commands for the control structure to facilitate writing non-deterministic programs
- date types:
  - bit, bool, byte, short, int
  - one-dimensional arrays
- features needed for building concurrent systems:
  - processes
  - atomic actions
  - channels
  - correctness properties (ltl, assertions)

SPIN: Promela

- **Pro**tocol/**Pro**cess **Me**ta **La**nguage
- language with limited set of features
- syntax and semantics similar to C
- leverages Dijkstra's guarded commands for the control structure to facilitate writing non-deterministic programs
- date types:
  - bit, bool, byte, short, int
  - one-dimensional arrays
- features needed for building concurrent systems:
  - processes
  - atomic actions
  - channels
  - correctness properties (ltl, assertions)
- IMPORTANT: Promela is NOT a programming language, it is a language to describe a design!

SPIN: Promela (cont.)

Defining a thread:

```
active [2] proctype HelloWorld()
{
    printf("Hello World\n");
}
```

SPIN: Promela (cont.)

Special thread (similar to ‘main‘):

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

SPIN: Promela (cont.)

Control structures:

```
if
    :: guard -> statement;
    :: guard -> statement;
    :: else -> statement;
fi;
```

SPIN: Promela (cont.)

Control structures:

```
do
    :: guard -> statement;
    :: guard -> statement;
    :: else -> statement;
od;
```

SPIN: Promela (cont.)

Data types:

Type	Typical Range	Sample Declaration
int	$[-2^{31}, 2^{31}-1]$	int y = 2;
bool	{false, true}	bool check = true;
byte	$[0, 255]$	byte test = 1;
arrays	*other types*	int array[5];
chan	*FIFO*	chan com = [2] of mtype, byte
mtype	$[0, 255]$	mtype name;

Data types:

Type	Typical Range	Sample Declaration
int	$[-2^{31}, 2^{31} - 1]$	int y = 2;
bool	{false, true}	bool check = true;
byte	$[0, 255]$	byte test = 1;
arrays	*other types*	int array[5];
chan	*FIFO*	chan com = [2] of mtype, byte
mtype	$[0, 255]$	mtype name;

```
mtype = { Sheldon, Penny, Leonard };
mtype name = Penny;
```

SPIN: Promela (cont.)

Atomic execution:

```
atomic{ <command> }
```

Run procedures on demand:

```
run <procedure>
```

send (!) and receive (?) message <msg> on channel <ch>

```
<ch>!<msg>
<ch>?<msg>
```

Jumps:

```
goto <label>
```

Check assertions:

```
assert(<expression>)
```

Check linear temporal logic expressions:

```
ltl <name> {<expression>}
```

Hands-On: First example

```
// file ex_1a.pml
init {
    byte i  // initialized to 0 by default
    do
    :: i++
    od
}
```

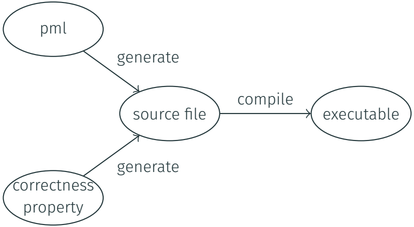
Hands-On: How to run it

- open terminal, navigate to the examples folder and try out the following commands (you will need to rename the spin executable and copy it to the examples folder)

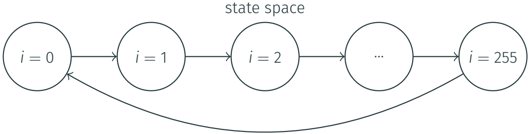
```
spin -u514 -p -l ex_1a.pml // perform 514 simulation steps

spin -run -d ex_1a.pml     // show state machine

spin -run ex_1a.pml        // compile and run verification
```

SPIN: How does it work

- Preparation:

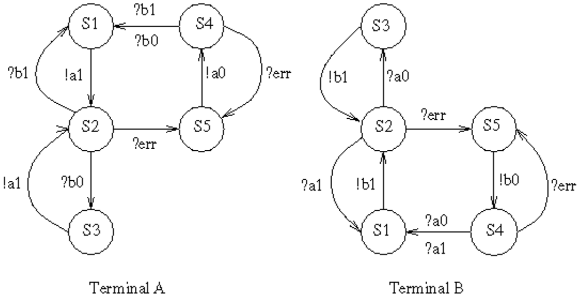


- Model checking:



- Question: can we find an input that invalidates our correctness property, in other words: ”a program is correct if there is no computation negating correctness”

- designed to make it possible to transmit information reliably over noisy telephone lines with low-speed modems (*ex_2.pml*)

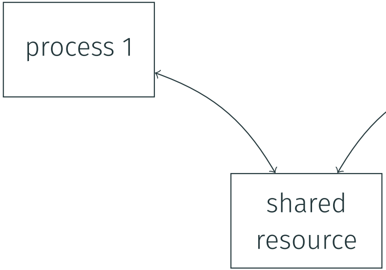


Task:

- verify protocol
- make every state reachable

Hands-On: Verifying a Mutual Exclusion Algorithm

- designed to give access to a shared resource to competing processes (*ex_3c.pml*)

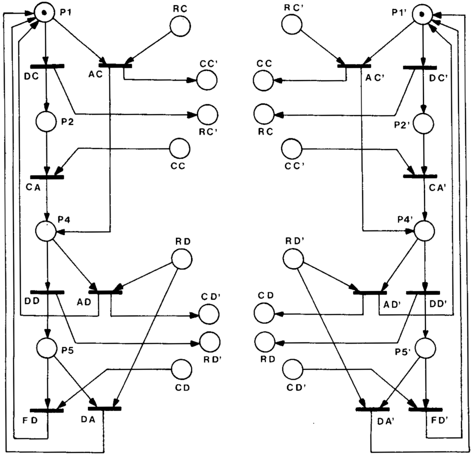


Task:

- verify algorithm

Hands-On: Verifying a Communication Protocol (I)

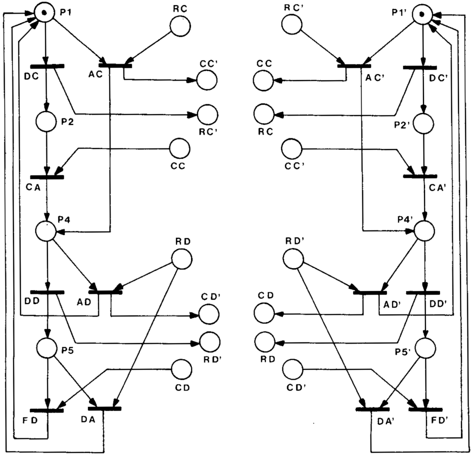
- Petri net describing a communication protocol, this protocol was proven to be correct (*ex_4.pml*)



Task:

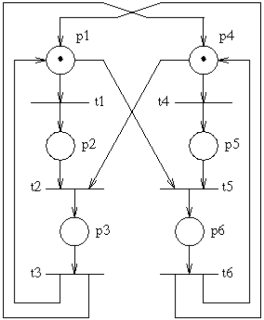
- verify protocol
- which transitions lead to the deadlock?

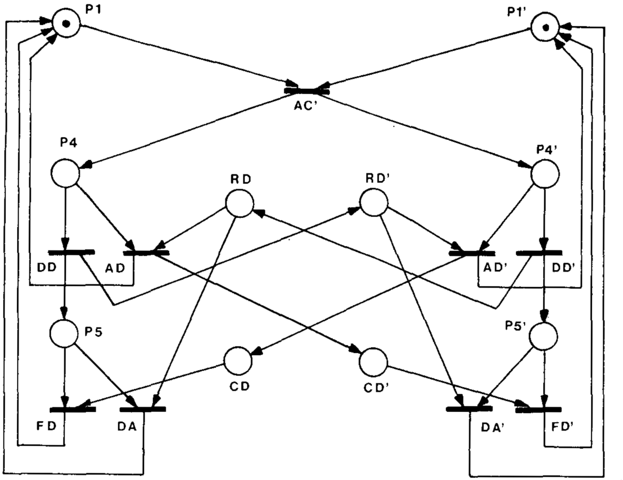
- Petri net describing a communication protocol, this protocol was proven to be correct (*ex_4.pml*)



Task:

- verify protocol
- which transitions lead to the deadlock?





Task:

- write a Promela file describing the Petri net shown on the left
- verify your model

◇ q means that q should eventually be true in the future.

□ q means that q should always be true.

□ (p → ◇ q) means that whenever p is true, q should eventually be true

"[...] In LTL, one can encode formulae about the future of path, e.g., a condition will eventually be true, a condition will be true until another fact becomes true, etc. [...]

LTL was first proposed for the formal verification of computer Programs by Amir Pnueli in 1977." [source: Wikipedia]

□ (p → q) means that whenever p is true, q should be true at the same time.

Linear Temporal Logic

Logic	Code	Description
$\Box\ p$	[] p	always
$\Diamond\ p$	<> p	eventually
$q \bigcup p$	q U p	until
$q\ R\ p$	q release p	release
$\neg\ p$	! p	negation
$q \wedge p$	q && p	and
$q \vee p$	q || p	or
$q \implies p$	q -> p	implication
$q \iff p$	q <-> p	equivalence

Logic	Code	Description
$\Box \Diamond p$	[] <> p	progress
$\Diamond \Box p$	<> [] p	stability
$q \implies \Diamond p$	q -> <> p	response

Promela-Syntax:

```
ltl <name> { <logical_expression> }
```

Advantage: LTL-expression gets checked after every Promela statement; asserts only where it was manually placed

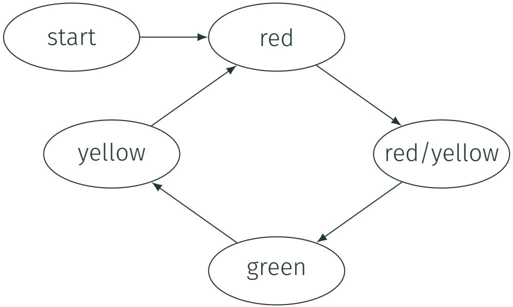
LTL: safety and liveness properties

With the symbols from table (previous slide), we can express both safety and liveness properties of a system:

- A safety property says that the system never does anything bad. This can be expressed as $\Box A$: "It is always true that A". If A is the statement, "a dangerous voltage is detected within 23ms," then $\Box A$ becomes the safety property that a dangerous voltage is always detected within 23ms.
- A liveness property says that the system eventually does something good: $\Diamond A$. If A is "the elevator stops," then $\Diamond A$ is the liveness property that the elevator always eventually stops. This is a weak fairness property.

LTL: traffic light example

System Description:



Possible Conditions:

- The traffic light is green infinitely often. (liveness)
  $\Box \Diamond$ green
- Once red, the light always becomes green eventually after being yellow for some time. (ordering)
  $\Box(\text{red} \implies (\Diamond \text{green} \wedge (\neg \text{green} \; \mathcal{U} \; \text{yellow})))$

[source: Richard M. Murray, Caltech CDS]

Hands-on: LTL

State	q	p
0	True	False
1	True	True
2	False	False
3	True	False
4	True	False
5	True	False
	...	

Are the following statements true:

1. In state 0: $\Box q$
2. In state 3: $\Box q$
3. In state 0: $\Diamond p$
4. In state 3: $\Diamond p$
5. In state 0: p release q
6. In state 3: p release q

Hands-On: Verifying a Mutual Exclusion Algorithm (revisited)

- designed to give access to a shared resource to competing processes (*ex_3c.pml*)

Task:

- change the assertion to a statement in LTL
- analyze the algorithm again

Conclusion

- SPIN can either proof the correctness of a design or give an counter example
- but Formal Methods have limits in regards to design complexity
- you have to take care to Model the design correctly, otherwise you will only proof that your Model is correct but not your design

SPIN: Cheat Sheet

```
spin -u514 -p -l file.pml   // -u : limit the number of simulation steps
                            // -p : print info about every step
                            // -l : print info about local variables

spin -run file.pml          // compile and run (defaults to dfs)

spin -run -bfs file.pml     // compile and run with bfs

spin -p -replay file.pml.   // replay trail

spin -run -m100000 file.pml // -m : set search depth
```