Seminar 6: Advanced SPIN

Non-deterministic Algorithms, Model Extraction

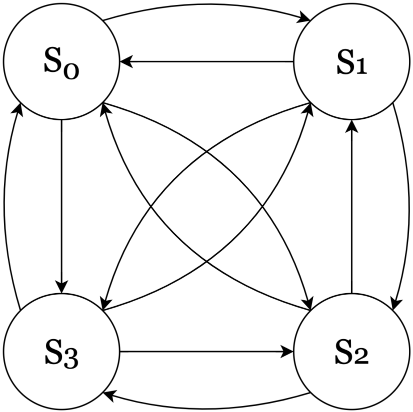
Di Marco, Okwieka

University of Rome "La Sapienza"

June 9th, 2023

TSP

Travelling Salesman Problem



	0	1	2	3
0	-	7	9	2
1	4	-	3	7
2	6	7	-	8
3	2	3	8	-

Promela

Ruys & Holzmann (2004): *Advanced SPIN Tutorial*

- Single process looking for a path
- Variable `cost` to hold the path cost so far.
- From a state we jump **non-deterministically** to another state **not yet visited**.

Promela

Ruys & Holzmann (2004): *Advanced SPIN Tutorial*

- Single process looking for a path
- Variable `cost` to hold the path cost so far.
- From a state we jump **non-deterministically** to another state **not yet visited**.

```
bit visited[4];
int cost;

active proctype TSP()
{
S0:  atomic {
        if
        :: !visited[3] -> cost = cost+2; goto S3;
        :: !visited[1] -> cost = cost+7; goto S1;
        :: !visited[2] -> cost = cost+9; goto S2;
        fi;
     }
S1:  atomic {
        visited[1] = true;
        if
        :: !visited[2] -> cost = cost+3; goto S2;
        :: !visited[3] -> cost = cost+7; goto S3;
        :: else        -> cost = cost+4; goto end;
        fi;
     }
     // ...
end: skip;
}
```

Find a solution

[Ruys & Brinksma - TACAS 1998]

- We can use SPIN model checking to find the **lower bound** for the variable `cost`.
- We will verify *iteratively*:
  1. $\Diamond(\texttt{cost} \geq 1000)$
     - The counterexample will be a path shorter than 1000.
     - *Ex.* a path with cost 20.
  2. $\Diamond(\texttt{cost} \geq 20)$
     - *Ex.* a path with cost 14
  3. $\Diamond(\texttt{cost} \geq 14)$
     - Satisfied

Find a solution

[Ruys & Brinksma - TACAS 1998]

- We can use SPIN model checking to find the **lower bound** for the variable `cost`.
- We will verify *iteratively*:
  1. $\Diamond(\texttt{cost} \geq 1000)$
     - The counterexample will be a path shorter than 1000.
     - *Ex.* a path with cost 20.
  2. $\Diamond(\texttt{cost} \geq 20)$
     - *Ex.* a path with cost 14
  3. $\Diamond(\texttt{cost} \geq 14)$
     - Satisfied

Algorithm Find Min Cost

```
min := guess of maximum cost
error := true
while error do
    Verify M ⊨ ◇(cost ≥ min)
    if error then
        min := cost
    end if
end while
```

Optimization

- Model Checkers are already being used for serious optimization problems.
- Although the original idea works, it is **inefficient**
  - The state space already contains the optimal solution.
  - Iteratively checking $\Diamond(\texttt{cost} \geq \mathit{min})$ is not needed.
- With *SPIN*'s **on-the-fly** model checking we won't explore the whole state space
  - We can save the best path found in the past for future checks

C code

- We can implement C code inside our Promela model!
    1. **`c_expr`**: executes a C expression and can return a value to the model.
    2. **`c_code`**: executes C code as an atomic statement
    3. **`c_state`**: can be used to track memory, holding state information

C code (2)

Ruys & Holzmann (2004): *Advanced SPIN Tutorial*

- In the declaration of the model we can add:
  c_state "int min_cost" "Hidden" "1000"
- at the **end** label we can add:

```
c_code {
        if(now.cost < min_cost) {
                min_cost = now.cost;
        }
}
```

C code (3)

Ruys & Holzmann (2004): *Advanced SPIN Tutorial*

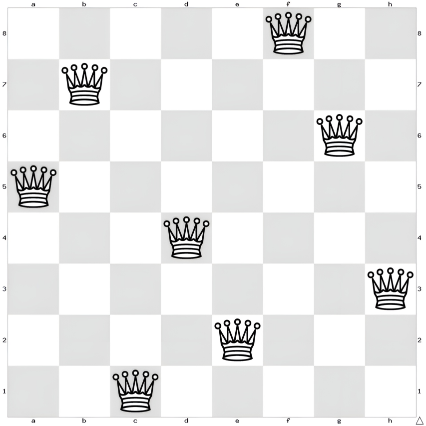
- At the beginning of each state ***Si***:

```
Si: atomic {
            visited[i] = true;
            if
            :: c_expr { now.cost > min_cost } -> goto end;
            :: else -> skip;
            fi;

            if
            :: !visited[1] -> cost = ... ; goto S1;
            :: ...
            :: else        -> cost = ... ; goto end;
            fi;
      }
```

8-Queens

8-Queens Problem



Promela

Ben-Ari (2008): *Principles of the SPIN Model Checker*

1. **Non-deterministically** choose a row for each Queen
2. Check if the placement is valid
3. If it isn't, the run gets **stuck**
   - All the traces that get to the **end** are valid solutions
4. The solution will be the counterexample of:
   `ltl sol {<> !(Queens@end)};`

Promela

Ben-Ari (2008): *Principles of the SPIN Model Checker*

1. **Non-deterministically** choose a row for each Queen
2. Check if the placement is valid
3. If it isn't, the run gets **stuck**
   - All the traces that get to the **end** are valid solutions
4. The solution will be the counterexample of:
   `ltl sol {<> !(Queens@end)};`

```
byte result[8]; // queens placement
bool a[8];      // row
bool b[15];     // diagonal of type /
bool c[15];     // diagonal of type \

active proctype Queens()
{
        byte col = 1;
        byte i = 0;
        byte row;
        do
        :: Choose(row);
                !a[row-1];
                !b[row+col-2];
                !c[row-col+7];
                a[row-1] = true;
                b[row+col-2] = true;
                c[row-col+7] = true;
                result[col-1] = row;
                if
                :: (col >= 8) -> break;
                :: else -> col++;
                fi;
        od;
end:skip;
}
```

Generate a **random** number

Ben-Ari (2008): *Principles of the SPIN Model Checker*

- *SPIN* does not support Real numbers
- When model checking, we will check **every possible choice**
  - We only need randomness when during **simulations**
- We can use **non-determinism** to decide whether to increment or not the number
  - Within **bounds**.

Generate a **random** number

Ben-Ari (2008): *Principles of the SPIN Model Checker*

- *SPIN* does not support Real numbers
- When model checking, we will check **every possible choice**
  - We only need randomness when during **simulations**
- We can use **non-determinism** to decide whether to increment or not the number
  - Within **bounds**.

```
inline Choose(row)
{
        row = 1; // min
        do       // max
        :: (row < 8) -> row++;
        :: break;
        od;
}
```

Model Extraction

Motivation

Holzmann (2001): *From Code to Models*

Classical approach: write model manually given (prelim.) system design

- Difficult for evolving systems: model may not match anymore
- Writing model manually is complex (similar to programming)
- Cannot detect errors introduced in implementation
- Unclear rules to derive abstraction from concrete systems
- Low expressiveness of modelling languages $\implies$ can be huge

Idea

Holzmann (2001): *From Code to Models*

Idea: derive model from concrete implementation

- How much semantic details to keep?
- How to keep model size manageable?
- Finite model from program?
- How to keep verification context while implementation evolves?

Observation: a C program's control structure is finite

Also, it can be mechanically transferred to Promela

Data is the main problem (say, 64-bit `long`)

Verification Context

Holzmann (2001): *From Code to Models*

Verification Context = Test Drivers + Native Code + Instrumented Code

Test Drivers: user-written Promela, implements environment

Native Code: C code that is not extracted, merely embedded into Promela (`c_code`, ...)

Instrumented Code: C code extracted to Promela

Extraction is guided by optional *replacement rules/filters*.

Only thread operations *need* filter rules ("this call creates a new process").

Handshaking Example

Holzmann (2001): *From Code to Models*

```
extern const int p0; enum msg_type { Msg, Ack, TimeOut };

void handshake(void) {
        send(p0, Msg);
        set_timer(16000); /* msec */
        int resp = wait_recv();
        switch (resp) {
        case Ack:       reset_timer(); break;
        case TimeOut: /* handle */ break;
        default: reset_timer(); error("meh"); break;
        }
}
```

Timer

Holzmann (2001): *From Code to Models*

Timer process can be modelled in Promela

Problem: Integer variable $\Longrightarrow$ huge state space

Observation: we only care about nonzero (timer ticking) vs zero (timeout)!

Improved Timer

Holzmann (2001): *From Code to Models*

```
chan timer = [0] of { mtype, chan, int };

mtype = { Msg, Ack, Other, Set, Reset, TimeOut };

active proctype timer_p()
{ chan who = 0;
        do
        :: timer?Set(who,_)
        :: timer?Reset(who,_)
        :: who != 0 -> who!TimeOut
        od
}
```

Handshake (raw extraction)

Holzmann (2001): *From Code to Models*

```
active proctype handshake() { int resp;
        c_code { send(now.p0,Msg); };
        c_code { set_timer(16000); };
        c_code { Phandshake->resp=wait_recv(); };
        do
        :: c_expr{ Ack == Phandshake->resp };
                c_code { reset_timer(); };
                break; goto C_0
        [...]
        od;
}
```

Replacement Rules Table

Holzmann (2001): *From Code to Models*

`set_timer(16000)`	`timer!Set(q0,16000)`
`reset_timer()`	`timer!Reset(q0,0)`
`send(p0,Msg)`	`p0!Msg`
`resp=wait_recv()`	`q0?resp`
`error(. . .`	`assert(false)`

Handshake (using rules)

Holzmann (2001): *From Code to Models*

```
active proctype handshake() {
        int resp;
        p0!Msg;
        timer!Set(q0,16000);
        q0?resp;
        do
        :: c_expr {Ack == Phandshake->resp};
                timer!Reset(q0,0);
                break; goto C_0
        [...]
        od;
}
```

Modex

Modex 2.8 User Guide

Previous paper, as primer: FeaVer model extractor

Modex is a similar tool

Packages test harness in a single `.prx` file

Composed of sections and commands such as

```
%P
<Promela code>
%%
%X <C function name to extract>
```

Fin.