An Introduction to the Model Checker Spin

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at
1. The Model Checker Spin

2. Checking a Client/Server System with Spin
The Model Checker Spin

- **Spin system:**
  - Gerard J. Holzmann et al, Bell Labs, 1980–.
  - Freely available since 1991.

- **Spin resources:**

**Goal:** verification of (concurrent/distributed) software models.
The Model Checker Spin

On-the-fly LTL model checking.

- Explicit state representation
  - Contrast to “Symbolic Model Checking” based e.g. on BDDs.
  - Representation of system $S$ by automaton $S_A$.
- On-the-fly model checking.
  - Reachable states of $S_A$ are only expended on demand.
  - Partial order reduction to keep state space manageable.
- LTL model checking.
  - Property $P$ to be checked described in LTL (linear temporal logic).
  - Description converted into property automaton $P_A$.
    - Automaton accepts only system runs that satisfy the property.

Model checking based on automata theory.
The Spin System Architecture

Fig. 1. The structure of SPIN simulation and verification.
System Descriptions: PROMELA

Process Meta Language.

```c
active proctype not_euclid(int x, y) 
{
    if
        :: x < y -> L: x = x-y;
        :: x < y -> y = y-x;
        :: x==y  -> assert(x!=y); goto L
    fi;
    printf("%d\n", x);
}
```

Wolfgang Schreiner  http://www.risc.uni-linz.ac.at  6/22
Example: A Traffic Simulation

/* road A: 0 -> 1 -> 2 -> 3 -> 0 
 * road B: 0 -> 1 -> 2 -> 3 -> 0 

bool roadA[4] = true;    // 4 cars on road A
bool roadB[4] = false;   // no car on road B
active proctype traffic()
{
    do
        /* from open road to crossing */

        /* A[1] enters the crossing */
          ...
    }

Wolfgang Schreiner  http://www.risc.uni-linz.ac.at  7/22
Propositional Linear Temporal Logic.

- LTL formulas describe properties of system runs.
  - A system run is a sequence of states.
    - $\Box p$ .... “$p$ always holds”.
    - $\Diamond p$ .... “$p$ eventually holds”.
    - $\Box \Diamond p$ .... “$p$ holds infinitely often”.

- Such a property can be translated into an automaton.
  - Automaton has special “accepting” states.
  - A run is accepted, if it infinitely often passes an accepting state.
  - An automaton for $\Box \Diamond p$:

```
true   T0   true
      p   accept
```

Wolfgang Schreiner
http://www.risc.uni-linz.ac.at
Example: A Traffic Simulation

- □¬stopped.
  - Is a deadlock impossible?
  - Answer: no; counterexample shows deadlock situation.
  - Is this the only deadlock situation?
  - Answer: yes.
- ◊(B[0] ∨ B[2]).
  - Is eventually some car on the open road B?
  - Answer: yes.
  - Is it impossible to transfer all cars from road A to road B?
  - Answer: no; counterexample shows actions for transfer.

Many interesting questions about system behavior.
Model Checking

- Logic: $S \models P$.
  - System $S$ satisfies property $P$.
  - Every run of $S$ satisfies $P$.

- Automata: $\mathcal{L}(S_A) \subseteq \mathcal{L}(P_A)$.
  - Automata $S_A$ and $P_A$ corresponding to $S$ and $P$.
  - Equivalent to: $\mathcal{L}(S_A) \cap \mathcal{L}(P_A) = \emptyset$.
  - Equivalent to: $\mathcal{L}(S_A) \cap \mathcal{L}(\neg P_A) = \emptyset$.

- Synchronous automata product: $A \times B$.
  - Always performs simultaneously one step of $A$ and one of $B$.
  - Both automata $A$ and $B$ have same vocabulary.
  - Then: $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$.

- Core question: $\mathcal{L}(S_A \times (\neg P)_A) = \emptyset$.

Emptiness check is the core of Spin model checking.
Product of two Automata

Figure 9.5
An automaton for infinite number of a’s (left) and an automaton for an infinite number of b’s (right).

Figure 9.6
An automaton for words with an infinite number of a’s and b’s.
On the Fly Model Checking

To check \( \mathcal{L}(S_A \times (\neg P)_A) = \emptyset \), we need not determine all states of \( A \).

- State of product automaton is a pair \( \langle s, p \rangle \).
  - State \( s \) is from \( S_A \), state \( p \) is from \( (\neg P)_A \).
- Compute successors \( \langle s', p' \rangle \) of \( \langle s, p \rangle \).
  - All states of \( (\neg P)_A \) have been already precomputed.
  - Need not compute \( s' \) of \( S_A \) if the labeling of transition \( s \rightarrow s' \) does not agree with the labeling of any transition \( p \rightarrow p' \).

Since the construction of \( S \) is guided by \( P \), the number of states computed by the model checker is often considerably reduced.
Partial Order Reduction

Check property $\square (g = 0 \lor g > x)$.

Only two representative runs have to be considered.
1. The Model Checker Spin

2. Checking a Client/Server System with Spin
A Client/Server System

■ System of one server and two clients.
  ■ Three concurrently executing system components.
■ Server manages a resource.
  ■ An object that only one system component may use at any time.
■ Clients request resource and, having received an answer, use it.
  ■ Server ensures that not both clients use resource simultaneously.
  ■ Server eventually answers every request.

Set of system requirements.
System Implementation

Server:
    local given, waiting, sender
begin
    given := 0; waiting := 0
loop
    sender := receiveRequest()
    if sender = given then
        if waiting = 0 then
            given := 0
        else
            given := waiting; waiting := 0
        endif
    elsif given = 0 then
        given := sender
    else
        waiting := sender
    endif
    sendAnswer(given)
endloop
end Server

Client(ident):
    param ident
begin
loop
    ...
    sendRequest()
    receiveAnswer()
    ... // critical region
    sendRequest()
endloop
end Client
Reasoning about Concurrent Systems

- **Property**: mutual exclusion.
  - At no time, both clients are in critical region.
    - Critical region: program region after receiving resource from server and before returning resource to server.
  - The system shall only reach states, in which mutual exclusion holds.

- **Property**: no starvation.
  - Always when a client requests the resource, it eventually receives it.
  - Always when the system reaches a state, in which a client has requested a resource, it shall later reach a state, in which the client receives the resource.

- **Problem**: each system component executes its own program.
  - Multiple program states exist at each moment in time.
  - Total system state is combination of individual program states.
  - Not easy to see which system states are possible.

How can we check that the system has the desired properties?
Implementing the System in PROMELA

/* definition of a constant MESSAGE */
mtype = { MESSAGE };

/* two arrays of channels of size 2, each channel has a buffer size 1 */
chan request[2] = [1] of { mtype };
chan answer [2] = [1] of { mtype };

/* two global arrays for monitoring the states of the clients */
bool inC[2] = false;
bool wait[2] = false;

/* the system of three processes */
init
{
  run client(1);
  run client(2);
  run server();
}

/* the client process type */
proctype client(byte id)
{
  do :: true ->
    request[id-1] ! MESSAGE;
    wait[id-1] = true;
    answer[id-1] ? MESSAGE;
    wait[id-1] = false;
    inC[id-1] = true;
    skip; // the critical region
    inC[id-1] = false;
    request[id-1] ! MESSAGE
  od;
}
/* the server process type */
proctype server()
{
    /* three variables of two bit each */
    unsigned given : 2 = 0;
    unsigned waiting : 2 = 0;
    unsigned sender : 2;

doo:: true ->

    /* receiving the message */
    if
        :: request[0] ? MESSAGE ->
            sender = 1
        :: request[1] ? MESSAGE ->
            sender = 2
    fi;

    /* answering the message */
    if
        :: sender == given ->
            if
                :: waiting == 0 ->
                    given = 0
                :: else ->
                    given = waiting;
                    waiting = 0;
                    answer[given-1] ! MESSAGE
            fi;
        :: given == 0 ->
            given = sender;
            answer[given-1] ! MESSAGE
        :: else
            waiting = sender
    fi;

    od;
}
Simulating the System Execution in Spin
Specifying a System Property in Spin

```
/* Formula Ax Type: [] !(c1 && c2) */
/* The Never Claim Below Corresponds */
/* To The Negated Formula !([] !(c1 && c2)) */
/* (formalizing violations of the original) */

never { /* !([] !(c1 && c2)) */
```

Verification Result: valid
Checking the System Property in Spin

(Spin Version 4.2.2 -- 12 December 2004)
+ Partial Order Reduction

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

State-vector 48 byte, depth reached 477, errors: 0
   499 states, stored
   395 states, matched
   894 transitions (= stored+matched)
   0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
...  
0.00user 0.01system 0:00.01elapsed 83%CPU (0avgtext+0avgdata 0maxresident)k 
0inputs+0outputs (0major+737minor)pagefaults 0swaps