Model Checking (Part 4)

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

The Model Checker Spin

- Spin system:
  - Gerard J. Holzmann et al, Bell Labs, 1980–.
  - Freely available since 1991.
  - Workshop series since 1995 (12th workshop "Spin 2005").
  - ACM System Software Award in 2001.
- Spin resources:
  - Web site: http://spinroot.com.
  - Survey paper: Holzmann "The Model Checker Spin", 1997.
  - Book: Holzmann "The Spin Model Checker — Primer and Reference Manual", 2004.

Goal: verification of (concurrent/distributed) software models.

The Model Checker Spin

On-the-fly LTL model checking.

- Explicit state representation
  - Representation of system S by automaton S_A.
  - There exist various other approaches (discussed later).
- 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 PLTL.
    - Propositional linear temporal logic.
  - Description converted into property automaton P_A.
    - Automaton accepts only system runs that do not satisfy the property.

Model checking based on automata theory.

The Spin System Architecture

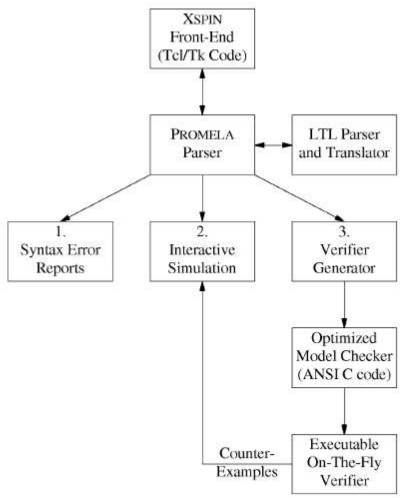


Fig. 1. The structure of SPIN simulation and verification.

Features of Spin

- System description in Promela.
  - Promela = Process Meta-Language.
    - Spin = Simple Promela Interpreter.
  - Express coordination and synchronization aspects of a real system.
  - Actual computation can be e.g. handled by embedded C code.
- Simulation mode.
  - Investigate individual system behaviors.
  - Inspect system state.
  - Graphical interface XSpin for visualization.
- Verification mode.
  - Verify properties shared by all possible system behaviors.
  - Properties specified in PLTL and translated to "never claims".
    - Promela description of automaton for negation of the property.
  - Generated counter examples may be investigated in simulation mode.

Verification and simulation are tightly integrated in Spin.

Some New Promela Features

Active processes, inline definitions, atomic statements, output.

```
mtype = { P, C, N }
mtype turn = P;

inline request(x, y) { atomic { x == y -> x = N } }
inline release(x, y) { atomic { x = y } }
#define FORMAT "Output: %s\n"

active proctype producer()
{
  do
  :: request(turn, P) -> printf(FORMAT, "P"); release(turn, C);
  od
}

active proctype producer()
{
  do
  :: request(turn, C) -> printf(FORMAT, "C"); release(turn, P);
  od
}
```


Some New Promela Features

Embedded C code.

```
/* declaration is added locally to proctype main */
c_state "float f" "Local main"

active proctype main()
{
  c_code { Pmain->f = 0; }
  do
    :: c_expr { Pmain->f <= 300 };
       c_code { Pmain->f = 1.5 * Pmain->f ; };
       c_code { printf("%4.0f\n", Pmain->f); };
  od;
 }
```

Can embed computational aspects into a Promela model (only works in verification mode where a C program is generated from the model).

Spin Usage for Simulation

Command-line usage of spin: `spin --`.

- Perform syntax check.

  `spin -a` *file*
- Run simulation.

No output:	`spin` *file*
One line per step:	`spin -p` *file*
One line per message:	`spin -c` *file*
Bounded simulation:	`spin -u`*steps* *file*
Reproducible simulation:	`spin -n`*seed* *file*
Interactive simulation:	`spin -i` *file*
Guided simulation:	`spin -t` *file*

Spin Usage for Verification

- Generate never claim

  spin -f "*nformula*" >*neverfile*
- Generate verifier.

  spin -N *neverfile* -a *file*

```
ls -la pan.*
-rw-r--r--  1 schreine schreine   3073 2005-05-10 16:36 pan.b
-rw-r--r--  1 schreine schreine 150665 2005-05-10 16:36 pan.c
-rw-r--r--  1 schreine schreine   8735 2005-05-10 16:36 pan.h
-rw-r--r--  1 schreine schreine  14163 2005-05-10 16:36 pan.m
-rw-r--r--  1 schreine schreine  19376 2005-05-10 16:36 pan.t
```

- Compile verifier.

  cc -O3 -DNP -DMEMLIM=128 -o pan pan.c
- Execute verifier.

Options:	./pan --
Find non-progress cycle:	./pan -l
Weak scheduling fairness:	./pan -l -f
Maximum search depth:	./pan -l -f -m*depth*

Spin Verifier Generation Options

cc -O3 *options* -o pan pan.c

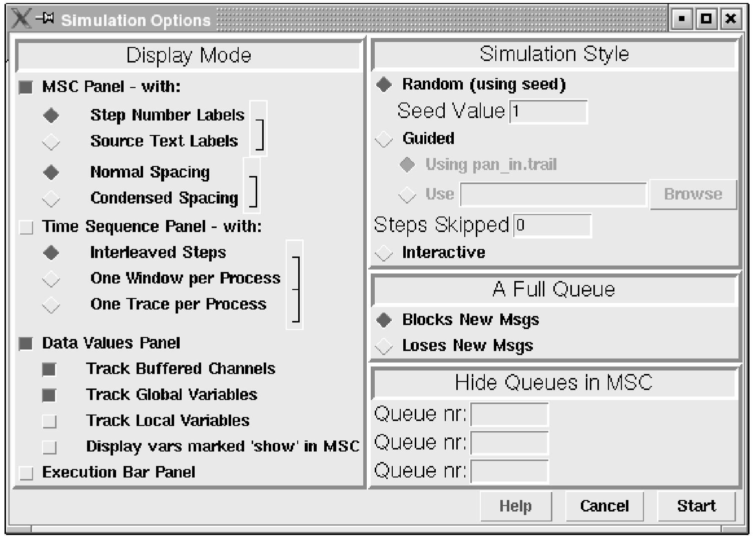
-DNP	Include code for non-progress cycle detection
-DMEMLIM=*N*	Maximum number of MB used
-DNOREDUCE	Disable partial order reduction
-DCOLLAPSE	Use collapse compression method
-DHC	Use hash-compact method
-DBITSTATE	Use bitstate hashing method

For detailed information, look up the manual.

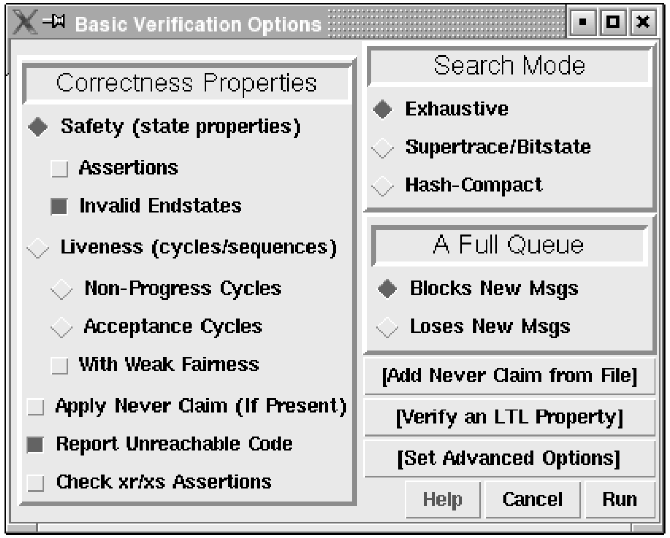
Spin Verifier Output

```
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
(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 52 byte, depth reached 587, errors: 0
     861 states, stored
     856 states, matched
    1717 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 1 (resolved)

Stats on memory usage (in Megabytes):
...
2.622   total actual memory usage
...
```

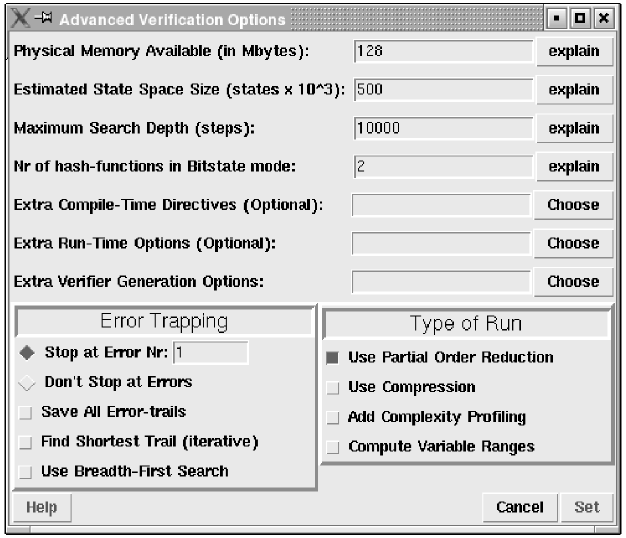

XSpin Simulation Options



XSpin Basic Verification Options



XSpin Advanced Verification Options



Other Approaches to Model Checking

There are fundamentally different approaches to model checking than the automata-based one implemented in Spin.

- Symbolic Model Checking (e.g. SMV, NuSMV).
  - Core: binary decision diagrams (BDDs).
    - Data structures to represent boolean functions.
    - Can be used to describe state sets and transition relations.
  - The set of states satisfying a CTL formula P is computed as the BDD representation of a fixpoint of a function (predicate transformer) F_P.
    - If all initial system states are in this set, P is a system property.
  - BDD packages for efficiently performing the required operations.
- Bounded Model Checking (e.g. NuSMV2).
  - Core: propositional satisfiability.
    - Is there a truth assignment that makes propositional formula true?
  - There is a counterexample of length at most k to a LTL formula P, if and only if a particular propositional formula $F_{k,P}$ is satisfiable.
    - Problem: find suitable bound k that makes method complete.
  - SAT solvers for efficiently deciding propositional satisfiability.

Other Approaches to Model Checking

- Counter-Example Guided Abstraction Refinement (e.g. BLAST).
  - Core: model abstraction.
    - A finite set of predicates is chosen and an abstract model of the system is constructed as a finite automaton whose states represent truth assignments of the chosen predicates.
  - The abstract model is checked for the desired property.
    - If the abstract model is error-free, the system is correct; otherwise an abstract counterexample is produced.
    - It is checked whether the abstract counterexample corresponds to a real counterexample; if yes, the system is not correct.
    - If not, the chosen set of predicates contains too little information to verify or falsify the program; new predicates are added to the set. Then the process is repeated.
  - Core problem: how to refine the abstraction.
    - Automated theorem provers are applied here.

Many model checkers for software verification use this approach.