SPINS: Extending LTSmin with Promela through SpinJa

Alfons Laarman

Joint with Freark van der Berg

Sept 17, 2012

Imperial College, London, UK
Spin Model Checker

Process Meta-Language (Promela)

Spin’s strengths

- Popular tool - early adopter of latest techniques
- Highly optimized C code
### Spin Model Checker

**Process Meta-Language (Promela)**

<table>
<thead>
<tr>
<th>Spin’s strengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Popular tool - early adopter of latest techniques</td>
</tr>
<tr>
<td>▶ Highly optimized C code</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Weakness</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ Hard to extend</td>
</tr>
</tbody>
</table>
SPINJa Model Checker

A Java reimplementation of SPIN
by Mark de Jonge & Theo Ruys - University of Twente

Strengths

▶ Layered OO Design - Easier to maintain & extend
A Java reimplementation of Spin by Mark de Jonge & Theo Ruys - University of Twente

Strengths
- Layered OO Design - Easier to maintain & extend

Weaknesses
- No parallel algorithms, no state compression, etc
- At least a factor 5 slower
Introducing the **LTS\textsubscript{MIN}** Model Checker

Initially, an LTS manipulation tool (explore, store, minimize). Developed at University of Twente

Grown to a full-blown model checking tool set:

- Multi-core, Distributed, Symbolic, Sequential
  (algorithmic backends)
  \times
- LTL, CTL, $\mu$-calculus, invariants, etc (properties)
  \times
- POR, state compression, saturation, chaining (optimizations)
  \times
- $\mu$CRL, $\text{mCRL2}$, DVE (DiVinE), UPPAAL, PBES, ETF
  (language frontends)
LTSmin’s goals

1. develop new model checking algorithms
2. reuse existing model checking algorithms
3. compare model checking algorithms
LT$\text{Smin}$’s goals

1. develop new model checking algorithms
2. reuse existing model checking algorithms
3. compare model checking algorithms

Good Promela support enables reuse of our algorithms and a multitude of comparisons!
**SpinJa’s and SpinS’ workflow:**

```
Model.prom  →  SpinJa  →  Model.java  →  Model.class  →  SpinJa  →  result
          ↓    ↓    ↓    ↓    ↓    ↓    ↓    ↓    ↓    ↓
          parse generate compile load verify
```

---

**UNIVERSITY OF TWENTE.**

SpinS: Extending LTSmin with Promela through SpinJa

Sept 17, 2012 6 / 19
**Approach**

**SpinJa’s and SpinS’ workflow:**

1. **SpinJa** workflow:
   - Model.prom → Parse → **SpinJa** → Generate → Model.java → Compile → Model.class → Load → **SpinJa** → Verify → Result

2. **SpinS** workflow:
   - Model.prom → Parse → **SpinS** → Generate → Model.c → Compile → Model.spins → Load → **LTSMin** → Verify → Result

---

UNIVERSITY OF TWENTE.

Sept 17, 2012 6 / 19
Approach

**SpinJa’s and SpinS’ workflow:**

- **SpinJa**:
  - Model.prom (parse) \(\rightarrow\) SpinJa \(\rightarrow\) Model.java \(\rightarrow\) Model.class \(\rightarrow\) compile \(\rightarrow\) load \(\rightarrow\) SpinJa \(\rightarrow\) verify \(\rightarrow\) result

- **SpinS**:
  - Model.prom (parse) \(\rightarrow\) SpinS \(\rightarrow\) Model.c \(\rightarrow\) Model.spins \(\rightarrow\) compile \(\rightarrow\) load \(\rightarrow\) LTSmin \(\rightarrow\) verify \(\rightarrow\) result
The Partitioned Next-State Interface

<table>
<thead>
<tr>
<th>pins defines:</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶ A state vector type: ( S: \langle s_1, \ldots, s_n \rangle )</td>
</tr>
<tr>
<td>▶ An initial state function: \text{INITIAL}() : S</td>
</tr>
<tr>
<td>▶ A ( k )-partitioned next-state function: \text{NEXT-STATE}_i(S) : S</td>
</tr>
<tr>
<td>▶ A dependency matrix: ( D_{k \times n} ) with ( D_{i,j} \in 2{\text{read,write}} )</td>
</tr>
</tbody>
</table>
The Partitioned Next-State Interface

pins defines:

- A state vector type: $S: \langle s_1, \ldots, s_n \rangle$
- An initial state function: \texttt{INITIAL}(): $S$
- A $k$-partitioned next-state function: \texttt{NEXT-STATE}_i(S): $S$
- A dependency matrix: $D_{k \times n}$ with $D_{i,j} \in 2\{\text{read, write}\}$

A few additional dependency matrixes with guard-information for partial order reduction.
```
int x = 0;
chan c;

active proctype p1() {
    c?;
}
proctype p2() {
    byte y = 1;
    c!;
    x = x + y;
}
init {
    run p2();
    x > 0;
}
```
From Promela to a PINS state vector

```c
int x = 0;
chan c;

active proctype p1() {
    c?;
}
proctype p2() {
    byte y = 1;
    c!;
    x = x + y;
}
init {
    run p2();
    x > 0;
}
```
From Promela to a PINS state vector

```c
int x = 0;
chan c;

active proctype p1() {
    c?;
}
proctype p2() {
    byte y = 1;
    c!;
    x = x + y;
}
init {
    run p2();
    x > 0;
}
```

```c
typedef struct state_s {
    int x;
    struct proctype_p1 {
        int _pc;
    } p1;
    struct proctype_p2 {
        int _pc;
        char y;
    } p2;
    struct proctype_init {
        int _pc;
    } init;
} state_t;
```
int x = 0;
chan c;

active proctype p1() {
    c?;
}
proctype p2() {
    byte y = 1;
    c!;
    x = x + y;
}

init {
    run p2();
    x > 0;
}

typedef struct state_s {
    int x;
    struct proctype_p1 {
        int __pc;
    } p1;
    struct proctype_p2 {
        int __pc;
        char y;
    } p2;
    struct proctype_init {
        int __pc;
    } init;
} state_t;

state_t *initial() {
    state_t *s = malloc(sizeof(state_t));
    s->x = 0;
    s->p1.__pc = 0;
    s->p2.__pc = -1;
    s->p2.y = 1;
    s->init.__pc = 0;
    return s;
}
int x = 0;
chan c;

active proctype p1() {
    c?;
}
proctype p2() {
    byte y = 1;
    c!;
    x = x + y;
}
init {
    run p2();
    x > 0;
}
int x = 0;
chan c;

active proctype p1() {
    c?;
}
proctype p2() {
    byte y = 1;
    c!;
    x = x + y;
}
init {
    run p2();
    x > 0;
}
From **PROMELA** to a **PINS** next-state and dependencies

```plaintext
int x = 0;
chan c;

active proctype p1() {
  c?;
}
proctype p2() {
  byte y = 1;
  c!;
  x = x + y;
}
init {
  run p2();
  x > 0;
}
```

Diagram: [Diagram showing the interaction between p1, p2, init states]
From Promela to a PINS next-state and dependencies

\[ \begin{align*}
\text{init}^0 & \rightarrow \text{init}^1 \rightarrow \text{init}^2 \\
\text{p*}^0 & \rightarrow \text{p*}^1 \rightarrow \text{p2}^2
\end{align*} \]
From **Promela** to a **pINS** next-state and dependencies

**Dependency matrix:**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>init0</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
</tr>
<tr>
<td>init1</td>
<td>rw</td>
<td>rw</td>
<td>rw</td>
</tr>
<tr>
<td>init2</td>
<td>r</td>
<td>rw</td>
<td>rw</td>
</tr>
</tbody>
</table>

**Code snippet:**

```
state t∗next−state(int i, state t∗in) {
    switch (i) {
        . . .
        case 2:
            if (in->p2.pc == 1) {
                state t∗out = malloc(sizeof(state t));
                memcpy(out, in, sizeof(state t));
                out->p2.pc = 2;
                out->x = out->x + out->p2.y;
                return out;
            }
            break;
        . . .
    }
}
```
state_t *next_state(int i, state_t *in) {
    switch (i) {
    ...
    case 2:
        if (in->p2._pc == 1) {
            state_t *out = malloc(sizeof(state_t));
            memcpy(out, in, sizeof(state_t));
            out->p2._pc = 2;
            out->x = out->x + out->p2.y;
            return out;
        }
        break;
    ...
    }}
state_t *next_state(int i, state_t *in) {
    switch (i) {
        ...
        case 2:
            if (in->p2._pc == 1) {
                state_t *out = malloc(sizeof(state_t));
                memcpy(out, in, sizeof(state_t));
                out->p2._pc = 2;
                out->x = out->x + out->p2.y;
                return out;
            }
            break;
        ...
    }
}

Dependency matrix:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>p1</th>
<th>p2</th>
<th>y</th>
<th>init</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>rw</td>
<td>rw</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>rw</td>
<td></td>
<td>rw</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>rw</td>
<td></td>
<td></td>
<td>rw</td>
</tr>
<tr>
<td>4</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td>rw</td>
</tr>
</tbody>
</table>
We extended SpinJa with:

- channel operations (empty, full, etc)
- user-defined structures (typedef)
- pre-defined variables (_pid and _nr_pr)
- channel polling and random receives (?[] and ??),
- remote references (@)
- preprocessor (#if, #ifdef, #define f(a,b), inline, and #include)
- and others
We extended SpinJa with:
- channel operations (empty, full, etc)
- user-defined structures (typedef)
- pre-defined variables (_pid and _nr_pr)
- channel polling and random receives (?[ ] and ??),
- remote references (@)
- preprocessor (#if, #ifdef, #define f(a,b), inline, and #include)
- and others

We were able to correctly compile and verify:

protocols BRP, Needham, I-protocol, Snoopy, SMCS, Chappe, x509
academic DBM, Phils, Peterson, pXXX, Bakery.7, Lynch, Chain, Sort
controller FGS, Zune, Elevator2.3 and Relay
BEEM all translated models from the BEEM database
huge GARP protocol [Konnov, Vienna]
Experiments (Scalability with Multi-Core)

Reachability with DiVinE, Spin and LTSmin using 48 cores
Experiments (Scalability with Multi-Core)

Reachability with DIVINE, SPIN and LTSMIN using 48 cores

Legend
- divine-table
- Itsmin-cleary-tree
- Itsmin-table
- Itsmin-tree
- spin-hc
- spin-nohc

- 1.8 sec
- 6.3 sec
- 20 sec
- 100 sec

Reachability with **DiVinE**, **Spin** and **LTSmin** using 48 cores

Promela model: Bakery protocol, other results:

Experiments (Memory usage)

Compression

- tree: → 8 byte per state
- cleary-tree: → 4 byte per state
- hc: 4 byte per state (lossy)
Experiments (Memory usage)

**Compression**

- tree: → 8 byte per state
- cleary-tree: → 4 byte per state
- hc: 4 byte per state \((\text{lossy})\)
Experiments (Memory usage)

Compression

tree: → 8 byte per state

clearly-tree: → 4 byte per state

hc: 4 byte per state (lossy)

<table>
<thead>
<tr>
<th></th>
<th>SPIN</th>
<th>DIVINE</th>
<th>LTSMIN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>hc</td>
<td>nohc</td>
<td>collapse</td>
</tr>
<tr>
<td>hc</td>
<td>1.5e+4</td>
<td>1.4e+5</td>
<td>4.9e+4</td>
</tr>
<tr>
<td>Bakery.7</td>
<td>1.3e+4</td>
<td>9.0e+4</td>
<td>6.4e+3</td>
</tr>
<tr>
<td>Peterson4</td>
<td>5.7e+3</td>
<td>4.4e+4</td>
<td>5.5e+3</td>
</tr>
<tr>
<td>n/a</td>
<td>4.8e+3</td>
<td>n/a</td>
<td>8.7e+3</td>
</tr>
<tr>
<td>n/a</td>
<td>1.1e+3</td>
<td>1.1e+3</td>
<td>2.8e+3</td>
</tr>
<tr>
<td>n/a</td>
<td>1.0e+2</td>
<td>4.0e+2</td>
<td>1.5e+2</td>
</tr>
<tr>
<td>1.0e+2</td>
<td>9.0e+2</td>
<td>2.5e+2</td>
<td>2.5e+2</td>
</tr>
</tbody>
</table>
Experiments (LTL with Multi-Core)

LTL with DiVINE (owcty), SPIN (piggybag) and LTSMIN (cndfs) using 48 cores
Experiments (LTL with Multi-Core)

**LTL with DiVINE (owcty), SPIN (piggybag) and LTSMin (cndfs) using 48 cores**

**Properties**

<table>
<thead>
<tr>
<th></th>
<th>distr.</th>
<th>on-the-fly</th>
<th>exact</th>
</tr>
</thead>
<tbody>
<tr>
<td>cndfs</td>
<td>- -</td>
<td>++</td>
<td>y</td>
</tr>
<tr>
<td>owcty</td>
<td>++</td>
<td>+</td>
<td>y</td>
</tr>
<tr>
<td>piggybag</td>
<td>+</td>
<td>- -</td>
<td>n</td>
</tr>
</tbody>
</table>
Experiments (LTL with Multi-Core)

LTL with DiVINE (owcty), SPIN (piggybag) and LTSmin (cndfs) using 48 cores

Legend
- divine-owcty
- Itsmin-cndfs
- spin-pb

Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>distr.</th>
<th>on-the-fly</th>
<th>exact</th>
</tr>
</thead>
<tbody>
<tr>
<td>cndfs</td>
<td>- -</td>
<td>++</td>
<td>y</td>
</tr>
<tr>
<td>owcty</td>
<td>++</td>
<td>+</td>
<td>y</td>
</tr>
<tr>
<td>piggybag</td>
<td>+</td>
<td>- -</td>
<td>n</td>
</tr>
</tbody>
</table>

PROMELA model: Elevator controller
Promela model: GARP protocol [Konnov, Vienna]

LTSmin completely explored all $3.11 \cdot 10^{11}$ states in under 3 minutes using only 300MB.

Next step: use CTL to verify liveness properties
### Experiments (Partial order reduction)

<table>
<thead>
<tr>
<th>Model</th>
<th>No POR</th>
<th>LTSmin</th>
<th>Spin</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>POR</td>
<td>POR</td>
</tr>
<tr>
<td></td>
<td>Transitions</td>
<td>States</td>
<td>States</td>
</tr>
<tr>
<td>GARP</td>
<td>48,363,145</td>
<td>247,135,869</td>
<td>4%</td>
</tr>
<tr>
<td>i-protocol2</td>
<td>14,309,427</td>
<td>48,024,048</td>
<td>16%</td>
</tr>
<tr>
<td>Peterson4</td>
<td>12,645,068</td>
<td>47,576,805</td>
<td>3%</td>
</tr>
<tr>
<td>BRP</td>
<td>3,280,269</td>
<td>7,058,556</td>
<td>100%</td>
</tr>
<tr>
<td>Sort</td>
<td>659,683</td>
<td>3,454,988</td>
<td>19%</td>
</tr>
<tr>
<td>X.509</td>
<td>9,028</td>
<td>35,999</td>
<td>62%</td>
</tr>
<tr>
<td>DBM</td>
<td>5,112</td>
<td>20,476</td>
<td>100%</td>
</tr>
<tr>
<td>SMCS</td>
<td>5,066</td>
<td>19,470</td>
<td>28%</td>
</tr>
<tr>
<td>Needham2</td>
<td>4,143</td>
<td>10,752</td>
<td>100%</td>
</tr>
</tbody>
</table>
Conclusions

Evaluation

- with little effort we could extend \texttt{LTSmin} with \texttt{Promela}
- \texttt{Promela} verification benefits from \texttt{LTSmin}'s capabilities
- we compared hc vs tree compression and cndfs vs pg vs owcty
LTSmin Bibliography & Acknowledgements

Multi-core:

- **table** Laarman, van de Pol & Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. FMCAD’10
- **tree** Laarman, van de Pol & Weber. Parallel Recursive State Compression for Free. SPIN’11
- **cleary** Laarman & van der Vegt. A Parallel Compact Hash Table. MEMICS’11
- **cleary-tree** van der Berg & Laarman. SPINS: Extending LTSmin with Promela through SpinJa. PDMC’12
- **cndfs** Evangelista, Laarman, Petrucci & van de Pol. Improved Multi-Core Nested Depth-First Search. ATVA’12
- **UPPAAL** Dalsgaard, Laarman, Larsen, Olesen & van de Pol. Multi-Core Reachability for Timed Automata. FORMATS’12

Distributed & symbolic:

- ▶ Blom, van de Pol & Weber. LTSmin: Distributed and Symbolic Reachability. CAV’10
- ▶ van Dijk, Laarman & van de Pol. Multi-core BDD Operations for Symbolic Reachability. PDMC’12
- ▶ Siaw. Saturation for LTSmin. 2012. Thesis

Other techniques:

- **PBES** Kant & van de Pol. Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games. Graphite’12

Download LTSmin 2.0 from: [http://fmt.cs.utwente.nl/tools/ltsmin/](http://fmt.cs.utwente.nl/tools/ltsmin/)

Thanks to LTSmin crew: Jaco van de Pol, Michael Weber, Stefan Blom, Elwin Pater, Tom van Dijk, Gijs Kant and Jeroen Ketema