UNIVERSITY OF TWENTE.
formal methods & tools.

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)

Spin's strengths

- Popular tool - early adopter of latest techniques
- Highly optimized C code

Weakness

- Hard to extend

SpinJa Model Checker

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

JAVA

Strengths

- Layered OO Design - Easier to maintain & extend

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

Weaknesses

- No parallel algorithms, no state compression, etc
- At least a factor 5 slower

Introducing the LTSmin 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, μ-calculus, invariants, etc (properties)
$\times$
POR, state compression, saturation, chaining (optimizations)
$\times$
μCRL, 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

Goals

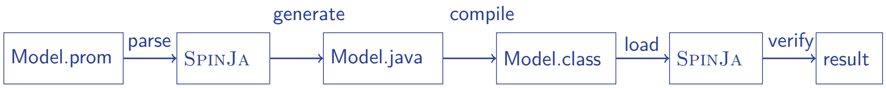
LTSmin's goals

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!

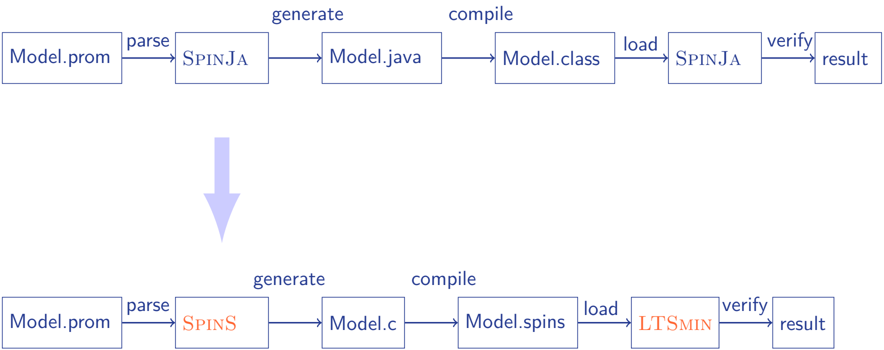
Approach

SpinJa's and SpinS' workflow:



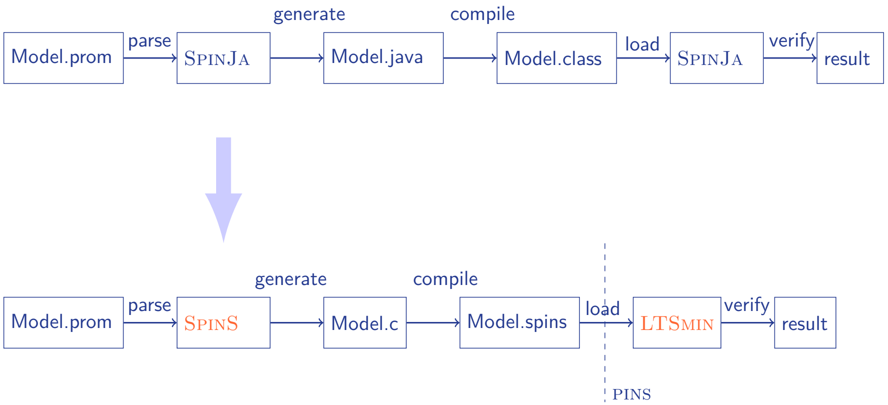
Approach

SpinJa's and SpinS' workflow:



Approach

SpinJa's and SpinS' workflow:



The Partitioned Next-State Interface

pins defines:

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

The Partitioned Next-State Interface

pins defines:

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

A few additional dependency matrixes with guard-information for partial order reduction.

PROMELA Example (simplified)

```
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

```
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

```
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;
```

From PROMELA to a PINS state vector

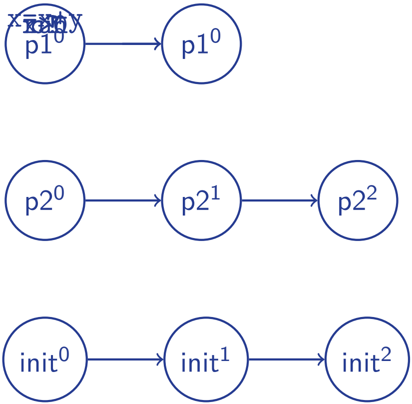
```
int x = 0;
chan c;

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;
}
```

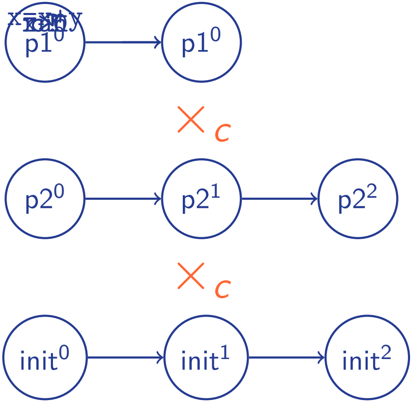
From PROMELA to a PINS next-state and dependencies

```
int x = 0;
chan c;

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

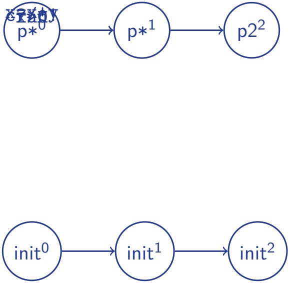
```
int x = 0;
chan c;

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



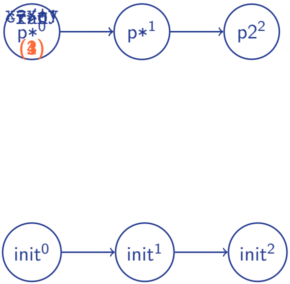
```
int x = 0;
chan c;

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

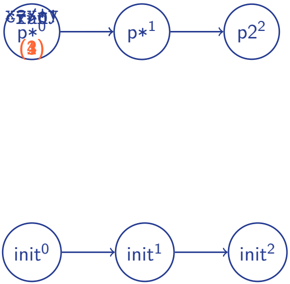


From PROMELA to a PINS next-state and dependencies



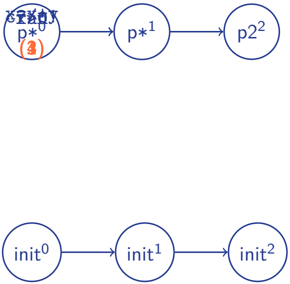
From PROMELA to a PINS next-state and dependencies






```
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;
. . .
}}
```

From PROMELA to a PINS next-state and dependencies




```
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:

	x	p1	p2	y	init
1		rw	rw		
2	rw		rw	r	
3			rw		rw
4	r				rw

SpinS Extends SpinJa

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

SpinS Extends SpinJa

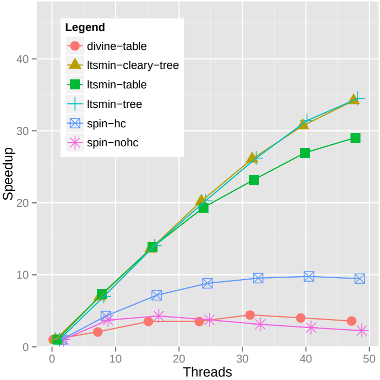
We extended SpinJa with:

- 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:

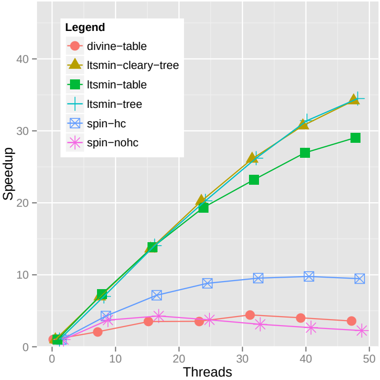
protocols	BRP, Needham, I-protocol, Snoopy, SMCS, Chappe, x509
academic	DBM, Phils, Peterson, pXXX, Bakery.7, Lynch, Chain, Sort
controller	FGS, Zune, Elevator2.3 and Relay
BEEM	all translated models from the BEEM database
huge	GARP protocol [Konnov, Vienna]

Reachability with DiViNE, Spin and LTSmin using 48 cores



Experiments (Scalability with Multi-Core)

Reachability with DiVinE, Spin and LTSmin using 48 cores



← 1.8 sec

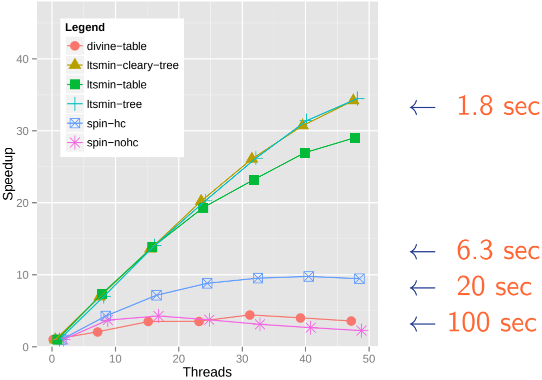
← 6.3 sec

← 20 sec

← 100 sec

Experiments (Scalability with Multi-Core)

Reachability with DiVinE, Spin and LTSmin using 48 cores



Promela model: Bakery protocol, other results:
http://wwwhome.cs.utwente.nl/~laarman/papers/pdmc2012/

Experiments (Memory usage)

Compression

tree: $\rightarrow$ 8 byte per state

cleary-tree: $\rightarrow$ 4 byte per state

hc: 4 byte per state (lossy)

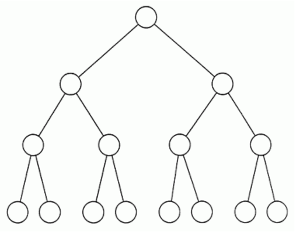
Experiments (Memory usage)

Compression

tree: $\rightarrow$ 8 byte per state

cleary-tree: $\rightarrow$ 4 byte per state

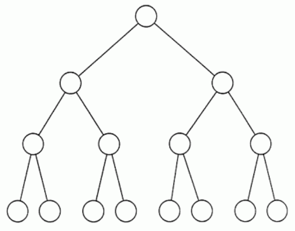
hc: 4 byte per state (lossy)



Experiments (Memory usage)

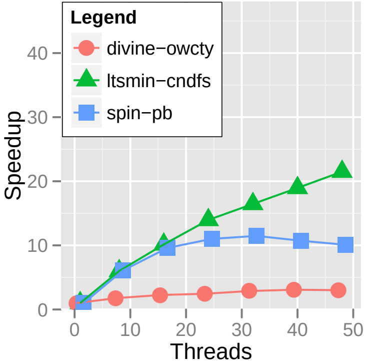
Compression

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



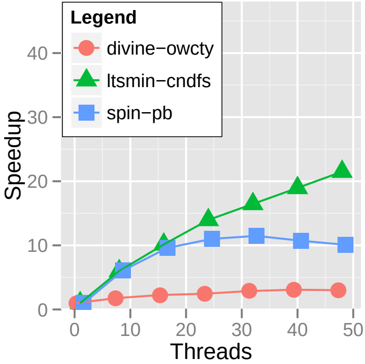
	Spin			DiVinE	LTSmin		
	hc	nohc	collapse	table	table	tree	cleary
GARP1	1.5e+4	1.4e+5	4.9e+4	n/a	8.7e+3	1.1e+3	9.0e+2
Bakery.7	1.3e+4	9.0e+4	6.4e+3	4.8e+3	2.8e+3	4.0e+2	2.5e+2
Peterson4	5.7e+3	4.4e+4	5.5e+3	n/a	1.3e+3	1.5e+2	1.0e+2

LTL with DiVinE (owcty), Spin (piggybag) and LTSmin (cndfs) using 48 cores



Experiments (LTL with Multi-Core)

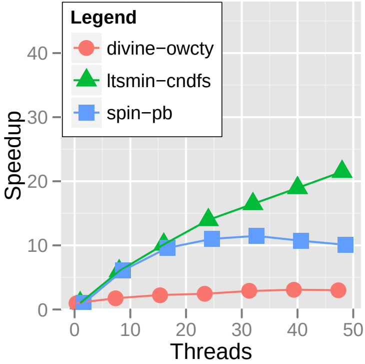
LTL with DiVinE (owcty), Spin (piggybag) and LTSmin (cndfs) using 48 cores



Properties

	distr.	*on-the-fly*	*exact*
cndfs	- -	++	*y*
owcty	++	+	*y*
piggybag	+	- -	*n*

LTL with DiVinE (owcty), Spin (piggybag) and LTSmin (cndfs) using 48 cores



Properties

	distr.	on-the-fly	exact
cndfs	- -	++	y
owcty	++	+	y
piggybag	+	- -	n

Promela model: Elevator controllor
http://wwwhome.cs.utwente.nl/~laarman/papers/pdmc2012/

Experiments (Symbolic)

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)

Model	No POR States	Transitions	Time	LTSMIN POR States	Trans	Time	SPIN POR States	Trans	Time
GARP	48,363,145	247,135,869	95.6	4%	1%	45.2	18%	9%	15.5
i-protocol2	14,309,427	48,024,048	15.5	16%	10%	18.7	24%	16%	4.5
Peterson4	12,645,068	47,576,805	13.8	3%	1%	2.3	5%	2%	0.3
BRP	3,280,269	7,058,556	3.7	100%	100%	7.0	58%	39%	1.6
Sort	659,683	3,454,988	1.9	19%	5%	2.6	0%	0%	0.0
X.509	9,028	35,999	0.1	62%	36%	0.0	68%	34%	0.0
DBM	5,112	20,476	0.0	100%	100%	0.1	100%	100%	0.0
SMCS	5,066	19,470	0.0	28%	14%	0.1	25%	11%	0.0
Needham2	4,143	10,752	0.0	100%	100%	0.0	100%	100%	0.1

Conclusions

Evaluation

- with little effort we could extend LTSmin with Promela
- Promela verification benefits from 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:

- Elwin Pater. Partial Order Reduction for PINS. 2011. Thesis (to TACAS'13)

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

GARP Konnov & Letichevsky Jr. Model Checking GARP Protocol using Spin and VRS. AAIT'10.

Download LTSMIN 2.0 from: 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