SpinS: Extending LTSmin with Promela through SpinJa

Freark van der Berg[1] Alfons Laarman[2]

Formal Methods and Tools, University of Twente, The Netherlands

Abstract

We show how Promela can be supported by the high-performance generic model checker LTSmin. The success of the Spin model checker has made Promela an important modeling language. SpinJa was created as a Java implementation of Spin, in an effort to make the model checker easily extendible and reusable while maintaining some of its efficiency. While these goals were certainly met, the downside of SpinJa remained its dependability on Java, degrading performance by a factor 5 and obstructing support for embedded C code in Promela models.
LTSmin aims at language-independence through the definition of the generic Partitioned Next-State Interface (pins). The toolset has shown that a generic model checker can indeed be competitive in terms of efficiency by supporting several languages from different paradigms and implementing many analysis algorithms that compete with other state-of-the-art model checkers.
We extended SpinJa to emit C code that implements the pins interface. Our new version of SpinJa, called SpinS (Spin + pins), also improves Promela support, greatly extending the support of models beyond toy and academic examples. In this paper, we demonstrate the usage of LTSmin's analysis algorithms: multi-core model checking of assertion violations, deadlocks and never claims (full LTL), inspection of error trails, partial order reduction (POR), state compression, symbolic reachability using (multi-core) decision diagrams and distributed reachability. Our experiments show that the performance of these methods beats other leading model checkers.

Keywords: model checking, Spin, LTSmin, SpinJa, Promela, multi-core, LTL, state compression, symbolic, decision diagram, distributed, partial order reduction

1 A New Promela Frontend for LTSmin: SpinS

Historically Promela (process meta language) was created to specify software systems for the Spin model checker [7]. By generating optimized C code from Promela models, Spin has flourished as an efficient model checker that even supports embedded C code for easy model program translation to

[1] Email: f.i.vanderberg@student.utwente.nl
[2] Email: a.w.laarman@cs.utwente.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

Promela. However due to the many optimizations Spin is also hard to extend. Therefore, efforts have been made to support Promela outside of Spin. For example, nips [19] defines a virtual machine language to compile Promela to; and SpinJa [9] is basically a reimplementation of Spin in Java.

LTSmin [3,14] is a language-independent model checking tool set. Through its pins interface, it abstracts away language-specific features with a state vector format and a next-state function. At the same time it exposes internal structure in the form of locality information through dependency matrices:

Definition 1.1 pins [2] defines a state vector format $S \equiv \langle s_0, s_1, \ldots, s_n \rangle$ with a fixed number of n slots and fixed domains $|s_i|$, an initial-state and partitioned next-state function: initial(): S and next-state$_k$(S): S, and a dependency matrix $D_{k \times n}$ recording read/write dependencies between transitions and slots.

In the past, we have shown that this locality information can yield large (order of magnitude) performance gains, especially for LTSmin's distributed and symbolic algorithms [3]. To additionally enable POR in our enumerative reachability and LTL model checking tools, several other matrices were added: maybe-coenabled, necessary disabling and necessary enabling set [16], the latter two are optional for better reductions. Although less dependent on the dependency matrices, LTSmin's multi-core backend was shown to be the leading tool in the area of parallel (LTL) model checking [13,15,12,11].

LTSmin already supported a subset of Promela through a nips connection. To enable more extensive and high-performance Promela support, we created SpinS; a modified and extended version of SpinJa that generates C code implementing the pins interface. SpinS *is included in the* LTSmin *distribution.*[3] Promela-specific properties, like *assertion violations, (in)valid end states* and *never claims* are exported as pins state and transition labels (not in Def. 1.1), for support in LTSmin. This enables the full power of all analysis algorithms in LTSmin as the following sections demonstrate.

Moreover, SpinS extends SpinJa with many new features: a preprocessor with support for conditionals (`#if, #ifdef`, etc), defines with arguments (`#define` and `inline`) and includes (`#include`), channel operations (`empty`, `full`, etc), user-defined structures (`typedef`), pre-defined variables (`_pid` and `_nr_pr`), channel polling and random receives (`?[]` and `??`), remote references (@), and many other Promela constructs.[4] Thereby, we were able to handle the models used in the following sections for the first time.

Promela is an extensive and evolving language, hence it is not yet supported in full. The most important, but still lacking, features (the ones that are actually used in Promela case studies) are: `timeout`, user-defined structures/channels in channel buffers and indirect channel references.

[3] The LTSmin website: `http://fmt.cs.utwente.nl/tools/ltsmin`
[4] See generally: `http://spinroot.com/spin/Man/promela.html`

2 Implementing PINS with SpinS

A Promela model $\mathcal{M}$ contains channel declarations ($\mathcal{C}$), global variable declarations (V^G) and at least one proctype definition ($\mathcal{P}$) containing statements to be executed and local variable declarations: $\mathcal{M} \equiv (\mathcal{P}_1, \ldots, \mathcal{P}_P, \mathcal{C}, V^G, v_0)$, where v_0 is the initial valuation of V^G. Proctypes are *instantiated* N times via an `active[N]` directive, or dynamically via `run` statements. Furthermore:

Definition 2.1 [Variables, channels and actions] V is a finite set of (global and local) variables with finite domains $Dom(V)$, $\mathcal{C}$ is a finite set of channels, and $\mathcal{A}$ an action of the form: 'V = $\mathcal{E}$' (assignment), '$\mathcal{E}$' (guard), 'c?' and 'c!' (channel synchronization), where $c \in \mathcal{C}$ and $\mathcal{E}$ is an expression. Expressions include boolean/arithmetic operators, but also operations, e.g.: `run`. They are parsed to abstract syntax trees (ASTs), but here we simply write code in single braces with AST variables in italics. An action a has enabling conditions ($en(\mathcal{A}): \mathcal{E}^*$), e.g.: $en(\text{'run(p)'}) = \langle \text{'\_nr\_pr < 256'} \rangle$.

Definition 2.2 [Process Automaton (PA)] A process automaton is a quintuple $\mathcal{P} \equiv (\mathcal{L}^\mathcal{P}, \mathcal{T}^\mathcal{P}, V^\mathcal{P}, l_0^\mathcal{P}, v_0^\mathcal{P})$, where: $\mathcal{L}^\mathcal{P}$ is a finite set of program locations, $V^\mathcal{P}$ is a set local variables, $\mathcal{T}^\mathcal{P} \subseteq \mathcal{L}^\mathcal{P} \times \mathcal{A}^* \times \mathcal{L}^\mathcal{P}$ is a set of transitions, $l_0^\mathcal{P} \in \mathcal{L}^\mathcal{P}$ is an initial location and $v_0^\mathcal{P} \in Dom(V)^{|V^\mathcal{P}|}$ the initial variable valuation.

With a sequence of actions $A \in \mathcal{A}^*$ with $A \equiv \langle a_0, \ldots \rangle$, we support atomic `d_step`s; A is enabled iff a_0 is, hence: $en(A) = en(a_0)$. The following subsections describe our PINS implementation of the Promela semantics (see [4]).

Automata creation. First, the Promela code is parsed into $\mathcal{M}$. Each proctype becomes a PA $\mathcal{P}$, actions become transitions, conditions ('`if`...`fi`'$\in \mathcal{A}$) become branches and loops ('`do`...`od`'$\in \mathcal{A}$) become cycles. A never claim is also parsed as a PA $\mathcal{N}$. Then, SpinS creates an *instance automaton* $\mathcal{I}_i^P$ by copying $\mathcal{P}$ (and its local variables) for each possible instantiation i.

State vector creation. At this stage, the state vector can be created. In the Promela semantics model, a global system state comprises of the values of the local variables and process counters of all proctype instances and the global variables. A system state can be easily mapped to a PINS state vector $S: \langle V, \mathcal{L}^{\mathcal{I}_1}, V^{\mathcal{I}_1}, \ldots, \mathcal{L}^{\mathcal{I}_I}, V^{\mathcal{I}_I} \rangle$ by adding additional *program counters* $pc(\mathcal{I}_i)$ to accommodate $\mathcal{L}^{\mathcal{I}_i}$ for all I instance automata $\mathcal{I}_i$. The implementation of INITIAL becomes: $\langle v_0, l_0^{\mathcal{I}_1}, v_0^{\mathcal{I}_1}, \ldots, l_0^{\mathcal{I}_I}, v_0^{\mathcal{I}_I} \rangle$.

In reality, V is not a flat structure, but may contain user-defined types, channels buffers and combinations thereof. Our state vector implementation S reflects this structure and is used to generate a C struct "`S`" in the final step. Variables can therefore be referenced symbolically while generating code, e.g.: $print(s, x) =$"`s.init[0].x`", where $s: S$ is a state vector with $name(s) =$"`s`", $x \in V^{\mathcal{I}_0}$ a local variable with $name(x) =$"`x`" and $name(\mathcal{I}_0) =$"`init`". While $print(s, pc(\mathcal{I}_0)) =$"`s.init[0].__pc`"; "`__pc`" is a reserved name for $pc(\mathcal{I}_0)$.

To adhere to the PINS interface, we need to fix I. Therefore, SpinS prompts the user for a fixed number of maximum process instances $M^{\mathcal{P}}$ for each dynamically started proctype. To fix $|s_i|$ all variables are padded to the size of an integer using compiler directives. The introduced overhead is mitigated by PINS as our performance and memory benchmarks show (Sec. 4). Sec. 3 shows how $M^{\mathcal{P}}$ can be encoded in the model.

Model transition creation. The set $\mathcal{T}$ of all transitions in all $\mathcal{I}$'s represents the asynchronous system as implemented by the Promela model, modulo channel synchronization. Hence, next, we transform it into a set of synchronizing transitions: $\mathcal{T}' \subseteq 2^{\mathcal{L}} \times \mathcal{A}^* \times 2^{\mathcal{L}}$. To this end, all channel send actions are replaced by synchronous pairs for all possible synchronization partners: $\mathcal{T}' := \{(\{l_1, l_3\}, \langle A, B\rangle, \{l_2, l_4\}) \mid (l_1, A, l_2) \in \mathcal{T}^{\mathcal{I}_1} \wedge (l_3, B, l_4) \in \mathcal{T}^{\mathcal{I}_2} \wedge \text{`}c\texttt{!}\text{'} \in A \wedge \text{`}c\texttt{?}\text{'} \in B \wedge c \in \mathcal{C} \wedge \mathcal{I}_1 \neq \mathcal{I}_2\}$.[5] Non-sync. actions are copied: $\mathcal{T}' := \mathcal{T}' \cup \{(\{l_1\}, A, \{l_2\}) \mid (l_1, A, l_2) \in \mathcal{T} \wedge \forall c \in \mathcal{C} : \text{`}c\texttt{?}\text{'} \notin A \wedge \text{`}c\texttt{!}\text{'} \notin A\}$. If a never claim exists, the synchronous product of $\mathcal{T}'$ and the never automaton is also calculated: $\mathcal{T}' := \{(L_1 \cup \{l_3\}, \langle A, B\rangle, L_3 \cup \{l_4\}) \mid (L_1, A, L_2) \in \mathcal{T}' \wedge (l_3, B, l_4) \in \mathcal{T}^{\mathcal{N}}\}$.

We decorate $T \in \mathcal{T}'$ where $T \equiv (L_1, A, L_2)$ with action and location guards: $en(T) = en(A) \cup \{\text{`}p \texttt{ == } l^{\mathcal{I}_i}\text{'} \mid l^{\mathcal{I}_i} \in L_1 \wedge p = pc(\mathcal{I}_i)\}$. We also add assignment actions for the location transfer function: $act(T) = A \cup \{\text{`}p \texttt{ = } l^{\mathcal{I}_i}\text{'} \mid l^{\mathcal{I}_i} \in L_2 \wedge p = pc(\mathcal{I}_i)\}$. Operations are replaced by simple actions, e.g.: `'run(p)'` becomes '`s.p_`i`.__pc = `$l_0^{\mathcal{I}_i}$' s.t. $name(\mathcal{I}_i) =$ "`p`" and $\mathcal{I}_i$ is a nonactive instance to be determined by additional (prior) actions.

C code generation. $T_i \in \mathcal{T}'$ becomes the blueprint for our partitioned next-state function with $k = |\mathcal{T}'|$. Alg. 1 shows C code for a NEXT-STATE$_i$(S) function.The square braces contain code generation templates. The *print* function generates conjunctions of the expressions $e \in en(T_i)$ and C statements for actions $a \in act(T_i)$. Again, it is parameterized by the state vector to be used for variable printing (in: S or out: S). Since Promela statements are similar to C, an implementation of *print* is straightforward.

Algorithm 1 C code template for NEXT-STATE$_i$

```
1 S next_state_[i](S in) {
2   if ([ print(in, en(T_i)) ]) {
3     S out = in;  // copy
4     [ print(out, act(T_i)) ]
5     return out;
6 }}
```

Dependency matrices. For $D_{k \times n}$ we traverse the ASTs $en(T)$ and $act(T)$ for all $T \in \mathcal{T}'$; POR dependency matrices require some additional analysis.

For this brief explanation, we considered only *rendezvous* channels, and abstracted away from atomic states and accepting state labels. *Buffered* channels only require some actions handling buffer bookkeeping. Accepting states are exported by adding $\mathcal{L}^{\mathcal{N}}$ as PINS state labels (not in Def. 1.1). Finally, atomic states (including loss and transfer of atomicity) are implemented using an internal (generated) reachability algorithm limited to a specific process instance.

[5] Promela's semantical constraints allow only one channel action per transition: the first.

3 Using LTSmin on Promela Models

The `spins` command calls SpinS to generate C code and compiles the result to a `.prom` library implementing the pins interface. The user is prompted to provide a fixed number of maximum instances for each dynamic proctype $\mathcal{P}$ ($M^{\mathcal{P}}$ in the previous section). This information can also be encoded in the model via a macro definition: `#define __instances_[proctype] [num]`. In many cases, the number of instantiated processes can be inferred statically [2, Def.5], but we did not implement this yet.

For this paper, we compiled the following set of models from the Spin distribution, [17] and a database [6]: BRP, GARP, Needham, I-protocol, Snoopy, SMCS, Chappe and x509 are protocol models, DBM, Phils, Peterson, pXXX, Bakery.7, Lynch, Chain and Sort are academic examples, and FGS, Zune, Elevator2.3 and Relay are models of controllers. X509 contains an assertion error ($Done < 6$) and Zune a never claim expressing $\neg\Box(@S \Rightarrow \Diamond @E)$ in LTL. We used two models of the GARP protocol: GARP1[6] and GARP2 is not publicly available [10]. We verified that indeed all these models are correctly explored by our tools (see Sec. 4). To this end, we had to turn off control flow optimization (`-o3`) in some cases, due to its limited implementation in SpinS. The following subsections present different verification strategies on these models with LTSmin and give some background on the used algorithms.

Model checking Promela-specific properties. The following command uses the *sequential tool* to detect assertion violations (`--action=assert`) :

```
prom2lts-seq --action=assert --trace=trace.gcf X.509.prm.prom
```

The first error trace is written to `trace.gcf`, which contains line numbers in the original Promela code, and can be pretty printed using the command `ltsmin-printtrace`. Similarly, deadlocks can be detected using the `-d` option.

Never claim violations can be detected with the NDFS algorithm [11]:

```
prom2lts-mc --strategy=ndfs --trace=lasso.gcf zune.pml.prom
```

The typical lasso-formed error trail can be best inspected using the command: `ltsmin-tracepp --table lasso.gcf | less -S`.

Multi-core model checking. One of the areas in which LTSmin excels is parallel model checking. For safety properties (deadlocks, invariants and assertion violations), we can enable parallel exploration in randomized (`-prr`) pseudo depth-first (`dfs`) order in the *multi-core tool*:

```
prom2lts-mc --threads=48 --strategy=dfs -prr -d smcs.pml.prom
```

While our parallel exploration algorithms tend to yield linear speedups for full verification [13,15], the randomized dfs order can potentially yield super-linear speedups in presence of counter-examples [12].

For parallel LTL model checking, we can use our latest and best multi-core

[6] The Promela database: http://www.albertolluch.com/research/promelamodels

NDFS algorithm CNDFS [6]. While this algorithm is heuristic in nature, we found that on a large set (over 400) of examples it scales rather well, i.e., speedups of 10 to 48 on a 48-core machine. It outperforms our earlier best algorithm [12]. The following command line uses this algorithm (randomization is enabled automatically in this setting):

```
prom2lts-mc --threads=48 --strategy=cndfs zune.pml.prom
```

Since CNDFS is on-the-fly, we may also obtain super-linear speedups in presence of bugs [12, Sec. 4].

Memory-efficient model checking. By default, LTSmin uses the option `--state=tree` to store states in binary tree form in a single hash table containing tuples of 32-bit references (for details refer to [15]). The tree compression can yield optimal compressed state sizes of 2 references (8 byte), while maintaining the excellent performance and scalability of uncompressed hash table storage [13] (`--state=table`). Recently, we added some optimizations to the tree. By splitting the table in two, one for root nodes and one for internal nodes, we can accommodate more than 2^{32} states (`-s32`), while maintaining the optimal compression ratio of 8 byte per state! By default, the root table is 4 times larger than the internal node table (`--ratio=2`) allowing a maximum of 2^{34} states to be stored using $1\frac{1}{4} \cdot 8\text{B} \cdot 2^{34} = 160\text{GB}$. Higher ratios allow us to store more states, e.g.: `-s35 --ratio=3` (notice how the internal node table remains $2^{35}/2^3 = 2^{32}$ in size, thus supporting the 32-bit internal references, hence the 8 byte optimal compressed sizes).

Typically, input models are asynchronous systems exhibiting *high locality*, i.e., all transitions read/write only few variables in the state vector. The resulting combinatorial space of state vectors often yields the near-optimal tree compression of almost 8 bytes per state. But some models might yield worse compression, then LTSmin gives the error *node table full*. In such cases, we need to lower the ratio, e.g., `--ratio=1` (ratio $= 2^1 = 2$), increasing compressed sizes to 12 byte per state.

To further improve compression, we combined the tree tables with compact hashing. Compact hash tables only store the key modulo the hashed location. The latter can be reconstructed using three additional accounting bits [18]. By replacing the root of the tree table with our lockless Cleary table [18], the compressed sizes approach 4 byte per state. For example, the options `--state=cleary-tree -s34 --ratio=2` allow us to store 2^{34} states in only $(\frac{1}{4} \cdot 8\text{B} + 4\text{B}) \cdot 2^{34} = 96\text{GB}$ provided that the model exhibits compression ratios close to $1\frac{1}{4}$ of the optimum. Over half of 350 diverse models [17] exhibit this [15, median in Fig. 7]. All our compression techniques are compatible with both the algorithms for LTL and safety properties.

Orthogonally, partial order reduction (POR) can further reduce state spaces (`--por`). Our POR method uses a language-independent notion of dependency relations expressed in terms of transition guards and exported via PINS matri-

ces [16]. POR is fully compatible with our (multi-core) algorithms for safety properties (`--strategy=[bfs,dfs,sbfs]`; pseudo bfs/dfs and strict bfs order described in [4]). LTL model checking however requires: (1) the use of a cycle proviso `--proviso=[closedset,color,stack]` (refer to [16, Sec. 4.6.4-6]), (2) the sequential tool (`prom2lts-seq`) as we have not yet found a way to combine the cycle proviso with our parallel LTL algorithms, and (3) a crossproduct calculated by LTSmin (option `--ltl=[formula]`) so that actions relevant to the invisibility proviso can be recorded [16, Sec. 4.6.3].

Symbolic model checking. The tool `prom2lts-sym` implements *symbolic model checking*, learning the symbolic transition relation on-the-fly [2]. This approach also works well on models with high locality. As such models have a sparse PINS dependency matrix, our reordering algorithms (`-rga`) can optimize them further for BDDs. Using a chaining heuristic [3], we can explore $> 10^{20}$ states in a second: `prom2lts-sym -rga --order=chain peterson5.prom` LTSmin also implements exploration in parallel [5] and with saturation (see documentation of `--saturation`). Additionally the symbolic tool can verify properties expressed in μ-calculus (see `--mu`) and CTL (see `--ctl`).

Distributed model checking. The tool `prom2lts-dist` supports distributed exploration and storage of the state space [3]. State spaces are stored distributedly and can be reduced modulo bisimulation using `ltsmin-reduce-dist`.

4 Performance, Scalability, Memory and Correctness

To compare the performance of Promela model checkers, we benchmarked Spin 6.2.1 [8] and LTSmin 2.0[3] [14] on a 48-core machine (a four-way AMD Opteron™ 6168). Each time we include one BEEM model [17] to allow comparison with DiVinE 2.5.2 [1]. We show here a representative selection. [7]

Performance and scalability. For high performance in Spin, we compiled models with parallel BFS [8]: `-DNOBOUNDCHECK -DSAFETY -DNOREDUCE -DBFS_PAR -DBFS_MAXPROCS=48`. By default, this enables a lossy hash compaction (hc) state storage, hence we also compiled using `-DNOHC`. DiVinE is configured as described in [13]. In LTSmin, we used a hash table, a tree table and a cleary-tree (all non-lossy). All experiments use a fixed table size of 2^{28}. To accommodate a master thread, Spin and DiVinE are limited to 47 threads.

Fig. 1 shows the obtained speedups. While speedups in LTSmin are good, we also observe in Table 1 that the sequential runtimes are on par with those in Spin. The 48-core runtimes show that LTSmin's multi-core algorithms are a good addition for Promela model checking. Furthermore, we can see that (Cleary-)tree compression introduces little or no overhead.

[7] For complete results see http://fmt.cs.utwente.nl/tools/ltsmin/pdmc-2012

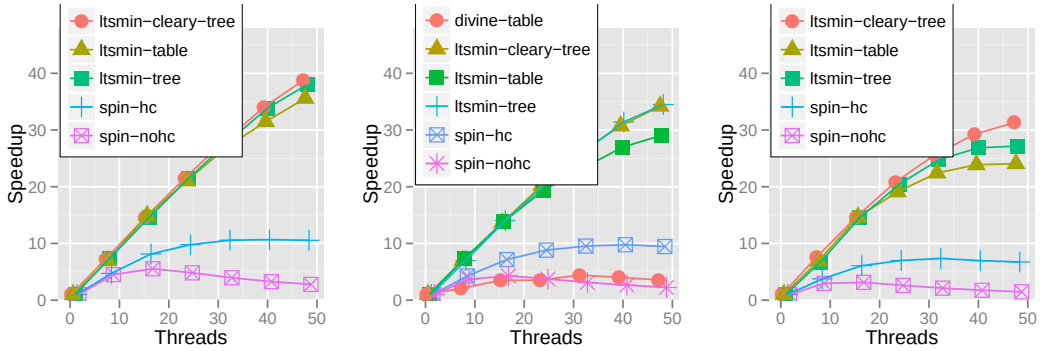

Fig. 1. Speedups of GARP1, Bakery.7 and Peterson4 in Spin, DiVinE and LTSmin

Fig. 2 shows speedups of two models obtained with DiVinE's owcty algorithm, Spin's piggyback (PB) algorithm [8] (with hash compaction) and LTSmin's CNDFS [6] algorithm (with hash table). CNDFS shows the best speedups and is sequentially faster than the PB algorithm (by 60%), which comes second in terms of speedup. Three other aspects are of interest when comparing the three algorithms: CNDFS/OWCTY are exact LTL algorithms while the PB may miss counter-examples [8], CNDFS is on-the-fly while the PB explores the whole state space before reporting a counter-example [8] and owcty typically explores a large portion of it [6, Sec. 4.2], and CNDFS is found to return even shorter counter-examples than a parallel BFS-based algorithm [6, Sec. 4.3]! On the other hand, the BFS-based algorithms owcty and PB can be distributed on a cluster, as DiVinE demonstrates [1].

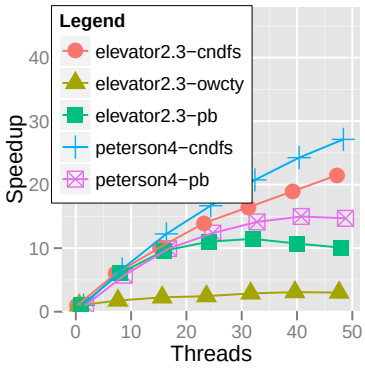

Fig. 2: Peterson4 ($\Box\Diamond p$), Elevator2.3 [8] (Speedup)

Memory usage. We measured the memory usage of DiVinE, LTSmin with and without tree compression and of Spin with and without collapse compression (col) and hash compaction. Table 2 shows the memory usage of all these combinations. The first thing we noticed, is that the memory usage is almost independent of the number of threads, showing that the model checkers add little overhead for parallel operation. Spin's memory usage is measured

Table 1
Runtimes (sec) in Spin (hc/nohc), DiVinE and LTSmin (table, tree and cleary-tree)

	States	Spin-hc		Spin-nohc		DiVinE		LTSmin-table		LTSmin-tree		LTSmin-cleary	
		1	47	1	47	1	47	1	48	1	48	1	48
GARP1	1.6e8	458.0	43.4	820.0	295.0	n/a	n/a	187.9	5.3	**175.8**	**4.6**	196.9	5.1
Bakery.7	2.7e7	66.0	6.3	169.0	38.4	32.2	9.0	**52.0**	1.8	60.0	**1.7**	69.4	2.0
Peterson4	9.5e6	23.1	2.6	56.9	18.3	n/a	n/a	29.6	1.2	**22.3**	**0.8**	26.9	0.9

Table 2
Memory usage (MB) in Spin, DiVinE and LTSmin is almost independent of number of threads

	Spin-hc		Spin-nohc		col	DiVinE		LTSmin-table		LTSmin-tree		LTSmin-cleary	
	1	47	1	47	1	1	47	1	48	1	48	1	48
GARP1	1.5e4	1.6e4	1.4e5	1.4e5	4.9e4	n/a	n/a	8.7e3	8.8e3	1.1e3	1.3e3	**9.0e2**	**1.1e3**
Bakery.7	1.3e4	1.5e4	9.0e4	6.0e4	6.4e3	4.8e3	4.9e3	2.8e3	2.9e3	4.0e2	4.2e2	**2.5e2**	**2.8e2**
Peterson4	5.7e3	6.2e3	4.4e4	2.5e4	5.5e3	n/a	n/a	1.3e3	1.3e3	1.5e2	1.6e2	**1.0e2**	**1.0e2**

Table 3
POR performance in LTSmin and Spin

	No POR		LTSmin POR		Spin POR	
Model	States	Transitions	States	Transitions	States	Transitions
GARP1	48,363,145	247,135,869	**1,742,585**	**3,669,890**	8,718,209	22,412,803
i-protocol2	14,309,427	48,024,048	**2,308,898**	**4,585,530**	3,436,166	7,778,563
BRP	3,280,269	7,058,556	3,280,269	7,058,556	**1,906,691**	**2,733,018**
Sort	659,683	3,454,988	123,583	170,134	**182**	**182**
Snoopy	81,013	273,781	**9,251**	**11,639**	13,380	18,550
X.509	9,028	35,999	**5,569**	12,787	6,094	**12,336**
SMCS	5,066	19,470	1,425	2,784	**1,244**	**2,134**
Chappe	1,203	3,017	**363**	**466**	1,203	3,018

by reducing the hash table size to exactly fit the state count, hence overestimated by at most 50%. We can however conclude that tree compression provides great reduction compared to full-state storage in a hash table *making lossy hash compaction redundant*. And the cleary-tree improves upon this by almost a factor of two. In [15], we compared compression methods in detail.

We see in Table 3 that LTSmin's POR is competitive to Spin's. However, especially for the Sort model, Spin yields better reductions. We attribute this to the fact it uses the extra xs and xr annotations in the model.

Symbolic results. Using our symbolic tools, we exhaustively explored the GARP2 model [10]. This model was never before fully explored with Spin except with lossy compression techniques. With regrouping and chaining, we could explore the model within 3 minutes using only 250MB of memory for $3.3 \cdot 10^{11}$ states. For the Phils model with 30 dining philosophers, we obtain $7.8 \cdot 10^{20}$ states in 0.18 sec and 39MB. It takes about one minute to explore the $8.3 \cdot 10^{8}$ states of Peterson5 using only 36MB. However, for many other models with fewer locality, runtimes and memory usage can increase steeply because many small operations need to be executed on large BDDs.

Correctness. To ensure correctness of our implementation of the Promela semantics, we verified that state, transition and deadlock counts are exactly equal to those reported by Spin for all models discussed in this paper. Also we

checked that LTSmin reports the same (LTL) counter-examples. We also found and excluded some models that yield different state counts in LTSmin, these were however only related to the corner-case semantics concerning loss of atomicity and jumps from and to atomic statements. Notable examples include a model for a steam generator controller, and the PLC and GIOP protocols.[6]

5 Conclusions

We presented SpinS: a new frontend for the LTSmin toolset that handles Promela models. We demonstrated how the many capabilities of LTSmin can be exploited and with experiments we showed great enhancements for model checking of Promela models: through C code generation its performance is on par with Spin's, scalability of reachability is better than Spin's latest parallel BFS algorithm, tree compression reduces memory usage with a factor 5 compared to collapse compression and maintains performance, POR can compete with Spin's POR, exact scalable parallel LTL is available for Promela for the first time, and we were able to fully verify a model symbolically that could never before be handled by Spin [10].

But SpinS opens more perspectives for better model checking. By choosing the C language as a target, we can easily add support for Promela's embedded C code (a lack of example models has prevented us from doing so thus far). Furthermore, by reimplementing Promela's semantics in Java[8], we can more easily loosen the semantic's dependencies on implementation details. For example, we think SpinS can easily support more flexible process creation methods as proposed by Holzmann.[9] For the current version, however, we aimed to implement Promela's semantics as close as possible to Spin's; the state and transition counts for all the models discussed in this paper are equal to Spin's.

Acknowledgements. Special thanks goes to Elwin Pater for implementing many of LTSmin's features, including but not limited to: POR, LTL, CTL and μ-calculus crossproducts, trace pretty printing, reordering, and the DiVinE frontend. Elwin also worked on a direct connection between Spin and LTSmin, which he gave up only because support for the pins matrices required a reimplementation of Spin anyway. We also thank Michael Weber. His ideas and efforts laid the basis for the current state of LTSmin. We thank Stefan Blom for his work on our distributed and symbolic backends. Finally, Jaco van de Pol contributed to the μCRL/mCRL2 frontends, and made substantial contributions to the symbolic backends together with Jeroen Ketema. Jaco also commented on early versions of this paper.

[8] Recall that SpinS is based on SpinJa but generates C code instead of Java code.
[9] Spin model checking projects: http://spinroot.com/spin/projects.html

References

[1] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker. In *Parallel/Distributed Methods in Verification & High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010.

[2] S.C.C. Blom, J.C. van de Pol, and M. Weber. Bridging the Gap between Enumerative and Symbolic Model Checkers. Technical Report TR-CTIT-09-30, University of Twente, 2009.

[3] S.C.C. Blom, J.C. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In T. Touili, B. Cook, and P. Jackson, editors, *CAV'10*, volume 6174 of *LNCS*, pages 354–359, Berlin, July 2010. Springer.

[4] A.E. Dalsgaard, A.W. Laarman, K.G. Larsen, M.Chr. Olesen, and J.C. Pol. Multi-core Reachability for Timed Automata. In M. Jurdziński and D. Ničković, editors, *FORMATS'12*, volume 7595 of *LNCS*, pages 91–106. Springer, 2012.

[5] T. van Dijk, A.W. Laarman, and J.C. van de Pol. Multi-core BDD Operations for Symbolic Reachability. In *PDMC'12*, volume current of *ENTCS*. Springer, 2012.

[6] S. Evangelista, A.W. Laarman, L. Petrucci, and J.C. van de Pol. Improved Multi-Core Nested Depth-First Search. In S. Ramesh, editor, *ATVA'12*, volume 7561 of *LNCS*, pages 269–283. Springer, 2012.

[7] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2011.

[8] G.J. Holzmann. Parallelizing the spin model checker. In A. Donaldson and D. Parker, editors, *SPIN'12*, volume 7385 of *LNCS*, pages 155–171. Springer, 2012.

[9] M. de Jonge and T. Ruys. The SpinJa Model Checker. In J. van de Pol and M. Weber, editors, *SPIN'10*, volume 6349 of *LNCS*, pages 124–128. Springer, 2010.

[10] I. Konnov and O.A. Letichevsky Jr. Model Checking GARP Protocol using Spin and VRS. *International Workshop on Automata, Algorithms, and Information Technologies*, May 2010.

[11] A.W. Laarman, R. Langerak, J.C. van de Pol, M. Weber, and A. Wijs. Multi-Core Nested Depth-First Search. In T. Bultan and P. A. Hsiung, editors, *ATVA'11*, volume 6996 of *LNCS*, pages 321–335, London, July 2011. Springer.

[12] A.W. Laarman and J.C. van de Pol. Variations on Multi-Core Nested Depth-First Search. In J. Barnat and K. Heljanko, editors, *PDMC*, volume 72 of *EPTCS*, pages 13–28, 2011.

[13] A.W. Laarman, J.C. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In N. Sharygina and R. Bloem, editors, *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design, Lugano, Swiss*, USA, October 2010. IEEE Computer Society.

[14] A.W. Laarman, J.C. van de Pol, and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *LNCS*, pages 506–511, Berlin, July 2011. Springer.

[15] A.W. Laarman, J.C. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In A. Groce and M. Musuvathi, editors, *SPIN'11*, LNCS, pages 38–56. Springer, 2011.

[16] E. Pater. Partial Order Reduction for PINS, MSc thesis, University of Twente, 2011.

[17] R. Pelánek. BEEM: Benchmarks for explicit model checkers. In *SPIN'07*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.

[18] S. van der Vegt and A.W. Laarman. A Parallel Compact Hash Table. In T. Vojnar, editor, *MEMICS'11*, volume 7119 of *LNCS*, pages 191–204. Springer, 2011.

[19] M. Weber. An Embeddable Virtual Machine for State Space Generation. In D. Bošnački and S. Edelkamp, editors, *SPIN'07*, volume 4595 of *LNCS*, pages 168–186. Springer, 2007.