LTSmin: High-Performance, Language-Independent Model Checking

Gijs Kant,[1,⋆] Alfons Laarman,[1,2,§] Jeroen Meijer[1], Jaco van de Pol[1], Stefan Blom,[1,‡] and Tom van Dijk,[1,∥]

[1] Formal Methods and Tools, University of Twente, The Netherlands
[2] Formal Methods in Systems Engineering, Vienna University of Technology, Austria

Abstract. In recent years, the LTSmin model checker has been extended with support for several new modelling languages, including probabilistic (Mapa) and timed systems (Uppaal). Also, connecting additional language front-ends or ad-hoc state-space generators to LTSmin was simplified using custom C-code. From symbolic and distributed reachability analysis and minimisation, LTSmin's functionality has developed into a model checker with multi-core algorithms for on-the-fly LTL checking with partial-order reduction, and multi-core symbolic checking for the modal μ-calculus, based on the multi-core decision diagram package Sylvan. In LTSmin, the modelling languages and the model checking algorithms are connected through a Partitioned Next-State Interface (Pins), that allows to abstract away from language details in the implementation of the analysis algorithms and on-the-fly optimisations. In the current paper, we present an overview of the toolset and its recent changes, and we demonstrate its performance and versatility in two case studies.

1 Introduction

The LTSmin model checker has a modular architecture which allows a number of modelling language front-ends to be connected to various analysis algorithms, through a common interface. It provides both symbolic and explicit-state analysis algorithms for many different languages, enabling multiple ways to attack verification problems. This connecting interface is called *Partitioned Next-State Interface* (Pins), the basis of which consists of a state-vector definition, an initial state, a *partitioned* successor function (NextState), and labelling functions. Pins defines an implicit state space, abstracting away from modelling language details.

The main difference with other language interfaces, such as the Open/Cæsar interface [21] of CADP [22] and the CESMI interface of DiVinE [3], is the structure that Pins exposes by exporting dependencies between the *partitioned*

[⋆] Supported by the NWO under grant 612.000.937 (VOCHS).

[§] Supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) grant VRG11-005.

[‡] Partially funded by NWO under grant 612.063.817 (SYRUP).

[∥] Supported by the NWO under grant 612.001.001 (MaDriD).

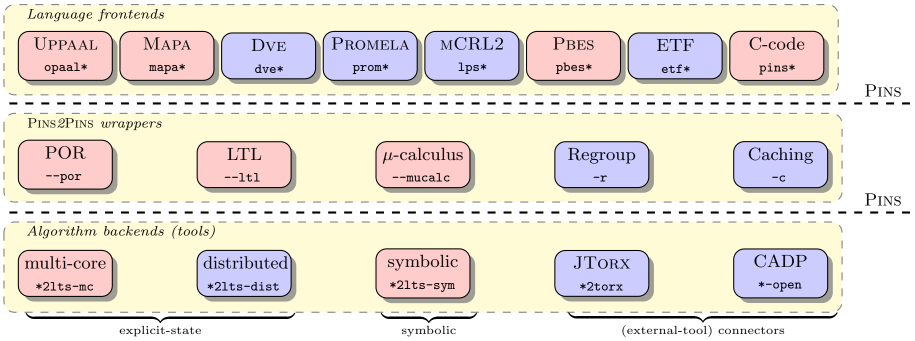

Fig. 1: LTSmin's Pins architecture.

successor function and the variables in the state vector in the form of *dependency matrices*. Our approach is also dissimilar from NuSMV's [9], where the transition *relation* is specified directly, imposing restrictions on the language. In the past, we have shown that these dependencies enable symbolic reachability using BDDs/MDDs with multiple orders of magnitude performance improvement [7, 8] as well as (explicit-state) distributed reachability with state compression [6].

Recently, we extended Pins with separate read and write dependency matrices, special state labels for guards, and guard/transition dependency matrices. This extended interface, which we call Pins+, enables further performance improvements for the symbolic tools [40] and on-the-fly *partial-order reduction* (POR) [32] for the explicit-state tools. Pins+ will be presented in Section 2.

LTSmin offers extensive analysis of implicit state spaces through Pins: reachability analysis, including deadlock detection, action detection and invariant/assertion checking, but since recently also verification of Linear Time (LTL) and modal μ-calculus properties. The toolset and its architecture have been previously presented in, e.g., [7], [5] and [8]. This article covers the changes since then. An up-to-date overview of LTSmin is in Figure 1. The toolset is open source.[3]

The languages supported by LTSmin are listed in Table 1, including newly added support for probabilistic (Mapa) and timed systems (Uppaal) and for boolean equation systems with data types (Pbes). New is also the possibility to add a new language front-end by providing a dynamically loaded `.so`-library implementing the Pins interface. These additions will be presented in Section 3.

Table 1: Languages supported by LTSmin.

Uppaal	Timed automata.
Mapa	Process algebra for Markov automata from the Scoop tool.
Dve	The modelling language of the DiVinE model checker.
Promela	The modelling language of the Spin model checker.
mCRL2	Process algebra.
Pbes	Parameterised Boolean Equation Systems.
Etf	Built-in symbolic format.

[3] Source code available at: `https://github.com/utwente-fmt/ltsmin`.

In the explicit-state tools, new multi-core algorithms provide scalable LTL checking with POR and state compression. The symbolic tools have been extended with the multi-core *decision diagram* (DD) packages Sylvan and LDDmc, enabling multi-core symbolic analysis algorithms by parallelising their DD operations. A symbolic parity game solver has been added, enabling multi-core symbolic μ-calculus checking. These additions will be described in Section 4 and 5.

2 The Pins-architecture of LTSmin

The starting point of our approach is a generalised implicit model of systems, called the *P̲artioned N̲ext-S̲tate I̲nterface* (Pins). An overview of Pins functions is in Table 2. The functions InitialState, NextState and StateLabel are mandatory, as well as the state vector length N, the number of transition groups K, and the names and types of labels, slots and actions. Together, these give rise to a transition system, see Section 2.1.

On top of this basic Pins interface, we distinguish several axes of extensions, A1 till A∞, which together form the extended Pins+ interface. These extensions allow to expose enough structure, in the form of information about dependency relations, to enable high-performance algorithms. The first axis of such information is provided by the functions labelled A1: the read and write dependency matrices (see [40]). LTSmin's POR layer (see Section 4.3) requires guards, exported as special state labels, and the GuardMatrix, StateLabelM and DoNotAccord dependency matrices – the functions labelled A2. The definitions of the dependencies and guards are given in Section 2.2.

The simulation relation over states provided by the CoveredBy function, labelled A3, allows powerful *subsumption abstraction* [14] in our algorithms. Timed language formalisms allow such abstractions as described in Section 3.2. In the future, a symmetry-reduction layer, a la [17], could implement CoveredBy.

Other named matrices, can be added to the generic GetMatrix function, which we label A∞. This is used to increase POR's precision and facilitate statistical systems such as with Mapa (see Section 3.1).

We write $\textsc{Matrix}(x)$ as shorthand for $\{y \mid (x,y) \in \textsc{Matrix}\}$.

Table 2: Functions in Pins+ are divided along multiple axes.

Level	Function	Type	Description
B0	InitialState	$S_{\mathcal{P}}$	Initial state.
B0	NextState$_i$	$S_{\mathcal{P}} \to \wp(\mathcal{A} \times S_{\mathcal{P}})$	Successors and action label for group i.
B0	StateLabel	$S_{\mathcal{P}} \times \mathcal{L} \to \mathbb{N}$	State label.
A1	ReadMatrix	$\mathbb{B}^{K \times N}$	Read dependency matrix (Definition 2).
A1	WriteMatrix	$\mathbb{B}^{K \times N}$	Write dependency matrix (Definition 3).
A2	GuardMatrix	$\mathbb{B}^{K \times G}$	Guard/transition group matrix (Definition 5).
A2	StateLabelM	$\mathbb{B}^{G \times N}$	State label dependency matrix (Definition 4).
A2	DoNotAccord	$\mathbb{B}^{K \times K}$	Matrix for non-commutativity of groups [32].
A3	CoveredBy	$S_{\mathcal{P}} \times S_{\mathcal{P}} \to \mathbb{B}$	State covering function.
A∞	GetMatrix$_{Name}$	$\mathbb{B}^{X \times Y}$	Predefined $X \times Y$ matrix named *Name*.

2.1 Partitioned transition systems

In Pins, states are vectors of N values. We write $\langle x_1, \ldots, x_N \rangle$, for vector variables, or simply $\boldsymbol{x}$ for the state vector. Each position j in the vector ($1 \leq j \leq N$) is called a *slot* and has a unique identifier and a type, which are used in the language front-ends to specify conditions and updates. The NextState function, which computes the successors of a state, is partitioned in K disjunctive *transition groups*, such that $\textsc{NextState}(\boldsymbol{x}) = \bigcup_{1 \leq i \leq K} \textsc{NextState}_i(\boldsymbol{x})$. We have action labels $a \in \mathcal{A}$ and a set of M state labels $\mathcal{L}$. A model, available through Pins, gives rise to a *Partitioned Transition System* (PTS).

Definition 1. *A PTS is a structure $\mathcal{P} = \left\langle S_\mathcal{P}, \rightarrow_\mathcal{P}, \boldsymbol{s}^0, L \right\rangle$, where*

- $S_\mathcal{P} = S_1 \times \cdots \times S_N$ *is the set of states $\boldsymbol{s} \in S_\mathcal{P}$, which are vectors of N values,*
- $\rightarrow_\mathcal{P} = \bigcup_{i=1}^{K} \rightarrow_i$ *is the labelled transition relation, which is a union of the K transition groups $\rightarrow_i \subseteq S_\mathcal{P} \times \mathcal{A} \times S_\mathcal{P}$ (for $1 \leq i \leq K$),*
- $\boldsymbol{s}^0 = \left\langle s_1^0, \ldots, s_N^0 \right\rangle \in S_\mathcal{P}$ *is the initial state, and*
- $L : S_\mathcal{P} \times \mathcal{L} \rightarrow \mathbb{N}$ *is a state labelling function.*

We write $\boldsymbol{s} \xrightarrow{a}_i \boldsymbol{t}$ when $(\boldsymbol{s}, a, \boldsymbol{t}) \in \rightarrow_i$ for $1 \leq i \leq K$, and $\boldsymbol{s} \xrightarrow{a}_\mathcal{P} \boldsymbol{t}$ when $(\boldsymbol{s}, a, \boldsymbol{t}) \in \rightarrow_\mathcal{P}$. Considering $\mathcal{L}$ as binary state labels, $L(s)$ denotes the set of labels that hold in state s, i.e. we define $L(\boldsymbol{s}) := \{\ell \mid L(\boldsymbol{s}, \ell) \neq 0\}$.

When the LTL layer is used, the output PTS is interpreted as a *Büchi automaton*, where accepting states are marked using a special state label. When using the μ-calculus layer or the Pbes front-end, the output PTS is interpreted as a *parity game*, where two state labels encode the *player* and the *priority*. When using the Mapa front-end, the output is a *Markov automaton*, where transitions are decorated with labels, representing hyperedges with rates. For all these interpretations, the same Pins interface is used.

2.2 Dependencies and guards

The partitioning of the state vector into *slots* and of the transition relations into *transition groups*, enables to specify the *dependencies* between the two, i.e., which transition groups touch which slots of the vector.

Previously, we used a single notion of dependency; now we distinguish read, write and label dependencies [32, 40]. The read and write dependencies allow to *project* state vectors to relevant slots only, improving performance of both caching, state compression and the symbolic tools. Label dependencies enable POR. The following definitions apply to each PTS $\mathcal{P} = \left\langle S_\mathcal{P}, \rightarrow_\mathcal{P}, \boldsymbol{s}^0, L \right\rangle$.

Definition 2 (Read independence). *Transition group i is* read-independent *from state slot j, if for all $\boldsymbol{s}, \boldsymbol{t} \in S_\mathcal{P}$ with $\boldsymbol{s} \rightarrow_i \boldsymbol{t}$, we have:*

$$\forall r_j \exists r_j' \in S_j \colon \langle s_1, \ldots, r_j, \ldots, s_N \rangle \rightarrow_i \left\langle t_1, \ldots, r_j', \ldots, t_N \right\rangle \wedge r_j' \in \{r_j, t_j\} \ ,$$

i.e., whatever value r_j we plug in, the transition is still possible, the values t_k ($k \neq j$) do not depend on the value of r_j, and the value of state slot j is either copied ($r_j' = r_j$) or overwritten ($r_j' = t_j$).

Definition 3 (Write independence). *Transition group i is* write-independent *from state slot j, if:*

$$\forall \boldsymbol{s}, \boldsymbol{t} \in S_{\mathcal{P}} \colon \langle s_1, \ldots, s_j, \ldots, s_N \rangle \rightarrow_i \langle t_1, \ldots, t_j, \ldots, t_N \rangle \Rightarrow (s_j = t_j) \ ,$$

i.e., state slot j is never modified in transition group i.

Definition 4 (Label independence). *Label $l \in \mathcal{L}$ is* independent *from state slot j, if:*

$$\forall \boldsymbol{s} \in S_{\mathcal{P}}, t_j \in S_j \colon L(\langle s_1, \ldots, s_j, \ldots, s_N \rangle, l) = L(\langle s_1, \ldots, t_j, \ldots, s_N \rangle, l) \ .$$

Definition 5 (Guards). *Transition guards are represented as a subset of labels $\mathcal{G} \subseteq \mathcal{L}$. To each transition group we associate a set of guards. The guards associated with group i, denoted $\mathcal{G}(i)$, are evaluated conjunctively, i.e., transition group i is only enabled in state $\boldsymbol{s}$ if all guards $g \in \mathcal{G}(i)$ hold: if $\boldsymbol{s} \rightarrow_i \boldsymbol{t}$ then $\mathcal{G}(i) \subseteq L(\boldsymbol{s})$.*

We have provided semantic requirements for read, write and label independence relations. The language front-end must provide these dependency matrices. It can approximate dependencies using static analysis, for instance by checking occurrence of variables in expressions. Note that it is always safe to assume that groups/labels do depend on a state slot.

3 Language front-ends

LTSmin already supported the languages mCRL2 [11], DiVinE [3], and Spin's Promela [26] (through SpinS [4]). Since recently, also Mapa, Uppaal and Pbes are available, as well as the ability to load a model from a binary `.so`-file, all of which will be discussed in the current section.

3.1 MAPA: Markov Automata Process Algebra

For verification of quantitative aspects of systems, we support Mapa: *Markov Automata Process Algebra*. MA's are automata with non-deterministic choice, probabilistic choice and stochastic rates, generalizing LTS, PA, MDP and CTMC. The Scoop tool [43] offers state-space generation for Mapa specifications, applying several reduction techniques. It is part of the MaMa toolchain [25] for the quantitative analysis of Markov automata. LTSmin has been extended with a Mapa language module based on Scoop, allowing for high-performance state space generation for Mapa specifications. This language module uses Pins+ A∞ to add an *inhibit matrix* and a *confluence matrix*. The maximum progress assumption in the semantics of Markov automata forbids taking stochastic rate transitions when some action-labelled transition is enabled. This has been implemented using a *inhibit matrix*: when the higher priority transition is enabled, other transitions are inhibited. The distributed and symbolic tools of LTSmin have been extended to handle inhibit matrices for Mapa. The distributed tool also includes confluence reduction.

3.2 Uppaal: Timed Automata

The langague frontend for Uppaal timed automata is based on Opaal [12]. The C-code generated by Opaal implements the Pins+ A3 interface. Timed automata require symbolic states to handle time, for which Opaal relies on the *difference bounds matrices* from the DBM-library. This is supported by Pins, by dedicating two reserved state slots for a pointer to a symbolic time abstraction.

Subsumption abstraction can prune large parts of the PTS on-the-fly. LTSmin checks if two states subsume each other ($s \sqsubseteq t$) via the CoveredBy-relation in Pins+, which is implemented by a call to the DBM-library. The reduced state space only consists of $\sqsubseteq$-maximal states. Since $\sqsubseteq$ is a simulation relation [14], the reduced PTS is a valid abstraction of the original PTS. The reachability algorithms in the multi-core tool perform this abstraction (`opaal2lts-mc -u2`). To maintain $\sqsubseteq$-maximal states, the pointers to the DBMs are stored in a separate lockless multi-map [12].

A new LTL model checking algorithm with subsumption [36] is also supported, by extending the multi-core cndfs algorithm (see Section 5.1).

3.3 PBES: Parameterized Boolean Equation Systems

Parameterised Boolean Equation Systems (Pbess) extend Boolean equations with nested fixed points and data parameters. [24, 39] Several verification tasks are mapped to Pbess by the mCRL2 and CADP toolsets, such as model checking modal μ-calculus properties and equivalence checking. The mCRL2 toolset offers various tools for manipulating and solving Pbess. LTSmin now provides high-performance generation of parity games from Pbess [29], by viewing them as PTSs with special labels for players and priorities. The Pbes language module is available via the `pbes2lts-*` tools. The generated parity games can be solved by the means described in Section 5.

3.4 C-code via the dlopen interface

The `UNIX/POSIX dlopen`-interface allows to specify a model or a language directly in C. We show an example of how this can be done for the Sokoban game board in Figure 2. The goal of sokoban is for the player (`@`) to move all boxes (`$`) in the room to destination locations (`.`) without hitting walls (`#`). This behavior is implemented in the function `next_state` in Listing 1. For each place in the board, we reserve one slot in the state vector. We add a state label `goal`, to distinguish states where the game is finished. Finally, an `initial_state` function is defined, and functions returning dependency matrices. These need to be set using the `GBset*` functions in Listing 2. Setting the name of the plugin is also required. `sokoboard.c` is then compiled as shared library:
`gcc -shared -o sokoboard.so dlopen-impl.o sokoboard.o`.
To analyse the reachability of the goal label, call, e.g., the multi-core tool:
`pins2lts-mc sokoboard.so --invariant="!goal" --trace=solution.gcf`.

```
#####
#.$@#
#####
```

Fig. 2: Example board

Listing 1: sokoboard.c

```
void next_state(int group, int* src,
   void (*callback)(int* dst, int action))
{ int dst[3]; int action;
  memcpy(dst, src, 3);
  if (group == 0
    && src[1] == EMPTY && src[2] == MAN)
  { dst[1] = MAN; dst[2] = EMPTY;
    action = WALK_LEFT;
    callback(dst, action);
  }
  else if (group == 1
    && src[1] == MAN && src[2] == EMPTY)
  { dst[1] = EMPTY; dst[2] = MAN;
    action = WALK_RIGHT;
    callback(dst, action);
  }
  else if (group == 2 && src[0] == EMPTY
    && src[1] == BOX && src[2] == MAN)
  { dst[0] = BOX; dst[1] = MAN;
    dst[2] = EMPTY; action = PUSH_LEFT;
    callback(dst, action);
  }
}

int state_label(int* src, int label)
{return label == LABEL_GOAL && src[0] == BOX;}
int* initial_state()
{ return {EMPTY, BOX, MAN}; }

int* read_matrix()
{ return {{0,1,1}, {0,1,1}, {1,1,1}}; }

int* write_matrix()
{ return {{0,1,1}, {0,1,1}, {1,1,1}}; }

int* label_matrix()
{ return {{1,0,0}}; }
```

Listing 2: dlopen-impl.c

```
#include <ltsmin/pins.h>
#include <ltsmin/dlopen-api.h>
#include <sokoboard.h>
char pins_plugin_name[] = "sokoban";
void pins_model_init(model_t m)
{ GBsetInitialState(m, initial_state());
  GBsetNextStateLong(m, next_state);
  GBsetStateLabelLong(m, state_label);
  GBsetDMInfoRead(m, read_matrix());
  GBsetDMInfoMustWrite(m, write_matrix());
  GBsetStateLabelInfo(m, label_matrix());
}
```

4 Intermediate layers

Between language front-ends and the model checking back-ends, PINS2PINS-wrappers provide performance optimisations, state space reductions, and support for verification of LTL and μ-calculus properties. The *caching layer* reduces the number of next-state calls to the language module by storing the projected results of previous calls. The *regrouping layer* provides variable reordering, useful for the symbolic analysis tool, and reduces overhead by merging transition groups. The current section describes recent innovations in the intermediate layers, which are all language-independent and agnostic of the underlying model checking algorithm.

4.1 The LTL layer

LTSMIN supports Linear Time Logic (LTL) formulae defined by the grammar:

$\lambda ::= \texttt{true} \mid \texttt{false} \mid v\texttt{==}n \mid \texttt{!}\lambda \mid \texttt{[]}\lambda \mid \texttt{<>}\lambda \mid \texttt{X}\,\lambda \mid \lambda\texttt{\&\&}\lambda \mid \lambda\texttt{||}\lambda \mid \lambda\texttt{->}\lambda \mid \lambda\texttt{<->}\lambda \mid \lambda\texttt{U}\lambda \mid \lambda\texttt{R}\lambda$

The negated formula is translated to a Büchi automaton using `ltl2ba` [23]. The product of the PTS and the Büchi automaton is computed on-the-fly, i.e., the layer does not perform reachability in advance. Instead, it wraps the NEXTSTATE function of a language module in its own NEXTSTATE function, which synchronises the translated Büchi automaton on the state labels or slot values of successor states (the expression $v \texttt{==} n$ can refer to a label or a slot named v). The synchronised successors are then passed to the analysis algorithm. A label added by the layer allows the algorithm to distinguish Büchi accepting states. On-the-fly accepting cycle detection algorithms are described in Section 5.1.

4.2 The μ-calculus layer

The modal μ-calculus layer supports formulae defined by the grammar:

$$\varphi ::= \texttt{true} \mid \texttt{false} \mid \{v \texttt{=} e\} \mid \texttt{!}\{v \texttt{=} e\} \mid \varphi \,\texttt{\&\&}\, \varphi \mid \varphi \,\texttt{||}\, \varphi \mid \texttt{Z} \mid \sigma\texttt{Z}\,\texttt{.}\,\varphi \mid \texttt{[}\alpha\texttt{]}\varphi \mid \texttt{<}\alpha\texttt{>}\varphi \ ,$$

where v is a state variable, e is a value, $\sigma \in \{\texttt{mu}, \texttt{nu}\}$ is a minimal (`mu`) or maximal (`nu`) fixpoint operator, and α is an action label.

The μ-calculus Pins2Pins layer reads a modal μ-calculus property φ from a file, provided using the `--mucalc` option, and generates a parity game, which is the product $\mathcal{P} \times \varphi$ of the formula and a system $\mathcal{P}$ that is explored through Pins. Like the Büchi automaton, this game is generated on-the-fly. The explicit-state tools can write the parity game to a file which can be converted to a format that is readable by the tools `pgsolver` [20] and `pbespgsolve` (from mCRL2). The symbolic tools can write the game to a file, which can be solved by the new LTSmin tool `spgsolver`. The symbolic tools also have an alternative implementation for μ-calculus model checking (available through the `--mu` option), which is a fixpoint algorithm applied to the system after reachability. This implementation also supports CTL* through the translation in [13] (the `--ctl-star` option).

4.3 The partial-order reduction layer

Partial-Order Reduction (POR, [30, 44]) greatly reduces a PTS by pruning irrelevant interleavings. LTSmin implements POR as an intermediate layer (cf. Figure 1). This POR layer (`--por`) wraps the next-state function of any language module, and provides a reduced state space to any analysis tool by replacing it with an *on-the-fly* reduction function: $\textsc{PorState}(\boldsymbol{s}) \subseteq \textsc{NextState}(\boldsymbol{s})$.

We rephrased the stubborn set method [44] in terms of guards [32] to achieve language independence. For any state, a set of (enabled or disabled) stubborn transitions is computed, and $\textsc{PorState}(\boldsymbol{s})$ corresponds to the enabled stubborn transitions. The stubborn set should (1) contain at least one enabled transition if one exists; (2) contain all non-commuting transitions for the *enabled* selected transitions; and (3) contain a *necessary-enabling set* of transitions for the *disabled* selected transitions.

To compute stubborn sets, LTSmin needs structural model information via the Pins+ A2 interface. For effective POR, we extended Pins transitions with guards (Definition 5). In particular, a language module must declare when transitions commute, and the dependencies of guards (Definition 4). The former is declared with the $\textsc{DoNotAccord} : \mathbb{B}^{K\times K}$-matrix. It should satisfy:

Definition 6 (Do-not-accord). *Transition groups i and j are according, if*

$$\forall s, s_i, s_j \in \boldsymbol{s} : s \to_i s_i \wedge s \to_j s_j \Rightarrow \exists t \in \boldsymbol{s} : s_i \to_j t \wedge s_j \to_i t$$

Otherwise, they must be declared conflicting in the DoNotAccord *matrix.*

Next, the POR layer derives an enabling relation from the provided dependency information. A transition i can only enable guard g, if i writes to a variable that g depends on: $\textsc{EnableMatrix}^{K\times G} \equiv \{(i,g) \mid \textsc{WriteMatrix}(i) \cap \textsc{StateLabelM}(g) \neq \emptyset\}$. A set of necessary-enabling transitions for a disabled transition j can then be found by selecting *one* disabled guard g of j and taking all transitions that may enable g: $\textsc{EnableMatrix}(g)$.

Optionally, these notions can be further refined by the language module for more reduction, by providing additional PINS+ A_∞ matrices. For example, the language module can compute a detailed ENABLEMATRIX by static analysis on assignment expressions and guards, or a DISABLEMATRIX and a CO-ENABLED-MATRIX on guards to leverage the power of *necessary disabling sets* [32].

LTSMIN contains heuristics to compute small stubborn sets efficiently. The user can select a fast heuristic search (`--por=heur`) or the subset-minimal deletion algorithm (`--por=del`). POR preserves at least all deadlock states. The multi-core algorithms in LTSMIN preserves all liveness properties, but this requires additional interaction with the LTL layer (to know the visible state properties) and the analysis algorithm (to avoid the so-called ignoring problem, see Section 5.1).

The POR layer is incompatible with the symbolic analysis tool, since after partial-order reduction all locality and dependence information is lost. The distributed analysis tool currently only supports POR for deadlock detection.

5 Algorithmic back-ends

LTSMIN has *distributed* [6], *multi-core* [12,19,35,37], and *symbolic* [16] back-ends. Furthermore, connectors to the model-based testing tool JTORX, are available as the `*2torx` tools, and to the CADP toolset, through the OPEN/CÆSAR interface, as the `*-open` tools. Since its early origins, LTSMIN has a sequential (`ltsmin-reduce`) and a distributed (`ltsmin-reduce-dist`) reduction tool. Both provide strong and branching bisimulation minimisation, while the sequential tool also supports divergence sensitivity, cycle elimination and minimisation modulo lumping. In the current section, we highlight the multi-core algorithms for explicit-state and symbolic model checking, and the symbolic parity game solver.

5.1 Multi-core reachability, POR and LTL checking

Since [37], LTSMIN's multi-core tools were extended beyond reachability analysis, while improving state compression.

At the basis of our multi-core algorithms is still a lockless hash or tree table (`--state=table/tree`) for shared state storage coupled with a dynamic load balancer [33,34]. However, state compression has been enhanced by extending the tree with a concurrent Cleary compact hash table [10,45] (`--state=cleary-tree`), regularly yielding compressed sizes of 4 bytes per state [35, Tab. 11.4] without compromising completeness. *Incremental tree compression* [35, Sec. 3.3.4] uses the WRITEMATRIX from PINS+ to limit the number of hash computations, ensuring similar scalability and performance to plain hash tables [34].

LTSmin's state storage provides ample flexibility for different search orders, enabling LTL verification by traditial linear-time algorithms, in particular nested depth-first search (NDFS). The cndfs algorithm (`--strategy=cndfs`) runs multiple semi-independent DFS searches which are carefully synchronized in the backtrack [19]. DFS_{fifo} (`--strategy=dfsfifo`) combines this with breadth-first search to find livelocks, an important subet of LTL [31]. The latter algorithm avoids the *ignoring problem* in POR [18], but the combination of POR and full LTL was until recently not possible in multi-core LTSmin.

The ignoring problem occurs when POR consistently prunes the same relevant action infinitely often [18]. It can be solved by fully exploring one state $\boldsymbol{s}$ along each cycle in the PTS ($\text{PorState}(\boldsymbol{s}) := \text{NextState}(\boldsymbol{s})$). The problem of detecting cycles while constructing the PTS on-the-fly is usually solved with DFS [18], which is hard to parallelize [2]. Exploiting the DFS-based parallel algorithms, this problem is efficiently solved with a new *parallel cycle proviso* [38] (`--proviso=cndfs`). Cycles are exchanged with the POR layer via Pins.

We have shown before [4, 31] that our multi-core reachability approach exhibits almost ideal scalability up to 48 cores, even for very fast NextState implementations, like SpinS. cndfs outperforms [4, 19] other algorithms for multi-core LTL model checking [1, 27]. For further information on multi-core algorithms and data structures, see [35].

5.2 Multi-Core decision diagrams

The symbolic back-end of LTSmin has been improved in several ways. *First*, it has been extended with the multi-core decision diagram packages Sylvan and LDDmc [16] (`--vset=sylvan/lddmc`). *Second*, two parallel reachability algorithms have been added, based on the task-based parallelism framework Lace [15, 16]. *Third*, the distinction between read and write dependencies in Pins+ improves the symbolic algorithms by reducing the size of transitions relations [40].

5.3 Symbolic parity game solving

We implemented Zielonka's recursive algorithm [46] using decision diagrams, which is available in the symbolic tools (`--pg-solve`) or stand-alone in `spgsolver`. The tool solves symbolic parity games, generated by the symbolic tool, and returns whether the game has a winning strategy for player **0**. When the game has been generated using the μ-calculus layer, this answer corresponds to whether $\mathcal{P} \models \varphi$. The solver currently does not output a strategy.

6 Case studies

The following two case studies demonstrate the use of having both explicit-state and symbolic approaches to attack problems. The second case also demonstrates the power of μ-calculus model checking for solving games.[4]

[4] Installation instructions and case studies: `https://github.com/utwente-fmt/ltsmin-tacas2015`.

6.1 Attacking the RERS Challenge

LTSmin participated in the RERS [28,42] challenges of 2012, 2013 [41] and 2014, winning several first prizes. The flexibility of LTSmin allowed us to address the RERS challenge problems from different angles. We will discuss three ways to connect LTSmin to the challenge problems. We also demonstrate how LTSmin's backend tools check for assertion errors and temporal properties.

Each RERS problem consists of a large C-program, encoding a system of Event-Condition-Action rules. The program operates in an infinite loop modifying the global state. In each iteration, the program gets an input from a small alphabet and checks for assertion errors. If the condition of one of the rules is met, it generates an output symbol and changes the state for the next iteration.

Linking LTSmin *to RERS programs.* In the first approach, a RERS C-program is translated to a modelling language that is already supported by LTSmin. We took this approach in 2012, by translating RERS programs to Promela and to mCRL2. The translations are rather straightforward, since the ECA-rules can be readily reverse-engineered from the C-programs.

A fundamentally different approach is to create a new language module for (a subclass of) C-programs. This was our approach in 2013 and 2014. In 2013, we just wrapped the body of the main-loop into a single, monolithic next-state function, compiled in a separate binary (`.so` file). This is a robust solution, since the original code is run during model checking.

This monolithic approach worked fine for multi-core model checking. However, it leads to a lack of "locality": there is only one transition, which reads and writes all state variables. In order to apply symbolic model checking, our 2014 approach was to adapt the C-language module, by providing a separate transition group for each ECA rule, checking its conditions and applying the state change. Edge labels are added, to indicate the input and output values and the assertion violations. In this partitioned view, every transition group only touches a couple of variables, enabling symbolic model checking. With Sylvan linked to LTSmin, RERS 2014 was the first large case to which we applied multi-core symbolic model checking.

Using LTSmin *to check properties.* We show here how LTSmin can be used to check properties of Problem2.c from the RERS challenge 2014. The original C-code is optimized and transformed as indicated above. We assume that the transformed code is compiled and available in a shared object `Problem.so`.

In the following dialogue, we request the symbolic model checker to find all actions with prefix `error`. Flag `--no-exit` avoids that LTSmin exits after finding the first error. We also request to store concrete error traces in a subdirectory and print one of them in human readable format. LTSmin quickly finds 23 errors.

```
> pins2lts-sym Problem.so --action=error --trace=Error/ --no-exit
pins2lts-sym: writing to file: Error/error_6.gcf
pins2lts-sym: writing to file: Error/error_8.gcf
^C
```

```
> ltsmin-printtrace Error/error_6.gcf | grep action | cut -f3 -d=
"input_3" "output_20" ... "input_3" "output_26" "error_6"
```

Actually, the state space of this example is very big and LTSmin keeps searching for more errors. In order to do an exhaustive search, we throw more power, by using the parallel BFS strategy and Sylvan's (enlarged) multi-core multi-way decision diagrams [16]. We also request static variable reordering, to keep the MDDs small. With `--when`, we request timing information. The following experiments are run on a 48-core machine with 132 GB RAM. The parallel symbolic model checker of LTSmin computes the full state space within 2 minutes. All 1.75 billion states, divided over 480 BFS levels, are stored in about 1 million MDD nodes.

```
> pins2lts-sym Problem.so --order=par-prev --regroup=gs --when \
    --vset=lddmc --lddmc-tablesize=30 --lddmc-cachesize=28
pins2lts-sym: Using 48 CPUs
pins2lts-sym, 28.076: level 90 is finished
pins2lts-sym, 113.768: level 480 is finished
pins2lts-sym: ... 1750528171 (~1.75e+09) states, 1158486 BDD nodes
```

Alternatively, we may decide to switch to the *explicit-state* multi-core reachability engine [33]. We request a strict breadth-first strategy, to facilitate comparison with the symbolic run. To squeeze the maximum out of our machine, we combine recursive tree compression [34] with Cleary's compact hashing [10, 45]. Within a minute we learn that there are no new errors up to depth 90. LTSmin is able to traverse the full state space exhaustively within 5 minutes, generating over 1.75 billion states and 2.4 billion transitions.

```
> pins2lts-mc Problem.so --strategy=sbfs --state=cleary-tree --when
pins2lts-mc(18/48), 45.791: ~90 levels ~125829120 states ~191075472 trans
pins2lts-mc( 0/48), 298.565: Explored 1750528171 states 2445589869 trans
```

The explicit multi-core tool can also check LTL properties, using multi-core NDFS (`cndfs`, [19]). The LTL formula refers to integer variables in the original C-program `a94` and `a95`. With `--ltl-semantics=ltsmin` we insist on checking infinite paths only, i.e., we don't consider traces that end in an assertion error. The violated trace can be printed as above, and will end in a lasso.

```
> pins2lts-mc Problem.so --ltl='a94==9 U a95==12' \
     --strategy=cndfs --ltl-semantics=ltsmin --trace=Error/ltl.gcf
pins2lts-mc( 0/48): Accepting cycle FOUND at depth 11!
pins2lts-mc( 3/48): Writing trace to ltl.gcf
```

6.2 Solving Connect Four

We explore the Connect Four game, originally played on a 7×6 board between two players: *yellow* and *red*, which is available in the examples directory of mCRL2. For the first run, we reduced the board size to 5×4, for which the model has 7,039,582 states.

```
rrrrrrrrrrrrrrrrrrrr+
rrrrrrrrrrrrrrrrrrrr+
+----+----+----+----+
--+----+----+----+--+
-+----+----+----+---+
----+----+----+----++
---+----+----+----+-+
+----+----+----+----+
--+----+----+----+--+
-+----+----+----+---+
----+----+----+----++
---+----+----+----+-+
```

(a) Monolithic.

```
+----+----+----+----+
--+----+----+----+--+
-+----+----+----+---+
----+----+----+----++
---+----+----+----+-+
rrrr-----------------
---r---r---r---r-----
r-----r-----r-----r--
r----r----r----r-----
----------rrrr-------
-----rrrr------------
-rrrr----------------
----r---r---r---r----
-r-----r-----r-----r-
--r----r----r----r---
-r----r----r----r----
---------------rrrr--
-----------rrrr------
------rrrr-----------
----r----r----r----r-
---r----r----r----r--
----------------rrrr-
```

(b) Separated.

Fig. 3: Matrix.

The matrix is in Fig. 3a. LTS generation using `lps2lts` (mCRL2) takes 85 seconds and 539 MB. Using 16 cores, multi-core LTSmin takes 22 seconds and 63 MB. But symbolic LTSmin needs 124 seconds.

This is caused by the monolithic summands in the specification (the dense rows in Figure 3a), representing the winning condition. We split the condition in separate parts and let the game continue after a winning move has been done. The matrix of the problem becomes more sparse, see Figure 3b. Note that the four `r`'s in a row correspond to the four winning tiles. The symbolic tool now generates a different state space of the same game (5,464,758 states) in less than a second. mCRL2 takes 88 seconds for this version. The exploration time of LTSmin for a 6×5 board is 2.4 seconds, for 9.78×10^9 states in 41,239 MDD nodes.

Next, we generate a PBES with mCRL2, to encode the μ-calculus property (in file `yellow_wins.mcl`) that player Yellow has a winning strategy, and solve it with LTSmin:

```
mu X . [Wins(Red)]false && <Move>(<Wins(Yellow)>true || [Move]X)
> lps2pbes -s -f yellow_wins.mcl four5x4.lps four5x4.pbes
> pbes2lts-sym --mcrl2=-rjitty --regroup=gs --pg-solve  \
    --vset=lddmc --order=par-prev four5x4.pbes
```

For the 5×4-board mCRL2 takes 193 seconds, but the symbolic tool of LTSmin 77 seconds, to compute that the starting player has no winning strategy.

7 Discussion

There are several toolsets that take a similar approach, supporting a generic interface, or offer similar, multi-core or symbolic, analysis algorithms. The table below provides a brief qualitative comparison of the available types of algorithms and the supported logics. The last column indicates wether multiple input languages are supported, and if so, through which interface.

The Pins interface is the main differentiator of LTSmin. It is sufficiently general to support a wide range of modelling languages. At the same time, the

Toolset	multi-core	distributed	symbolic	μ-calculus	LTL	POR	Language
LTSmin	yes	yes	yes	yes	yes	yes	`any` (Pins)
mCRL2 [11]	no	no	no	yes	no	no	`fixed`
CADP [22]	no	yes	no	yes*	no	yes	`any` (Open/C)
DiVinE [3]	yes	yes	no	no	yes	yes	`any` (CESMI)
Spin	yes	yes	no	no	yes	yes	`fixed`
NuSMV [9]	no	no	yes	no	yes	no	`fixed`

* CADP supports μ-calculus formulae up to alternation depth 2.

dependency matrices provide sufficient structural model information to exploit locality in the analysis algorithms. As a consequence, LTSmin is the only language-agnostic model checker that supports on-the-fly symbolic verification and full LTL model checking with POR. Due to the modular architecture, the user can freely choose a verification strategy depending on the problem at hand.

References

1. Barnat, J., Brim, L., Ročkai, P.: A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In: ICFEM'09. LNCS, vol. 5885, pp. 407–425. Springer (2009)
2. Barnat, J., Brim, L., Ročkai, P.: Parallel Partial Order Reduction with Topological Sort Proviso. In: SEFM 2010. pp. 222–231. IEEE (2010)
3. Barnat, J., et al.: DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: CAV 2013. LNCS, vol. 8044, pp. 863–868. Springer (2013)
4. van der Berg, F.I., Laarman, A.W.: SpinS: Extending LTSmin with Promela through SpinJa. In: PDMC 2012. ENTCS, vol. 296, pp. 95–105 (2013)
5. Blom, S.C.C., van de Pol, J.C., Weber, M.: Bridging the Gap between Enumerative and Symbolic Model Checkers. TR-CTIT-09-30, University of Twente (2009)
6. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A Database Approach to Distributed State-Space Generation. Journal of Logic and Computation 21(1), 45–62 (2009)
7. Blom, S., van de Pol, J.: Symbolic Reachability for Process Algebras with Recursive Data Types. In: ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer (2008)
8. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and Symbolic Reachability. In: CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer (2010)
9. Cimatti, A., et al.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: CAV 2002. LNCS, vol. 2404. Springer (2002)
10. Cleary, J.G.: Compact Hash Tables Using Bidirectional Linear Probing. IEEE Transactions on Computers C-33(9), 828–834 (1984)
11. Cranen, S., et al.: An Overview of the mCRL2 Toolset and Its Recent Advances. In: TACAS 2013. LNCS, vol. 7795, pp. 199–213. Springer (2013)
12. Dalsgaard, A.E., et al.: Multi-core Reachability for Timed Automata. In: FORMATS 2012. LNCS, vol. 7595, pp. 91–106. Springer (2012)
13. Dam, M.: Translating CTL* into the modal μ-calculus. Report ECS-LFCS-90-123, LFCS, University of Edinburgh (1990)
14. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer (1998)
15. van Dijk, T., van de Pol, J.C.: Lace: non-blocking split deque for work-stealing. In: MuCoCoS 2014. LNCS, vol. 8806, pp. 206–217 (2014)
16. van Dijk, T., van de Pol, J.C.: Sylvan: Multi-core Decision Diagrams. In: TACAS 2015. Springer (2015)
17. Emerson, E.A., Wahl, T.: Dynamic symmetry reduction. In: TACAS 2005. LNCS, vol. 3440, pp. 382–396. Springer (2005)
18. Evangelista, S., Pajault, C.: Solving the Ignoring Problem for Partial Order Reduction. STTT 12, 155–170 (2010)
19. Evangelista, S., et al.: Improved Multi-core Nested Depth-First Search. In: ATVA 2012. LNCS, vol. 7561, pp. 269–283. Springer (2012)
20. Friedmann, O., Lange, M.: PGSolver (2008), https://github.com/tcsprojects/pgsolver

21. Garavel, H.: OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In: TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer (1998)
22. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT 15(2), 89–107 (2013)
23. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer (2001)
24. Groote, J.F., Willemse, T.A.C.: Model-checking processes with data. Science of Computer Programming 56(3), 251–273 (2005)
25. Guck, D., et al.: Analysis of Timed and Long-Run Objectives for Markov Automata. Logical Methods in Computer Science 10(3) (2014)
26. Holzmann, G.J.: The model checker SPIN. IEEE TSE 23, 279–295 (1997)
27. Holzmann, G.J.: Parallelizing the SPIN Model Checker. In: SPIN 2012, LNCS, vol. 7385, pp. 155–171. Springer (2012)
28. Howar, F., et al.: Rigorous examination of reactive systems. STTT 16(5) (2014)
29. Kant, G., van de Pol, J.: Generating and Solving Symbolic Parity Games. In: GRAPHITE 2014. EPTCS, vol. 159, pp. 2–14 (2014)
30. Katz, S., Peled, D.: An efficient verification method for parallel and distributed programs. In: REX Workshop. LNCS, vol. 354, pp. 489–507. Springer (1988)
31. Laarman, A., Faragó, D.: Improved On-The-Fly Livelock Detection. In: NFM 2013. LNCS, vol. 7871, pp. 32–47. Springer (2013)
32. Laarman, A., Pater, E., van de Pol, J.C., Hansen, H.: Guard-based partial-order reduction. STTT pp. 1–22 (2014)
33. Laarman, A., van de Pol, J., Weber, M.: Boosting Multi-Core Reachability Performance with Shared Hash Tables. In: FMCAD 2010. pp. 247–255. IEEE (2010)
34. Laarman, A., van de Pol, J., Weber, M.: Parallel Recursive State Compression for Free. In: SPIN 2011. LNCS, vol. 6823, pp. 38–56. Springer (2011)
35. Laarman, A.: Scalable Multi-Core Model Checking. Ph.D. thesis, University of Twente (2014)
36. Laarman, A., Olesen, M.C., Dalsgaard, A.E., Larsen, K.G., van de Pol, J.: Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction. In: CAV 2013, LNCS, vol. 8044, pp. 968–983. Springer (2013)
37. Laarman, A., van de Pol, J., Weber, M.: Multi-Core LTSmin: Marrying Modularity and Scalability. In: NFM 2011. LNCS, vol. 6617, pp. 506–511 (2011)
38. Laarman, A., Wijs, A.: Partial-Order Reduction for Multi-Core LTL Model Checking. In: HVC 2014. LNCS, vol. 8855, pp. 267–283. Springer (2014)
39. Mateescu, R.: Local Model-Checking of an Alternation-Free Value-Based Modal Mu-Calculus. In: VMCAI 1998 (1998)
40. Meijer, J.J.G., Kant, G., van de Pol, J.C., Blom, S.C.C.: Read, Write and Copy Dependencies for Symbolic Model Checking. In: HVC 2014. LNCS, vol. 8855, pp. 204–219. Springer (2014)
41. van de Pol, J., Ruys, T.C., te Brinke, S.: Thoughtful brute-force attack of the RERS 2012 and 2013 Challenges. STTT 16(5), 481–491 (2014)
42. RERS – Rigorous Examination of Reactive Systems, `http://rers-challenge.org/`
43. Timmer, M.: Efficient modelling, generation and analysis of Markov automata. Ph.D. thesis, University of Twente (2013)
44. Valmari, A.: Eliminating Redundant Interleavings During Concurrent Program Verification. In: PARLE, LNCS, vol. 366, pp. 89–103. Springer (1989)
45. van der Vegt, S., Laarman, A.W.: A parallel compact hash table. In: MEMICS 2011. LNCS, vol. 7119, pp. 191–204. Springer (2012)
46. Zielonka, W.: Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. Theoretical Computer Science 200(1–2), 135–183 (1998)