Symbolic Reachability Analysis of B Through ProB and LTSmin

Jens Bendisposto[1], Philipp Körner[1], Michael Leuschel[1], Jeroen Meijer[2(✉)], Jaco van de Pol[2], Helen Treharne[3], and Jorden Whitefield[3]

[1] Institut für Informatik, Heinrich Heine University Düsseldorf, Düsseldorf, Germany
{bendisposto,leuschel}@cs.uni-duesseldorf.de,
p.koerner@uni-duesseldorf.de
[2] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{j.j.g.meijer,j.c.vandepol}@utwente.nl
[3] Department of Computer Science, University of Surrey, Guildford, UK
{h.treharne,j.whitefield}@surrey.ac.uk

Abstract. We present a symbolic reachability analysis approach for B that can provide a significant speedup over traditional explicit state model checking. The symbolic analysis is implemented by linking ProB to LTSmin, a high-performance language independent model checker. The link is achieved via LTSmin's Pins interface, allowing ProB to benefit from LTSmin's analysis algorithms, while only writing a few hundred lines of glue-code, along with a bridge between ProB and C using ØMQ. ProB supports model checking of several formal specification languages such as B, Event-B, Z and TLA$^+$. Our experiments are based on a wide variety of B-Method and Event-B models to demonstrate the efficiency of the new link. Among the tested categories are state space generation and deadlock detection; but action detection and invariant checking are also feasible in principle. In many cases we observe speedups of several orders of magnitude. We also compare the results with other approaches for improving model checking, such as partial order reduction or symmetry reduction. We thus provide a new scalable, symbolic analysis algorithm for the B-Method and Event-B, along with a platform to integrate other model checking improvements via LTSmin in the future.

Keywords: B-Method · Event-B · ProB · LTSmin · Symbolic reachability

1 Introduction

In this paper we describe the process, technique and design decisions we made for integrating the two tooling sets: LTSmin and ProB. Bicarregui et al. suggested, in a review of projects which applied formal methods [6], that providing useable

J. Meijer—Supported by STW SUMBAT grant: 13859.
J. van de Pol—Supported by the 3TU.BSR project.
J. Whitefield—Partly supported by EPSRC grant: EP/M506655/1.

E. Ábrahám and M. Huisman (Eds.): IFM 2016, LNCS 9681, pp. 275–291, 2016.
DOI: 10.1007/978-3-319-33693-0_18

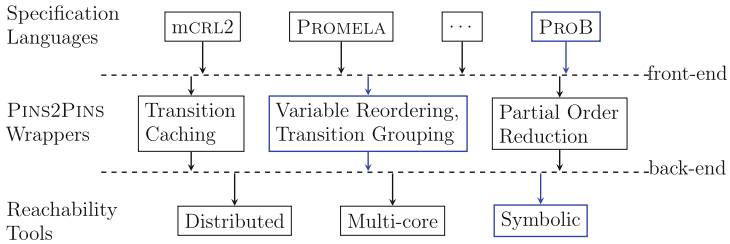

Fig. 1. Modular Pins architecture of LTSmin [17]

tools remained a challenge. Recent use of the ProB tool in a rail system case study [16], where model checking large industrial sized complex specifications was performed, illustrated that there continues to be limitations with the tooling. Model checking CSP‖B [28] specifications in ProB was the original motivator for this research, and based on a promising initial exploration [30], this paper defines a systematic integration of the two tooling sets.

LTSmin is a high-performance language-independent model checker that allows numerous modelling language front-ends to be connected to various analysis algorithms, through a common interface, as shown in Fig. 1. It offers a wide spectrum of parallel and symbolic algorithms to deal with the state space explosion of different verification problems. This connecting interface is called the Partitioned Interface to the Next-State function (Pins), the basis of which consists of a state-vector definition, an initial state, a partitioned successor function (NextState), and labelling functions [17]. It is through Pins that we have been able to leverage the ProB tool, therefore allowing us to take advantage of LTSmin's algorithmic back-ends. In this paper we focus on the new ProB language front-end, the grouping of transitions, and the symbolic back-end. In Sect. 5 we also briefly discuss state variable orders.

ProB [19] is an animator and model checker for many different formal languages [26], including the classical B-Method [2], Event-B [1], CSP, CSP‖B, Z and Tla$^+$. ProB can perform automatic or step by step animation of B machines, and can be used to systematically verify the behaviour of machines. The verification can identify states which do not meet the invariants, do not satisfy assertions or that deadlock. At the heart of ProB is a constraint solver, which enables the tool to animate and model check high-level specifications. The built-in model checker is a straightforward, explicit state model checker (albeit augmented with various features such as symmetry reduction [20] or partial order reduction [11]). The explicit state model checker Tlc can also be used as a backend [12].

The purpose of this paper is to make use of the advanced features of the LTSmin model checker, such as symbolic reachability analysis, by linking the ProB state exploration engine with LTSmin. This is achieved through a C programming interface [4] within the ProB tool, allowing the representation of a state to be compatible for LTSmin's consumption. In this paper the integration

focuses on what is required in order to perform symbolic reachability analysis of B-Method and Event-B specifications. The contribution of this research is a new tool integration, which can be used as a platform for further extensions.

The paper is structured as follows: Sect. 2 presents an overview of the B-Method, a running example and an illustration of definitions of transition systems used by LTSmin. Section 3 details the symbolic reachability analysis and Sect. 4 outlines the implementation details. Section 5 provides empirical results from performing reachability analysis benchmarking examples in ProB alone and using the new integration of the two tools. The paper concludes in Sect. 6 with reflections and future work.

2 Preliminaries: B-Method and Transition Systems

In this section we provide an overview of the B-Method and the foundations used within LTSmin.

A B machine consists of a collection of clauses and a collection of operations. The **MACHINE** clause declares the abstract machine and gives it its name. The **VARIABLES** clause declares the variables that are used to carry the state information within the machine. The **INVARIANT** clause gives the type of the variables, and more generally it also contains any other constraints on the allowable machine states. The **INITIALISATION** clause determines the initial state(s) of the machine. Operations in a machine are events that change the state of a machine and can have input parameters. Operations can be of the form **SELECT** P **THEN** S **END** where P is a guard and S is the action part of the operation. The predicate P must include the type of any input variables and also give conditions on when the operation can be performed. When the guard of an operation is true then the operation is enabled and can be performed. If the guard is the simple predicate *true* then the operation form is simplified to **BEGIN** S **END**. An operation can also be of the form **PRE** P **THEN** S **END** so that the predicate is a precondition and if the operation is invoked outside its precondition then this results in a divergence (we do not illustrate this in our running example). Finally, the action part of an operation is a *generalised substitution*, which can consist of one or more assignment statements (in parallel) to update the state or assign to the output variables of an operation. Conditional statements and nondeterministic choice statements are also permitted in the body of the operation. The example in Fig. 2 illustrates the *MutexSimple* machine with three variables and five operations. Its initial state is deterministic and *wait* is set to MAXINT. For MAXINT=1 we get 4 states; the state space constructed by ProB can be found in Fig. 3. From the initial state only the guards for `Enter` and `Leave` are true. Following an `Enter` operation the value of the `cs` variable is true which means that the guard of the `CS_Active` operation is true and the system can indicate that it is in the critical section by performing the `CS_Active` operation.

The example presented could also be considered as an **Event-B** example since it is a simple guarded system. We do not elaborate further on the notation

```
1  MACHINE MutexSimple
2  VARIABLES cs, wait, finished
3  INVARIANT
4       cs: BOOL & wait: NATURAL & finished: NATURAL
5  INITIALISATION cs := FALSE || wait := MAXINT || finished := 0
6  OPERATIONS
7       Enter      = SELECT cs = FALSE & wait > 0 THEN
                       cs := TRUE || wait := wait - 1 END;
8       Exit       = SELECT cs = TRUE THEN
                       cs := FALSE || finished := finished + 1 END;
9       Leave      = BEGIN cs := FALSE END;
10      CS_Active  = SELECT cs = TRUE THEN skip END;
11      Restart    = SELECT finished > 0 THEN
                       wait := wait + 1 || finished := finished - 1 END
12 END
```

Fig. 2. *MutexSimple* B-Method machine example

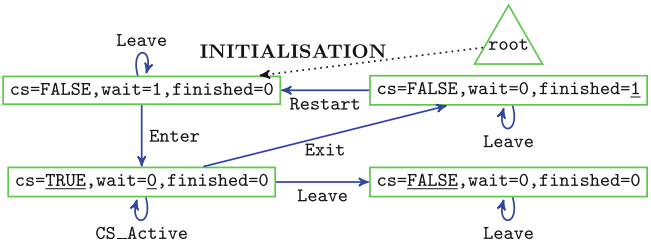

Fig. 3. *MutexSimple* statespace for `MAXINT=1`

of Event-B in this paper but note that the results in the subsequent sections are also applicable to Event-B.

As far as symbolic reachability analysis is concerned, a formal model is seen to denote a transition system. LTSMIN adopts the following definition:

Definition 1 (Transition System). *A Transition System (TS) is a structure* $(S, \rightarrow, I)$*, where* S *is a set of states,* $\rightarrow \subseteq S \times S$ *is a transition relation and* $I \subseteq S$ *is a set of initial states. Furthermore, let* $\rightarrow^*$ *be the reflexive and transitive closure of* $\rightarrow$*, then the set of reachable states is* $R = \{s \in S \mid \exists\, s' \in I \,.\, s' \rightarrow^* s\}$.

A B-Method and Event-B model induces such a transition system: initial states are defined by the initialisation clause and the individual operations together define the transition relation $\rightarrow$. Figure 3 shows the transition system[1] for the machine in Fig. 2. As can be seen in Fig. 3, the transition relation is annotated with operation names. For symbolic reachability analysis it is actually very important that we divide the transition relation into groups, leading to the concept of a partitioned transition system:

[1] One subtle issue is that LTSMIN actually only supports a single initial state; this is solved by introducing the artificial *root* state linked to the initial states proper. We ignore this technical issue in the paper.

Definition 2 (Partitioned Transition System). *A Partitioned Transition System (PTS) is a structure* $\mathcal{P} = (S^{\mathrm{N}}, G, \rightarrow^{\mathrm{M}}, I^{\mathrm{N}})$*, where*

- $S^{\mathrm{N}} = S_1 \times \ldots \times S_{\mathrm{N}}$ *is the set of states, which are vectors of* N *values,*
- $G = (\rightarrow_1, \ldots, \rightarrow_{\mathrm{M}})$ *is a vector of* M *transition groups* $\rightarrow_i \subseteq S^{\mathrm{N}} \times S^{\mathrm{N}}$ $(\forall 1 \leq i \leq \mathrm{M})$
- $\rightarrow^{\mathrm{M}} = \bigcup_{i=1}^{\mathrm{M}} \rightarrow_i$ *is the overall transition relation induced by* G*, i.e., the union of the* M *transition groups, and*
- $I^{\mathrm{N}} \subseteq S^{\mathrm{N}}$ *is the set of initial states.*

We write $s \rightarrow_i t$ *when* $(s,t) \in \rightarrow_i$ *for* $1 \leq i \leq \mathrm{M}$*, and* $s \rightarrow^{\mathrm{M}} t$ *when* $(s,t) \in \rightarrow^{\mathrm{M}}$*.*

For example $I^{\mathrm{N}} = \{(FALSE, MAXINT, 0)\}$ in the running example. Note that G in Definition 2 does not necessarily form a partition of $\rightarrow^{\mathrm{M}}$, overlap is allowed between the individual groups.

3 Symbolic Reachability Analysis for B

Computing the set of reachable states (R) of a transition system can be done efficiently with symbolic algorithms if many transition groups $\rightarrow_i$ touch only a few variables. This concept is known as event locality [9]. Many models of transition systems in the B-Method employ event locality. In the B-Method event locality occurs in operations, where only a few variables are read from, or written to. For example in Fig. 2 operation `CS_Active` only reads from `cs` and `Leave` only writes to `cs`. This event locality benefits the symbolic reachability analysis, so that the algorithm is capable of coping with the well known state space explosion problem. Since the B-Method employs event locality we build on the foundations of earlier work on LTSMin [7,23] and extend it to ProB. To perform symbolic reachability analysis of the B-Method, ProB should provide LTSMin with read matrices and write matrices. These matrices inform LTSMin about the locality of events in the B-Method.

Read independence is an important concept, it allows one to reuse the successor states computed in one state s for all states s' which differ just by read-independent variables from s, and vice versa.

Definition 3 (Read Independence). *Two state vectors* s, s' *are equivalent except on index* j*, denoted by* $s \approx_j s'$*, iff* $\forall\, k \neq j : s_k = s'_k$*.*

Transition group i *is* read-overwrite independent *from state variable* j*, iff* $\forall\, s, s', t \in S^{\mathrm{N}}$ *such that* $s \approx_j s'$ *and* $s \rightarrow_i t$*, we have that* $s' \rightarrow_i t$*.*

Transition group i *is* read-copy independent *from state variable* j*, iff* $\forall\, s, s', t \in S^{\mathrm{N}}$ *such that* $s \approx_j s'$ *and* $s \rightarrow_i t$*, we have that* $s' \rightarrow_i (t_1, \ldots, t_{j-1}, s'_j, t_{j+1}, \ldots, t_{\mathrm{N}})$*.*

A transition group is read independent *iff it is either read-overwrite or read-copy independent.*

If an event never reads but may write to a variable j it generally does not satisfy the above definition. For example, the operation `MayReset` = IF cs = **true**

THEN *wait* := 0 END would neither be read-copy nor read-overwrite independent (for state vectors with $cs = \textbf{false}$ it satisfies the definition of the former and for $cs = \textbf{true}$ the latter, but neither for all state vectors). LTSmin can also deal with more liberal independence notions, but we have not yet implemented this in the present paper.

Definition 4 (Write Independence). *Transition group i is* write-independent *from state variable j, if* $\forall s, t \in S^{\mathrm{N}}$: $(s_1, \ldots, s_j, \ldots, s_{\mathrm{N}}) \rightarrow_i (t_1, \ldots, t_j, \ldots, t_{\mathrm{N}}) \Longrightarrow (s_j = t_j)$, *i.e. state variable j is never modified by transition group i.*

We illustrate the above definitions below.

Definition 5 (Dependency Matrices). *For a PTS* $\mathcal{P} = (S^{\mathrm{N}}, G, \rightarrow^{\mathrm{M}}, I^{\mathrm{N}})$, *the* write matrix *is an* $\mathrm{M} \times \mathrm{N}$ *matrix* $WM(\mathcal{P}) = WM^{\mathcal{P}}_{\mathrm{M}\times\mathrm{N}} \in \{0,1\}^{\mathrm{M}\times\mathrm{N}}$, *such that* $(WM_{i,j} = 0) \Longrightarrow$ *transition group i is* write independent *from state variable j. Furthermore, the* read matrix *is an* $\mathrm{M} \times \mathrm{N}$ *matrix* $RM(\mathcal{P}) = RM^{\mathcal{P}}_{\mathrm{M}\times\mathrm{N}} \in \{0,1\}^{\mathrm{M}\times\mathrm{N}}$, *such that* $(RM_{i,j} = 0) \Longrightarrow$ *transition group i is* read independent *from state variable j.*

In this paper we will use sufficient syntactic conditions to ensure Definitions 3 and 4 and obtain the read and write matrix from Definition 5. Indeed, we compute for every operation syntactically which variables are read from and which variables are written to.

- If an operation does not write to a variable, its transition group is write independent according to Definition 4 and the corresponding entry in *WM* is 0.
- If an operation does not read a variable, its transition group is read independent according to Definition 3, unless it maybe written to (e.g., because the assignment is in the branch of an if-then-else). In this case, we will mark the variable as both write and read independent. Also, note that when the assignment within an operation is of the form `f(X) := E` then the operation should have a read dependency on the function `f` (in addition to the write dependency).

For our example in Fig. 2 the syntactic read-write information is as follows:

From the matrices we can infer if a variable is read-copy or read-overwrite independent: a variable that is read independent and not written to (i.e., write independent) is read-copy independent, otherwise it is read-overwrite independent.

We can thus infer that:

- the transition group of `Enter` is read-copy and write independent on `finished`.
- `Exit` is read-copy and write independent on `wait`.
- `Leave` is read-copy and write independent on `wait` and `finished` and read-overwrite independent on `cs`.
- `CS_Active` is read-copy and write independent on `wait` and `finished` and write independent on `cs` (but not read-independent on `cs`).
- `Leave` is read-copy and write independent on `cs`.

	cs	wait	finished
Enter	**1**	**1**	0
Exit	**1**	0	**1**
Leave	0	0	0
CS_Active	**1**	0	0
Restart	0	**1**	**1**

(a) Read matrix (RM)

	cs	wait	finished
Enter	**1**	**1**	0
Exit	**1**	0	**1**
Leave	**1**	0	0
CS_Active	0	0	0
Restart	0	**1**	**1**

(b) Write matrix (WM)

Fig. 4. Dependency matrices

3.1 Exploration Algorithm

We now present the core of the symbolic reachbility analysis algorithm of LTSMIN. Algorithm 1 computes the set of reachable states R (represented as a decision diagram) and it uses the independence information to minimise the number of next state computations that have to be carried out, i.e., re-using the next states $\{t \mid s \rightarrow_i t\}$ computed for a single state s for many other states s' according to Definitions 3 and 4. Algorithm 1 will, while it keeps finding new states, expand the partial transition relation with potential successor states, and apply the expanded relation to the set of new states.

Four key functions that make Algorithm 1 highly performant are the following. [2] The (1) *read projection* $\pi_i^r = \pi_i^{RM}$ and (2) *write projection* $\pi_i^w = \pi_i^{WM}$ take as argument a state vector and produce a state vector restricted to the read and write dependent variables of group i, respectively. Furthermore these function are extended to apply to sets directly, e.g., given the examples in Figs. 2 and 4, a read projection for `Leave` is $\pi_3^r(\{(\underline{FALSE}, 0, 0), (\underline{FALSE}, 0, 1), (\underline{FALSE}, 1, 0)\}) = \{(\underline{FALSE})\}$. This is illustrated in Fig. 6 and used at Line 2 in Algorithm 2. The read projection prevents LTSMIN from doing two unnecessary next state calls to PROB, since `Leave` is *read-copy* independent on `wait` and `finished`.

The function (3) NEXTSTATE$_i$ takes a read projected state and projects (with π_i^w) all successor states of transition group i. The partial transition relation $\hookrightarrow_i^{\mathrm{p}}$ is learned on the fly, and NEXTSTATE$_i$ is used to expand $\hookrightarrow_i^{\mathrm{p}}$. An example next state call for `Enter` is NEXTSTATE$_1((FALSE, 1)) = \{(TRUE, 0)\}$.

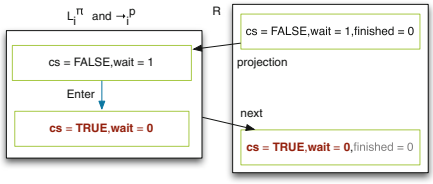

Fig. 5. One iteration with `Enter`

Lastly, (4) NEXT takes a set of states, a partial transition relation, a row of the read and write matrix and outputs a set of successor states. For example, applying the partial relation of `Enter` to the initial state yields NEXT$(\{(\underline{FALSE}, \underline{1}, 0)\}, \{((FALSE, 1), (TRUE, 0))\}, (1, 1, 0), (1, 1, 0)) =$

[2] We refrain from giving their formal definitions; they can be found in [23].

$\{(\underline{TRUE}, \underline{0}, 0)\}$. Note that in this example `Enter` is *read-copy* independent on `finished` and thus NEXT will copy its value from the initial state.

The usage of these four key functions is also illustrated in Fig. 5. The figure shows that first the projection is done for `Enter`, then $\hookrightarrow^{\mathrm{p}}_i$ is expanded with a $\textsc{NextState}_i$ call, lastly relation $\hookrightarrow^{\mathrm{p}}_i$ is applied to the initial state, producing the first successor state.

Figure 6 shows for each operation the transition relation $\hookrightarrow^{\mathrm{p}}_i$ and the projected states on which they are computed. Definition 3 ensures that the projected state space shown in Fig. 6 can be used to compute the effect of each of these operations for the *entire* state space (using `next`).

Algorithm 1. REACHBREADTH-FIRST

Input : $I^{\mathrm{N}} \subseteq S^{\mathrm{N}}, \mathrm{M} \in \mathbb{N}, RM, WM$
Output: The set of reachable states $\mathcal{R}$

$\mathcal{R} \leftarrow I^{\mathrm{N}}$; $\mathcal{L} \leftarrow \mathcal{R}$;
for $1 \leq i \leq \mathrm{M}$ **do** $\mathcal{R}^{\mathrm{p}}_i \leftarrow \varnothing$; $\hookrightarrow^{\mathrm{p}}_i \leftarrow \varnothing$;
while $\mathcal{L} \neq \varnothing$ **do**
  LEARNTRANS(); $\mathcal{N} \leftarrow \varnothing$;
  for $1 \leq i \leq \mathrm{M}$ **do**
    $\mathcal{N} \leftarrow \mathcal{N} \cup \textsc{next}(\mathcal{L}, \hookrightarrow^{\mathrm{p}}_i, RM_i, WM_i)$;
  $\mathcal{L} \leftarrow \mathcal{N} - \mathcal{R}$; $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{N}$;
return $\mathcal{R}$

Algorithm 2. LEARNTRANS

Description: Extends $\hookrightarrow^{\mathrm{p}}_i$

for $1 \leq i \leq \mathrm{M}$ **do**
  $\mathcal{L}^{\mathrm{p}} \leftarrow \pi^r_i(\mathcal{L})$;
  for $s^{\mathrm{p}} \in \mathcal{L}^{\mathrm{p}} - \mathcal{R}^{\mathrm{p}}_i$ **do**
    $\hookrightarrow^{\mathrm{p}}_i \leftarrow \hookrightarrow^{\mathrm{p}}_i \cup \{(s^{\mathrm{p}}, d^{\mathrm{p}}) \mid d^{\mathrm{p}} \in \textsc{NextState}_i(s^{\mathrm{p}})\}$;
  $\mathcal{R}^{\mathrm{p}}_i \leftarrow \mathcal{R}^{\mathrm{p}}_i \cup \mathcal{L}^{\mathrm{p}}$;

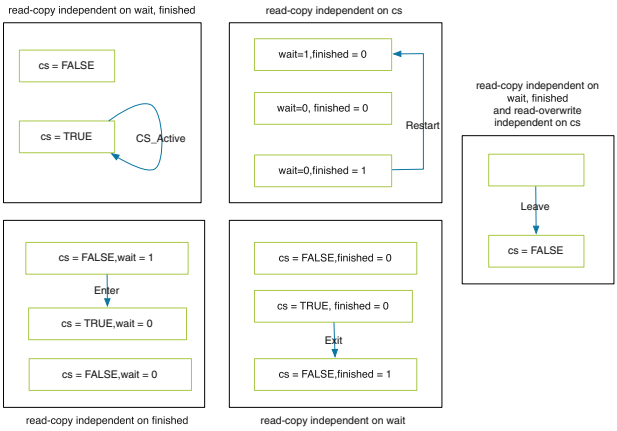

Fig. 6. MutexSimple, operations computed on their projected state space

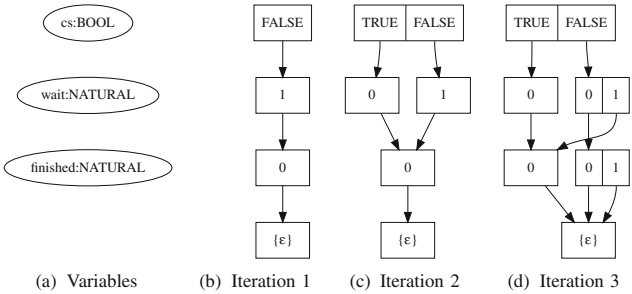

Fig. 7. LDDs of the reachable states

3.2 List Decision Diagrams

The symbolic reachability algorithm in Sect. 3.1 uses List Decision Diagrams (LDDs) to store the reachable states and transition relations. Similar to a Binary Decision Diagram, an LDD [7] represents a set of vectors. Due to the sharing of state vectors within an LDD, the memory usage can be very low, even for very large state spaces. Three example LDDs for the running example are given in Fig. 7. The LDDs represent the set of reachable states $\mathcal{R}$ in Algorithm 1 at each iteration of Line 3. In an LDD every path from the top left node to $\{\epsilon\}$ is a state, e.g., the initial state $(FALSE, 1, 0)$ in Fig. 7b. A node in an LDD represents a unique set of (sub) vectors, e.g., $\{\epsilon\}$ represents the set of zero-length vectors and the right-most 0 of variable `wait` in Fig. 7d encodes the set $\{(0,0), (0,1), (1,0)\}$. Figure 7c shows that firing `Enter` will add $(TRUE, 0, 0)$ to R. In Fig. 7d $(FALSE, 0, 0)$ and $(FALSE, 0, 1)$ are added to R, by firing `Leave` and `Exit` respectively. The benefit of using LDDs for state storage is due to the sharing of state vectors. For example, the subvector $(FALSE)$ of the states $\{(FALSE, 0, 0), (FALSE, 0, 1), (FALSE, 1, 0), (FALSE, 1, 1)\}$ in iteration 3 is encoded in the LDD with a single node. For bigger state spaces the sharing can be huge; resulting in a low memory footprint for the reachability algorithm.

3.3 Performance: NextState Function

There are two big differences of Algorithm 1 with classical explicit state model checking as used by ProB [19]. First, the state space is represented using an LDD datastructure, which enables sharing amongst states. Second, independence is used to apply the NextState function not state by state, but for entire sets of states in one go. For each of the 4 states in Fig. 3, the explicit model checking algorithm of ProB would check whether each of the 5 operations is enabled; resulting in 20 next-state calls. With LTSmin's symbolic reachability Algorithm 1, only 12 NextState calls are made. This is shown in the following table, where + means enabled, − means disabled, and C means that LTSmin has reused the results of a previous call to ProB.

If we initialise `wait` with MAXINT = 500, the state space has 251,002 states. The runtime with ProB is 70 s, with LTSmin+ProB 48 s and LTSmin performs

State#	cs	wait	finished	Enter	Exit	Leave	CS_Active	Restart
1	FALSE	1	0	+	C	C	C	–
2	TRUE	0	0	–	+	+	+	–
3	FALSE	0	0	–	–	C	–	C
4	FALSE	0	1	C	–	C	C	+

only 6012 NextState calls. The example does not have a lot of concurrency and uses only simple data structures (and thus the overhead of the LTSmin's ProB front-end is more of a factor compared to the runtime of ProB for computing successor states); other examples will show greater speedups (see Sect. 5). But the purpose of this example is to illustrate the principles.

4 Technical Aspects and Implementation

We used a distributed approach to integrate ProB and LTSmin. Both tools are stand-alone applications, so a direct integration, i.e., turning one of the tools into a shared library would require considerable effort. We therefore added extensions to both tools that convert the data formats and use sockets to communicate with each other. A high level view of the integration is shown in Fig. 8. We use the ØMQ [14] library for communication. ØMQ is oriented around message queues and can be used as both, a networking library with very high throughput and as a concurrency framework. We have chosen ØMQ because it worked very well in previous work [4]. Although we do not (yet) have to care about concurrency in this work, the reactor abstraction provided by ØMQ was very handy in the ProB extension. It allows to implement a server that receives and processes messages without much effort. The communication is always initiated by LTSmin; it sends a message and blocks until it receives the answer from ProB. We usually run both tools on a single computer using interprocess (IPC) sockets, but it is only a matter of configuration to run the tools on different machines using TCP sockets. We currently only support Linux and Mac OS. The communication protocol is straightforward. Reachability analysis is initiated from LTSmin by sending an initialisation packet. ProB answers with a message containing the relevant static information about a model, such as the dependency matrices that LTSmin requires (see Sect. 3). Each matrix is encoded as a 2-dimensional array, which is not optimal for a sparse matrix but is not an issue because we

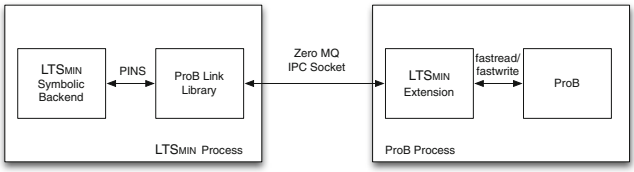

Fig. 8. High level design showing the integration

only send the matrices once. The packet also contains the list of variables, their types, the list of transition groups, and the initial state.

States are represented as a list of so called **chunks**. A chunk is one of the elements in the state vector according to Definition 2. In the case of B, each chunk is a value of one of the state variables. Because LTSmin will not look inside the chunks, we simply use the binary representation of ProB's Prolog term that represents the value of a variable. This has the advantage, that ProB does not have to keep information about the state space. It can always recover a state from the chunks that are sent by LTSmin. The transition groups correspond to B operations as explained in Sect. 2. Like chunks the transition groups are only used as names in LTSmin.

Once the initial setup is done, LTSmin will start to ask ProB for successor states for **specific** transition groups. It will send a next-state message containing a state and a transition group. The state, that LTSmin sends is a list of chunks and ProB's LTSmin extension can directly consume them and construct a Prolog term that internally represents a state. Using this constructed state and the transition group, the extension will then ask ProB for all successor states. The result is a list of Prolog terms, each representing a successor state. The extension transforms the list of states into a list of lists of chunks and sends them back to LTSmin. This is repeated until LTSmin has explored all necessary states and sends a termination signal.

The next-state messaging is similar to Fig. 5, the projection is achieved by replacing all read independent variables by a dummy value.

5 Experiments

To demonstrate that the combination of ProB and LTSmin improves the performance of the reachability analysis and deadlock detection compared with the standalone version of ProB, we use a wide range of B and Event-B models. Our benchmark suite contains puzzles (e.g., towers of Hanoi) as well as specifications of protocols (e.g., Needham-Schroeder), algorithms (e.g., Simpson's four slot algorithm) and industrial specifications (e.g., a choreography model by SAP, a cruise control system by Volvo and a fault tolerant automatic train protection system by Siemens).[3]

The experiments were run on Ubuntu 15.10 64-bit, with 8 GB RAM, 120 GB SSD and an Intel Sandybridge Mobile i5 2520M 2.50 GHz Dual core. The version of ProB used in this paper is 1.5.1-beta3, and LTSmin tag LTSminProB-iFM2016.[4]

Figure 9 summarises a selection of the experiments that we ran. The last two models are Event-B models. In these experiments we used Breadth-First Search (BFS) and looked for deadlocks. A deadlock was found only for the Philosophers model (this is also why there are no next state call statistics for this model). The table also contains the number of next state calls for ProB reachability

[3] More detailed descriptions can be found in [5].

[4] Reproduction notes at https://github.com/utwente-fmt/ProB-LTSmin-iFM16.

Benchmark	Events	States	ProB Nxt St Calls	LTSmin NxtSt Calls	ProB Wall (ms)	LTSmin CPU (ms)	LTSmin Wall (ms)	Speedup
CAN_BUS	21	132600	2784560	3534	122850	660	1590	77.264
ConcurrentCounters	4	110813	443249	113032	21820	2760	13820	1.579
Cruise_finite1	26	1361	35361	1667	2900	100	1020	2.843
file_system	8	698	5577	1198	1900	180	4660	0.41
MutexSimple	5	10	46	26	10	10	190	0.053
Philosophers	5				480	40	590	0.814
SiemensMiniPilot_Abrial0	9	181	1621	182	100	20	260	0.385
Simpson_Four_Slot	9	46658	419906	2089	17310	200	860	20.128
Train1_Lukas_POR	8	24637	197082	101441	33660	6480	50260	0.670
nota	11	80719	887899	588	287970	130	660	436.318
pkeyprot2	10	4412	44111	2004	22190	210	1710	12.977
Ref5_Switch_mch	38	29861	1134681	1281	160600	490	1260	127.460
obsw_M001	21	589279	12374779	23406	2051320	1620	12420	165.163

Fig. 9. B and Event-B Machines, with BFS and deadlock detection

analysis on its own and when called from LTSmin's symbolic reachability analysis algorithm (i.e., our new integration see Sect. 3.3) without deadlock checking. One can clearly see that we obtain a considerable reduction in wallclock time. The ProB time is the walltime of the ProB reachability analysis and initial state computation and does not include parsing and loading. The LTSmin CPU time column shows how much time is spent in the LTSmin side of the symbolic reachability analysis algorithm. The LTSmin wall time shows the total walltime, and this also contains the time spent in the communication layer and waiting for the ProB process to compute the next states. To compare the benefit of our new algorithm we compute the speedup of the walltime in the last column by dividing the ProB walltime from column 5 with the LTSmin walltime in column 7.

We can see that for some of the smaller models the overhead of setting up LTSmin does not pay off. However, for all larger models, except for the Train1_Lukas_POR model considerable speedups were obtained.

A major result we achieved with non default settings for LTSmin, is for elevator12.eventb. This model is not listed in Fig. 9, because ProB runs out of memory on the hardware configuration used for this experiment. LTSmin computed in 34 s, with 96,523 NextState calls, that the model has 1,852,655,841 states. As reachability algorithm we chose chaining [27], and to compute a better variable order, we ran Sloan's bandwidth reduction algorithm [29] on the dependency matrix.

As far as memory consumption is concerned; when performing reachability analysis on CAN_BUS, the ProB process consumes 370 MB real memory, while the LTSmin process consumes 633 MB, with the default settings. With the default settings LTSmin will allocate 2^{22} elements ($\approx$ 100 MB) for the node table and 2^{24} elements ($\approx$ 500 MB) for the operations cache. If we choose a smaller node table and operations cache for the LDD package (both 2^{18} elements),

LTSmin consumes only 22 MB. The default settings for LTSmin are geared towards larger symbolic state spaces than that of CAN_BUS. The default node table and cache are too big for CAN_BUS, and thus not completely filled during reachability.

We have also run our new symbolic reachability analysis on Z and TLA$^+$ models. For example, we successfully validated the video rental Z model from [10]. For 2 persons and 2 titles and maximum stock level of 4, LTSmin generates the 23009 states in 1.75 s compared to 52.4 s with ProB alone. The model contained useless constants; after removing them ProB runs in 1.6 s; the runtime of LTSmin stays unchanged. We were unable to use the output of z2sal [10] using SAL [25] and its symbolic model checker for comparison.

In summary, Fig. 9 shows that for several non-trivial B and Event-B models, considerable improvements can be obtained using the symbolic reachability analysis technique described in this paper.

Alternate Approaches. Other techniques for improving model checking for B-Method and Event-B models have been developed and evaluated in the recent years. We have run a further set of experiments using a selection of those methods; the complete results can be found in [5] For technical reasons, the experiments were run on different hardware than above, a MacBook Air with 2.2 GHz i7 processor and 8 GB of RAM. We summarise the findings here and compare the results with our new symbolic model checking algorithm.

Benchmark	ProB POR		ProB Hash		Tlc		ProB no opt
	ms	Speedup	ms	Speedup	sec	Speedup	ms
CAN_BUS	138720	0.80	98390	1.12	3	37	110400
ConcurrentCounters	50	345.8	18400	1.06	1	17	17290
file_system	2380	0.37	210	4.24	29	0.03	890
Simpson_Four_Slot	20860	0.70	9550	1.52	1	15	14530
Train1_Lukas_POR	34030	0.75	28930	0.88	4	6	25740
nota	490	509.22	14780	16.88	10	25	249520
Ref5_Switch_mch	215160	0.59	124500	1.01	6	21	126170
obsw_M001	2150520	0.80	76190	22.53	55	31	1716770

The authors in [12] presented a translation from the B-Method to TLA$^+$, with the goal of using the **Tlc** model checker [32] as backend. Tlc has no constraints solving capabilities, and as such that it can only deal with lower level models. On the other hand, its execution can be considerably faster than ProB, and its explicit state model checking engine (which stores fingerprints) is very efficient. On the downside, there is a small probability that fingerprint collisions can occur. The experiments show that Tlc does not deal well with benchmark programs which require constraint solving (graph isomorphism, JobsPuzzle, ...), running up to three orders of magnitude slower than ProB or LTSmin with ProB.

However, it does deal very well with lower level models, e.g., it is faster than LTSMin for ConcurrentCounters. For many benchmark models, even those not requiring constraint solving, our symbolic reachability analysis is faster.

For example, for the nota example, Tlc runs in about 10 s—faster than ProB without any optimisation—but slower than LTSMin by less than a second.

Symmetry reduction [20] can be very useful; but exponential improvements usually occur only on academic examples. Here we have experimented with the hash marker symmetry reduction, which is ProB's fastest symmetry method, but is generally not guaranteed to explore all states. The method gives the best results for certain models (e.g., file_system). But for several of the larger, industrial examples shown above, its benefit is not of the same scale as LTSMin. In future, we will investigate combining ProB's symmetry reduction with the new LTSMin algorithm.

We have also experimented with **partial order reduction**. [11] uses a semantic preprocessing phase to determine independence (different from our purely syntactic determination; see Sect. 3). As such, it can induce a slow down for some examples where this does not pay off (e.g., file_system). ProB's partial order reduction obtains the best times for certain models with a large degree of concurrency (ConcurrentCounters, SiemensMiniPilot_Abrial, and nota). However, once we start doing invariant checking, [11] does not scale nearly as well (e.g., it takes 134 s instead of 0.5 s for LTSMin checking the nota model). But even without invariant checking, there are plenty examples where the symbolic reachability analysis approach is better (e.g., Cruise_finite1, Philosophers, Simpson_Four_Slot and almost two orders of magnitude for CAN_BUS). In summary:

- Tlc is good for models not requiring constraint solving. It is a very efficient, explicit state model checker. However, models often have to be rewritten (such as CAN_BUS), and there is a small chance of having fingerprint collisions.
- Symmetry reduction excels when models make use of deferred sets. However, the hash marker method is not guaranteed to explore all states.
- Partial order reduction is very good for models with a large degree of concurrency. However, it can cause slow downs and is less suited for invariant checking.
- The new symbolic reachability analysis algorithm deals well with concurrency and is by far the fastest method for certain larger, industrial models, such as CAN_BUS, obsw_M001, elevator12, the ABZ landing gear model or Abrial's mechanical press. LTSMin is currently the only tool set that uses a symbolic representation of the state space that is connected to ProB.

6 More Related Work, Future Work and Conclusion

We have already evaluated the use of Tlc [32] for model checking B. Another explicit state model checker for B has been presented in [21], which uses lazy enumeration. Symbolic model checking [8] has been used for railway applications in [31]. The best known symbolic model checker is probably SMV [22], which uses a low-level input language. Some comparisons between using SMV and ProB

have been conducted in [15], where models were translated by hand. For abstract state machines there is the AsmetaSMV [3] tool, which automatically translates ASM specifications to SMV. It is our impression that the translation often leads to a considerable blowup of the model, encoded in SMV's low-level language, also affecting performance. We did one experiment on a Tic-Tac-Toe model provided for AsmetaSMV: NuSMV 2.6 took over 13 s to find a configuration where the cross player wins; ProB (without LTSmin) took 0.2 s model checking time for the same property on a similar B model. Another experiment involved puzzle 3 of the RushHour game: ProB solves this in 5 s, while NuSMV still had not found a solution after 120 min.

Other symbolic model checkers that perform comparable well to LTSmin include Marcie [13] and PetriDotNet[24].

The paper provides a stable architectural link between ProB and LTSmin that can be extended. First, we plan to provide LTSmin with more fine-grained information about the models, both statically and dynamically. Dynamically, ProB will transmit to LTSmin which variables have actually been written by an operation, enabling a more extensive independence notion to be used. Statically, ProB will transmit the individual guards of operations and provide variable read matrices for the guards. We will also transmit the individual invariants in the same manner, to enable analysis of the invariants. (It is actually already possible to check invariants using the present integration, simply by encoding invariants as operations. We have done so with success for some of the examples, e.g., the nota from Sect. 5.) When ProB transmits individual guards, we also hope to use the guard-based partial order optimisations of LTSmin [18] and enable LTL model checking with LTSmin.

These future directions will strengthen the capability of the verification tools and hence further encourage the application of formal methods within industry as identified in [6], for example to support complex railway systems verification in CSP‖B. This will require both more fine-grained static and dynamic information.

In summary, we have presented a new scalable, symbolic analysis algorithm for the B-Method and Event-B, along with a platform to integrate other model checking improvements via LTSmin in the future.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.: The B-Book - Assigning Programs to Meanings. Cambridge University Press, Cambridge (2005)
3. Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 61–74. Springer, Heidelberg (2010)
4. Bendisposto, J.: Directed and Distributed Model Checking of B Specifications. Ph.D. thesis, University of Düsseldorf (2015). http://docserv.uni-duesseldorf.de/servlets/DocumentServlet?id=34472

5. Bendisposto, J., Körner, P., Leuschel, M., Meijer, J., van de Pol, J., Treharne, H., Whitefield, J.: Symbolic Reachability Analysis of B through ProB and LTSmin. CoRR abs/1603.04401 (2016)
6. Bicarregui, J.C., Fitzgerald, J.S., Larsen, P.G., Woodcock, J.C.P.: Industrial practice in formal methods: a review. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 810–813. Springer, Heidelberg (2009)
7. Blom, S., van de Pol, J.: Symbolic reachability for process algebras with recursive data types. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer, Heidelberg (2008)
8. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. IC **98**(2), 142–170 (1992)
9. Ciardo, G., Marmorstein, R.M., Siminiceanu, R.: The saturation algorithm for symbolic state-space exploration. STTT **8**(1), 4–25 (2006)
10. Derrick, J., North, S., Simons, A.J.H.: Z2SAL - building a model checker for Z. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 280–293. Springer, Heidelberg (2008)
11. Dobrikov, I., Leuschel, M.: Optimising the ProB model checker for B using partial order reduction. In: Giannakopoulou, D., Salaün, G. (eds.) SEFM 2014. LNCS, vol. 8702, pp. 220–234. Springer, Heidelberg (2014)
12. Hansen, D., Leuschel, M.: Translating B to TLA$^+$ for validation with TLC. In: Ait Ameur, Y., Schewe, K.-D. (eds.) ABZ 2014. LNCS, vol. 8477, pp. 40–55. Springer, Heidelberg (2014)
13. Heiner, M., Rohr, C., Schwarick, M.: MARCIE – model checking and reachability analysis done efficiently. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. LNCS, vol. 7927, pp. 389–399. Springer, Heidelberg (2013)
14. Hintjens, P.: ZeroMQ: Messaging for Many Applications. O'Reilly Media Inc, Sebastopol (2013)
15. Hörne, T., van der Poll, J.A.: Planning as model checking: the performance of ProB vs NuSMV. In: SAICSIT Conference ACM ICPS, vol. 338, pp. 114–123. ACM (2008)
16. James, P., Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H., Trumble, M., Williams, D.: Verification of scheme plans using CSP||B. In: Counsell, S., Núñez, M. (eds.) SEFM 2013. LNCS, vol. 8368, pp. 189–204. Springer, Heidelberg (2014)
17. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015)
18. Laarman, A., Pater, E., Pol, J., Hansen, H.: Guard-based partial-order reduction. Int. J. Softw. Tools Technol. Transfer, 1–22 (2014). doi:10.1007/s10009-014-0363-9
19. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008)
20. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. Ann. Math. Artif. Intell. **59**(1), 81–106 (2010)
21. Matos, P.J., Fischer, B., Marques-Silva, J.: A lazy unbounded model checker for Event-B. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 485–503. Springer, Heidelberg (2009)
22. McMillan, K.L.: Symbolic Model Checking. Ph.D. thesis, Boston (1993)
23. Meijer, J., Kant, G., Blom, S., van de Pol, J.: Read, write and copy dependencies for symbolic model checking. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 204–219. Springer, Heidelberg (2014)

24. Molnár, V., Darvas, D., Vörös, A., Bartha, T.: Saturation-based incremental LTL model checking with inductive proofs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 643–657. Springer, Heidelberg (2015)
25. de Moura, L., Owre, S., Shankar, N.: The SAL language manual. Technical report, SRI International, technical Report SRI-CSL-01-02 (Rev. 2) (2003)
26. Plagge, D., Leuschel, M.: Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. STTT **11**, 9–21 (2010)
27. Roig, O., Cortadella, J., Pastor, E.: Verification of asynchronous circuits by BDD-based model checking of petri nets. In: Proceedings ATPN, pp. 374–391 (1995)
28. Schneider, S., Treharne, H.: CSP theorems for communicating B machines. Formal Asp. Comput. **17**(4), 390–422 (2005)
29. Sloan, S.W.: A FORTRAN program for profile and wavefront reduction. Int. J. Numer. Meth. Eng. **28**(11), 2651–2679 (1989)
30. Whitefield, J.: Linking ProB and LTSmin (2015), Final Year Dissertation, University of Surrey
31. Winter, K.: Optimising ordering strategies for symbolic model checking of railway interlockings. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012, Part II. LNCS, vol. 7610, pp. 246–260. Springer, Heidelberg (2012)
32. Yu, Y., Manolios, P., Lamport, L.: Model checking TLA$^+$ specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999)