State-of-the-Art Model Checking for B and Event-B Using ProB and LTSmin

Philipp Körner[1] ✉[0000−0001−7256−9560], Michael Leuschel[1], Jeroen Meijer[2]*

[1] Institut für Informatik, Universität Düsseldorf, Germany
{p.koerner, leuschel@cs.} uni-duesseldorf.de
[2] Formal Methods and Tools, University of Twente, The Netherlands
j.i.g.meijer@utwente.nl

Abstract. In previous work, we presented symbolic reachability analysis by linking ProB, an animator and model checker for B and Event-B, and LTSmin, a language-independent model checker offering state-of-the-art model checking algorithms. Although the results seemed very promising, it was a very basic integration of these tools and much potential of LTSmin was not covered by the implementation.
In this paper, we present a much more mature version of this tool integration. In particular, explicit-state model checking, efficient verification of state invariants, model checking of LTL properties, as well as partial order reduction and proper multi-core model checking are now available. The (improved) performance of this advanced tool link is benchmarked on a series of models with various sizes and compared to ProB.

1 Introduction

Formal methods, e.g., the B-Method [3], are vital to ensure correctness in software where failure means loss of money or even risking human lives. Yet, for industrial application, tooling often remains unsatisfactory [6,35]. One such tool is ProB, an animator and model checker for B and Event-B. While ProB is fairly mature after hundreds of man-years of engineering effort, it may still struggle with industrial-sized models containing several millions of states. LTSmin, however, is a language-independent model checker that offers symbolic algorithms and many optimizations in order to deal with the state space explosion problem.

In [4], we linked LTSmin with ProB in order to obtain a symbolic reachability analysis for B and Event-B. ProB was computing the B operational semantics and providing static information about possible state transitions while LTSmin was performing the symbolic reachability algorithm.

LTSmin offers further model checking algorithms and optimizations, both for its symbolic and for its sequential backend. In this paper, we describe how we extended the link between LTSmin and ProB using ZeroMQ [19] in order to obtain a model checking tool for B and Event-B and evaluate the performance

* Supported by STW SUMBAT grant: 13859

on a set of real life, industrial-sized models. LTSmin's language frontend that interacts with ProB can directly be used with any model checker that speaks the same protocol via ZeroMQ. The extension is based on [21] and includes:

- invariant checking (for both the symbolic and sequential backend),
- guard splitting for symbolic analysis of B models,[1]
- partial order reduction (with the sequential backend),
- (parallel) LTL model checking (with the sequential/multi-core backend),
- effective caching of transitions and state labels,
- short states, to transmit only relevant variables for each transition.

1.1 LTSmin

LTSmin [9] is an open-source, language-independent, state-of-the-art model checker that offers many model checking algorithms including partial order reduction, LTL verification and distributed model checking. An overview of its architecture can be found in Fig. 1. By implementing the Pins, i.e., the partitioned interface to the next-state function, new language frontends can employ these algorithms. At its core, Pins consists of three functions: one that provides an initial state vector, a second, partitioned transition function that calculates successor states and, lastly, a state labelling function.

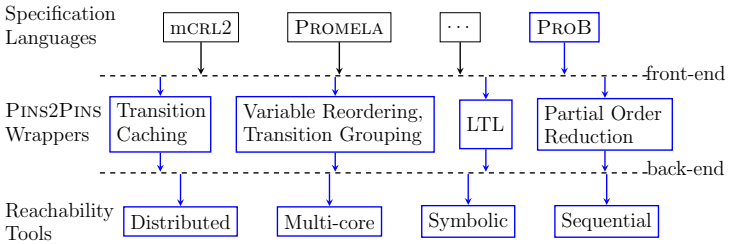


Fig. 1: Modular Pins architecture of LTSmin [20]

LTSmin provides four backends:

- a sequential backend that implements an explicit state model checking algorithm similar to the one implemented in ProB,
- a symbolic backend that stores states as LDDs (List Decision Diagrams) [7],
- a multi-core backend that works similar to the sequential backend, but is capable of using multiple CPU cores on the same machine,
- a distributed backend in order to utilize multiple machines for model checking.

For this article, we will focus on the advances of the integration with ProB using the sequential and symbolic backends but also experiment with the multi-core backend. We have not done any experiment with the distributed backend yet.

[1] Due to technical limitations in ProB, we have not added this for Event-B yet.

1.2 ProB and the B-Method

ProB [26] is an open-source animator and model checker for several formalisms including B, Event-B, CSP, Z and TLA$^+$. It can be used in order to find invariant or assertion violations or deadlock states in machine specifications. While it implements a straightforward explicit state model checker, it also ships more advanced techniques, e.g., symmetry reduction [30], partial order reduction [16,17] or symbolic model checking [23]. This style of symbolic model checking [10], where states are stored as predicates, must not be confused with the symbolic model checking that LTSmin provides, where states are stored as decision diagrams. ProB's core is written in SICStus Prolog [11] and may also employ SMT solvers [24], such as Z3 and CVC4, or SAT solvers, such as Kodkod [29].

When integrating ProB into LTSmin, we focus on two formalisms: B (sometimes referred to as "classical B") is part of the B-Method [3], where software is developed starting with a very abstract model that iteratively is refined to a concrete implementation. This method aims for software to be "correct by construction". Event-B [1] is considered to be the successor of B that does not include constructs that often hinder formal proof in the language, e.g., conditional assignments or loops. Both formalisms offer a very high level of abstraction and are based on set theory and first-order logic.

1.3 Theoretical Background

We repeat the most important definitions used in [4] on the following contrived example:

```
MACHINE example
CONSTANTS c
PROPERTIES c = 100
VARIABLES x,y
INVARIANT x : INTEGER & y : INTEGER &
          x <= c & x + y <= 2 * c
INITIALISATION x := 0 || y := 0
OPERATIONS
  incx        = SELECT x < c THEN x := x + 1 END;
  doublex     = SELECT x < c/2 & x > 0 THEN x := x * 2 END;
  incy(n)     = SELECT n > 0 & n < c & y < c
                THEN y := y + n END;
  incxmaybey = SELECT x < c
                THEN x := x + 1 ||
                     IF x mod 2 = 0 THEN y := y + 1 END
END
```

Fig. 2: Contrived B Machine example

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

Such transition systems are induced by both B and Event-B models. As all variables have to be typed, the set of states $\mathcal{S}$ is the Cartesian product of all types. All possible initial states are given in the `INITIALISATION` machine clause. The union of all operations define the transition relation $\rightarrow$. For symbolic model checking however, it is very important that the transition relation is split into groups.

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

- $S^{\mathrm{N}} = S_1 \times \ldots \times S_{\mathrm{N}}$ *is the set of states, which are vectors of* N *values,*
- $\mathcal{T} = (\rightarrow_1, \ldots, \rightarrow_{\mathrm{M}})$ *is a vector of* M *relations, called 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* $\mathcal{T}$*, 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 $\boldsymbol{s} \rightarrow_i \boldsymbol{t}$ *when* $(\boldsymbol{s}, \boldsymbol{t}) \in \rightarrow_i$ *for* $1 \leq i \leq \mathrm{M}$*, and* $\boldsymbol{s} \rightarrow^{\mathrm{M}} \boldsymbol{t}$ *when* $(\boldsymbol{s}, \boldsymbol{t}) \in \rightarrow^{\mathrm{M}}$.

Strictly speaking, the transition relation in Definition 2 is not partitioned, as individual B operations can have same effect. We implemented an easy mental model where each transition group represents exactly one operation in the B model.

For the example in Fig. 2, the only initial state is $init^{(c,x,y)} = (100, 0, 0)$. We agree on a notation where the ordering of the variables in an individual state is given once as a superscript. In LTSmin, the ordering of the variables is fixed and unambiguous. Then, $I^{\mathrm{N}} = \{(100, 0, 0)\}$, $S^{\mathrm{N}} = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ and $\mathcal{T} = (\texttt{incx}, \texttt{doublex}, \texttt{incy}, \texttt{incxmaybey})$ with, e.g., $\texttt{incx} = \{(100, 0, 0) \rightarrow_1 (100, 1, 0)\,, (100, 0, 1) \rightarrow_1 (100, 1, 1)\,, (100, 1, 0) \rightarrow_1 (100, 2, 0)\,, \ldots\}$.

Symbolic Model Checking and Event Locality In many B models, operations only read from and write to a small subset of variables, which is known as event locality [12]. Symbolic model checking benefits from event locality, allowing reuse of successor states when only variables changed that are irrelevant to the state transition.

In order to employ LTSmin's symbolic algorithms [8,27,28], ProB provides several dependency matrices about the B model that shall be checked: A *read* matrix and *may-write* matrix is used in order to determine independence between transition groups for symbolic model checking. These two matrices for Fig. 2 are given in Fig. 3. Entries are set to 1, if the operation reads or writes the variable, and otherwise to 0. Further matrices are shown once their use-case is introduced.

	c	x	y
incx	**1**	**1**	0
doublex	**1**	**1**	0
incy	**1**	0	**1**
incxmaybey	**1**	**1**	**1**

(a) Read Matrix

	c	x	y
incx	0	**1**	0
doublex	0	**1**	0
incy	0	0	**1**
incxmaybey	0	**1**	**1**

(b) May-Write Matrix

Fig. 3: Dependency Matrices

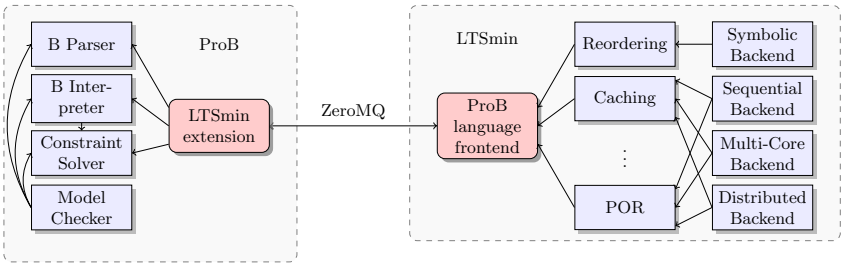


Fig. 4: Overview of the Tool Integration LTSmin↔ProB

2 Architecture Overview of LTSmin↔ProB Integration

LTSmin and ProB typically run as two separate processes which can be launched both manually or start each other. They are linked as shown in Fig. 4. The processes are linked via one IPC (local inter-process communication) socket per tool provided by ZeroMQ [19], a library offering distributed messaging. We employ a request-reply pattern on these sockets, where LTSmin sends requests and ProB responds. In order to handle messaging in ProB, we added a small layer in C.

In order to expose information about the loaded model to LTSmin, ProB's response to an initial request includes, amongst others, names of state variables and transition groups, an initial state and dependency matrices.

LTSmin expects a model to have a single initial state, while B and Event-B models allow for nondeterministic initialization. Thus, the initial state transferred to LTSmin consists of dummy values for all variables. Furthermore, we add a state variable named `is_init` that is initially set to false. Via a special transition `$init_state` which is only applicable if `is_init` is false, the actual initial states of the specification are exposed. For all these states (and their successors), `is_init` is set to true. This technicality leads to special cases in the entire implementation which we will omit.

In order to call ProB's next-state function, LTSmin sends a request containing the transition group and a state. ProB then will answer with a list of successor states. Since ProB is implemented in Prolog, it is hard to exchange states reasonably. Prolog terms have a limited life-span when using SICStus' foreign function interface. Thus, we serialize and deserialize state variables into/from blobs (binary large objects) by making use of an undocumented Prolog library named `fastrw`. Each variable is stored in a separate blob, such that, a

state is only a vector of blobs for LTSmin. Naturally, repeated (de)serialization comes with an overhead that we chose to accept for now.

The labelling function is called in the same way, providing a label name and a state. ProB will answer with either true or false.

3 Implementation

In order to extend the prior integration, ProB needs to expose more of the B model to LTSmin. In this section, we describe what information is additionally exchanged, how it is calculated and used by considering the running example in Fig. 2.

3.1 State Labels and Invariant Checking

In a labelled transition system, a set of atomic propositions is assumed. An atomic proposition is any predicate that we will call "state label". ProB will implement the labelling function, i.e., it will receive a state and a state label and return true or false.

The entire invariant can be seen as a single atomic proposition. However, if only some variables change from one state to its successor, not all conjuncts need to be re-evaluated. Thus, we split the invariant into its conjuncts. Each conjunct is announced as a state label to LTSmin by providing a unique identifier. In order to expose which state label depends on which variable, additionally a state label matrix is included.

In our example in Fig. 2, initially, four state labels are created. The corresponding state label matrix is shown in Fig. 5.

$$\begin{array}{l} \\ x \in \mathbb{Z} \\ y \in \mathbb{Z} \\ x \leq c \\ x + y \leq 2c \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{c} & \texttt{x} & \texttt{y} \end{array} \\ \begin{bmatrix} 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} \\ \mathbf{1} & 0 & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & \mathbf{1} \end{bmatrix} \end{array}$$

Fig. 5: State Label Dependency Matrix

Since predicates are split at conjunctions, well-definedness issues might arise. As an example, consider the following predicate: $x \neq 0 \wedge 100 \; mod \; x = 1$. ProB's constraint solver will reorder the conjuncts in order to exclude any division by zero. Once the predicate is split into $x \neq 0$ and $100 \; mod \; x = 1$, it will only receive a single conjunct and cannot do any reordering. Thus, in a state with $x = 0$, the second conjunct on its own will result in a well-definedness error.

Thus, if a well-definedness error arose, another part of the original predicate has to be unsatisfied and we can assume the offending conjunct to be false as well.

Proof Information Event-B models exported from Rodin [2] include information about discharged proof information. ProB uses this information, e.g., in order to avoid checking an invariant that has been proven to be preserved when a specific action is executed [5]. If an invariant has been fully proven to be correct, i.e., that it holds in the initial states and all transitions preserve it, we can exclude it from the list of invariants exposed to LTSmin.

Even though the machine in Fig. 2 is written in classical B, i.e., there is no proof information available, the two invariant conjuncts $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$ are dropped. The type checker can already prove that they will hold. Thus, there is no need to check them separately.

3.2 Short States

In order to call ProB via an interface function, LTSmin needs to transmit the entire state to ProB. ProB then deserializes all variables individually before the interpreter is called. Obviously, not all values are required in order to calculate a state transition or evaluate a predicate, rendering some overhead obsolete. Such an interface function is called *long* function, analogously a state that consists of all state variables is called a *long* state.

Instead of transmitting the entire state, LTSmin can make use of the dependency matrix in order to *project* a long state to state vector that only contains the values of accessed variables. Such a state is named *short* state and is relative to either a state label or transition group. An interface function that receives a short state is, analogously, named a *short* function. Short states can also be *expanded* back to long states by inserting values for the missing variables, e.g., those of a predecessor or initial state.

Short states come in two flavors: firstly, *regular* short states are of a fixed size per transition group or state label and only contain values both written to and read from. Consider the initial state $init^{(c,x,y)} = (100, 0, 0)$ from Fig. 2. The corresponding short state for `incx` only contains c and x since y is not accessed. The projected short state, thus, is $init^{(c,x)}_{short} = (100, 0)$.

On the other hand, *R2W* short states (read to write) differ in the contained variables. LTSmin passes only read variables to ProB which in turn answers with written variables only. For `incx`, the R2W short state that is passed to ProB also is $init^{(c,x)}_{read} = (100, 0)$ because all written variables also are read. However, c is a constant, thus the value does not change. Thus, the returned state only consists of x, i.e., $init^{(c,x)}_{read} \rightarrow_1 = init^{(x)}_{write}$ with $init^{(x)}_{write} = (1)$.

Because there might be non-deterministic write accesses to variables ("may write"), a so-called *copy vector* is additionally passed to LTSmin. This copy vector is a bitfield marking which variables were actually written to and which values are taken from an earlier state. Additionally, the must-write matrix is given to LTSmin in order to expose which variables will be updated every single time.

The must-write matrix for Fig. 2 is given in Fig. 6. The difference to the may-write matrix is printed in bold. Note that if an entry in the must-write matrix

$$\begin{array}{l} \\ \texttt{incx} \\ \texttt{doublex} \\ \texttt{incy} \\ \texttt{incxmaybey} \end{array} \begin{array}{c} \begin{array}{ccc} \texttt{c} & \texttt{x} & \texttt{y} \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & \mathbf{0} \end{bmatrix} \end{array}$$

Fig. 6: Must-Write Matrix

is 1, it has to be 1 in the may-write matrix, but not vice versa. Then, consider the operation `incxmaybeincy` from Fig. 2. This operation only writes to y when x is an even number. Thus, for $s_{read}^{(c,x,y)} = (100, 2, 2)$ and $s \rightarrow_4 s'$, $s'^{(x,y)}_{write} = (3, 3)$ and $cpy_{s \rightarrow s'}^{(x,y)} = (0, 0)$, because both variables were actually written to. However, for $\hat{s}_{read}^{(c,x,y)} = (100, 3, 2)$ and $\hat{s} \rightarrow_4 \hat{s}'$, $\hat{s}'^{(x,y)}_{write} = (4, 2)$. Yet, $cpy_{\hat{s} \rightarrow \hat{s}'}^{(x,y)} = (0, 1)$ since y was not written to and the old value was copied.

Internally, LTSmin may use both long and short states and convert freely between them. The ProB language frontend always communicates R2W short states in order to minimize overhead by communication and (de)serialization. However, the caching layer works on regular short states, while, e.g., the variable reordering uses long states.

3.3 Caching Mechanism

While the symbolic backend calls the next-state function once with a state that represents a set of states, the sequential backend calls it for each of the states and transition groups. Analogously, the same holds true for state labels and calls to the labelling function.

Since we already calculate dependency matrices, which contain information about which variables are read and, in case of the next-state function, are written to, we can calculate the corresponding short state instead. Then, in order to avoid transferring and (de)serializing states as well as calculating the same state transition or state label multiple times, we can store results in hash maps, one per transition group and state label each. These hash maps map the corresponding short state to either a list of (short) successors states or a Boolean value in case of state labels. Only if the lookup in the hash table fails, the state is transferred to ProB. Otherwise, LTSmin can calculate the result by itself.

Currently, all operations are cached. Obviously, as more variables are accessed by an operation, caching offers lower benefit in exchange to the amount of memory consumed.

3.4 Guard Splitting

Operations (aka events or state transitions) are guarded, i.e., the action part that substitutes variables may only be executed if the guard predicate is satisfied. When LTSmin asks ProB to calculate successor states for a given transition

group, ProB will evaluate the guard and, if applicable, try to find all (or a limited amount of) successors.

It is easy to make the following two observations: firstly, it is often more performant to evaluate single conjuncts of a guard individually. Usually, they access only a very limited amount of variables and can easily be cached. As an example, the guard $x < c$ of `incx` in Fig. 2 only reads two state variables (of which one is constant). Secondly, the same conjunct might guard multiple operations and, if evaluated for one operation in the same state, does not require additional evaluation for another operation. In Fig. 2, both `incx` and `incxmaybey` share the same guard $x < c$.

LTSmin's symbolic backend supports splitting the action from evaluating its guard. A new interface function next-action is provided that works similar to the next-state function, but assumes the guard of an operation to be true. Then, only the action part is evaluated. A special matrix (reads-action) is required for the next-action function that only contains variables that are read during the action part.

Additionally, each guard predicate is split into its conjuncts and associated with the corresponding transition groups in the guard matrix. Each conjunct is added to the state labels announced to LTSmin and their accessed variables are stored in the state label matrix. Parameter constraints however are considered to be part of the action. E.g., the guard $n > 0 \wedge n < c \wedge y < c$ of `incy(n)` in Fig. 2 is split into two: only $y < c$ is the actual guard for LTSmin and $n > 0 \wedge n < c$ is evaluated when calling the next-action function. The new matrices and the new rows in the state label matrix can be found in Fig. 7. Differences between the read matrix from Fig. 3 and the reads-action matrix are highlighted in bold.

	c	x	y
incx	**0**	1	0
doublex	**0**	1	0
incy	**0**	0	1
incxmaybey	1	1	1

(a) Reads-Action Matrix

	c	x	y
x < c	1	1	0
x < c/2	1	1	0
x > 0	0	1	0
y < c	1	0	1

(b) Extension of the State Label Matrix

	x < c	x < c/2	x > 0	y < c
incx	1	0	0	0
doublex	0	1	1	0
incy	0	0	0	1
incxmaybey	1	0	0	0

(c) Guard Matrix

Fig. 7: Matrices for Guard-Splitting

3.5 Partial Order Reduction

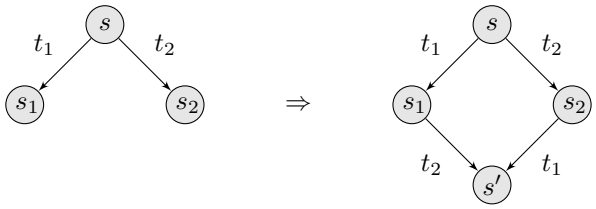
Partial order reduction (POR) [31,32] is a technique that reduces the amount of considered states based on a property that is checked. This is achieved by making use of additional information that can usually be inferred by static analysis.

In the following, we describe which relationships need to be calculated in order to achieve the best reduction with LTSmin.

Definition 3 (According with, based on [25]). *Let $t_1, t_2 \in \mathcal{T}$ be any two operations. We define t_1 to be according with t_2, iff*

$$\forall s, s_1, s_2 \in \mathcal{R} : s \xrightarrow{t_1} s_1 \wedge s \xrightarrow{t_2} s_2 \implies \exists s' : s_1 \xrightarrow{t_2} s' \wedge s_2 \xrightarrow{t_1} s'$$

or as graphical representation:



We define that no $t \in \mathcal{T}$ accords with itself.

Accordance of transition groups expresses that they are independent from each other, i.e., depending on the property, not all interleavings have to be considered. ProB underapproximates the according-with relationship. Instead, the constraint $\forall s, s_1, s_2 \in \mathcal{S} : s \xrightarrow{t_i} s_1 \wedge s \xrightarrow{t_j} s_2 \implies \exists s' : s_1 \xrightarrow{t_j} s' \wedge s_2 \xrightarrow{t_i} s'$ is evaluated for a given pair $t_i, t_j \in \mathcal{T}$, $i \neq j$, considering the guards and the before-after predicates of both transitions. This does not ensure that any state is reachable. However, it is a valid *over*approximation of the does *not* accord relationship matrix that is passed to LTSmin by negating all entries.

LTSmin uses a heuristic in order to determine which of the calculated stubborn sets [31] might yield the best state space reduction. It requires a good approximation of the necessary enabling sets to do so:

Definition 4 (Necessary Enabling Set (NES), based on [25]). *Let g be any state label that is **disabled** in some state $s \in \mathcal{R}$, i.e. $\neg\mathbf{g}(\mathbf{s})$.*

*A set of transitions $\mathcal{N}_g$ is called the necessary **enabling** set for g in s, if for all states $s' \in \mathcal{R}$ with some sequence $s \xrightarrow{t_1,\ldots,t_n} s'$ and $\mathbf{g}(\mathbf{s}')$, for at least one transition t_i $(1 \leq i \leq n)$ we have $t_i \in \mathcal{N}_g$.*

We can use an already existing implementation of the `test_path` procedure (cf. [15], definition 2) in order to calculate the necessary enabling set. In the implementation, it is just tested for any given state, whether a single transition can enable the guard label. In particular, this means that the states s and s' in Definition 4 may not be reachable at all. This results in a safe approximation but may lose precision.

Along with the NES, a necessary *disabling* set (NDS) is approximated. It is calculated in the same way but uses the negation of the state label.

Information about the NES and NDS matrices can syntactically be approximated from the dependency matrix. Solving the given constraints often results in a better approximation and, thus, a better reduction.

Furthermore, with additional information one might prove that out of at least three transition, e.g., t_1, t_2 and t', some transitions, e.g, t_2 and t' might not be enabled at the same time. Then, not all interleaving of t_1 and t_2 need to be considered. This situation is depicted in Fig. 8, where s_1 does not have to be visited. Then, a may-be co-enabled matrix is calculated, based on the following definition:

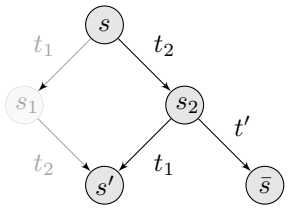


Fig. 8: Reduction Using Co-Enabledness

Definition 5 (Co-Enabledness, based on [25]). *Two state labels $l, l' \in \mathcal{L}$ are co-enabled in a state $s \in \mathcal{R}$ iff they both evaluate to true in s, i.e. $l(s) = true = l'(s)$.*

Again, instead of working on reachable states, it is checked whether there is *any* state in the Cartesian product of types where both labels are enabled. If the co-enabledness of two state labels cannot be determined, they are considered as may-be co-enabled.

While ProB offers an implementation partial order reduction as well [16,17], the partial order reduction algorithm implemented in LTSmin uses a finer heuristic. ProB checks whether a transition can enable another transition, LTSmin uses information about whether individual guards of the event can be enabled, often resulting in a better reduction.

However, this reduction comes with a tradeoff: to calculate the additional matrices, more constraints have to be solved by ProB in order to determine the additional relationships, resulting in a longer analysis time. ProB only calculates accordance of transitions and one row in the NES matrix per transition instead of per guard conjunct. Furthermore, LTSmin does not allow both transitions to be enabled and not generating any successors at the same time. This is possible in ProB when no suitable parameters exist. Thus, an additional guard is added which is an existential quantification of the parameters and often rather costly to evaluate. This quantification has to be solved both on evaluation of the guard and computation of successor states.

3.6 LTL Formula Checking

In order to check LTL properties, both tools need to have access to the formula: only ProB is capable of dealing with atomic propositions properly since it implements the syntax and semantics of the B language. LTSmin, however, requires the formula in order to generate the corresponding Büchi automaton.

Thus, ProB parses the formula first. All atomic propositions in the formula are replaced with a newly generated state label. In order to evaluate these new atomic propositions, the state labelling function is extended in ProB. Furthermore, the formula is wrapped in a "next" operator in order to skip the artificial

initial state introduced earlier. This modified formula is then pretty printed into a format that LTSmin can parse.

4 Evaluation

In this section, we will evaluate the performance of the tool integration of LTSmin and ProB using both the sequential and symbolic backend. We will compare the model checking time on several models from literature and industrial applications which are publicly available under `https://github.com/pkoerner/prob-ltsmin-models`. Benchmark scripts are included in the repositories as well.

Furthermore, we will compare the impact of the implementations of partial order reduction for a different set of models, where the state space can be reduced.

Each benchmark was run on a machine featuring two Intel Xeon IvyBridge E5-2697 with twelve cores each running at 2.70 GHz and 100 GB of RAM. Two CPUs were reserved for each run of invariant verification, partial order reduction and LTL model checking. For multi-core benchmarks, we reserved as many CPUs as there are worker threads plus one CPU for ZeroMQ overhead.

The given values are the median value of ten repetitions.

4.1 Invariant Checking

We benchmarked three backends on multiple B and Event-B models:

- ProB: the vanilla ProB model checker
- LTSmin (seq): the sequential backend of LTSmin with the ProB interpreter,
- LTSmin (sym): the symbolic backend of LTSmin without guard-splitting and the ProB interpreter.

We omit results with guard-splitting enabled since they are very similar to the symbolic backend without guard-splitting for applicable models, i.e., those written in classical B.

Runtimes and memory consumption can be found in Table 1. Runtimes of LTSmin's sequential and symbolic backend do not include ProB's startup time which includes parsing and minor analysis of the model. None of the considered models has any invariant violation and, thus, all models have to be explored exhaustively.

Only for one of models benchmarked, the "Set Laws" machine, a single backend of LTSmin is slower than ProB. Apart from that, we can observe speed-ups ranging from two-fold up to more than two hundred times. For most models, LTSmin is at least an order of magnitude faster than ProB.

The "Train" machine cannot be checked by vanilla ProB on the benchmarking machine, as it runs out of main memory after three days, exploring about half the state space. Both LTSmin backends manage to verify the entire

	Tool	ProB	LTSmin (seq)	LTSmin (sym)
Four Slot	Runtime	26.33	0.99	1.12
(Simpson's Algorithm)	Speed-up	1.00	26.60	23.51
46 657 states	Memory	227.21	11.14	426.07
Landing Gear	Runtime	61.38	1.04	0.65
[18]	Speed-up	1.00	59.02	94.43
43 306 states	Memory	244.01	11.95	425.77
RETHER protocol	Runtime	77.75	4.87	6.09
[34]	Speed-up	1.00	15.97	12.77
42 253 states	Memory	304.08	12.61	430.36
Set Laws	Runtime	232.37	120.57	301.03
	Speed-up	1.00	1.93	0.77
35 937 states	Memory	428.45	87.05	501.86
Earley Parser	Runtime	24612.00	15153.00	6476.00
(J.-R. Abrial)	Speed-up	1.00	1.62	3.80
472 886 states	Memory	4218.62	5224.57	4833.13
CAN Bus	Runtime	131.51	2.68	2.80
(John Colley)	Speed-up	1.00	49.07	46.97
132 599 states	Memory	346.74	24.14	435.50
Mercury Orbiter	Runtime	2608.28	14.14	10.76
[14]	Speed-up	1.00	184.46	242.41
589 278 states	Memory	2360.06	68.66	428.34
Mode Protocol	Runtime	1393.97	317.90	381.20
‡ [14]	Speed-up	1.00	4.38	3.66
336 648 states	Memory	1097.70	151.36	536.55
Core	Runtime	1921.58	315.77	320.05
	Speed-up	1.00	6.09	6.00
160 946 states	Memory	1751.64	314.94	742.17
Train	Runtime	600000 †	33124.00	49120.00
[1]	Speed-up	1.00	18.11	12.21
61 648 077 states	Memory	> 100 000	18 887.32	19 815.79
Train	Runtime	98.59	51.79	71.01
(reduced version)	Speed-up	1.00	1.90	1.39
24 636 states	Memory	198.48	39.00	493.39

Table 1: Invariant Checking Performance (Runtime in Seconds, Memory in MB) of ProB alone compared to LTSmin with ProB. †: Estimated Runtime, ‡: Limited Amount of Initializations

state space in several hours. Only in few instances, LTSmin requires more memory than ProB. Surprisingly, the sequential backend often requires an order of magnitude less memory, even though it maintains a cache. In the only instance where it uses more memory, i.e., "Earley Parser", only few variables re-use the same values. Thus, almost no sharing between in states is possible and the entire fastrw representation for almost every state has to be stored.

Overall, LTSmin is able to outperform ProB in almost all instances. Obviously, caching is really important for the sequential backend. We tried two implementations of a similar caching mechanism for ProB in order to benefit from similar speed-ups: firstly, by hashing short states as well as asserting them as Prolog facts, and, secondly, by serializing the state via the fastrw library and storing the result in a hash map in C. However, neither implementation had a similar impact. Often, they even slowed ProB down. In the first case, we assume that the hashing algorithms for Prolog terms are too slow. For the second implementation, the overhead to serialize every state, in particular for cache lookups, was too costly.

4.2 LTL Model Checking

We benchmarked LTL model checking on a few of aforementioned models that are reasonably sized. Since no LTL formulas were included in the models, we arbitrarily picked some that allow reasoning about the models. ProB's special syntax is used in the formulas: $e(x)$ means that the operation x is enabled. The results are given in Table 2

We can see that for LTL formulas that hold, the good speed-ups and low memory consumption of LTSmin that was presented in Section 4.1 can also be observed for LTL model checking. If a formula is not satisfied, LTSmin can be more than thousandfold faster. While ProB generates the state space of the transition system first, LTSmin features an on-the-fly LTL model checking algorithm where the state space consisting of the Cartesian product of the corresponding Büchi automaton and the actual transition system is generated as necessary. Thus, accepting loops can be found quickly without exploring the entire transition system and speed-ups may be more than thousandfold.

4.3 Partial Order Reduction

From benchmarks conducted in [21,17], it became clear that, in typical B and Event-B models, partial order reduction usually does not work well in combination with invariant verification. For the models above, none or very little reduction was achievable. Instead, we compare the results of partial order reduction for deadlock checking.

In ProB, we use the "least" heuristic for the partial order reduction. In order to reduce analysis time for LTSmin, the timeout is set to 20 milliseconds per predicate.

Results are shown in Table 3 comparing the performance of ProB's POR for deadlock checking with the one of LTSmin. B models that offer no reduction

RETHER protocol [34]	Tool	ProB	LTSmin
G (*not* $e(reserve)$	Runtime	76.67	0.14
$\implies X(e(grant)$ & $e(no\_grant)))$ †	Speed-up	1.00	547.64
	Memory	335.92	5.57
RETHER protocol [34]	Tool	ProB	LTSmin
$GF\ e(pass\_token)$	Runtime	77.50	4.79
	Speed-up	1.00	16.18
	Memory	335.31	15.02
CAN Bus (John Colley)	Tool	ProB	LTSmin
$GF\ e(Update)$	Runtime	125.49	3.18
	Speed-up	1.00	39.46
	Memory	439.01	29.21
CAN Bus (John Colley)	Tool	ProB	LTSmin
$G\ (e(T1Wait) \implies X(T1\_timer > 0))$ †	Runtime	126.80	0.24
	Speed-up	1.00	528.33
	Memory	435.25	6.23
Mode Protocol [14]	Tool	ProB	LTSmin
$GF\ e(evt\_DeliverOK)$ †	Runtime	1406.91	0.30
	Speed-up	1.00	4689.70
	Memory	1336.25	6.39

Table 2: Runtimes (in Seconds) and Speed-ups of LTL Model Checking.
†: LTL formula does not hold. Speed-up compared to ProB without LTSmin.

with either tool are omitted (which are more than half). Again, the startup time of ProB that includes parsing the machine file, is not included.

As can be seen in Table 3, LTSmin is – as discussed in Section 3.5 – able to find a reduction that is equal to ProB's or even better (for the "Set Laws" machine, the additional state is the artificial initial state). Indeed, for the "Mercury Orbiter" and "Landing Gear", ProB cannot reduce the state space at all with the given heuristic, whereas LTSmin reduces the state space by about a quarter up to a half.

However, fine-tuning of the time-outs for the static analysis is important. For machines with many unique guards, analysis time can easily exceed model checking time. This can be observed for the "Landing Gear" and the extremely reduced "Four Slot". With the default time-out value of 300 milliseconds, the analysis time of the "Mercury Orbiter" can exceed a minute, which is more than four times the time required for model checking without any reduction. A reason could be that the constraint solver cannot solve the necessary predicates easily and time-outs are raised often.

Four Slot	Tool	ProB	LTSmin
46 657 states	Analysis Time	0.08	0.23
	Model Checking Time	24.37	0.90
	States (after reduction)	44 065	44 065
Landing Gear [18]	Tool	ProB	LTSmin
43 306 states	Analysis Time	0.98	2.43
	Model Checking Time	87.94	1.25
	States (after reduction)	43 306	29 751
Set Laws	Tool	ProB	LTSmin
35 937 states	Analysis Time	0.30	0.37
	Model Checking Time	0.14	0.07
	States (after reduction)	33	34
CAN Bus (John Colley)	Tool	ProB	LTSmin
132 599 states	Analysis Time	1.38	1.49
	Model Checking Time	94.90	2.24
	States (after reduction)	85 515	67 006
Mercury Orbiter [14]	Tool	ProB	LTSmin
589 278 states	Analysis Time	51.00	9.29
	Model Checking Time	2757.33	16.19
	States (after reduction)	589 278	316 164

Table 3: Runtimes (in Seconds) and Impact of POR in ProB Alone and LTSmin with ProB for Deadlock Checking.

4.4 Multi-Core Model Checking

In order to evaluate the performance of the multi-core backend, we performed multi-core invariant verification on the five B models with the largest runtime in Section 4.1.

The runtime of runs with different amount of workers are shown in Table 4. Speed-ups are visualized in Fig. 9. This time, ProB's startup time is included because the multi-core extension of LTSmin starts up ProB.

Currently, each worker thread maintains its own cache. Since caching works fairly well for many B machines, it is quite costly to fill each cache individually. This explains that often, for the first few workers the speed-up is nearly linear, but grows slower when more than ten workers participate.

An exception is the "Earley Parser": the standalone sequential backend does not offer much speed-up (cf. Section 4.1). Thus, the caching effects are lower. Additionally, ProB spends much time deserializing the state variables because the Prolog terms grow quite large. Hence, a more linear speed-up is possible for many workers.

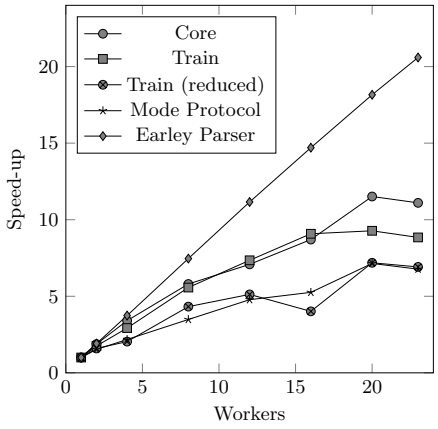


Fig. 9: Speed-ups of Multi-Core LTSmin+ProB Model Checking for Complex Models

Earley Parser	Workers	1	4	8	12	16	20	23
(J.-R. Abrial)	Runtime	15152	4051	2030.76	1358.06	1030.59	834.46	735.76
472 886 states	Speed-up	1.00	3.74	7.46	11.16	14.70	18.16	20.59
Mode Protocol	Workers	1	4	8	12	16	20	23
‡ [14]	Runtime	328.29	150.71	94.23	68.74	62.46	45.96	48.48
336 648 states	Speed-up	1.00	2.18	3.48	4.78	5.26	7.14	6.77
Core	Workers	1	4	8	12	16	20	23
	Runtime	317.48	93.18	54.73	44.82	36.48	27.57	28.60
160 946 states	Speed-up	1.00	3.41	5.80	7.08	8.70	11.52	11.10
Train	Workers	1	4	8	12	16	20	23
[1]	Runtime	32805	11215	5889	4464	3612	3536.60	3710
61 648 077 states	Speed-up	1.00	2.93	5.57	7.35	9.08	9.28	8.84
Train	Workers	1	4	8	12	16	20	23
(reduced)	Runtime	62.39	30.59	14.43	12.18	15.54	8.68	9.02
24 636 states	Speed-up	1.00	2.04	4.32	5.12	4.01	7.19	6.92

Table 4: Runtimes (in Seconds) and Speed-ups of Multi-Core LTSmin with ProB Model Checking on the High-Performance Cluster. ‡: Limited Amount of Initializations

5 Conclusion, Related and Future Work

In this paper, we presented and evaluated significant improvements of the existing link of ProB and LTSmin. It allows state-of-the-art model checking of industrial-sized models with large state spaces, where the vanilla ProB model checker struggles due to time or memory constraints. E.g., the "Train" example can only be checked successfully with ProB on its own by distributing the workload onto several machines, whereas with LTSmin, it could be verified on an ordinary notebook or workstation.

We have compared symbolic reachability analysis to the impact of alternative techniques that can speed up state space generation, e.g., partial order reduction and symmetry reduction in [4]. A discussion of the impact of algorithms that are implemented in both LTSMIN and PROB can also be found in Section 4.

Additionally, there is another toolset named `libits` [13]. It supports, like LTSMIN, symbolic model checking using decision diagrams, variable reordering and LTL as well as CTL model checking. As we understand, its input formalisms are translated into its guarded action language (GAL). An integration for B might prove to be infeasible because a constraint solver is required in order to compute some state transitions and would have to be implemented in GAL itself. We do not know yet whether linking PROB with `libits` in order to compute state transitions is possible.

The caching performed by LTSMIN seems to be particularly efficient. For several realistic examples, the PROB and LTSMIN link achieves two to three orders of magnitude improvements in runtime. Yet, there are several aspects that require future work: currently, the PROB front-end of LTSMIN does not support parallel symbolic model checking with Sylvan [33]. Furthermore, caches are local per worker. A shared cache will most likely improve the scaling for massive parallel model checking. Additionally, the cache does not implement R2W semantics, which loses information about write accesses and requires more memory and additional state transformations.

Moreover, there is room for interesting research: while we know from experience that, for most B models, partial order reduction often only has little to no impact on the state space, we are unsure why. Would alternative algorithms perform better? Is the constraint solver not strong enough? Is the approximation given to LTSMIN not precise enough? Does partial order reduction not perform with the modeling style employed in B? If so, are there any patterns which hinder it? Additionally, a proper survey of distributed model checking of B and Event-B specifications – which we did not touch upon due to page limitations – between, e.g. LTSMIN's distributed backend, PROB's *distb* [22] and TLC [36] should be considered.

With the gained experience and shared know-how about both LTSMIN and PROB, we now aim to extend the implementation for CSP‖B, where the execution of classical B machine is guided by a CSP specification. While PROB provides both an animator and model checker for this formalism, PROB currently is not a satisfactory tool for this formalism. Interleaving of several processes causes an enormous state space explosion that we hope the symbolic capabilities of LTSMIN can manage.

Acknowledgement. Computational support and infrastructure was provided by the "Centre for Information and Media Technology" (ZIM) at the University of Düsseldorf (Germany). We also thank Ivaylo Dobrikov and Alfons Laarman for their helpful explanations concerning partial order reduction algorithms.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 1st edition, 2010.
2. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International journal on software tools for technology transfer*, 12(6):447–466, 2010.
3. J.-R. Abrial, M. K. Lee, D. Neilson, P. Scharbach, and I. H. Sørensen. The B-method. In *Proceedings VDM*, volume 552 of *LNCS*. Springer, 1991.
4. J. Bendisposto, P. Körner, M. Leuschel, J. Meijer, J. van de Pol, H. Treharne, and J. Whitefield. Symbolic Reachability Analysis of B through ProB and LTSmin. In *Proceedings iFM*, volume 9681 of *LNCS*. Springer, 2016.
5. J. Bendisposto and M. Leuschel. Proof assisted model checking for B. In *International Conference on Formal Engineering Methods*, volume 9675 of *LNCS*, pages 504–520. Springer, 2009.
6. J. C. Bicarregui, J. S. Fitzgerald, P. G. Larsen, and J. Woodcock. Industrial practice in formal methods: A review. In *International Symposium on Formal Methods*, volume 5850 of *LNCS*, pages 810–813. Springer, 2009.
7. S. Blom and J. van de Pol. Symbolic Reachability for Process Algebras with Recursive Data Types. In J. S. Fitzgerald, A. E. Haxthausen, and H. Yenigun, editors, *Proceedings ICTAC*, volume 5160 of *LNCS*, pages 81–95. Springer, 2008.
8. S. Blom and J. van de Pol. Symbolic Reachability for Process Algebras with Recursive Data Types. In *Proceedings ICTAC*, volume 5160 of *LNCS*, pages 81–95. Springer, 2008.
9. S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and Symbolic Reachability. In *Proceedings CAV*, volume 6174 of *LNCS*. Springer, 2010.
10. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.-J. Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.
11. M. Carlsson, J. Widen, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland. *SICStus Prolog user's manual*. Swedish Institute of Computer Science Kista, 1988.
12. G. Ciardo, R. M. Marmorstein, and R. Siminiceanu. The saturation algorithm for symbolic state-space exploration. *STTT*, 8(1):4–25, 2006.
13. M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg. Towards distributed software model-checking using decision diagrams. In *International Conference on Computer Aided Verification*, pages 830–845. Springer, 2013.
14. D. Deliverable. D20–Report on Pilot Deployment in the Space Sector. *FP7 ICT DEPLOY Project. January*, 2010. Available at `http://www.deploy-project.eu/html/deliverables.html`.
15. I. Dobrikov and M. Leuschel. Optimising the ProB Model Checker for B using Partial Order Reduction. In D. Giannakopoulou and G. Salan, editors, *Proceedings SEFM 2014*, volume 8702 of *LNCS*, pages 220–234, 2014.
16. I. Dobrikov and M. Leuschel. Optimising the ProB model checker for B using partial order reduction. *Formal Aspects of Computing*, 28(2):295–323, 2016.
17. I. M. Dobrikov. *Improving Explicit-State Model Checking for B and Event-B*. PhD thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf, 2017.
18. D. Hansen, L. Ladenberger, H. Wiegard, J. Bendisposto, and M. Leuschel. *Validation of the ABZ Landing Gear System Using ProB*, volume 433 of *CCIS*. Springer, 2014.

19. P. Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.
20. G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *Proceedings TACAS*, pages 692–707, 2015.
21. P. Körner. An Integration of ProB and LTSmin. Master's thesis, Heinrich Heine Universität Düsseldorf, February 2017.
22. P. Körner and J. Bendisposto. Distributed Model Checking Using ProB. In *Proceedings NFM 2018*, volume 10811 of *LNCS*. Springer, 2018.
23. S. Krings and M. Leuschel. Proof assisted symbolic model checking for B and Event-B. In *Proceedings ABZ*, pages 135–150. Springer, 2016.
24. S. Krings and M. Leuschel. SMT Solvers for Validation of B and Event-B Models. In *Proceedings iFM*, volume 9681, pages 361–375. Springer, 2016.
25. A. Laarman, E. Pater, J. Van De Pol, and M. Weber. Guard-based partial-order reduction. In *Proceedings SPIN Workshop*, pages 227–245. Springer, 2013.
26. M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME*, volume 2805 of *LNCS*. Springer, 2003.
27. J. Meijer, G. Kant, S. Blom, and J. van de Pol. Read, Write and Copy Dependencies for Symbolic Model Checking. In *Proceedings HVC*, volume 8855 of *LNCS*, pages 204–219. Springer, 2014.
28. J. Meijer and J. van de Pol. Bandwidth and Wavefront Reduction for Static Variable Ordering in Symbolic Reachability Analysis. In *Proceedings NFM*, volume 9690 of *LNCS*, pages 255–271. Springer, 2016.
29. D. Plagge and M. Leuschel. Validating B,Z and TLA + Using ProBand Kodkod. In D. Giannakopoulou and D. Méry, editors, *Proceedings FM*, pages 372–386. Springer, 2012.
30. C. Spermann and M. Leuschel. ProB gets nauty: Effective symmetry reduction for B and Z models. In *Proceedings TASE*, pages 15–22. IEEE, 2008.
31. A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings ICATPN*, volume 483 of *LNCS*, pages 491–515. Springer, 1989.
32. A. Valmari. A stubborn attack on state explosion. In *Proceedings CAV*, volume 531 of *LNCS*, pages 156–165. Springer, 1990.
33. T. van Dijk and J. van de Pol. Sylvan: multi-core framework for decision diagrams. *STTT*, 19(6):675–696, 2017.
34. C. Venkatramani and T.-c. Chiueh. Design, implementation, and evaluation of a software-based real-time ethernet protocol. *ACM SIGCOMM Computer Communication Review*, 25(4), 1995.
35. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys (CSUR)*, 41(4), 2009.
36. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Proceedings CHARME*, volume 1703 of *LNCS*. Springer, 1999.