Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more

Daniel Plagge, Michael Leuschel

Softwaretechnik und Programmiersprachen
Institut für Informatik, Heinrich-Heine-Universität Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany
e-mail: {plagge,leuschel}@cs.uni-duesseldorf.de

The date of receipt and acceptance will be inserted by the editor

Abstract. The size of formal models is steadily increasing and there is a demand from industrial users to be able to use expressive temporal query languages for validating and exploring high-level formal specifications. We present an extension of LTL, which is well adapted for validating B, Z and CSP specifications. We present a generic, flexible LTL model checker, implemented inside the PROB tool, that can be applied to a multitude of formalisms such as B, Z, CSP, B||CSP, as well as Object Petri nets, compensating CSP, and dSL. Our algorithm can deal with deadlock states, partially explored state spaces, past operators, and can be combined with existing symmetry reduction techniques of PROB. We establish correctness of our algorithm in general, as well as combined with symmetry reduction. Finally, we present various applications and empirical results of our tool, showing that it can be applied successfully in practice.

Key words: Validation and Verification – Notations and Languages – LTL – model checking – B-method – CSP – Z – Integrated Methods – symmetry reduction.[1]

1 Introduction and Motivation

The B-Method and Z are used in railway systems [14], the automotive sector [39], as well as avionics [21]. The size of the formal models is steadily increasing and there is a big demand from industrial users to be able to animate and validate high-level specifications [15], in order to ensure that the correct system is built. PROB is an animator and model-checker primarily designed for B [29]. By now PROB supports other formalisms like Z specifications [37], CSP, B||CSP, Object Petri nets, compensating CSP and dSL (a programming language used for industrial controllers). Beside animation, it can also be used to detect invariant violations, deadlocks and check refinement. However, there is also an industrial demand for expressive temporal query and validation languages[2], in order to validate temporal properties of the system (not easily expressed in B or Z), as well as to navigate in the state space, and ask questions about the future and past of the current state.

In this paper we present a methodology and implementation to satisfy this industrial need by

- using LTL as the core and—based on feedback from case studies—extending it to enable convenient property specification by the user,
- implementing the model checking algorithm and integrating it into the PROB tool set. Due to the flexible, high-level implementation our technology is not limited to B and Z, but can also be applied to CSP, combinations of B, CSP and Z, as well as to a few other domain specific formalisms.
- providing a practical evaluation of our language and tool, showing that we can express a large class of problems (covering many described in earlier literature) and also solve those problems in practice using our implementation.

2 LTL for Formal Models

LTL is a popular temporal logic for model checking [9], and is now considered to be more expressive, intuitive and practically useful than CTL (see, e.g. [49]). Despite an apparent complexity problem (model checking LTL is

[1] This research is partially supported by the EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

[2] Private communication from Kimmo Varpaaniemi, Space Systems Finland.

exponential in the size of the formula), "efficient" algorithms exist for LTL model checking, notably by negating an LTL formula and translating it into a Büchi automata. The most prominent model checking tool that supports LTL is probably SPIN [24]. But note that newer versions of SMV now also support LTL. Despite its popularity and usefulness, there are a number of formalisms which are still lacking an automatic LTL model checking tool.

- The B-Method [1]
  There has also been considerable interest in trying to verify temporal properties for B specifications. In [2] proof obligations are defined for liveness properties in B. A way to reason about temporal properties of B systems is described in [4] amongst others, e.g., checking properties about when operations are enabled. The work in [20] and the associated JAG tool [19] aim to prove LTL properties of B machines by translating Büchi automata into a B representation and generating suitable proof obligations. However, none of these provide a fully automatic model checker, as proof obligations still need to be discharged. The works [3] and [8] also study the use of LTL for B specifications, in particular examining the link with refinement. The work in [36] describes a LTL model checker for B based on CLPS-B [5].The system does not cover the full B language (e.g., no power set construction, no lambda abstractions nor set comprehensions are supported) and deals with standard LTL (albeit with fairness constraints).
  Finally, the model checker PROB [29] is a fully automatic tool, but in its current form can only check safety properties, as well as perform refinement checks. In summary, to our knowledge there is no automatic tool available to check LTL properties for full B (or at least a large subset thereof). The same can be said for the composition of B and CSP (see, e.g. [47] and [6]).
- CSP [41]
  This formalism is supported by the refinement checker FDR [18]. Here, the idea is to model both the system and the property in the *same* formalism, e.g., as CSP processes, and perform refinement checks.
  The relationship between refinement checking and LTL model checking has been studied (e.g., [42] and [13]) and we ourselves have even proposed a way to perform LTL model checking for CSP using FDR in [33], by translating Büchi automata into CSP processes, language intersection into CSP synchronisation and the emptiness check into a refinement check. However, this approach is not that useful in practice (because the complexity is on the wrong side of the refinement check for FDR to be efficient, and because it requires several tools to be applied in sequence).

Contributions: In the rest of this paper we describe an extension of LTL, called $\mathrm{LTL}^{[e]}$, which is well adapted for validating B, Z and CSP specifications, by allowing us to reason about enabled operations and the execution of operations. In addition, we present the implementation of a $\mathrm{LTL}^{[e]}$ model checking algorithm inside PROB, which can

- deal with deadlock states and partially explored state spaces,
- be applied in conjunction with symmetry reduction,
- be directly applied to multiple formalisms, such as B, CSP, B ‖ CSP, Z, Object Petri nets, StAC (CSP with compensations), and dSL.

We establish correctness of our algorithm in general, as well as combined with symmetry reduction. In addition we provide various applications and useful $\mathrm{LTL}^{[e]}$ patterns, as well as empirical results. We also briefly present an extension to allow Past LTL operators [27].

Discussion about the approach: The interested reader may ask the question: "Why did we not translate our formal models into, e.g., Promela and use the SPIN LTL model checker?" Indeed, this approach is perfectly valid, and has proven to be successful for some lower-level languages (e.g., for Java in [23] or dSL in [50]). For very high-level languages, however, this approach becomes much more difficult. Indeed, translating B directly into Promela would be extremely challenging (it is already difficult enough to write a B interpreter in Prolog with constraint solving like we did for PROB), and it is furthermore very difficult to avoid additional state space explosion due to the smaller granularity of Promela (see, [28]).[3] Another option would be to compute the state space with PROB, and then translate it to a Promela model. We have actually implemented such a translation, but it has so far not proven to be practically useful. First, the overhead of starting up an external tool can be considerable (typically 6 seconds were needed for SPIN to generate and compile the pan.c files). Also, translating the high-level properties into atomic Promela properties can be expensive, and it is not obvious how to exploit the symmetry present in the high-level model in the Promela model. Most importantly, the extensions of the LTL language, which are needed for most interesting practical applications discussed in Section 6, are not supported by SPIN. Still, we plan to reevaluate this approach in the future.

3 $\mathrm{LTL}^{[e]}$

We want to use the LTL model checker for models specified in B, Z, etc. Those models can have deadlock states, but usually LTL formulas are defined over Kripke structures such that every state must have at least one successor state. To support models with deadlock states, we

[3] This process was actually attempted in the past — without success — within the EPSRC funded project ABCD at the University of Southampton.

simply extend the definition of a Kripke structure to also allow states without successors.

Another approach to treat deadlock states is to add a 'dummy' loop transition to each deadlock state or even to add an additional state with a loop. But this introduces subtle differences, e.g. when a dummy transition is added, the formula Xp is true in a deadlock state iff p is true in the state itself. We think that Xp should not be true at all because there is no next state. Our approach has the advantage that the underlying labeling transition system remains unaltered whereas the performance impact is very low.

From preliminary case studies it became clear that often it is interesting to know which kind of operation has been performed to get into a certain state. Especially in CSP models, we are often only interested in the operation performed, not in the state between operations. We add labels on transitions in the relation of the Kripke structure. LTL with support for labelled transitions can also be found in [7], but the definition there is limited to infinite paths.

Propositions on transitions give us the possibility to express statements like 'the next operation on the path is *op1*(x) with $x > 3$'. In the formalism below we denote such statements with $[t]$, where t is the proposition on the transition.

Definition 1. A *labelled Kripke structure* M with possible deadlocks over atomic propositions AP and transition propositions TP is a tuple $M = (S, S_0, R, L)$ consisting of a set of states S, a set of initial states $S_0 \subseteq S$, a ternary relation between states $R \subseteq S \times 2^{TP} \times S$, and a labeling function $L \in S \rightarrow 2^{AP}$.

For our purposes we do *not* restrict the relation to be total, so the structure may have deadlock states. The set of deadlock states is

$$\mathit{deadlocks} = \{s \in S | \neg\exists t, s' : (s, t, s') \in R\}.$$

Definition 2. A path π in M can be either infinite or finite ending in a deadlock state:

- A finite path of length $|\pi| = k$, $k \geq 1$ is a finite sequence $\pi = s_0 \xrightarrow{t_0} \ldots \xrightarrow{t_{k-2}} s_{k-1}$ with $s_0 \in S_0$, $s_{k-1} \in$ *deadlocks* and $\forall i : 0 \leq i < k-1 \Rightarrow (s_i, t_i, s_{i+1}) \in R$.
- Infinite paths have the form $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots$, $s_0 \in S_0$, $\forall i \geq 0 : (s_i, t_i, s_{i+1}) \in R$. We denote $|\pi| = \omega$ for the infinite length of π.

We denote π^i as the suffix of π without π's first i elements.

We extend the semantics of LTL formulas in two aspects: First we claim that a formula of the form $X\varphi$ is only true if the current state is not a deadlock. Second we allow to check if a transition proposition holds in the transition to the next state by using the $[t]$ construct. A state s in M satisfies a formula φ (denoted $M, s \models \varphi$) if all paths starting in s satisfy φ. Whether a path π satisfies a formula φ (denoted $M, \pi \models \varphi$, or shorter $\pi \models \varphi$ if M is unambiguous) is defined by:

$$
\begin{array}{ll}
\pi \models \mathit{true} & \\
\pi \models p & \Leftrightarrow \pi = s_0 \ldots \text{ and } p \in L(s_0) \\
 & \quad \text{for atomic propositions } p \in AP \\
\pi \models \neg\varphi & \Leftrightarrow \pi \not\models \varphi \\
\pi \models \varphi \vee \psi & \Leftrightarrow \pi \models \varphi \text{ or } \pi \models \psi \\
\pi \models X\varphi & \Leftrightarrow |\pi| \geq 2 \text{ and } \pi^1 \models \varphi \\
\pi \models \varphi U \psi & \Leftrightarrow \exists k < |\pi| : \pi^k \models \psi \\
 & \quad \text{and } \forall i : 0 \leq i < k \Rightarrow \pi^i \models \varphi \\
\pi \models [t] & \Leftrightarrow |\pi| \geq 2 \text{ and} \\
 & \quad \pi = s_0 \xrightarrow{t_0} \pi^1 \text{ with } t \in t_0 \\
 & \quad \text{for transition labels } t \in TP
\end{array}
$$

So far we have defined only a few basic LTL[e] operators. We introduce other operators like conjunction ($\wedge$), finally (F), globally (G), release (R) and weak until (W) in the usual way:

$$
\begin{aligned}
\mathit{false} &:= \neg\mathit{true} \\
\varphi \wedge \psi &:= \neg(\neg\varphi \vee \neg\psi) \\
F\varphi &:= \mathit{true}\, U \varphi \\
G\varphi &:= \neg F \neg\varphi = \neg(\mathit{true}\, U \neg\varphi) \\
\varphi R \psi &:= \neg(\neg\varphi U \neg\psi) \\
\varphi W \psi &:= G\varphi \vee \varphi U \psi = \neg(\mathit{true}\, U \neg\varphi) \vee \varphi U \psi
\end{aligned}
$$

4 The Model Checking Algorithm

Below we adapt the LTL model checking algorithm from [34] and [9]. One may ask why we did not use the "standard" LTL model checking algorithm based on Büchi automata. Our motivations were as follows:

- It can be easily extended to deal with "open" nodes, on which no information is available. This is especially useful for infinite state systems, where only part of the state space can be computed. Also, it is not clear to what extent Büchi automata can easily deal with the $[t]$ operator from LTL[e].
- The state space for B and Z specifications is, due to the high-level nature of the operations, typically much smaller than for other more low-level formalisms such as Promela. This is especially true when we apply symmetry reduction (cf. Section 7). Hence, the bottleneck is generally not to be found inside the LTL model checking algorithm.
- The algorithm can also later be extended to CTL* [9].

We implemented the algorithm in C, using SICStus Prolog's C-Interface to integrate it into the PROB tool. The model checking module is not specific to the B formalism, in fact it uses callback mechanism to let the Prolog code evaluate the atomic propositions and outgoing transitions for every state.

4.1 Overview of the algorithm

To check if a model satisfies a given LTL[e] formula, we use a modified version of the tableau algorithm given in [34] and [9]. We adapted the algorithm in a way that deadlock states and propositions on transitions are supported.

To check if a state s in the structure M satisfies a given LTL formula, we try to find a counter-example by searching for a path starting in s that satisfies the negated formula φ.

In the next paragraphs we explain how a graph can be constructed that contains some nodes (called atoms) for each state of the model. An atom represents a possible valuation of φ and its subformulas that is consistent with the corresponding state. E.g. for a formula $\varphi = a \vee Xb$ with $a, b \in AP$, the valuations of a and b are defined by the state but there are two atoms, one where Xb is true and one where Xb is false. There is an edge between two atoms A and B if there is a transition between the corresponding states and if subformulas of the form $X\psi$ in A have the same valuation as ψ in B.

Then we search for a path of atoms that serves as a counter-example with the following properties: The path starts with an atom (s_0, F_0) of the initial states $(s_0 \in S_0)$ where $\varphi \in F_0$. And for each atom on the path where $\psi_1 \, U \psi_2$ is true, ψ_1 is true until a state is reached where ψ_2 is true. A counter-example may be infinitely long, then it consists of a finite path, followed by a cycle. To find also those cycles, we search for a strongly connected component (SCC) with certain properties.

We adapt the original algorithm's rules of how atoms can be constructed and when a transition from one atom to another exists. And in contrast to the original algorithm we consider deadlock states in the requirements of the SCC we search for.

After presenting the algorithm, we show how nodes that are not yet explored can be handled and present a proof for the correctness of our extensions to the algorithm.

For the interested reader we provide our algorithm in full detail below. The reader not interested in this can skip to Section 5.

4.2 The closure of a formula

The *closure* $Cl(\varphi)$ of a formula φ is the smallest set of LTL[e] formulas satisfying the following rules:

$$
\begin{aligned}
\varphi &\in Cl(\varphi) \\
\psi \in Cl(\varphi) &\Rightarrow (\neg\psi) \in Cl(\varphi), \\
&\quad\;\text{identifying } \neg\neg\varphi \text{ with } \varphi \\
\psi_1 \vee \psi_2 \in Cl(\varphi) &\Rightarrow \psi_1 \in Cl(\varphi) \text{ and } \psi_2 \in Cl(\varphi) \\
X\psi \in Cl(\varphi) &\Rightarrow \psi \in Cl(\varphi) \\
\neg X\psi \in Cl(\varphi) &\Rightarrow X(\neg\psi) \in Cl(\varphi) \\
\psi_1 \, U \psi_2 \in Cl(\varphi) &\Rightarrow \psi_1 \in Cl(\varphi), \psi_2 \in Cl(\varphi), \\
&\quad\; X(\psi_1 \, U \psi_2) \in Cl(\varphi)
\end{aligned}
$$

Informally, the closure $Cl(\varphi)$ contains all formulas that determine if φ is true. This definition has been taken without modification from the original algorithm.

4.3 Atoms of a state

An *atom* is a pair (s, F) with $s \in S$ and F a consistent set of formulas $F \subseteq Cl(\varphi)$. F is consistent if it satisfies the following rules:

- $p \in F$ iff $p \in L(s)$ for atomic propositions $p \in AP$
- $\psi \in F$ iff $(\neg\psi) \notin F$ for $\psi \in Cl(\varphi)$
- $\psi_1 \vee \psi_2 \in F$ iff $\psi_1 \in F$ or $\psi_2 \in F$ for $\psi_1 \vee \psi_2 \in Cl(\varphi)$
- $\psi_1 U \psi_2 \in F$ iff $\psi_2 \in F$ or $\psi_1, X(\psi_1 U \psi_2) \in F$ for $\psi_1 \, U \psi_2 \in Cl(\varphi)$
- If $s \in$ *deadlocks* then $(\neg X\psi) \in F$ for $X\psi \in Cl(\varphi)$
- If $s \notin$ *deadlocks* then $X\psi \in F \Leftrightarrow (X\neg\psi) \notin F$ for $X\psi \in Cl(\varphi)$
- If $s \in$ *deadlocks* then $(\neg[t]) \in F$ for all transition propositions $t \in TP$.

Our changes to the original rules are as follows: We added the last rule for transition propositions and we introduced the distinction of the cases $s \in$ *deadlocks* and $s \notin$ *deadlocks* for the X operator.

Whether a formula $\psi \in Cl(\varphi)$ is in F or not for an atom (s, F) thus depends on the current state's atomic propositions and whether formulas with next operators $X\psi$ and the transitions propositions $[t]$ are in F or not. We denote the set of all possible atoms of a state s with $A(s)$ and all atoms of the set of states S with $A(S)$. The number of atoms of s is limited by $|A(s)| \leq 2^{N_x} \cdot 2^{N_t}$, where N_x is the number of next operators in $Cl(\varphi)$ and N_t the minimum of the number of transition propositions in $Cl(\varphi)$ and the maximum vertex degree $\Sigma(G)$. Thus the maximum number of atoms grows exponentially with the number of next and until operators (because an until operator implies an additional next operator in the closure) and N_t.

4.4 Search for self-fulfilling SCCs

We construct a directed graph G with the set of all atoms $A(S)$ as nodes. There is an edge from (s_1, F_1) to (s_2, F_2) labelled with t iff

- there is a transition in M from the state s_1 to s_2 with $(s_1, t, s_2) \in R$, and
- $X\psi \in F_1 \Leftrightarrow \psi \in F_2$ for all formulas $X\psi \in Cl(\varphi)$, and
- $[t'] \in F_1 \Leftrightarrow t' \in t$ for all $[t'] \in Cl(\varphi)$

The last rule is an addition to the original algorithm.

A strongly connected component (SCC) C is a maximal subgraph of G such that between all nodes in C there exists a path in C. We search for an SCC C that is reachable from an atom (s_0, F_0) of an initial state $s_0 \in S_0$ with $\varphi \in F_0$ and that has the following properties:

- For every atom (s, F) in the component C, and every formula $\psi_1 U \psi_2 \in F$ there exists an atom (s', F') in C with $\psi_2 \in F'$. ('self-fulfilling')
- There exists an edge in C ('nontrivial') or C consists of exactly one deadlock state.

In contrast to the original algorithm we added the exception for deadlock states.

We use Tarjan's algorithm [46] to identify the SCCs in the graph. If we find a nontrivial self-fulfilling SCC C, we can construct an α-path (a path of atoms in G)

$$\pi_\alpha = \underbrace{(s_0, F_0) \xrightarrow{t_0} \ldots}_{\pi_1} \underbrace{(s_c, F_c) \xrightarrow{t_c} \ldots (s_c, F_c)}_{\pi_2}$$

with $s_0 \in S_0$, $\varphi \in F_0$ and (s_c, F_c) in C. The first part π_1 is the α-path from an initial atom to an atom in the found SCC C. The second part π_2 is a loop in C that includes an atom (s, F) with $\psi_2 \in F$ for each $\psi_1 U \psi_2 \in Cl(\varphi)$.

The path $\pi = s_0 \xrightarrow{t_0} \ldots s_c \xrightarrow{t_c} \ldots s_c$ then acts as a counter-example.

If the found SCC consists of exactly one atom (s_d, F_d) of a deadlock state s_d, the found α-path has the form $\pi_\alpha = (s_0, F_0) \xrightarrow{t_0} \ldots (s_d, F_d)$, and the counter-example is $\pi = s_0 \xrightarrow{t_0} \ldots s_d$.

4.5 Handling of open nodes

An open node in the state space S is a node, whose outgoing transitions are not calculated yet. The algorithm explained above can be easily modified to work with state spaces that contain open nodes. Whenever the outgoing transitions of a node are needed in Tarjan's algorithm, we check if the current node is an open node. If so, all transitions starting in the node will be calculated. This way the LTL$^{[e]}$ model checker can drive the exploration of the state space. Also, part of the state space can remain unexplored, while still ensuring the correctness of the result.

Another alternative is to leave a node unexplored if we want to explore only a limited number of states or restrict the search to a given subset of states. Then we only have to check if the state is a deadlock. If not, we just store the information that we encountered an such a node. The constructed atoms have no outgoing transitions and are trivial SCCs. So the node won't lead to a false counter-example. If we do not find a counter-example we give the user a warning that not the whole state space was considered.

4.6 Correctness of the algorithm

The algorithm is used to find a counter-example for a given LTL formula. This is done by searching for a path of atoms to a self-fulfilling SCC or an SCC that consists of a deadlock state. The proof consists of two steps: First we show the equivalence between the existence of a counter-example and the existence of an eventuality sequence, then we show the equivalence of the existence of an eventuality sequence and the existence of a path to an SCC.

The complete proof (without our additions to the LTL semantics and the algorithm) can be found in [9], we extend it only in a way that it covers our additions.

Definition 3. An *eventuality sequence* π_α is an infinite α-path or finite α-path ending in an atom of a deadlock state such that if $\psi_1 U \psi_2 \in F_A$ for an atom A on π_α, there exists an atom B on π after A with $\psi_2 \in F_B$.

Lemma 1. *There exists a path π starting in s_0 with $\pi \models \varphi$ iff there exists an eventuality sequence starting at an atom (s_0, F_0) such that $\varphi \in F$.*

Proof. We first show that the existence of an eventuality sequence implies the existence of π, then we show the reverse direction.

1. Let $\pi_\alpha = (s_0, F_0) \xrightarrow{t_0} (s_1, F_1) \xrightarrow{t_1} \ldots$ be an eventuality sequence starting in $(s_0, F_0) = (s, F)$ with $\varphi \in F$. The corresponding path is $\pi = s_0, s_1, \ldots$. We prove $\pi \models \varphi$ by showing that $\pi^i \models \psi \Leftrightarrow \psi \in F_i$ holds for every $\psi \in Cl(\varphi)$ and $0 \leq i < |\pi|$. The proof is done by induction on the subformulas of φ. We describe only the two cases that are affected by our changes to the LTL semantics.
   (a) If $\psi = X\varphi_1$: If $s_i \in$ *deadlocks* then by definition $X\varphi_1 \notin F_i$ and $\pi_i \not\models X\varphi_1$.
   If $s_i \notin$ *deadlocks* then we have a transition from (s_i, F_i) to (s_{i+1}, F_{i+1}), implying that $X\varphi_1 \in F_i \Leftrightarrow \varphi_1 \in F_{i+1}$. By induction we know $\varphi_1 \in F_{i+1} \Leftrightarrow \pi^{i+1} \models \varphi_1$ and by definition $\pi^{i+1} \models \varphi_1 \Leftrightarrow X\pi^i \models \varphi_1$ holds.
   (b) If $\psi = [t]$ with $t \in TP$: If $s_i \in$ *deadlocks* then it follows from the definition of an atom that $[t] \notin F_i$ and from the definition of $\models$ if follows $\pi_i \not\models [t]$.

If $s_i \notin \mathit{deadlocks}$ then there exists a transition $(s_i, F_i) \xrightarrow{t'} (s_{i+1}, F_{i+1})$. By definition $[t] \in F_i \Leftrightarrow t = t'$ and $\pi^i = s_i \xrightarrow{t} s_{i+1} \ldots \Leftrightarrow \pi^i \models [t]$.

2. Let $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots$ be a path with $\pi \models \varphi$. Let $F_i = \{\psi \mid \psi \in Cl(\varphi) \wedge \pi^i \models \psi\}$. First we show that there is an α-path $\pi_\alpha = (s_0, F_0) \xrightarrow{t_0} (s_1, F_1) \xrightarrow{t_1} \ldots$.
   (a) All (s_i, F_i) are atoms. This can be seen by comparing the definition of $\models$ with the definition of consistency of an atom's formulas.
   (b) If $s_i \in \mathit{deadlocks}$: s_i is the last element of π. Because there is no transition in R starting in s_i, there is neither a transition starting in (s_i, F_i).
   If $s_i \notin \mathit{deadlocks}$: There is a state s_{i+1} in π. We show that there is a transition from (s_i, F_i) to (s_{i+1}, F_{i+1}): For every $X\psi \in Cl(\varphi)$ holds by definition that $\pi^i \models X\psi \Leftrightarrow \pi^{i+1} \models \psi$ and we have $X\psi \in F_i \Leftrightarrow \pi^i \models X\psi$ and $\psi \in F_{i+1} \Leftrightarrow \pi^{i+1} \models \psi$, it follows that $X\psi \in F_i \Leftrightarrow \psi \in F_{i+1}$.
   Also for every $[t] \in Cl(\varphi)$, $t \in TP$ holds $\pi^i \models [t] \Leftrightarrow t \in t_i$, $\pi^i \models [t] \Leftrightarrow [t] \in F_i$, it follows that $t \in F_i \Leftrightarrow t \in t_i$.

   For every $\psi_1 U \psi_2 \in F_i$ there is a F_j, $i \leq j < |\pi|$, such that $\psi_2 \in F$, because $\psi_1 U \psi_2 \in F_i \Leftrightarrow \pi_i \models \psi_1 U \psi_2$ and by definition there is a j with $i \leq j < |\pi|$, such that $\pi_j \models \psi_2$ and that implies $\psi_2 \in F_j$. So π_α is an eventuality sequence.

Lemma 2. *There exists an eventuality sequence starting at an atom (s_0, F_0) iff there is a path in G from (s_0, F_0) to a self-fulfilling SCC or an SCC consisting of a deadlock state.*

Proof. We only consider the case where the path ends in a deadlock state. The other, infinite, case can be found in the original proof.

1. Let $\pi_\alpha = (s_0, F_0) \xrightarrow{t_0} \ldots \xrightarrow{t_{d-1}} (s_d, F_d)$ be a finite eventuality sequence ending in the deadlock state s_d. We know that (s_d, F_d) has no outgoing edges, so the atom is the only node in the SCC.
2. Let there be a path $(s_0, F_0) \xrightarrow{t_0} \ldots \xrightarrow{t_{d-1}} (s_d, F_d)$ to an SCC consisting of just a deadlock state (s_d, F_d). We have to show that π_α is an eventuality sequence by proving that for every occurrence of $\psi_1 U \psi_2 \in F_i$ $(0 \leq i \leq d)$ there is an atom (s_j, F_j), $i \leq j \leq d$ with $\psi_2 \in F_j$. Then, by the definition of an atom, $\psi_2 \in F_i$ or $X(\psi_1 U \psi_2) \in F$ implies $\psi_1 U \psi_2 \in F_{i+1}$. Now we assume $\psi_2 \notin F_j$ for all j with $i \leq j \leq d$. Then we know that $X(\psi_1 U \psi_2) \in F_j$, it follows that $X(\psi_1 U \psi_2) \in F_d$. But s_d is a deadlock state, so $X(\psi_1 U \psi_2) \notin F_d$, we have a contradiction. So there is an atom (s_j, F_j) with $\psi_2 \in F_j$.

The lemmas 1 and 2 together show that there exists a counter-example to a formula iff the presented algorithm finds a suitable SCC.

5 LTL$^{[e]}$ with past

The definition of LTL$^{[e]}$ in Section 3 only considers operators that enable us to express properties about current or future states of the system. LTL with past [27] makes it possible to reason about previous states and transitions by introducing operators such as

– Y, which stands for "yesterday" and is the dual of the LTL$^{[e]}$ next state operator X, and
– S, which stands for "since" and is the dual of the LTL$^{[e]}$ operator until.

5.1 Definition of Past-LTL$^{[e]}$

To support Past-LTL$^{[e]}$, we first have to make small modifications to the formal definition of the LTL$^{[e]}$ operators. In the original definition of $\models$ in Section 3, the current state of a sequence $\pi = s_0 \xrightarrow{t_0} s_1 \ldots$ is always the first state s_0. This prevents us from using this notation to reason about the past of a sequence. E.g. $\pi \models X\varphi$ is defined by $\pi^1 \models \varphi$, so φ cannot consider the state s_0 anymore. To allow this, an index i is introduced that indicates which state of the sequence is the current one.

Whether $\pi = s_0 \xrightarrow{t_0} s_1 \ldots$ satisfies a Past-LTL$^{[e]}$ formula φ in the ith state of the sequence π in the Kripke structure M (denoted $M, (\pi, i) \models_p \varphi$ or shorter $(\pi, i) \models_p \varphi$), is defined by:

$$\begin{array}{lll}
(\pi,i) \models_p \mathit{true} & \Leftrightarrow 0 \leq i < |\pi| \\
(\pi,i) \models_p p & \Leftrightarrow 0 \leq i < |\pi| \text{ and } p \in L(s_i) \\
& \quad \text{for atomic propositions } p \in AP \\
(\pi,i) \models_p \neg\varphi & \Leftrightarrow (\pi,i) \not\models_p \varphi \\
(\pi,i) \models_p \varphi \vee \psi & \Leftrightarrow (\pi,i) \models_p \varphi \text{ or } (\pi,i) \models_p \psi \\
(\pi,i) \models_p X\varphi & \Leftrightarrow i+1 < |\pi| \text{ and } (\pi,i+1) \models_p \varphi \\
(\pi,i) \models_p \varphi U \psi & \Leftrightarrow \exists k : i \leq k < |\pi| \text{ with } (\pi,k) \models_p \psi \\
& \quad \text{and } \forall j : i \leq j < k \Rightarrow (\pi,j) \models_p \varphi \\
(\pi,i) \models_p [t] & \Leftrightarrow i+1 < |\pi| \text{ and } t \in t_i \\
& \quad \text{for transition labels } t \in TP \\
(\pi,i) \models_p Y\varphi & \Leftrightarrow i > 0 \text{ and } (\pi,i-1) \models_p \varphi \\
(\pi,i) \models_p \varphi S \psi & \Leftrightarrow \exists k : 0 \leq k \leq i \text{ with } (\pi,k) \models_p \psi \\
& \quad \text{and } \forall j : k < j \leq i \Rightarrow (\pi,j) \models_p \varphi
\end{array}$$

The definition of $(\pi, 0) \models_p \varphi$ is equivalent to $\pi \models \varphi$ for formulas that do not contain the past operators Y or S.

We introduce operators like once (O, dual to finally), history (H, dual to globally) and trigger (T, dual to release) as usual:

$$\begin{aligned}
O\varphi &:= \mathit{true}\, S \varphi \\
H\varphi &:= \neg O \neg\varphi = \neg(\mathit{true}\, S \neg\varphi) \\
\varphi T \psi &:= \neg(\neg\varphi\, S \neg\psi)
\end{aligned}$$

We decided not to include a dual operator to $[t]$, because it can be easily expressed by $Y[t]$.

5.2 Closure and atoms of a Past-LTL$^{[e]}$ formula

We add two rules to the definition of a closure $Cl(\varphi)$ (cf. Section 4.2).

$$Y\psi \in Cl(\varphi) \Rightarrow \psi \in Cl(\varphi)$$
$$\psi_1 S\psi_2 \in Cl(\varphi) \Rightarrow \psi_1 \in Cl(\varphi), \psi_2 \in Cl(\varphi), Y(\psi_1 S\psi_2) \in Cl(\varphi)$$

To the consistency definition of an atom (s, F), we add rules for Y and S (cf. Section 4.3):

- If $s \notin S_0$ then $Y\psi \in F$ iff $(Y\neg\psi) \notin F$ for $Y\psi \in Cl(\varphi)$
- If $s \in S_0$ then
  - $Y\psi \in F$ iff $(Y\neg\psi) \notin F$ for $Y\psi \in Cl(\varphi)$
  - or $Y\psi \notin F$ for $Y\psi \in Cl(\varphi)$
- $\psi_1 S\psi_2 \in F$ iff $\psi_2 \in F$ or $\psi_1, Y(\psi_1 S\psi_2) \in F$ for $\psi_1 S\psi_2 \in Cl(\varphi)$

The first rule is analogous to the non-deadlock case of the X operator. It states that either $Y\psi$ or $Y\neg\psi$ is true, but not both. But in the first state s_0 of a sequence $\pi = s_0 \xrightarrow{t_0} s_1 \ldots$, we have the additional case that all formulas $Y\psi$ are false, even $Y true$. Because there might be an s_i in π with $s_i = s_0$, we can have both cases in an initial state $s \in S_0$. The third rule is analogous to the rule of the U operator.

We need an additional condition when there is an edge in G from an atom (s_1, F_1) to an atom (s_2, F_2) (cf. Section 4.4):

- $\psi \in F_1 \Leftrightarrow Y\psi \in F_2$ for all formulas $Y\psi \in Cl(\varphi)$

The search for an SCC in G starts in the atoms (s_0, F_0) with $s_0 \in S_0$ where $Y\psi \notin F_0$ for all formulas $Y\psi \in Cl(\varphi)$.

The proofs of Lemma 1 and Lemma 2 can be easily extended to Past-LTL$^{[e]}$. In the case distinctions of the proofs, the operator Y can be handled analogously to the operator X and the operator S analogously to the operator U where the path ends in a deadlock state.

6 Some Examples and Experiments

6.1 LTL$^{[e]}$: Supported syntax and usage patterns

We provide several types of atomic propositions in our implementation:

- One can check if a B predicate holds in the current state by writing the predicate between curly braces, e.g. `{card(set) > 0}`.
- And with `e(op)` it can be tested if an operation `op` is currently enabled.
- If the user animates the model, he can use an atomic proposition `current` to check if a state is the state that is shown in the animator. This allows the user to check a formula f in that state for all paths that go through that state by using `current` $\implies f$.

And some types of transition propositions:

- With `[op]` it can be checked if *op* is the next executed operation in the path.
- We provide simple pattern matching for the arguments of an operation. E.g. with `[op(5,_)]` one can check if the next operation is *op* and if its first argument is 5. Any B expression (including constants and variables of the machine) or underscores as placeholders for any value can be used as an argument.
- In future, we also plan to support combined pre- and post-conditions like $x' > x$ as transition propositions (where x' refers to the value of x after the transition has been executed).

Note that while checking if an operation is enabled with `e(op)` is an atomic proposition, the check if the next transition is done via a certain operation with `[op]` depends on the actual computation path, not only on the current state.

Some useful patterns of LTL$^{[e]}$ formulas for B/Z specifications (and sometimes also CSP) are as follows:

- quasi-deadlock `G (e(O1) or ... or e(On))` where `O1`,..,`On` are the real state-changing operations (as opposed to query operations). To improve support for this pattern, we introduced another atomic proposition keyword `sink` that is true for states that have no outgoing transitions to other states.
- operation post-condition `G ( [Op] => X {Post} )` (`[Op]` tests if the next executed operation is `Op`)
- operation pre-condition `G ( e(Op) => {Pre} )`.
- While animating a model, one may ask if an operation `Op` is executed in the past of the current state, regardless of the computation path being taken to reach it: `current => YO[Op]`.

Later, in Section 6, we will see that LTL$^{[e]}$ is useful in practice to solve a variety of other problems, and can also be used to encode fairness constraints.

In the rest of this section we exhibit the flexibility and practical usefulness of our approach. Notably, we show how our tool can now be used to solve a variety of problems mentioned in the literature. We also show that the tool is practically useful on a variety of case studies.

All experiments were run on a Linux PC with an AMD Athlon 64 Dual Core Processor running at 2 GHz, and using PROB 1.2.8 built from SICStus Prolog 4.0.2. Our model checker can actually drive the construction of the state space on demand. However, to clearly separate the time required for the LTL checking and the state space construction, we have first fully explored the state space in the examples below.

6.2 B Examples: Volvo Vehicle Function, Robot, and Card Protocol

We have tried our tool on a case study performed at Volvo on a typical vehicle function (see [29]). The B machine has 15 variables, 550 lines of B specification, and

26 operations and was developed by Volvo as part of the European Commission IST Project PUSSEE (IST-2000-30103).

To explore the full state space (1360 states and 25696 transitions) PROB required 25.29 seconds. Some of the LTL$^{[e]}$ formulas we checked are as follows:

- `G (e(SetFunctionParameter) => e(FunctionBecomesNotAllowed))`
  The formula is valid and the model checking time is 0.12 seconds.
- `G (e(FunctionBecomesAllowed) => X e(SetFunctionParameter))`
  A counter-example was found after 0.14 seconds.
- `G ([FunctionBecomesAllowed] => X e(SetFunctionParameter))`
  The formula is valid and the model checking time is 0.20 seconds.
- `G(e(FunctionOff) => YO[SetFunctionParameter])`
  The formula is valid and the model checking time is 0.21 seconds.

We have also applied our tool to the (very) small robot specification from [19]. The original LTL formula `G(({Dt=TRUE} & X{Dt=FALSE}) => {De=FALSE})` from [19] can now be validated fully automatically (and instantaneously). It is interesting to observe that the intended temporal property can be more naturally encoded in our extension LTL$^{[e]}$ as follows: `G([Unload] => {De=FALSE})`.

We have applied our tool on the T=1 protocol[4] specification from [8]. Computing the state space took 0.02 seconds (for 15 nodes). We tested the formula P_1 = `G({CardF2=bl} => F{CardF2=lb})` from [8]. This took less than 0.01 seconds (and 36 atoms were computed). However, our model checker provided a counter-example. This is not surprising, as [8] also takes fairness constraints into account. These fairness constraints are written in [8] as $FAIRNESS = \{Eject, Csends\ if\ (CardF2 = bl), Rsends\ if\ (ReaderF2 = bl)\}$. Fortunately, these fairness constraints can be expressed in our LTL$^{[e]}$ language as follows:

f = `GF[Eject] & GF{CardF2=bl => GF[Csends])`
`& (GF{ReaderF2=bl} => GF[Rsends])`

Checking the formula $f \Rightarrow P_2$ was successful (no counter-example found); this took 12.65 seconds (98,304 atoms where computed). The time is an illustration that LTL model checking is exponential in the size of the formula; it may be worthwhile to investigate adapting our algorithm to incorporate fairness, rather than encoding fairness in the LTL$^{[e]}$ formula itself.

Finally, we believe that our LTL model checker can be used to check probabilistic Event B models ([22]), by enabling the encoding of the required fairness constraints.

6.3 CSP Examples: Peterson and Train Level-Crossing

First we tried a standard CSP example from the book web page of [44][5], Peterson's Algorithm version 1. Computing the state space, consisting of 58 nodes and 115 transitions, took 0.44 seconds with PROB (which has recently been extended to handle full CSP-M). Some of the LTL$^{[e]}$ formulas checked are as follows:

- `G ([p1critical] => X(!e(p2critical)) )`
  The formula is valid; the model checking time is 0.01 seconds.
- `G ([p1critical] => X((!e(p2critical)) W [p1leave] ))`
  The formula is valid; the model checking time is 0.01 seconds.
- `G ([p2critical] => X((!e(p1critical)) W [p2leave] ))`
  The formula is valid; the model checking time is 0.02 seconds.

We have also tested version 2 of the same algorithm. Computing the state space with 215 nodes and 429 transitions took 1.28 seconds.The CSP model is more generic and elegant than the first version, which enables us to write a single LTL$^{[e]}$ formula basically covering the last two formulas from above.

- `G ([critical] => X((!e(critical)) W [leave]))`
  The formula is valid; the model checking time is now 0.06 sec.

Another example we tested is `crossing.csp`, also from [44]. This model by Bill Roscoe describes a level crossing gate using discrete-time modelling in untimed CSP. Computing the state space, consisting of 5517 nodes and 12737 transitions, took 53.76 seconds. Some of the LTL$^{[e]}$ formulas we checked are as follows:

- `G F e(enter)` The formula is valid; the model checking time is 0.36 seconds.
- `G F [enter]` A counter-example (of length 554) was found after 0.17 seconds.

6.4 CSP ∥ B Examples: Control Annotations and Philosophers

In [25] it is proposed to check compatibility of a CSP controller with a particular B machine by adding proof obligations. For this the NEXT annotation is introduced, from which the proof obligations are derived. It turns out that these annotations can also be checked (now automatically) by our LTL model checker. For example, for

[4] En27816-3, European Standard—identification cards—integrated circuit(s) card without contacts—electronic signal and transmission protocols, 1992.

[5] http://www.cs.rhul.ac.uk/books/concurrency/

the traffic light controller from [25], the NEXT annotation for the `Stop_All` operation can be checked by the following LTL[e] formula: `G ([Stop_All] => X (e(Go_Moat) & e(Go_Square)))`. This check can be done instantaneously. We have checked all the NEXT assertions from [25] fully automatically and instantaneously.

We have also applied our LTL[e] model checker to a fully combined CSP and B model. The B model is the generic dining philosophers example from [32] instantiated for three philosophers and three forks, using symmetry reduction (cf. Section 7), and where the protocol is specified by a CSP Controller.

```
datatype BPhils = p1 | p2 | p3
channel think,eat,TakeLeftFork,
        TakeRightFork,DropFork : BPhils
MAIN = ||| p: BPhils @ PHIL(p)
PHIL(P) = think!P -> TakeLeftFork!P ->
          TakeRightFork!P -> eat!P ->
          DropFork!P -> DropFork!P -> PHIL(P)
```

We have validated the following LTL[e] formula in 0.06 seconds:

- `G (e(DropFork) U ([TakeLeftFork] or [TakeRightFork]))`

6.5 Z Examples: SAL Example and Workstation Protocol

In [37], PROB was extended to deal with Z specifications. We examined the example from [12], formalising the process of joining an organisation. We were able to check the three LTL formulas described there:

- `! F {card(member)>2}` (PROB provides a counter-example)
- `! F {card(waiting)>2}` (PROB provides a counter-example)
- `G {card(waiting)+card(member)<=3}` (the formula is true)

Model checking time is 0.08 sec to construct the state space plus less than 0.01 sec for each LTL check. This is faster than the times reported in [12] (ranging from 3 seconds to 12 hours depending on the translation to SAL). In addition, we were able to uncover an error in the specification, namely that it is possible to reach a quasi-deadlock state where only probing operations are possible and no "real" operation can be performed, i.e., the following LTL[e] formula is false:

- `G (e(Join) or e(JoinQ) or e(Remove))`

Note that this error was not uncovered in [12].

We have also tested our tool on the workstation protocol industrial case study from [37]. Computation of the state space for 2 workstations took 2.49 seconds, resulting in 68 states.

The formula `G([Transfer]=>X(e(ReadRequestOK) or e(ReadResponse)))` was checked in 0.04 seconds, using 421 atoms.

6.6 Other Formalisms: StAC, Object Petri nets, dSL

PROB has also the ability to load specifications via custom Prolog interpreters following the style of [31], describing the initial states, the properties and the transition relation by using the Prolog predicates `start/1`, `prop/2`, `trans/3`.

This directly opens up LTL model checking for three further formalisms, for which we have such interpreters: Compensating CSP (StAC) [17], Object Petri Nets, [16], and dSL [50].

We have applied our LTL model checker to a door control system specified in dSL. Computing the state space with 9968 nodes and 37357 transitions took 13.25 seconds. Checking a safety property `G e(action)` took 39.21 seconds.

7 LTL Model Checking with Symmetry Reduction

Combining full blown LTL model checking and symmetry reduction is not always easy. If one is not careful, the application of symmetry reduction can lead to unsoundness for more complicated LTL formulas. Quite often, only safety properties or some other subset of LTL is supported. E.g., murphi ([26]) only deals with safety properties. An exception is, e.g., SMC ([45]), which does deal with safety and liveness properties (expressed as an automata).

It turns out that the presented LTL[e] language is the ideal companion to the existing symmetry reduction techniques developed for PROB [30,32,48], i.e., we can apply the symmetry reduction techniques and need to impose no restrictions whatsoever on the LTL[e] formulas. This meant that, in preliminary experiments, we were actually able to model check some examples considerably faster, than using SPIN with partial order reduction on hand-translated Promela models.

Let us recall some of the results from [30]. First, the notion of a permutation is introduced, which can permute elements of deferred sets. DS is the set of all deferred sets of the B machine under consideration.

Definition 4. Let DS be a set of disjoint sets. A *permutation f over DS* is a total bijection from $\cup_{S \in DS} S$ to $\cup_{S \in DS} S$ such that $\forall S \in DS$ we have $\{f(s) \mid s \in S\} = S$.

The following results show that the deferred sets induce a symmetry in the state space: if in a given state s we permute the deferred set elements, the resulting state will be symmetrical to s.

Theorem 1. *For any expression E, predicate P, state $[V := C]$ and permutation function f:*

$$f(E[V := C]) = E[V := f(C)]$$
$$P[V := C] \Leftrightarrow P[V := f(C)]$$

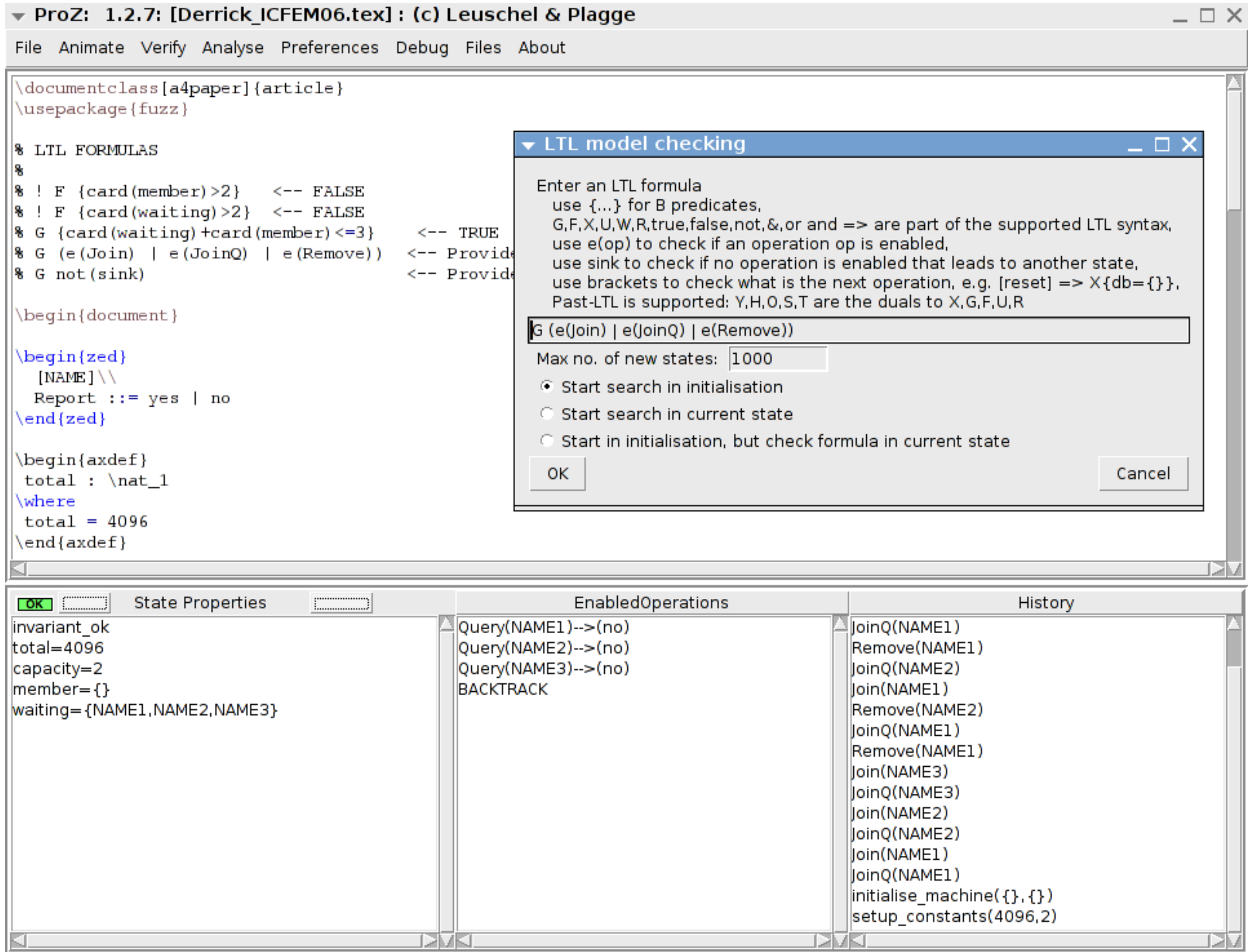

Fig. 1. Counter-example found for the Z model

Corollary 1. *Every state permutation f for a B machine M satisfies*

- $\forall s \in S : s \models I$ *iff* $f(s) \models I$
- $\forall s_1 \in S, \forall s_2 \in S$:
  $s_1 \rightarrow^{M}_{op.a.b} s_2 \quad \Leftrightarrow \quad f(s_1) \rightarrow^{M}_{op.f(a).f(b)} f(s_2)$.

From these theoretical results in [30] we can deduce a new result for LTL$^{[e]}$:

Proposition 1. *Let f be a permutation function, s a state of B machine M and ϕ a LTL$^{[e]}$ formula. Then $M, s \models \phi$ iff $M, f(s) \models \phi$.*

Proof. (Sketch)

- By Theorem 1, if a predicate { `Pred` } is true in a state then it is true in all permutation states.
- By Corollary 1, if a sequence of operations is possible in s then a permuted sequence is possible in the state $\pi(s)$. As the permutation does not affect the enabled operations nor the operation label itself (just the arguments): we can deduce that a LTL$^{[e]}$ formula is true in s iff it is true in $\pi(s)$.

In other words, there exists a LTL$^{[e]}$ counter-example for a B machine iff there exists one with symmetry reduction. As Z specifications are translated internally into B machines ([37]), all of the above also applies when model checking Z specifications.

8 Related and Future Work, Discussion and Conclusion

8.1 More Related Work

There are variety of relatively generic CTL model checkers, such as [31,35,11]. Both [35] and [11] are based on constraint logic programming. [35] requires constructive negation, and as such only a prototype implementation seems to exist. [11] is tailored towards verification of infinite state systems.[6] The CTL model checker in [31] is

[6] Unfortunately, the corresponding DMC prototype available at `http://www.disi.unige.it/person/DelzannoG/DMC/dmc.html` no longer runs on current SICStus Prolog versions and the code is no longer maintained.

generic, and can be applied to any specification language that can be encoded in Prolog. Unfortunately, the model checker relies on tabling, and as such it can only run on XSB Prolog [43], which does not support co-routines and hence the system can *not* be applied to our interpreters for CSP and B (and hence also Z). The same can be said for the pure LTL model checker from [38] or the XMC system [10, 40] for the modal mu-calculus and value-passing CCS.

8.2 Future Work

Adding fairness constraints to an $LTL^{[e]}$ formula leads to an exponential growth of the search graph. We plan to incorporate support for fairness directly into the algorithm by validating if a found SCC satisfies the fairness constraints or not.

We think that expressive transition propositions have many useful applications. We want to extend the model checker in a way that combined pre- and post-conditions can be checked, optionally with access to the parameters of the executed operations.

Currently, a counter-example is presented to the user by moving the animation into the final state of the path and having the complete path in the history of the animation (in Fig. 1, the box in the right bottom corner shows the history). This is unsatisfactory, as it is not always easy to see why the presented path is a counter-example. We plan to improve the presentation and investigate how the length of the counter-example can be minimised.

8.3 Conclusion

In summary, we have presented $LTL^{[e]}$ to conveniently express temporal properties of formal models. Indeed, $LTL^{[e]}$ can be used, e.g., to express pre- and post-conditions of operations, fairness constraints as of [8], the NEXT control annotations from [25], as well as a large class of interesting properties which cannot be directly expressed in pure LTL. We have shown an algorithm for $LTL^{[e]}$, proven it correct with and without symmetry reduction, and have integrated it into the PROB tool set. In the empirical section, we have shown that $LTL^{[e]}$ is expressive enough and that our tool is fast enough for a variety of practical applications.

References

1. J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. H. R. Barradas and D. Bert. Specification and proof of liveness properties under fairness assumptions in B event systems. In M. J. Butler, L. Petre, and K. Sere, editors, *IFM*, LNCS 2335, pages 360–379. Springer, 2002.
3. F. Bellegarde, C. Darlot, J. Julliand, and O. Kouchnarenko. Reformulation: A way to combine dynamic properties and b refinement. In J. N. Oliveira and P. Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 2–19. Springer, 2001.
4. D. Bert, M.-L. Potet, and N. Stouls. Genesyst: A tool to reason about behavioral aspects of B event specifications. application to security properties. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB 2005*, LNCS 3455, pages 299–318. Springer, 2005.
5. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - a constraint solver for B. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 188–204. Springer-Verlag, 2002.
6. M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.
7. S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, V17(4):461–483, December 2005.
8. S. Chouali, J. Julliand, P.-A. Masson, and F. Bellegarde. Pltl-partitioned model checking for reactive systems under fairness assumptions. *ACM Trans. Embedded Comput. Syst.*, 4(2):267–301, 2005.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
10. B. Cui, Y. Dong, X. Du, N. Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of ALP/PLILP'98*, LNCS 1490, pages 1–20. Springer-Verlag, 1998.
11. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *STTT*, 3(3):250–270, 2001.
12. J. Derrick, S. North, and T. Simons. Issues in implementing a model checker for Z. In Z. Liu and J. He, editors, *ICFEM*, LNCS 4260, pages 678–696. Springer, 2006.
13. J. Derrick and G. Smith. Linear temporal logic and Z refinement. In C. Rattray, S. Maharaj, and C. Shankland, editors, *AMAST 04*, LNCS 3116, pages 117–131. Springer, 2004.
14. D. Dollé, D. Essamé, and J. Falampin. B dans le tranport ferroviaire. L'expérience de Siemens Transportation Systems. *Technique et Science Informatiques*, 22(1):11–32, 2003.
15. D. Essamé and D. Dollé. B in large-scale projects: The Canarsie line CBTC experience. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, pages 252–254, Besancon, France, 2007. Springer-Verlag.
16. B. Farwer and M. Leuschel. Model checking object Petri nets in Prolog. In *Proceedings PPDP '04*, pages 20–31, New York, NY, USA, 2004. ACM Press.
17. C. Ferreira and M. Butler. A process compensation language. In T. Santen and B. Stoddart, editors, *Proceedings Integrated Formal Methods (IFM 2000)*, LNCS 1945, pages 424–435. Springer-Verlag, November 2000.
18. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual (version 2.8.2).*

19. J. Groslambert. A jag extension for verifying LTL properties on B event systems. In *Proceedings B'07*, pages 262–265, 2007.
20. J. Groslambert. Verification of LTL on B event systems. In *Proceedings B'07*, pages 109–124, 2007.
21. A. Hall. Using formal methods to develop an atc information system. *IEEE Software*, pages 66–76, March 1996. Reprinted in Industrial-Strength Formal Methods in Practice, M.G. Hinchey & J.P. Bowen, Springer, 1999.
22. T. S. Hallerstede, Stefan und Hoang. Qualitative probabilistic modelling in Event-B. In *IFM'2007*, LNCS 4591, pages 49–63, 2007.
23. J. Hatcliff and M. B. Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In K. G. Larsen and M. Nielsen, editors, *CONCUR*, LNCS 2154, pages 39–58. Springer, 2001.
24. G. J. Holzmann. The model checker Spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
25. W. Ifill, S. A. Schneider, and H. Treharne. Augmenting B with control annotations. In *Proceedings B'07*, pages 34–48, 2007.
26. C. N. Ip and D. L. Dill. Better verification through symmetry. In *Computer Hardware Description Languages and their Applications*, pages 97–111, 1993.
27. F. Laroussinie and P. Schnoebelen. A hierarchy of temporal logics with past. *Theor. Comput. Sci.*, 148(2):303–324, 1995.
28. M. Leuschel. The high road to formal validation:. In E. Börger, M. Butler, J. P. Bowen, and P. Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 4–23. Springer, 2008.
29. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
30. M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.
31. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Proceedings LOPSTR'99*, LNCS 1817, pages 63–82, Venice, Italy, 2000.
32. M. Leuschel and T. Massart. Efficient approximate verification of B via symmetry markers. In *Proceedings International Symmetry Conference*, pages 71–85, Edinburgh, UK, January 2007.
33. M. Leuschel, T. Massart, and A. Currie. How to make FDR spin: LTL model checking of CSP by refinement. In J. N. Oliviera and P. Zave, editors, *FME'2001*, LNCS 2021, pages 99–118, Berlin, Germany, March 2001. Springer-Verlag.
34. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings POPL '85*, pages 97–107, New York, NY, USA, 1985. ACM Press.
35. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model checking. In J. Lloyd, editor, *Proceedings of the International Conference on Computational Logic (CL'2000)*, LNAI 1861, pages 384–398, London, UK, 2000. Springer-Verlag.
36. B. Parreaux. *Vérification de systèmes d'événements B par model-checking PLTL*. Thèse de Doctorat, LIFC, Université de Franche-Comté, 08 Décembre 2000.
37. D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Proceedings IFM 2007*, LNCS 4591, pages 480–500. Springer-Verlag, 2007.
38. R. L. Pokorny and C. R. Ramakrishnan. Model checking linear temporal logic using tabled logic programming. In *Proceedings Tabling in Parsing and Deduction TAPD 2000*, Vigo, Spain, September 2000.
39. G. Pouzancre. How to diagnose a modern car with a formal B model?. In D. Bert, J. P. Bowen, S. King, and M. A. Waldén, editors, *ZB'2003*, LNCS 2651, pages 98–100. Springer, 2003.
40. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *Proceedings CAV'97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.
41. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1999.
42. A. W. Roscoe. On the expressive power of CSP refinement. *Formal Asp. Comput.*, 17(2):93–112, 2005.
43. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453, Minneapolis, Minnesota, May 1994. ACM.
44. S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 1999.
45. A. P. Sistla, V. Gyuris, and E. A. Emerson. Smc: a symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000.
46. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
47. H. Treharne and S. Schneider. How to drive a B machine. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB'2000*, LNCS 1878, pages 188–208. Springer, 2000.
48. E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, pages 25–34, Shanghai, China, June 2007. IEEE.
49. M. Y. Vardi. Branching vs. linear time: Final showdown. In T. Margaria and W. Yi, editors, *TACAS'01*, LNCS 2031, pages 1–22. Springer, 2001.
50. B. D. Wachter, A. Genon, T. Massart, and C. Meuter. The formal design of distributed controllers with $_{\rm d}$sl and Spin. *Formal Asp. Comput.*, 17(2):177–200, 2005.