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

Michael Leuschel and Daniel Plagge

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

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 deadlocking 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.
Keywords: Validation and Verification, Notations and Languages, LTL, model checking, B-method, CSP, Z, Integrated Methods, symmetry reduction.

1 Introduction and Motivation

The B-Method and Z are used in railway systems (Dollé et al., 2003), the automotive sector (Pouzancre, 2003), as well as avionics (Hall, 1996). 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 (Essamé and Dollé, 2007), in order to ensure that the correct system is built. The PROB tool set can be used to animate B (Leuschel and Butler, 2003) as well as Z specifications (Plagge and Leuschel, 2007). 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[1], 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

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

- 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 (Clarke et al., 1999), and is now considered to be more expressive, intuitive and practically useful than CTL (see, e.g., Vardi, 2001). Despite an apparent complexity problem (model checking LTL is 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 (Holzmann, 1997). 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 (Abrial, 1996)
  There has also been considerable interest in trying to verify temporal properties for B specifications. In Barradas and Bert (2002) proof obligations are defined for liveness properties in B. A way to reason about temporal properties of B systems is described in Bert et al. (2005) amongst others, e.g., checking properties about when operations are enabled. The work in Groslambert (2007b) and the associated JAG tool (Groslambert, 2007a) 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. Another approach is the work done in Parreaux (2000), Bellegarde et al. (2002) and Chouali et al. (2005). This work is actually very much in the spirit of our work; however, we were not able to download a version of the system, nor do the papers contain timing results. The system does not cover the full B language (e.g., no power set construction, no lambda abstractions nor set comprehensions are supported). Also, these works support standard LTL (albeit with fairness constraints).
  Finally, the model checker PROB (Leuschel and Butler, 2003) 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. Treharne and Schneider (2000) and Butler and Leuschel (2005)).
- CSP (Roscoe, 1999)
  This formalism is supported by the tool FDR (Formal Systems (Europe) Ltd). 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., Roscoe (2005) and Derrick and Smith (2004)) and we ourselves have even proposed a way to perform LTL model checking for CSP using FDR in Leuschel et al. (2001), 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 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 LTL$^{[e]}$ model checking algorithm inside PROB, which can

- deal with deadlocking 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 LTL$^{[e]}$ patterns, as well as empirical results. We also briefly present an extension to allow Past LTL operators (Laroussinie and Schnoebelen, 1995).

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., Hatcliff and Dwyer (2001) or Wachter et al. (2005)). 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 an interpreter in Prolog with constraint solving), and it is furthermore very difficult to avoid additional state space explosion due to the smaller granularity of Promela (also due to limitations of `dstep` and `atomic`).[2] 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 5, are not supported by SPIN. Still, we plan to reevaluate this approach in the future.

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

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

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 a label for each transition in the relation of the Kripke structure. LTL with support for labelled transitions can also be found in Chaki et al. (2005), but the definition there is limited to infinite paths.

Definition 1. A *labelled Kripke structure* M with possible deadlocks over atomic propositions AP and transition labels T 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 $R \subseteq S \times T \times S$, and a labeling function $L \in S \to 2^{AP}$. 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 $deadlocks = \{s \in S | \neg \exists t, s' : (s, t, s') \in R\}$.

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 $s_{k-1} \in deadlocks$ and $\forall i : 0 \leq i < k-1 \Rightarrow (s_i, t_i, s_{i+1}) \in R$.
- Infinite paths have the form $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots$, $\forall i \geq 0 : (s_i, t_i, s_{i+1}) \in R$. 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 certain operation t will be executed next 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 $M, \pi \models \varphi$) is defined by:

$$
\begin{array}{lll}
M, \pi \models & true & \\
M, \pi \models & p & \Leftrightarrow \quad p \in L(p) \text{ for atomic propositions } p \in AP \\
M, \pi \models & \neg\varphi & \Leftrightarrow \quad M, \pi \not\models \varphi \\
M, \pi \models & \varphi \vee \psi & \Leftrightarrow \quad M, \pi \models \varphi \text{ or } M, \pi \models \psi \\
M, \pi \models & X\varphi & \Leftrightarrow \quad |\pi| \geq 2 \text{ and } M, \pi^1 \models \varphi \\
M, \pi \models & \varphi U \psi & \Leftrightarrow \quad \exists k < |\pi| : M, \pi^k \models \psi \text{ and } \forall i : 0 \leq i < k \Rightarrow M, \pi^i \models \varphi \\
M, \pi \models & [t] & \Leftrightarrow \quad |\pi| \geq 2 \text{ and } \pi = s_0 \xrightarrow{t} s_1 \ldots \text{ for transition labels } t \in T
\end{array}
$$

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:

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

LTL$^{[e]}$: Syntax of atomic propositions We provide two types of atomic propositions $p \in AP$ 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.

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

- quasi-deadlock `G (e(O1) | ... | e(On))` where `O1`,..,`On` are the real state-changing operations (as opposed to query operations).
- operation post-condition `G ( [Op] => X {Post} )`
  (`[Op]` tests if the next executed operation is `Op`)
- operation pre-condition `G ( e(Op) => {Pre} )`.

Later, in Section 5, 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.

4 The Model Checking Algorithm

Below we adapt the LTL model checking algorithm from Lichtenstein and Pnueli (1985) and Clarke et al. (1999). 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 `[Op]` construct 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 6). Hence, the bottleneck is generally not to be found inside the LTL model checking algorithm.
- The algorithm can also later be extended to CTL* (Clarke et al., 1999).

We implemented the algorithm in C, using SICStus Prolog's C-Interface to integrate it into the PROB tool.

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 Lichtenstein and Pnueli (1985) and Clarke et al. (1999). We adapted the algorithm so that deadlock states and transition labels 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 atoms of the initial state where φ is true. 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.

For the interested reader we provide our algorithm in full detail below. The proof of the algorithm's correctness can be found in the technical report (Leuschel and Plagge, 2007).

4.2 Some Details of the Algorithm

Below we use the closure definition from Clarke et al. (1999). Informally, the closure $Cl(\varphi)$ contains all formulas that determine if φ is true. These are all subformulas of φ and their negations. Additionally, $X\psi \in Cl(\varphi)$ implies $X\neg\psi \in Cl(\varphi)$ and $\psi_1 U \psi_2 \in Cl(\varphi)$ implies $X(\psi_1 U \psi_2) \in Cl(\varphi)$.

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 t is not enabled in s then $(\neg[t]) \in F$ for transition labels $t \in T$.
- If $[t] \in F$ then $[t'] \notin F$ for all transition labels $t' \neq t$, $t, t' \in Cl(\varphi)$.

Our changes to the original rules are as follows: We added the two last rules for transition labels $[t]$ 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 labels $[t]$ are in F or not. There is at most one transition label $[t]$ in F. 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 $|A(s)| \leq 2^{N_x} \cdot (N_t + 1)$, where N_x is the number of next operators in $Cl(\varphi)$ and N_t the number of transition labels in $Cl(\varphi)$. The number of atoms grows exponentially with the number of next operators in $Cl(\varphi)$ and is linear with the number of transition labels in $Cl(\varphi)$.

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

- t is a transition in M from s_1 to s_2 ($(s_1, t, s_2) \in R$), and
- $X\psi \in F_1 \Leftrightarrow \psi \in F_2$ for all formulas $X\psi \in Cl(\varphi)$, and
- $t' \in F_1 \Leftrightarrow t' = t$ for all transition labels $t' \in Cl(\varphi)$

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 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 (Tarjan, 1972) 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)

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

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 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 $\pi = s_0 \xrightarrow{t_0} \ldots s_d$.

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.

5 Some Examples and Experiments

In 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.

5.1 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 Leuschel and Butler, 2003). 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; 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; the model checking time is 0.20 seconds.

We have also applied our tool to the (very) small robot specification from Groslambert (2007a). The original LTL formula `G(({Dt=TRUE}&X{Dt=FALSE})=>{De=FALSE})` from Groslambert (2007a) 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[3] specification from Chouali et al. (2005). Computing the state space took 0.02 seconds (for 15 nodes). We tested the formula P_1 = `G({CardF2=bl} => F{CardF2=lb})` from Chouali et al. (2005). 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 Chouali et al. (2005) also takes fairness constraints into account. These fairness constraints are written in Chouali et al. (2005) 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 (computing 98,304 atoms). 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.

5.2 CSP Examples: Peterson and Train Level-Crossing

First we tried a standard CSP example from the book web page of Schneider (1999)[4], Peterson's Algorithm version 1. Computing the state space, consisting of 58 nodes and 115 transitions, took 0.32 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 0.02 sec.
- `G ([p1critical] => X((!e(p2critical)) W [p1leave] ))`
  The formula is valid; the model checking time is 0.19 sec.

Another example we tested is `crossing.csp` also from Schneider (1999). 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 95.55 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 0.39 seconds.

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

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

– `G F [enter]` A counter-example (of length 554) was found after 0.36 seconds.

5.3 CSP || B Examples: Control Annotations and Philosophers

In Ifill et al. (2007) 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 (and now automatically) by our LTL model checker. For example, for the traffic light controller from Ifill et al. (2007), 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 Ifill et al. (2007) 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 Leuschel and Massart (2007) instantiated for three philosophers and three forks, using symmetry reduction (cf. Sect. 6), and where the protocol is specified by a CSP Controller.

5.4 Z Examples: SAL Example and Workstation Protocol

In Plagge and Leuschel (2007), PROB was extended to deal with Z specifications. We examined the example from Derrick et al. (2006), 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 Derrick et al. (2006) (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) | e(JoinQ) | e(Remove))`

Note that this error was not uncovered in Derrick et al. (2006). We have also tested our tool on the workstation protocol industrial case study from Plagge and Leuschel (2007). Computation of the state space for 2 workstations took 2.49 seconds, resulting in 68 states. The formula `G([Transfer]=>X(e(ReadRequestOK) | e(ReadResponse)))` was checked in 0.04 seconds, using 421 atoms.

5.5 Other Formalisms: StAC, Object Petri nets, dSL

PROB has also the ability to load specifications via custom Prolog interpreters following the style of Leuschel and Massart (2000), describing the initial states, the properties and the transition relation 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), Object Petri Nets, dSL (Wachter et al., 2005).

6 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.

It turns out that our LTL$^{[e]}$ language is the ideal companion to the existing symmetry reduction techniques developed for PROB (Leuschel et al., 2007; Leuschel and Massart, 2007; Turner et al., 2007), 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.

In Leuschel and Plagge (2007) we give the proof that, given an LTL$^{[e]}$ formula, there exists a counter-example for a B machine iff there exists one with symmetry reduction.

7 Future Work, Discussion and Conclusion

Concerning the expressivity of LTL$^{[e]}$, we would like to allow patterns to be used inside the `[.]` and `e(.)` constructs. E.g., one may wish to be able to check temporal formula such as `G ([sends!2] => X e(receives!2))` or even `G ([sends!x] => X e(receives!x))`. We have already implemented another extension of LTL$^{[e]}$, which allows reasoning about the past of a state, using, e.g., the Y (yesterday), S (since), O (once) Past-LTL operators. For the Volvo vehicle function, it was thus possible to validate the following property in 0.21 seconds: `G(e(FunctionOff) => YO[SetFunctionParameter])`, i.e., when the vehicle function can be turned off it must have been activated in the past.

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 Chouali et al. (2005), the NEXT control annotations from Ifill et al. (2007), 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

Abrial, J.-R. (1996). *The B-Book*. Cambridge University Press.

Barradas, H. R. and D. Bert (2002). Specification and proof of liveness properties under fairness assumptions in B event systems. In M. J. Butler, L. Petre, and K. Sere (Eds.), *IFM*, LNCS 2335, pp. 360–379. Springer.

Bellegarde, F., S. Chouali, and J. Julliand (2002). Verification of dynamic constraints for B event systems under fairness assumptions. In *ZB'2002*, LNCS 2272, pp. 477–496.

Bert, D., M.-L. Potet, and N. Stouls (2005). 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 (Eds.), *ZB 2005*, LNCS 3455, pp. 299–318. Springer.

Butler, M. and M. Leuschel (2005). Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, Newcastle upon Tyne, pp. 221–236. Springer-Verlag.

Chaki, S., E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha (2005). Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing V17*(4), 461–483.

Chouali, S., J. Julliand, P.-A. Masson, and F. Bellegarde (2005). Pltl-partitioned model checking for reactive systems under fairness assumptions. *ACM Trans. Embedded Comput. Syst. 4*(2), 267–301.

Clarke, E. M., O. Grumberg, and D. Peled (1999). *Model Checking*. MIT Press.

Derrick, J., S. North, and T. Simons (2006). Issues in implementing a model checker for Z. In Z. Liu and J. He (Eds.), *ICFEM*, LNCS 4260, pp. 678–696. Springer.

Derrick, J. and G. Smith (2004). Linear temporal logic and Z refinement. In C. Rattray, S. Maharaj, and C. Shankland (Eds.), *AMAST 04*, LNCS 3116, pp. 117–131. Springer.

Dollé, D., D. Essamé, and J. Falampin (2003). B dans le tranport ferroviaire. L'expérience de Siemens Transportation Systems. *Technique et Science Informatiques 22*(1), 11–32.

Essamé, D. and D. Dollé (2007). B in large-scale projects: The Canarsie line CBTC experience. In *Proceedings of the 7th International B Conference (B2007)*, LNCS 4355, Besancon, France, pp. 252–254. Springer-Verlag.

Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual*.

Groslambert, J. (2007a). A jag extension for verifying LTL properties on B event systems. In *Proceedings B'07*, pp. 262–265.

Groslambert, J. (2007b). Verification of LTL on B event systems. In *Proceedings B'07*, pp. 109–124.

Hall, A. (1996). Using formal methods to develop an atc information system. *IEEE Software*, 66–76. Reprinted in Industrial-Strength Formal Methods in Practice, M.G. Hinchey & J.P. Bowen, Springer, 1999.

Hatcliff, J. and M. B. Dwyer (2001). Using the bandera tool set to model-check properties of concurrent java software. In K. G. Larsen and M. Nielsen (Eds.), *CONCUR*, LNCS 2154, pp. 39–58. Springer.

Holzmann, G. J. (1997). The model checker Spin. *IEEE Trans. Software Eng. 23*(5), 279–295.

Ifill, W., S. A. Schneider, and H. Treharne (2007). Augmenting B with control annotations. In *Proceedings B'07*, pp. 34–48.

Laroussinie, F. and P. Schnoebelen (1995). A hierarchy of temporal logics with past. *Theor. Comput. Sci. 148*(2), 303–324.

Leuschel, M. and M. Butler (2003). ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli (Eds.), *FME 2003: Formal Methods*, LNCS 2805, pp. 855–874. Springer-Verlag.

Leuschel, M., M. Butler, C. Spermann, and E. Turner (2007). Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, Besancon, France, pp. 79–93. Springer-Verlag.

Leuschel, M. and T. Massart (2000). Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi (Ed.), *Proceedings LOPSTR'99*, LNCS 1817, Venice, Italy, pp. 63–82.

Leuschel, M. and T. Massart (2007). Efficient approximate verification of B via symmetry markers. *Proceedings International Symmetry Conference*, 71–85.

Leuschel, M., T. Massart, and A. Currie (2001). How to make FDR spin: LTL model checking of CSP by refinement. In J. N. Oliviera and P. Zave (Eds.), *FME'2001*, LNCS 2021, Berlin, Germany, pp. 99–118. Springer-Verlag.

Leuschel, M. and D. Plagge (2007). Seven at one stroke: LTL model checking for High-level Specifications in B, Z, CSP, and more. Technical Report STUPS/2007/02, Lehrstuhl für Softwaretechnik und Programmiersprachen, Institut für Informatik, Heinrich-Heine-Universität Düsseldorf. `http://www.stups.uni-duesseldorf.de/publications.php`.

Lichtenstein, O. and A. Pnueli (1985). Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings POPL '85*, New York, NY, USA, pp. 97–107. ACM Press.

Parreaux, B. (2000). *Vérification de systèmes d'événements B par model-checking PLTL*. Thèse de Doctorat, LIFC, Université de Franche-Comté.

Plagge, D. and M. Leuschel (2007). Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons (Eds.), *Proceedings IFM 2007*, LNCS 4591, pp. 480–500. Springer-Verlag.

Pouzancre, G. (2003). How to diagnose a modern car with a formal B model?. In D. Bert, J. P. Bowen, S. King, and M. A. Waldén (Eds.), *ZB'2003*, LNCS 2651, pp. 98–100. Springer.

Roscoe, A. W. (1999). *The Theory and Practice of Concurrency*. Prentice-Hall.

Roscoe, A. W. (2005). On the expressive power of CSP refinement. *Formal Asp. Comput. 17*(2), 93–112.

Schneider, S. (1999). *Concurrent and Real-time Systems: The CSP Approach*. Wiley.

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM J. Comput. 1*(2), 146–160.

Treharne, H. and S. Schneider (2000). How to drive a B machine. In J. P. Bowen, S. Dunne, A. Galloway, and S. King (Eds.), *ZB'2000*, LNCS 1878, pp. 188–208. Springer.

Turner, E., M. Leuschel, C. Spermann, and M. Butler (2007). Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, Shanghai, China, pp. 25–34. IEEE.

Vardi, M. Y. (2001). Branching vs. linear time: Final showdown. In T. Margaria and W. Yi (Eds.), *TACAS'01*, LNCS 2031, pp. 1–22. Springer.

Wachter, B. D., A. Genon, T. Massart, and C. Meuter (2005). The formal design of distributed controllers with $_{\mathrm{d}}$sl and Spin. *Formal Asp. Comput. 17*(2), 177–200.