Translating B to TLA$^+$ for Validation with TLC

Dominik Hansen and Michael Leuschel

Institut für Informatik, Universität Düsseldorf**
Universitätsstr. 1, D-40225 Düsseldorf
{hansen, leuschel}@cs.uni-duesseldorf.de

Abstract. The state-based formal methods B and TLA$^+$ share the common base of predicate logic, arithmetic and set theory. However, there are still considerable differences, such as the way to specify state transitions, the different approaches to typing, and the available tool support. In this paper, we present a translation from B to TLA$^+$ to validate B specifications using the model checker TLC. We provide translation rules for almost all constructs of B, in particular for those which are not built-in in TLA$^+$. The translation also includes many adaptations and optimizations to allow efficient checking by TLC. Moreover, we present a way to validate liveness properties for B specifications under fairness conditions. Our implemented translator, Tlc4B, automatically translates a B specification to TLA$^+$, invokes the model checker TLC, and translates the results back to B. We use ProB to double check the counter examples produced by TLC and replay them in the ProB animator. We also present a series of case studies and benchmark tests comparing Tlc4B and ProB.
Keywords: TLA$^+$, B-Method, Tool Support, Model Checking, Animation.

1 Introduction and Motivation

B [1] and TLA$^+$ [8] are both state-based formal methods rooted in predicate logic, combined with arithmetic, set theory and support for mathematical functions. However, as already pointed out in [5], there are considerable differences:

- B is strongly typed, while TLA$^+$ is untyped. For the translation, it is obviously easier to translate from a typed to an untyped language than vice versa.
- The concepts of modularization are quite different.
- Functions in TLA$^+$ are total, while B supports relations, partial functions, injections, bijections, etc.
- B is limited to invariance properties, while TLA$^+$ also allows the specification of liveness properties.
- The structure of a B machine or development is prescribed by the B-method, while in a TLA$^+$ specification any formula can be considered as a system specification.

** Part of this research has been sponsored by the EU funded FP7 project 287563 (ADVANCE) and the DFG funded project Gepavas.

As far as tool support is concerned, TLA$^+$ is supported by the explicit state model checker TLC [13] and more recently by the TLAPS prover [3]. TLC has been used to validate a variety of distributed algorithms (e.g., [4]) and protocols. B has extensive proof support, e.g., in the form of the commercial product AtelierB [2] and the animator, constraint solver and model checker ProB [9, 10]. Both AtelierB and ProB are being used by companies, mainly in the railway sector for safety critical control software. In an earlier work [5] we have presented a translation from TLA$^+$ to B, which enabled applying the ProB tool to TLA$^+$ specifications. In this paper we present a translation from B to TLA$^+$, this time with the main goal of applying the model checker TLC to B specifications. Indeed, TLC is a very efficient model checker for TLA$^+$ with an efficient disk-based algorithm and support for fairness. ProB has an LTL model checker, but it does not support fairness (yet) and is entirely RAM-based. The model checking core of ProB is less tuned than TLA$^+$. On the other hand, ProB incorporates a constraint solver and offers several features which are absent from TLC, in particular an interactive animator with various visualization options. One feature of our approach is to replay the counter-examples produced by TLC within ProB, to get access to those features but also to validate the correctness of our translation. In this paper, we also present a thorough empirical evaluation between TLC and ProB. The results show that for lower-level, more explicit formal models, TLC fares better, while for certain more high-level formal models ProB is superior to TLC because of its constraint solving capabilities. The addition of a lower-level model checker thus opens up many new application possibilities.

2 Translation

The complete translation process from B to TLA$^+$ and back to B is illustrated in Fig. 1.

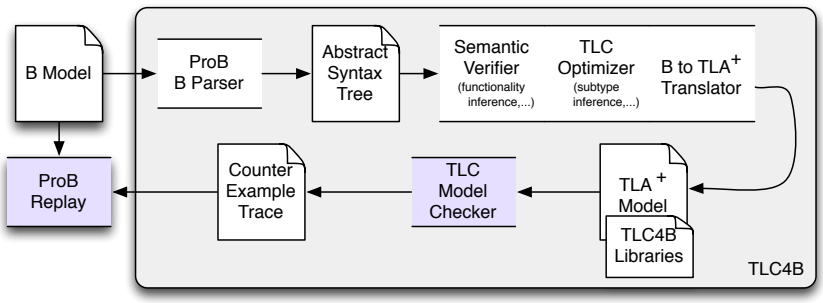

Fig. 1. The TLC4B Translation and Validation Process

Before explaining the individual phases, we will illustrate the translation with an example and explain the various phases based on that example. Translation rules of all data types and operators can be found in the extended version of our paper [6]. More specific implementation details will be covered in Section 4.

2.1 Example

Below we use a specification (adapted from [10]) of a process scheduler (Fig. 2). The specified system allows at most one process to be active. Each process can qualify for being selected by the scheduler by entering a FIFO queue. The specification contains two variables: a partial function *state* mapping each process to its state (a process must be created before it has a state) and a FIFO queue modeled as an injective sequence of processes. In the initial state no process is created and the queue is empty. The specification contains various operations to create (*new*), delete (*del*), or add a process to queue (*addToQueue*). Additionally, there are two operations to load a process into the processor (*enter*) and to remove a process from the processor (*leave*). The specification contains two safety conditions beside the typing predicates of the variables:

- At most one process should be active.
- Each process in the FIFO queue should have the state *ready*.

The translated TLA$^+$- specification is shown in Fig. 3. At the beginning of the module some standard modules are loaded via the *EXTENDS* statement. These modules contain several operators used in this specification. The enumerated set *STATE* is translated as a constant definition. The definition itself is renamed (into *STATE_1*) by the translator because *STATE* is keyword in TLA$^+$. The invariant of the B specification is divided into several definitions in the TLA$^+$ module. This enables TLC to provide better feedback about which part of the invariant is violated. We translate each B operation as a TLA$^+$ action. Substitutions are translated as before-after predicates where a primed variable represents the variable in the next state. Unchanged variables must be explicitly specified. Note that a parameterized operation is translated as existential quantification. The quantification itself is located in the next-state relation *Next*, which is a disjunction of all actions. Some of the operations' guards appear in the *Next* definition rather than in the corresponding action. This is an instance of our translator optimizing the translation for the interpretation with TLC. The whole TLA$^+$ specification is described by the *Spec* definition. A valid behavior for the system has to satisfy the *Init* predicate for the initial state and then each step of the system must satisfy the next-state relation *Next*.

To validate the translated TLA$^+$ specification with TLC we have to provide an additional configuration file (Fig 4) telling TLC the main (specification) definition and the invariant definitions. Moreover, we have to assign values to all con-

```
MODEL Scheduler
SETS
  PROCESSES;
  STATE = {idle, ready, active}
VARIABLES
  state,
  queue
INVARIANT
  state  ∈  PROCESSES ⇸ STATE
  & queue  ∈  iseq(PROCESSES)
  & card(state^-1[{active}]) ≤ 1
  & !x.(x  ∈  ran(queue) ⇒ state(x) = ready)
INITIALISATION
  state := {} || queue := [ ]
OPERATIONS
  new(p) = SELECT p  ∉  dom(state)
         THEN state := state ∪ {(p ↦ idle)} END;

  del(p) = SELECT p  ∈  dom(state) ∧ state(p) = idle
         THEN state := {p} ⩤ state END;

  addToQueue(p) = SELECT p  ∈  dom(state) ∧ state(p) = idle
         THEN state(p) := ready || queue := queue ← p END;

  enter = SELECT queue ≠ [ ] ∧ state^-1[{active}] = {}
         THEN state(first(queue)) := active || queue := tail(queue) END;

  leave(p) = SELECT p  ∈  dom(state) ∧ state(p) = active
         THEN state(p) := idle END
END
```

Fig. 2. MODEL Scheduler

stants of the module.[1] In this case we assign a set[2] of model values to the constant *PROCESSES* and single model values to the other constants. In terms of functionality, model values correspond to elements of a enumerated set in B. Model values are equal to themselves and unequal to all other model values.

[1] We translate a constant as variable if the axioms allow several solutions for this constant. All possible solution values will be enumerated in the initialization and the variable will be kept unchanged by all actions.

[2] The size of the set is a default number or can be specified by the user.

MODULE *Scheduler*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *Relations*, *Functions*,
FunctionsAsRelations, *SequencesExtended*, *SequencesAsRelations*

CONSTANTS *PROCESSES*, *idle*, *ready*, *active*

VARIABLES *state*, *queue*

$STATE\_1 \triangleq \{idle, ready, active\}$

$Invariant1 \triangleq state \in RelParFuncEleOf(PROCESSES, STATE\_1)$
$Invariant2 \triangleq queue \in ISeqEleOf(PROCESSES)$
$Invariant3 \triangleq Cardinality(RelImage(RelInverse(state), \{active\})) \leq 1$
$Invariant4 \triangleq \forall x \in Range(queue) : RelCall(state, x) = ready$

$$
\begin{aligned}
Init \triangleq\ & \land state = \{\} \\
& \land queue = \langle\rangle
\end{aligned}
$$

$$
\begin{aligned}
new(p) \triangleq\ & \land state' = state \cup \{\langle p, idle\rangle\} \\
& \land \text{UNCHANGED } \langle queue\rangle
\end{aligned}
$$

$$
\begin{aligned}
del(p) \triangleq\ & \land RelCall(state, p) = idle \\
& \land state' = RelDomSub(\{p\}, state) \\
& \land \text{UNCHANGED } \langle queue\rangle
\end{aligned}
$$

$$
\begin{aligned}
addToQueue(p) \triangleq\ & \land RelCall(state, p) = idle \\
& \land state' = RelOverride(state, \{\langle p, ready\rangle\}) \\
& \land queue' = Append(queue, p)
\end{aligned}
$$

$$
\begin{aligned}
enter \triangleq\ & \land queue \neq \langle\rangle \\
& \land RelImage(RelInverse(state), \{active\}) = \{\} \\
& \land state' = RelOverride(state, \{\langle Head(queue), active\rangle\}) \\
& \land queue' = Tail(queue)
\end{aligned}
$$

$$
\begin{aligned}
leave(p) \triangleq\ & \land RelCall(state, p) = active \\
& \land state' = RelOverride(state, \{\langle p, idle\rangle\}) \\
& \land \text{UNCHANGED } \langle queue\rangle
\end{aligned}
$$

$$
\begin{aligned}
Next \triangleq\ & \lor \exists p \in PROCESSES \setminus RelDomain(state) : new(p) \\
& \lor \exists p \in RelDomain(state) : del(p) \\
& \lor \exists p \in RelDomain(state) : addToQueue(p) \\
& \lor enter \\
& \lor \exists p \in RelDomain(state) : leave(p)
\end{aligned}
$$

$vars \triangleq \langle state, queue\rangle$

$Spec \triangleq Init \land \Box[Next]_{vars}$

Fig. 3. Module Scheduler

```
SPECIFICATION Spec
INVARIANT Invariant1, Invariant2, Invariant3, Invariant4
CONSTANTS
PROCESSES = {PROCESSES1, PROCESSES2, PROCESSES3}
idle = idle
ready = ready
active = active
```

Fig. 4. Configuration file for module Scheduler

2.2 Translating Data Values and Functionality Inference

Due to the common base of B and TLA$^+$, most data types exist in both languages, such as sets, functions and numbers. As a consequence, the translation of these data types is almost straightforward.

TLA$^+$ has no built-in concept of Relations[3], but TLA$^+$ provides all necessary data types to define relations based on the model of the B-Method. We represent a relation in TLA$^+$ as a set of tuples (e.g. $\{\langle 1, \mathit{TRUE}\rangle, \langle 1, \mathit{FALSE}\rangle \langle 2, \mathit{TRUE}\rangle\}$). The drawback of this approach is that in contrast to B, TLA$^+$'s own functions and sequences are not based on the relations defined is this way. As an example, we cannot specify a TLA$^+$ built-in function as a set of pairs; in B it is usual to do this as well as to apply set operators (e.g. the union operator as in $f \cup \{2 \mapsto 3\}$) to functions or sequences. To support such a functionality in TLA$^+$, functions and sequences should be translated as relations if they are used in a "relational way". It would be possible to always translate functions and sequences as relations. But in contrast to relations, functions and sequences are built-in data types in TLA$^+$ and their evaluation is optimized by TLC (e.g. lazy evaluation). Hence we extended the B type-system to distinguish between functions and relations. Thus we are able to translate all kinds of relations and to deliver an optimized translation.

We use a type inference algorithm adapted to the extended B type-system to get the required type information for the translation. Unifying a function type with a relation type will result in a relation type (e.g. $\mathbb{P}(\mathbb{Z} \times \mathbb{Z})$ for both sides of the equation $\lambda x.(x \in 1..3|x + 1) = \{(1, 1)\}$). However, there are several relational operators preserving a function type if they are applied to operands with a function type (e.g. *ran*, *front* or *tail*). For these operators we have to deliver two translation rules (functional vs relational). Moreover the algorithm verifies the type correctness of the B specification (i.e. only values of the same type can be compared with each other).

2.3 Translating Operators

In TLA$^+$ some common operators such as arithmetic operators are not built-in operators. They are defined in separate modules called standard modules that can

[3] Relations are not mentioned in the language description of [8]. In [7] Lamport introduces relations in TLA$^+$ only to define the transitive closure.

be extended by a specification.[4] We reuse the concept of standard modules to include the relevant B operators. Due to the lack of relations in TLA$^+$ we have to provide a module containing all relational operators (Fig. 5).

MODULE *Relations*

EXTENDS *FiniteSets*, *Naturals*, *TLC*

$Relation(X,\ Y) \triangleq \text{SUBSET } (X \times Y)$

$RelDomain(R) \triangleq \{x[1] : x \in R\}$

$RelRange(R) \triangleq \{x[2] : x \in R\}$

$RelInverse(R) \triangleq \{\langle x[2],\ x[1]\rangle : x \in R\}$

$RelDomRes(S,\ R) \triangleq \{x \in R : x[1] \in S\}$ Domain restriction

$RelDomSub(S,\ R) \triangleq \{x \in R : x[1] \notin S\}$ Domain subtraction

$RelRanRes(R,\ S) \triangleq \{x \in R : x[2] \in S\}$ Range restriction

$RelRanSub(R,\ S) \triangleq \{x \in R : x[2] \notin S\}$ Range subtraction

$RelImage(R,\ S) \triangleq \{y[2] : y \in \{x \in R : x[1] \in S\}\}$

$RelOverride(R1,\ R2) \triangleq \{x\ \in R : x[1] \notin RelDomain(R2)\} \cup R2$

$RelComposition(R1,\ R2) \triangleq \{\langle u[1][1],\ u[2][2]\rangle : u \in$
$\{x \in RelRanRes(R1,\ RelDomain(R2)) \times RelDomRes(RelRange(R1),\ R2) :$
$x[1][2] = x[2][1]\}\}$

$\vdots$

Fig. 5. An extract of the Module Relations

Moreover B provides a rich set of operators on functions such as all combinations of partial/total and injective/surjective/bijective. In TLA$^+$ we only have total functions. We group all operators on functions together in an additional module (Fig. 6). Sometimes there are several ways to define an operator. We choose the definition which can be best handled by TLC.[5]

Some operators exists in both languages but their definitions differ slightly. For example, the B-Method requires that the first operand for the modulo operator must be a natural number. In TLA$^+$ it can be also a negative number.

Operator	B-Method	TLA$^+$
a *modulo* b	$a \in \mathbb{N} \land b \in \mathbb{N}_1$	$a \in \mathbb{Z} \land b \in \mathbb{N}_1$

To verify B's well-definedness condition for modulo we use TLC's ability to check assertions. The special operator $Assert(P, out)$ throws a runtime exception with the error message *out* if the predicate P is false. Otherwise, *Assert* will be evaluated to TRUE. The B modulo operator can thus be expressed in TLA$^+$ as

[4] TLC supports operators of the common standard modules Integers and Sequences in a efficient way by overwriting them with Java methods.

[5] Note that some of the definitions are based on the *Cardinality* operator that is restricted to finite sets.

MODULE *Functions*

EXTENDS *FiniteSets*

$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$

$Image(f, S) \triangleq \{f[x] : x \in S\}$

$TotalInjFunc(S, T) \triangleq \{f \in [S \to T] :$
$\quad Cardinality(\text{DOMAIN } f) = Cardinality(Range(f))\}$

$ParFunc(S, T) \triangleq \text{UNION } \{[x \to T] : x \in \text{SUBSET } S\}$

$ParInjFunc(S, T) \triangleq \{f \in ParFunc(S, T) :$
$\quad Cardinality(\text{DOMAIN } f) = Cardinality(Range(f))\}$

$\vdots$

Fig. 6. An extract of the Module Functions

follows:

$$Modulo(a, b) \triangleq \text{IF } a \geq 0 \text{ THEN } a \,\%\, b \text{ ELSE } Assert(\text{FALSE}, \text{“WD ERROR”})$$

We also have to consider well-definedness conditions if we apply a function call to a relation:

$$\begin{aligned} RelCall(r, x) \triangleq\ & \text{IF } Cardinality(r) = Cardinality(RelDom(r)) \wedge x \in RelDom(r) \\ & \text{THEN } (\text{CHOOSE } y \in r : y[1] = x)[2] \\ & \text{ELSE } Assert(\text{FALSE}, \text{“WD ERROR”}) \end{aligned}$$

In summary, we provide the following standard modules for our translation:

- Relations
- Functions
- SequencesExtended (Some operators on sequences which are not included the standard module Sequences)
- FunctionsAsRelations (Defines all function operators on sets of pairs ensuring their well-definedness conditions)
- SequencesAsRelations (Defines all operators on sequences which are represented as sets of pairs.)
- BBuiltins (Miscellaneous operators e.g. modulo, min, max, sigma or pi)

2.4 Optimizations

Subtype Inference Firstly, we will describe how TLC evaluates expressions: In general TLC evaluates an expression from left to right. Evaluating an expression containing a bound variable such as an existential quantification ($\exists x \in S : P$), TLC enumerates all values of the associated set and then substitutes them for the bound variable in the corresponding predicate. Due to missing constraint solving techniques, TLC is not able to evaluate another variant of the existential quantification without an associated set ($\exists x : P$). This version is also a valid TLA$^+$ expression and directly corresponds to the way of writing an existential quantification in B ($\exists x.(P)$). However, we confine our translations to the subset of TLA$^+$

which is supported by TLC. Thus the translation is responsible for making all required adaptations to deliver an executable TLA$^+$ specification. For the existential quantification (or all other expressions containing bound variables), we use the inferred type τ of the bound variable as the associated set ($\exists x \in \tau_x : P.$) However, in most cases, it is not performant to let TLC enumerate over a type of a variable, in particular TLC aborts if it has to enumerate a infinite set. Alternatively, it is often possible to restrict the type of the bound variable based on a static analysis of the corresponding (typing) predicate. We use a pattern matching algorithm to find the following kind of expressions where x is a bound variable, e is an expression, and S is ideally a subset of the type[6]: $x = e$, $x \in S$, $x \subseteq S$ or $x \subset S$.

If more than one of these patterns can be found for one variable, we build the intersection to keep the associated set as small as possible:

B-Method	TLA$^+$
$\exists x.(x = e \land x \in S_1 \land x \subseteq S_2 \land P)$	$\exists x \in (\{e\} \cap S_1 \cap SUBSET\ S_2) : P$

This reduces the number of times TLC has to evaluate the predicate P. [7]

Lazy Evaluation Sometimes TLC can use heuristics to evaluate an expression. For example TLC can evaluate $\langle 1, 2, 1\rangle \in Seq(\{1, 2\})$ to true without evaluating the infinite set of sequences. We will show how we can use these heuristics to generate an optimized translation. As mentioned before functions have to be translated as relations if they are used in a relational way in the B specification. How then should we translate the set of all total functions $(S \rightarrow T)$? The easiest way is to convert each function to a relation in TLA$^+$:

$$MakeRel(f) \triangleq \{\langle x, f[x]\rangle : x \in DOMAIN\ f\}$$

The resulting operator for the set of all total functions is:

$$RelTotalFunctions(S, T) \triangleq \{MakeRel(f) : f \in [S \rightarrow T]\}$$

However this definition has a disadvantage, if we just want to check if a single function is in this set the whole set will be evaluated by TLC. Using the following definition TLC avoids the evaluation of the whole set:

$$\begin{aligned} RelTotalFunctionsEleOf(S, T) \triangleq\ & \{f \in SUBSET(S \times T) : \\ & \land\ Cardinality(RelDomain(f)) = Cardinality(f) \\ & \land\ RelDomain(f) = S\} \end{aligned}$$

In this case, TLC only checks if a function is a subset of the cartesian product (the whole Cartesian product will not be evaluated) and the conditions are checked

[6] The B language description in [2] requires that each (bound) variable must be typed by one of these patterns before use.

[7] In some cases, the associated set is still infinite and the user has to restrict the set to be finite.

only once. Moreover this definition fares well even if S or T are sets of functions (e.g. $S \rightarrow V \rightarrow W$ in B). The advantage of the first definition is that it is faster when the whole set must be evaluated. As a consequence, we use both definitions for our translation and choose the first if TLC has to enumerate the set (e.g. $\exists x \in \mathit{RelTotalFunctions}(S, T) : P$) and the second testing if a function belongs to the set (e.g. $f \in \mathit{RelTotalFunctionsEleOf}(S, T)$ as an invariant).

3 Checking Temporal Formulas

One of the main advantages of TLA$^+$ is that temporal properties can be specified directly in the language itself. Moreover the model checker TLC can be used to verify such formulas. But before we show how to write temporal formulas for a B specification we first have to describe a main distinction between the two formal methods. In contrast to B, the standard template of a TLA$^+$ specification ($\mathit{Init} \wedge \Box[\mathit{Next}]_{vars}$) allows stuttering steps at any time.[8] This means that a regular step of a TLA$^+$ specification is either a step satisfying one of the actions or a stuttering step leaving all variables unchanged. When checking a specification for errors such as invariant violations it is not necessary to consider stuttering steps, because such an error will be detected in a state and stuttering steps only allow self transitions and do not add additional states. For deadlock checking stuttering steps are also not regarded by TLC, but verifying a temporal formula with TLC often ends in a counter-example caused by stuttering steps. For example, assuming we have a very simple specification of a counter in TLA$^+$ with a single variable c

$$\mathit{Spec} \triangleq c = 1 \wedge \Box[c' = c + 1]_c$$

We would expect that the counter will eventually reach 10 ($\Diamond(c = 10)$). However TLC will report a counter-example, saying that at a certain state (before reaching 10), an infinite number of stuttering steps occurs and 10 will never reached. From the B side we do not want to deal with these stuttering steps. TLA$^+$ allows to add fairness conditions to the specification to avoid infinite stuttering steps. Adding weak fairness for the next-state relation ($WF_{vars}(\mathit{Next})$) would prohibit an infinite number of stuttering steps if a step of the next-state relation is possible (i.e. *Next* is always enabled):

$$\begin{aligned} WF_{vars}(A) \triangleq & \vee \Box\Diamond(\langle A\rangle_{vars}) \\ & \vee \Box\Diamond(\neg \text{ ENABLED } \langle A\rangle_{vars}) \end{aligned}$$

However this fairness condition is too strong: It asserts that either the action A will be executed infinitely often changing the state of the system (A must not be a stuttering step)

$$\langle A\rangle_{vars} \triangleq A \wedge vars' \neq vars$$

or A will be disabled infinitely often. Assuming weak fairness for the next state relation will also eliminate user defined stuttering steps. User defined stuttering

[8] $[\mathit{Next}]_{vars} \triangleq \mathit{Next} \vee \mathit{UNCHANGED\ vars}$

steps result from B operations which do not change the state of the system (e.g. skip or call operations). These stuttering steps may cause valid counter-examples and should not be eliminated. Hence, the translation should retain user defined stuttering steps in the translated TLA$^+$ specfication and should disable stuttering steps which are implicitly included. In [12], Richards describes a way to distinguish between these two kinds of stuttering steps in TLA$^+$. We use his definition of "Very Weak Fairness" applied to the next state relation ($VWF_{vars}(Next)$) to disable implicit stuttering steps and allow user defined stuttering steps in the TLA$^+$ specification:

$$\begin{aligned} VWF_{vars}(A) \triangleq\ & \vee \Box\Diamond(\langle A\rangle_{vars}) \\ & \vee \Box\Diamond(\neg \text{ ENABLED } \langle A\rangle_{vars}) \\ & \vee \Box\Diamond(\text{ENABLED } (A \wedge \text{UNCHANGED } \ vars)) \end{aligned}$$

The definition of $VWF(A)$ is identical to $WF(A)$ except for an additional third case allowing infinite stuttering steps if A is a stuttering action ($A \wedge$ UNCHANGED $vars$). We define the resulting template of the translated TLA$^+$ specification as follows:

$$Init \wedge \Box[Next]_{vars} \wedge VWF_{vars}(Next)$$

We allow the B user to use following temporal operators to define liveness conditions for a B specification:[9]

- $\Box f$ (Globally)
- $\Diamond f$ (Finally)
- $ENABLED(op)$ (Check if the operation op is enabled)
- $\exists x.(P \wedge f)$ (Existential quantification)
- $\forall x.(P \Rightarrow f)$ (Universal quantification)
- $WF(op)$ (Weak Fairness will be translated to VWF)
- $SF(op)$ (Strong Fairness will be translated to "Almost Strong Fairness"[10])

4 Implementation & Experiments

Our translator, called Tlc4B, is implemented in Java and it took about six months to develop the initial version. Figure 1 in Section 2 shows the translation and validation process of Tlc4B. After parsing the specification Tlc4B performs some static analyses (e.g. type checking or checking the scope of the variables) verifying the semantic correctness of the B specification. Moreover, as explained in Section 2, Tlc4B extracts required information from the B specification (e.g. subtype inference) to generate an optimized translation. Subsequently, Tlc4B creates a TLA$^+$ module with an associated configuration file and invokes the model checker TLC. The results produced by TLC are translated back to B. For example, a goal predicate is translated as a negated invariant. If this invariant is violated, a "Goal found"

[9] We demonstrate the translation of a liveness condition with a concrete example in the extended version of this paper [6].

[10] Analogically Richards defines "Almost Strong Fairness" (ASF) as a weaker version of strong fairness (SF) reflecting the different kinds of stuttering steps

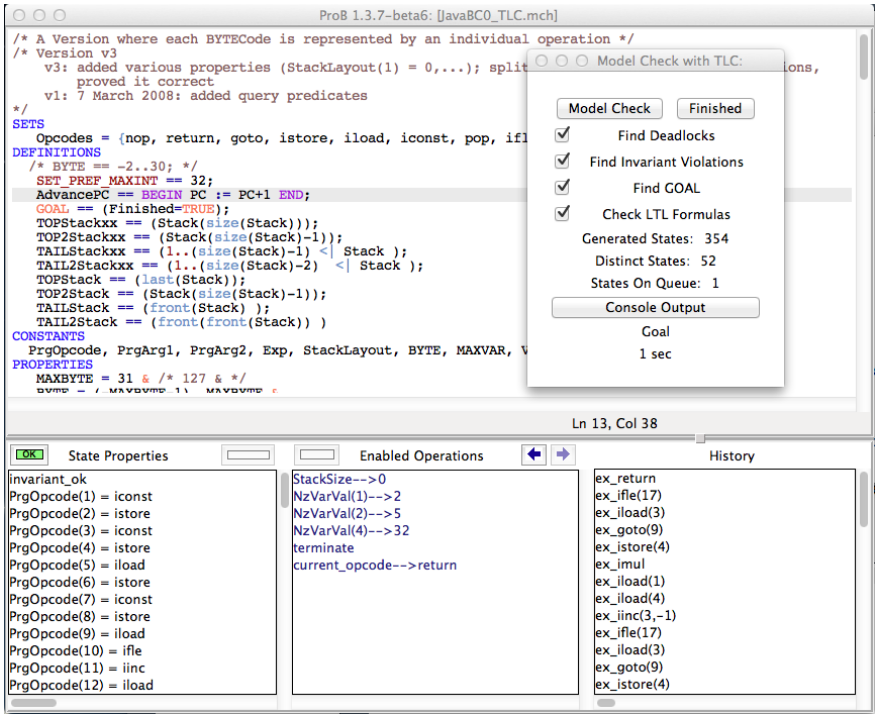

Fig. 7. ProB animator

message is reported. We expect TLC to find the following kinds of errors in the B specification:

- Deadlocks
- Invariant violations
- Assertion errors
- Violations of constants properties (i.e., axioms over the B constants are false)
- Well-definedness violations
- Violations of temporal formulas

For certain kinds of errors such as a deadlock or an invariant violation, TLC reports a trace leading to the state where the error occurs. A trace is a sequence of states where each state is a mapping from variables to values. Tlc4B translates the trace back to B (as a list of B state predicates). Tlc4B has been integrated into ProB as of version 1.3.7-beta10: The user needs no knowledge of TLA$^+$ because the translation is completely hidden. As shown in figure 7 counter-examples found by TLC are automatically replayed in the ProB animator (displayed in the history pane) to give the user an optimal feedback.

We have successfully validated several existing models from the literature (Fig. 8). The following examples show some fields of application of TLC4B. The experiments were all run on a Macbook Air with Intel Core i5 1,8 GHz processor, running TLC Version 2.05 and Prob version 1.3.7-beta9.

Can-Bus One example is a 314 line B specification of the Controller Area Network (CAN) Bus, containing 18 variables and 21 operations. The specification is rather low level, i.e., the operations consist of simple assignments of concrete values to variables (no constraint solving is required). TLC4B needs 1.5 seconds[11] to translate the specification to TLA$^+$ and less than 6 seconds for the validation of the complete state space composed of 132,598 states. ProB needs 192 seconds to visit the same number of states. Both model checkers report no errors. For this specification TLC benefits from its efficient algorithm for storing big state spaces.

Invariant violations We use a defective specification of a travel agency system (CarlaTravelAgencyErr) to test the abilities of TLC4B detecting invariant violations. The specification consists of 295 line of B code, 11 variables and 10 operations. Most of the variables are functions (total, partial and injective) which are also manipulated by relational operators. TLC4B needs about 3 seconds to translate the model and to find the invariant violation. 377 states are explored with the aid of the breadth first search and the resulting trace has a length of 5 states. ProB needs roughly the same time.

Benchmarks Besides the evaluation of real case studies, we use some specific benchmark tests comparing TLC4B and ProB. We use a specification of a simple counter testing TLC4B's abilities to explore a big (linear) state space. TLC4B needs 3 seconds to explore the state space with 1 million states. Comparatively, ProB takes 204 seconds. In another specification, the states of doors are controlled. The specification allows the doors to be opened and closed. We use two versions: In the first version the state of the doors are represented as a function (Doors_Functions) and in the second as a relation (Doors_Relations). The first version allows TLC4B to use TLA$^+$ functions for the translation and TLC needs 2 seconds to explore 32,768 states. For the second version TLC4B uses the newly introduced relations and takes 10 seconds. As expected, TLC can evaluate built-in operators faster than user defined operators. Hence the distinction TLC4B has between functions and relations can make a significant difference in running times. ProB needs about 100 seconds to explore the state space of both specifications. However, ProB needs less than a second using symmetry reduction.[12]

In summary, ProB is substantially better than TLC4B when constraint solving is required (NQueens, SumAndProduct, GraphIsomorphism[13]) or when naive enumeration of operation arguments is inefficient (GardnerSwitchingPuzzle). For some specifications (not listed in the table) TLC was not able to validate the translated TLA$^+$ specification because TLC had to enumerate an infinite set. On the other

[11] Most of this time is required to start the JVM and to parse the B specification.

[12] TLC's symmetry reduction does not scale for large symmetric sets.

[13] See `http://www.data-validation.fr/data-validation-reverse-engineering/` for larger industrial application of this type of task.

Model	Lines	Result	States	Transitions	ProB	Tlc4B	$\frac{ProB}{Tlc4B}$
Counter	13	No Error	1000000	1000001	186.5	3.7	50.653
Doors_Functions	22	No Error	32768	983041	103.2	3.3	31.194
Can-Bus	314	No Error	132598	340265	191.8	7.2	26.624
KnightsTour(1)	28	Goal	508450	678084	817.5	34.1	23.998
USB_4Endpoints	197	NoError	16905	550418	72.5	5.7	12.632
Countdown	67	Inv. Viol.	18734	84617	31.4	2.8	11.073
Doors_Relations	22	No Error	32768	983041	103.3	11.6	8.926
Simpson_Four_Slot	78	No Error	46657	11275	33.7	4.3	7.874
EnumSetLockups	34	No Error	4375	52495	6.5	2.1	3.105
TicTacToe(1)	16	No Error	6046	19108	7.5	3.1	2.435
Cruise_finite1	604	No Error	1360	25696	6.2	3.2	1.954
CarlaTravelAgencyErr	295	Inv. Viol.	377	3163	3.3	3.1	1.069
FinalTravelAgency	331	No Error	1078	4530	4.7	4.4	1.068
CSM	64	No Error	77	210	1.4	1.6	0.859
SiemensMiniPilot_Abrial(1)	51	Goal	22	122	1.5	1.7	0.849
JavaBC-Interpreter	197	Goal	52	355	1.7	2.4	0.708
Scheduler	51	No Error	68	205	1.4	2.1	0.682
RussianPostalPuzzle	72	Goal	414	1159	1.7	2.8	0.588
Teletext_bench	431	No Error	13	122	1.8	3.7	0.496
WhoKilledAgatha	42	No Error	6	13	1.5	5.2	0.295
GardnerSwitchingPuzzle	59	Goal	206	502	2.5	11.7	0.213
NQueens_8	18	No Error	92	828	1.4	23.2	0.062
JobsPuzzle	66	Deadlock	2	2	1.6	29.3	0.053
SumAndProduct(1)	51	No Error	1	1	9.7	420.8	0.023
GraphIsomorphism	21	Deadlock	512	203	1.8	991.5	0.002

(1) Without Deadlock Check

Fig. 8. Empirical Results: Running times of Model Checking (times in seconds)

hand, Tlc4B is substantially better than ProB for lower-level specifications with a large state space.

5 Correctness of the Translation

There are several possible cases where our validation of B models using TLC could be unsound: there could be a bug in TLC, there could be a bug in our TLA$^+$ library for the B operators, there could be a bug in our implementation of the translation from B to TLA$^+$, there could be a fundamental flaw in our translation.

We have devised several approaches to mitigate those hazards. Firstly, when TLC finds a counter example it is replayed using ProB. In other words, every step of the counter example is double checked by ProB and the invariant or goal predicate is also re-checked by ProB. This makes it very unlikely that we produce incorrect counter examples. Indeed, ProB, TLC, and our translator have been

developed completely independently of each other and rely on different technology. Such independently developed double chains are often used in industry for safety critical tools.

The more tricky case is when TLC finds no counter example and claims to have checked the full state space. Here we cannot replay any counter example and we have the added difficulty that, contrary to ProB, TLC stores just fingerprints of states and that there is a small probability that not all states have been checked. We have no simple solution in this case, apart from re-checking the model using either ProB or formal proof. In addition, we have conducted extensive tests to validate our approach. For example, we use a range of models encoding mathematical laws to stress test our translation. These have proven to be very useful for detecting bugs in our translation and libraries (mainly bugs involving operator precedences). In addition, we have uncovered a bug in TLC relating to the cartesian product.[14] Moreover, we use a wide variety of benchmarks, checking that ProB and TLC produce the same result and generate the same number of states.

6 More Related Work, Discussion and Conclusion

Mosbahi et al. [11] were the first to provide a translation from B to TLA$^+$. Their goal was to verify liveness conditions on B specifications using TLC. Some of their translation rules are similar to the rules presented in this paper. For example, they also translate B operations into TLA$^+$ actions and provide straightforward rules for operators which exist in both languages. However, there are also significant differences:

- Our main contribution is that we deliver translation rules for almost all B operators and in particular for those which are not build-in operators in TLA$^+$. E.g., we specified the concept of relations including all operators on relations.
- Moreover, we also consider subtle differences between B and TLA$^+$ such as different well-definedness conditions and provide an appropriate translation.
- Regarding temporal formulas, we provide a way that a B user does not have to care about stuttering steps in TLA$^+$.
- We restrict our translation to the subset of TLA$^+$ which is supported by the model checker TLC. Furthermore, we made many adaptations and optimizations allowing TLC to validate B specification efficiently.
- The implemented translator is fully automatic and does not require the user to know TLA$^+$.

In the future, we would like to improve our automatic translator:

- Supporting modularization and the refinement techniques of B.
- Improving the performance of TLC by implementing Java modules for the new standard modules.
- Integrating Tlc4B into Rodin and supporting Event-B specifications.

[14] TLC erroneously evaluates the expression $\{1\} \times \{\} = \{\} \times \{1\}$ to *FALSE*.

In conclusion, by making TLC available to B models, we have closed a gap in the tool support and now have a range of complementary tools to validate B models: Atelier-B (or Rodin) providing automatic and interactive proof support, ProB being able to animate and model check high-level B specifications and providing constraint-based validation, and now TLC providing very efficient model checking of lower-level B specifications. The latter opens up many new possibilities, such as exhaustive checking of hardware models or sophisticated protocols. A strong point of our approach is the replaying of counter examples using ProB. Together with the work in [5] we have now constructed a two-way bridge between TLA^+ and B, and also hope that this will bring both communities closer together.

Acknowledgements We are grateful to Ivaylo Dobrikov for various discussions and support in developing the Tcl/Tk interface of Tlc4B. We also would like to thank Stephan Merz and Leslie Lamport for very useful feedback concerning TLA^+ and TLC. and for giving us access to various specifications (like SumProduct). Finally, we are thankful to anonymous referees for their useful feedback.

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. ClearSy. B language reference manual. `http://www.tools.clearsy.com/resources/Manrefb_en.pdf`. Accessed: 2013-11-10.
3. D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA+ proofs. In *FM*, pages 147–154, 2012.
4. E. Gafni and L. Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.
5. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings iFM'2012*, LNCS 7321, pages 24–38. Springer, 2012.
6. D. Hansen and M. Leuschel. Translating B to TLA+ for validation with TLC. `http://www.stups.uni-duesseldorf.de/w/Special:Publication/HansenLeuschel_TLC4B_techreport`, 2013.
7. L. Lamport. The TLA+ hyperbook. `http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html`. Accessed: 2013-10-30.
8. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
9. 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.
10. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
11. O. Mosbahi, L. Jemni, and J. Jaray. A formal approach for the development of automated systems. In J. Filipe, B. Shishkov, and M. Helfert, editors, *ICSOFT (SE)*, pages 304–310. INSTICC Press, 2007.
12. M. Reynolds. Changing nothing is sometimes doing something. Technical Report TR-98-02, Department of Computer Science, King's College London, February 1998.
13. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In L. Pierre and T. Kropf, editors, *Proceedings CHARME'99*, LNCS 1703, pages 54–66. Springer-Verlag, 1999.