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

Dominik Hansen and Michael Leuschel

Institut für Informatik, University of 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. The translation 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. Tlc4B can also transmit constant values, precalculated by ProB to TLC. This allows the user to combine the strength of both tools, i.e. ProB's constraint solving abilities and TLC's highly tuned model checking core. Furthermore, we demonstrate an approach to optimize the model checking process by encoding proof information in the translated TLA$^+$specification. We also present a series of case studies and benchmark tests comparing Tlc4B and ProB.

Keywords: B-Method, TLA$^+$, Tool Support, Model Checking, Animation

1. Introduction and Motivation

B [1] and TLA$^+$ [2] 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 [3], 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.

As far as tool support is concerned, TLA$^+$ is supported by the explicit state model checker TLC [4] and more recently by the TLAPS prover [5]. TLC has been used to validate a variety of distributed algorithms (e.g. [6]) and protocols. B has extensive proof support, e.g., in the form of the commercial product AtelierB [7] and the animator, constraint solver and model checker ProB [8, 9]. Both AtelierB and ProB are being used by companies, mainly in the railway sector for safety critical control software. In an earlier work [3] 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 B application possibilities.

This article is the extended version of the ABZ 2014 conference paper [10]. For this article we extended our work in different aspects:

- We use a static analysis to deduce proof information which can be used to optimize the translation (Sect. 4). We also enable symmetry reduction.
- We demonstrate the use of temporal formulas in B with the aid of an example. In particular, we show how to specify fairness conditions on parameterized operations (Sect. 5).
- We extended the interaction of ProB and TLC by running TLC on precalculated values of constants produced by ProB.
- We extended the empirical evaluation by several non-trivial models form the literature (Sect. 7). Moreover, we present an evaluation of the optimization techniques and of running TLC in parallel.

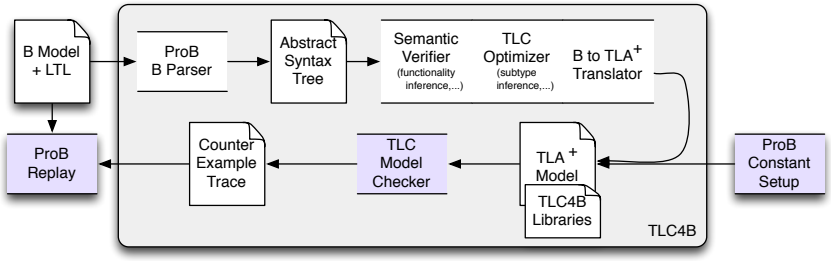

Figure 1: The TLC4B Translation and Validation Process

- We discuss the limitations and possible improvements in a separate section (Sect. 8).

2. Translation

The complete translation process from B to TLA$^+$ and back to B is depicted in Fig. 1. Before explaining the individual phases, we will illustrate the translation with an example and explain the various phases based on that example. More specific implementation details will be covered in Sect. 6.

2.1. Example

Below we use a specification (adapted from [9]) 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 *idle*.

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.

```
MODEL Scheduler
SETS
  PROCESSES;
  STATE = {idle, active}
VARIABLES
  state,
  queue
INVARIANT
  state  ∈  PROCESSES ⇸ STATE
  & queue  ∈  iseq(PROCESSES)
  & card(state^-1[{active}]) ≤ 1
  & !x.(x  ∈  ran(queue) ⇒ state(x) = idle)
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 ∧ p  ∉  ran(queue)
        THEN state := {p} ⩤ state
        END;

  addToQueue(p) =
        SELECT p  ∈  dom(state) ∧ state(p) = idle
        THEN 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
```

Figure 2: MODEL Scheduler

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 (described in Sect. 3). 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 constants of the module. If the axioms allow several solutions for a constant we will translate a constant as variable. All possible solution values will be enumerated in the initialization and the variable will be kept unchanged by all actions. In this case we assign a set[1] 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.

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[2], 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. $\{\langle 1, TRUE\rangle, \langle 1, FALSE\rangle \langle 2, TRUE\rangle\}$). 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.

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

[2] Relations are not mentioned in the language description of [2]. In [11] Lamport introduces relations in TLA$^+$ only to define the transitive closure.

MODULE *Scheduler*

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

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

VARIABLES *state*, *queue*

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

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

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

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

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

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

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

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

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

Figure 3: Module Scheduler

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

Figure 4: Configuration file for the module Scheduler

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 be extended by a specification.[3] 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).

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.[4]

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

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

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

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

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

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

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

⋮

Figure 5: An extract of the Module Relations

MODULE *Functions*

EXTENDS *FiniteSets*

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

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

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

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

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

⋮

Figure 6: An extract of the Module Functions

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 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 and max)

The complete translation rules of all data types and operators can be found in the technical report [12].

3. Optimizations

3.1. 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 an 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[5] where x is a bound variable, e is an expression, and S is ideally a subset of the type: $x = e$, $x \in S$, $x \subseteq S$, $x \subset S$ or $x \notin S$. In case of the last pattern we build the set difference of the type of the variable x and the set S:

B-Method	TLA$^+$
$\exists x.(x \notin S \wedge P)$	$\exists x \in (\tau_x \setminus S) : P$

[5]The B language description in [7] requires that each (bound) variable must be typed by one of these patterns before use.

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

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

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

3.1.1. 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 $(S \to T)$? 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:

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

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:

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

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 only once. Moreover this definition fares well even if S or T are sets of functions (e.g. $S \to V \to W$ 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. $\exists x \in RelTotalFunctions(S, T) : P$) and the second testing if a function belongs to the set (e.g. $f \in RelTotalFunctionsEleOf(S, T)$ as an invariant).

4. Proof guided model checking

The article [13] shows how to use proof information to optimize the consistency check of a model, i.e., checking whether the invariants of the model hold in all reachable states. The crucial idea is to deduce from the proof information that certain B operations (or events) are guaranteed to preserve the correctness of specific parts of the invariant. By keeping track of the operations leading to a certain

state in the state space within the model checker, one can avoid evaluating the corresponding parts of the invariant. In contrast to their approach, we will not implement this optimization in the verification tool, but will rather directly encode it in the generated TLA$^+$ model.

Deducing proof information. As a first approach we limit ourselves to use proof information which can be easily obtained by a simple static analysis:[6] A certain TLA$^+$ action preserves the correctness of an invariant if it only contains variables which are kept unchanged by the action. By revisiting the scheduler model (Fig. 3), we can deduce that the actions *new*, *del* and *leave* all preserve the correctness of the invariants *Invariant*2, and the action *addToQueue* preserves the correctness of *Invariant*1 and *Invariant*3 .

Encoding. To use this information, we specify an additional variable (*last_action*) to keep track of the action leading to each state. Each action assigns a specific value to this variable. The invariants are extended by a disjunction checking if the last action preserves the corresponding invariant. The extended scheduler module including proof informations is shown in Fig. 7 As already mentioned, TLC will evaluate a disjunction from left to right. If the left subformula is true, the right (possibly complex) subformula will not be evaluated. When successful, we replace evaluating a complex invariant by a simple membership test.

Evaluation. Using model with the additional state variable *last_action* can induce a significant overhead, i.e., the number of generated distinct states may be increased. Whenever two different actions lead to the same state in the original state space, two different states will now be produced for this state in the new state space. However, it is possible to preserve the original state space by modifying TLC's state function called the *view*[7] The default view is the tuple of all variables and two states are treated to be equal if they have the same view. Hence, we can instruct TLC to exclude the *last_action* variable from the view. This causes TLC to omit the evaluation of a state if another state with the same view has been evaluated previously.

Despite this, our optimization can sometimes still produce a small overhead due to more complex formulas. Moreover, its effectiveness depends on the order of executed actions. Still, for many larger models it produces a measurable benefit of about 20%.

5. 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

[6] In future work, we plan to use more elaborate proof information in the style of [13].
[7] The *view* is described in the tool section (14.3.3) of [2].

MODULE *Scheduler*

$\vdots$

CONSTANTS $\ldots$, *Init_action*, *new_action*, *del_action*, *enter_action*, *leave_action*, *addToQueue_action*
VARIABLES *state*, *queue*, *last_action*
$myView \triangleq \langle state, queue \rangle$

$Invariant1 \triangleq \lor last\_action \in \{addToQueue\_action\}$
$\lor state \in RelParFuncEleOf(PROCESSES, STATE\_1)$
$Invariant2 \triangleq \lor last\_action \in \{new\_action, leave\_action, del\_action\}$
$\lor queue \in ISeqEleOf(PROCESSES)$
$Invariant3 \triangleq \lor last\_action \in \{addToQueue\_action\}$
$\lor Cardinality(RelImage(RelInverse(state), \{active\})) \leq 1$
$Invariant4 \triangleq \forall x \in Range(queue) : RelCall(state, x) = idle$

$Init \triangleq \land state = \{\}$
$\land queue = \langle\rangle$
$\land last\_action = Init\_action$

$new(p) \triangleq \land state' = state \cup \{\langle p, idle \rangle\}$
$\land$ UNCHANGED $\langle queue \rangle$
$\land last\_action' = new\_action$

$\vdots$

Figure 7: Module Scheduler extended by proof informations

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 $(Init \land \Box[Next]_{vars})$ 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:

$$Spec \triangleq c = 1 \land \Box[c < 10 \land c' = c + 1]_c$$

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.

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

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 ($WF_{vars}(Next)$) would prohibit an infinite number of stuttering steps if a step of the next-state relation is possible (i.e. *Next* is always enabled):

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

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 steps result from B operations which do not change the state of the system (e.g. skip or query 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 [14], 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:

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

The definition of $VWF_{vars}(A)$ is identical to $WF_{vars}(A)$ except for an additional third case allowing infinite stuttering steps if A is a stuttering action. 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:

- $\Box f$ (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"[9])

Liveness conditions in B. Since temporal operators are not part of the B language, we insert liveness conditions (represented in string format) into the specification under the DEFINITIONS clause. As an example, we revisit the scheduler specification from Section 2.1 and specify the following liveness property:

> Whenever a process is added to the queue it will always eventually be loaded into the processor (i.e. gets the state *active*).

To satisfy this condition, we have to require WF for the operations *enter* and *leave*. Otherwise, there would be a infinite loop of creating a new process and deleting it immediately as a counter example. We can formulate this liveness condition with the aid of the newly introduced temporal operators and ordinary B predicates:

$$\begin{aligned}&\text{ASSERT\_LTL\_1} == \text{"WF(enter)} \wedge \text{WF(leave)}\\ &\quad \Rightarrow \Box(\forall p.(p \in \mathit{PROCESSES} \wedge p \in \mathit{ran}(\mathit{queue})\\ &\qquad \Rightarrow \Diamond p \in \mathit{dom}(\mathit{state}) \wedge \mathit{state}(p) = \mathit{active})\text{"}\end{aligned}$$

The translation of the liveness condition is almost straight forward:

$$\begin{aligned}&\mathit{ASSERT\_LTL\_1} \triangleq\\ &\quad \mathit{VWF}_{\mathit{vars}}(\mathit{enter}) \wedge \mathit{VWF}_{\mathit{vars}}(\exists\, p \in \mathit{RelDomain}(\mathit{state}) : \mathit{leave}(p))\\ &\qquad \Rightarrow \Box(\forall\, p \in \mathit{PROCESSES} : p \in \mathit{Range}(\mathit{queue})\\ &\qquad\quad \Rightarrow \Diamond(p \in \mathit{RelDomain}(\mathit{state}) \wedge \mathit{RelCall}(\mathit{state},\, p) = \mathit{active}))\end{aligned}$$

All temporal operators can be directly mapped to the existing TLA$^+$ operators. The translation of the temporal quantification is slightly different compared to the translation of a ordinary quantification. The model checker TLC can only substitute values of a constant set for the bound variable p. Hence, we only use the constant set *PROCESSES* as the associated set and do not build the intersection of *PROCESSES* and $\mathit{Range}(\mathit{queue})$ as described in the optimization section. If there are no assumptions on the parameters of an operation that is considered to be fair (here the *leave* operation) we will create an existential quantification in the fairness condition in order to get a valid TLA$^+$ expression. This will force TLC to ignore the parameter when it evaluates this fairness condition. Alternatively, the user can explicitly specify the parameter:

$$\forall p.(p \in \mathit{PROCESSES} \Rightarrow \mathit{WF}(\mathit{leave}(p))$$

The semantic meaning of the last formula is that the model should be fair with respect to the *leave* operation and all processes. The other fairness condition could allow to neglect a certain process by favouring other processes. However, for the scheduler model both conditions are equivalent because there is only one active process. Finally, we have to add an entry in configuration file (*PROPERTY ASSERT_LTL_1*) telling TLC to verify this liveness property.

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

6. Implementation

Our translator, called Tlc4B, is implemented in Java and it took about six months to develop the initial version. Tlc4B has been integrated into the ProB Tcl/Tk UI as of version 1.3.7-beta10. It can now also be called from the command-line version (probcli). Figure 1 in Section 2 shows the translation and validation process of Tlc4B. After parsing the specificationTlc4B 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" 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). As shown in figure 8 counter-examples found by TLC are automatically replayed in the ProB animator (displayed in the history pane) to give the user an optimal feedback. The user needs no knowledge of TLA^+ because the translation is completely hidden.

6.1. Constant Setup

ProB allows the user to partially validate infinite models. For example, assume we have a B specification containing a constant c and the only assumption is that c is a natural number ($c \in \mathbb{N}$). The ProB animator will provide the user various possible values for c satisfying the assumption. Moreover, we can run the ProB model checker which constrains the model to this selection. Otherwise, in order to run the TLC model checker on a TLA^+ specification, the user has to specify values for all constants oneself. Hence, we combine both tools by allowing the user to select a constant setup using ProB's convenient animation feature and then the translator Tlc4B encodes the constant setup in the translated TLA^+ module.

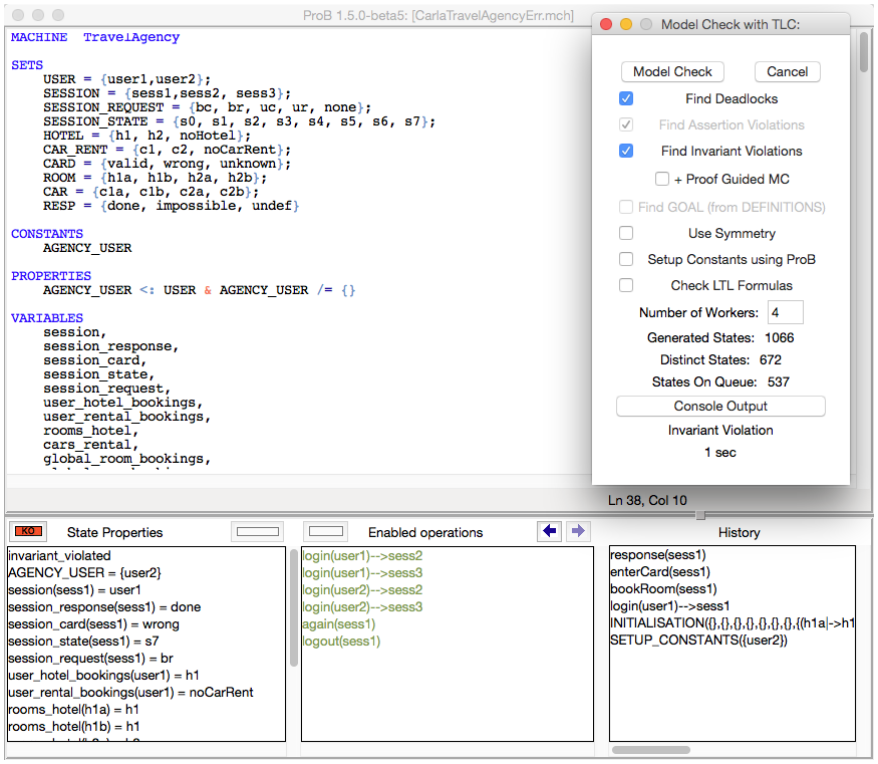

Figure 8: ProB animator

Note that the assumptions are still checked by TLC. Thus an invalid constants setup produced by ProB would be rejected by TLC, and the validation process of both tools is still independent. The combination of both tools opens up some new possibilities even for finite models. Assuming we have a model with a complex constants setup where constraint solving abilities are required. We can first use the strengths of ProB to setup the constants and then use TLC abilities to explore large state spaces.

Moreover, Tlc4B provides several further features in order to run TLC. For example, the user can choose the number of parallel workers and can make use of TLC's abilities for symmetry reduction. For the latter, Tlc4B automatically detects suitable sets (e.g. deferred sets) and instructs TLC.[10] All features can be easily accessed in the graphical user interface of ProB.

[10]Symmetry reduction is described in the tool section (14.3.4) of [2].

6.2. 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
- a bug in our TLA$^+$ libraries for the B operators,
- a bug in our implementation of the translation from B to TLA$^+$,
- 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 (TLC provides an estimation of this probability). 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.[11] Moreover, we use a wide variety of benchmarks, checking that ProB and TLC produce the same result and generate the same number of states.

7. Experiments

7.1. Model Checking Experiments

The following examples[12] 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. The examples contains several smaller benchmark models, industrial models and models from the literature (Table 1 contains citations, e.g., the library model is the information system

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

[12] The source code of the examples can be found at `http://www.stups.uni-duesseldorf.de/w/Pub:TLC4B_Benchmarks`.

example from [15] and the flight model is the system from [16]). Below we discuss some of the models and results from Table 1 in more detail.

Can-Bus. This 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 to animate the model). Tlc4B needs 1.5 seconds[13] 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 large state spaces.

Defective Specification. 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.

Specification with Complex Events We use Tlc4B to find a solution for the GardnerSwitchingPuzzle. Tlc4B needs about 12 seconds to find the trace leading to the desired state. ProB only needs 2.5 seconds to confirm the result. Both tools are instructed to use breadth first search. Hence, the reported traces is the same. The different running times result form different capacities to evaluate complex state transitions. The specifications contains parametric operations/events and ProB benefits from its constraints solving abilities in order to find valid values for the parameters satisfying the guards. TLC has to iterate over all possible values of the parameters and then has to check the guards for each combination. The static subtype inference presented in Sect. 3 can here assist TLC only to certain degree.

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 4 seconds to explore the state space with 1 million states. Comparatively, ProB takes 187 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 3 seconds to explore 32,768 states. For the second version Tlc4B uses the newly introduced relations and takes 12 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

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

Table 1: Empirical Results: Running times of Single-Core Model Checking (times in seconds)

Model	Lines	Result	States	Transitions	ProB	Tlc4B	$\frac{ProB}{Tlc4B}$
Counter	13	No Error	1,000,000	1,000,001	186.5	3.7	50.653
Doors_Functions(4)	22	No Error	32,768	983,041	103.2	3.3	31.194
Can-Bus	314	No Error	132,598	340,265	191.8	7.2	26.624
KnightsTour(1)	28	Goal	508,450	678,084	817.5	34.1	23.998
FlightSystem [16]	112	No Error	28,561	263,641	345.5	22.8	15.154
Library [15]	201	No Error	35,542	390,697	187.5	13.1	14.313
USB_4Endpoints	197	NoError	16,905	550,418	72.5	5.7	12.632
Countdown	67	Inv. Viol.	18,734	84,617	31.4	2.8	11.073
Doors_Relations(4)	22	No Error	32,768	983,041	103.3	11.6	8.926
Simpson_Four_Slot	78	No Error	46,657	11,275	33.7	4.3	7.874
Interlocking(2)[17]	187	No Error	672,173	2,244,109	4,995.2	1,456.7	3.429
EnumSetLockups	34	No Error	4,375	52,495	6.5	2.1	3.105
GSM_revue [18]	220	No Error	1,848	53,593	10.4	3.6	2.888
TicTacToe(2)	16	No Error	6,046	19,108	7.5	3.1	2.435
Cruise Control	604	No Error	1,360	25,696	6.2	3.2	1.954
CarlaTravelAgencyErr	295	Inv. Viol.	377	3,163	3.3	3.1	1.069
FinalTravelAgency	331	No Error	1,078	4,530	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
Satellite Mode Protocol(3)	201	No Error	62,616	185,287	288.1	402.7	0.715
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	1,159	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 [19]	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
(2) Optimized Model (reduced state space)
(3) Without Assertion Check
(4) 15 Doors

seconds to explore the state space of both specifications. However, ProB needs less than a second using symmetry reduction; see Sect. 7.2 below.

We have successfully validated several existing models from the literature (Table 1).

7.2. Symmetry Reduction

Both ProB and TLC provide symmetry reduction. In B, symmetry can be exploited for deferred sets, which are sets defined by the user but whose elements are not named (their exact composition is "deferred" until later in the refinement). Hence, any element of a deferred set can be substituted for another element, leading to symmetries in the state space. In Tlc4B we have implemented the possibility to provide annotations for TLC so as to enable its symmetry reduction. For example,

for the FlightSystem example TLC4B would (optionally) generate the following annotation in the TLA$^+$ translation, based on the two deferred sets *MEMBERID* and *BOOKID*:

```
Symmetry == Permutations(MEMBERID) \cup Permutations(BOOKID)
```

In the following table, we compare this new feature with ProB's approximate hash symmetry method [20] (the other symmetry methods lead to slower times). We use three of the specifications with deferred sets: Doors, Library [15] and FlightSystem [16].

Model	Def. Set Size(s)	States (not reduced)	ProB	ProB Hash Symm	TLC4B	TLC4B + Symm
Doors_Relations	7	128	1.9 s	1.7 s	2.7 s	7.9 s
	15	32,768	103.2 s	1.8 s	11.6 s	-
Library	3, 3	35,542	192.2 s	8.8 s	12.84 s	5.0 s
	3, 4	1,035,314	-	70.7 s	297.6 s	26.6 s
FlightSystem	3, 4	28,561	298.7	5.6 s	21.2 s	4.6 s
	4, 4	194,481	-	8.4 s	223.0 s	8.3 s

As we can see in the first example, TLC's symmetry reduction technique does not scale for large symmetry sets (i.e., deferred sets), and the runtime actually deterioriates here with a deferred set of size 7. If a symmetry set has actually more than 8 elements TLC will throw an exception ("Attempted to construct a set with too many elements (>1000000)"). Hence, our translator will automatically prevent symmetry for deferred set sizes of 9 or larger. The other two examples show that both TLC and ProB can gain from the symmetry reduction. The reduction achieved by TLC is less marked than for ProB, but the effect is still considerable and the symmetry reduction feature of TLC4B and TLC can be of practical use.

7.3. Parallel and Proof-Guided Model Checking of an Interlocking

In his book on Event-B [17], Abrial presents a model of a railway interlocking system. While this model is an academic model intended for teaching Event-B, it has a lot of features in common with real-life interlockings and formal models thereof.

Model checking cannot be used in an exhaustive fashion on the general interlocking model from [17], as there are infinitely many different topologies. However, model checking is an obvious candidate to verify the formal interlocking model for a particular topology. At first glance, the original topology from page 524 of [17] (with 5 signals, 5 points, one crossing and 14 tracks segments) looks quite small and one would hope that exhaustive model checking should quite quickly be able to check all possible states for the absence of deadlocks. However, the first refinement of the model has over 61 million distinct states and over 445 million transitions (aka events), and defied our initial efforts to exhaustively validate the model for this topology.

Using TLC4B we are able to explore the whole state spaces and to validate all invariants in about 5 hours on a six core 3.33 GHz Mac Pro with 16 GB of RAM and 12 workers (the experiments in this sub-section were thus run on another computer

than above). The proof guided model checking introduced in Sect. 4 reduces the model checking time to under 4 hours. Note, that only proof information deduced by the static analysis described in Sect. 4 is used. Since, Abrial's model is completely proven we omitted the invariant check in a further run for which Tlc4B needs 55 minutes.

More recently, ProB can be run in a parallel and distributed fashion [21]. ProB needs about 30 hours to check the same state space along with the invariants compared to the respectively 4 and 5 hours with Tlc4B; the proof directed model checking from the single core version of ProB is not yet available in this parallel version of ProB.

	States	Workers	Invariant Check	Proof Guided MC	Running time
Tlc4B	61 mio	1	No	-	191 min
Tlc4B	61 mio	12	No	-	55 min
Tlc4B	61 mio	12	Yes	No	297 min
Tlc4B	61 mio	12	Yes	Yes	227 min
ProB	61 mio	12	Yes	No	1787 min

This example nicely illustrates the potential of Tlc4B. ProB's features were, however, also useful in uncovering one cause for the somewhat unexpected state explosion (routes were not freed as soon as possible, leading to additional interleavings; this was detected by using ProB's features to inspect the state space); a hand optimized model has a considerably smaller statespace (672,174 states) and is the model we used in Table 1. This case study shows the advantage of having a variety of tools at our disposal.

7.4. Summary of Experiments

In summary, ProB is substantially better than Tlc4B when constraint solving is required (NQueens, SumAndProduct, GraphIsomorphism[14]) or when naive enumeration of operation arguments is inefficient (GardnerSwitchingPuzzle). On the other hand, Tlc4B is substantially better than ProB for lower-level specifications with a large state space. We have also shown that the symmetry detection of Tlc4B can be beneficial, as is our new partial invariant evaluation feature and the capability of running model checks in parallel.

8. Discussion, Future and Related Work

While Tlc4B covers almost all operators of an abstract B machine, TLC is not able to validate all of the translated TLA$^+$ specifications because sometimes TLC has to enumerate an infinite set. For example, assume we have an existential quantification $\exists x.(x \in \mathbb{Z} \ \wedge \ x + x = p)$ in B. Tlc4B will not be able to further restrict the type ($\mathbb{Z}$) of the bound variable x and TLC will abort to evaluate the

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

translated TLA$^+$ expression $\exists x \in \mathbb{Z} : x + x = p$. However, this issue would be indicated to the user and it should mostly be possible to rewrite the formula.

The experimental results from Sect. 7 imply that it would be suitable to apply Tlc4B to specifications at lower refinement levels. Normally, the state space increases and the needed constraint solving abilities decrease during a refinement process. Unfortunately, it is not easy to translate the operator for the sequential composition of operations ($op1;op2$) which is usually used in these specifications. Although there is a corresponding operator for the composition of actions in TLA$^+$, the operator is not supported by TLC. It seems to be possible to translate this operator using a program counter as an additional variable and to adjust the guards of consecutive actions. The imperative algorithm language PlusCal [22] is translated to TLA$^+$ in this way. However, this approach would create additional states and would make it much harder to verify the correctness of our translation.

The current version of the translator does not yet inherently support the modularization and refinement constructs of B. ProB's pretty-printing feature can be used to transform a compound set of models into a single model which can be validated by Tlc4B. We have also used this feature in Sect. 7.3 to be able to model check the Event-B interlocking model. However, our approach should not rely on ProB in order to ensure an independent validation.

When we started to implement Tlc4B the model checker TLC was not yet an open source project. In order to improve the performance of TLC, one can now modify the implementation of TLC. For example, one could now try to implement the optimization of Sect. 4 within the Java code of the model checker. Also, we can implement optimized Java modules for the new library modules (e.g. *Relations*). Such Java modules already exist for most standard modules (e.g. *Naturals* and *Sequences*). While this example is on top on our work list we have avoided modifying the implementation of TLC thus far for various reasons. The core of the TLC model checker is a finalized system; other TLA$^+$ tools confine themselves to the provided interface (e.g. the Toolbox IDE). While our translation is optimized for a validation with TLC, the translated TLA$^+$ modules are independent from TLC and could be used by other validation tools. It seems to be much harder to detect errors in the implementation than in a TLA$^+$ module.[15] Finally, the encoding of optimizations directly in TLA$^+$ seems to be transparent and comprehensible to the user.

Mosbahi et al. [23] were the first to provide a translation from B to TLA$^+$. Their goal was to verify liveness conditions on B specifications using TLC. Stuttering steps as a subtle difference between B and TLA$^+$ are not considered by their translation. 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, we provide translation rules for almost all constructs of B, in particular for those

[15] TLC (or, more precisely, the parser Sany) checks a TLA$^+$ formula for syntactic correctness. Moreover, various semantic conditions are also checked. These checks revealed some subtle bugs in our library modules.

which are not built-in in TLA$^+$. Moreover, we detected several issues in their translation rules, which seems be unsound (e.g. they translate a total injective function ($S \rightarrowtail T$) in B to a total function $S \rightarrow T$ in TLA$^+$) or not suitable for a generic translation tool. Another difference is that we have restricted our translation to the subset of TLA$^+$ which is executable by TLC. We have contrasted their and our translation rules in a technical report [12].

There have been various other works which translate B to other formalisms: B to the Express language [24], B to the Kodkod library [25, 26], B to SMTLib for proving [27]. It is interesting to see how these various technologies and translations interact and can be of benefit to the current B user: the SMTLib translation [27] is useful for proving, the proof information in turn can benefit ProB via proof-directed model checking [13], the Kodkod translation [26] is beneficial for constant finding and constraint-based checking, while the translation to TLA$^+$ can achieve efficient explicit state model checking, benefitting from ProB for constant finding and trace replay. In this context it may also be relevant to mention the works that translate the Z language to other formalisms: Z to Alloy [28], Z to SAL [29].

Leuschel et al. [30] presented a LTL model checker, implemented inside ProB that can verify liveness properties on B specification. The LTL model checker also supports Past-LTL and propositions on transitions such as that an certain operation have to be executed next. Currently the model checker does not support fairness and temporal quantifications. The way to inject a temporal formula into a B specification presented in this paper is derived from the LTL model checker. We use the same parser for temporal formulas and TLC as an alternative back end. However, the intersection of supported temporal formulas of both model checkers is currently too small to present a comprehensive comparison.

9. Conclusion

We presented a translation from B to TLA$^+$, which make use of a (sub) type inference algorithm and translates a large subset of B to TLA$^+$. Our main contribution is that we deliver translation rules for almost all B operators and in particular for those which are not built-in operators in TLA$^+$. For example, 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. 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. We provide a way that a B user can specify and validate temporal properties including fairness conditions on B specifications and does not have to care about stuttering steps in TLA$^+$.

The implemented translator Tlc4B is fully automatic and does not require the user to know TLA$^+$. By integrating Tlc4B into the ProB UI we provide a convenient way to use TLC to validate B specification. The interaction of ProB and TLC allows the user to benefit from the strength of both tools by preserving

an independent validation. The empirical evaluation shows that our approach fares well not only for toy examples.

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 [3] we have now constructed a two-way bridge between TLA^+ and B, and we also hope that this will bring both communities closer together.

Acknowledgements

Part of this research has been sponsored by the EU funded FP7 project 287563 (ADVANCE) and the DFG funded project Gepavas. We are grateful to Ivaylo Dobrikov for various discussions and support in developing the Tcl/Tk interface of Tlc4B.

References

[1] J.-R. Abrial, The B-Book, Cambridge University Press, 1996.

[2] L. Lamport, Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers, Addison-Wesley, 2002.

[3] D. Hansen, M. Leuschel, Translating TLA+ to B for validation with ProB, in: Proceedings iFM'2012, LNCS 7321, Springer, 2012, pp. 24–38.

[4] Y. Yu, P. Manolios, L. Lamport, Model checking TLA+ specifications, in: L. Pierre, T. Kropf (Eds.), Proceedings CHARME'99, LNCS 1703, Springer-Verlag, 1999, pp. 54–66.

[5] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, H. Vanzetto, TLA+ proofs, in: FM, 2012, pp. 147–154.

[6] E. Gafni, L. Lamport, Disk Paxos, Distributed Computing 16 (1) (2003) 1–20.

[7] ClearSy, B language reference manual, `http://www.tools.clearsy.com/resources/Manrefb_en.pdf`, accessed: 2013-11-10.

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

[9] M. Leuschel, M. J. Butler, ProB: an automated analysis toolset for the B method, STTT 10 (2) (2008) 185–203.

[10] D. Hansen, M. Leuschel, Translating b to tla + for validation with tlc, in: Proceedings ABZ'14, LNCS 8477, 2014, pp. 40–55.

[11] L. Lamport, The TLA+ hyperbook, `http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html`, accessed: 2013-10-30.

[12] D. Hansen, M. Leuschel, Translating B to TLA+ for validation with TLC, `http://www.stups.uni-duesseldorf.de/w/Special:Publication/HansenLeuschel_TLC4B_techreport` (2013).

[13] J. Bendisposto, M. Leuschel, Proof assisted model checking for B, in: K. Breitman, A. Cavalcanti (Eds.), Proceedings of ICFEM 2009, Vol. 5885 of Lecture Notes in Computer Science, Springer, 2009, pp. 504–520.

[14] M. Reynolds, Changing nothing is sometimes doing something, Tech. Rep. TR-98-02, Department of Computer Science, King's College London (February 1998).

[15] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, M. Ouenzar, Comparison of model checking tools for information systems, in: J. S. Dong, H. Zhu (Eds.), Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods,

ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings, Vol. 6447 of Lecture Notes in Computer Science, Springer, 2010, pp. 581–596. doi:10.1007/978-3-642-16901-4_38.
URL http://dx.doi.org/10.1007/978-3-642-16901-4_38

[16] A. Mammar, M. Frappier, Proof-based verification approaches for dynamic properties: application to the information system domain, Formal Aspects of Computing (2014) 1–40doi:10.1007/s00165-014-0323-x.
URL http://dx.doi.org/10.1007/s00165-014-0323-x

[17] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, 2010.

[18] E. Bernard, B. Legeard, X. Luck, F. Peureux, Generation of test sequences from formal specifications: GSM 11-11 standard case study, International Journal of Software Practice and Experience 34 (10) (2004) 915–948.

[19] M. Leuschel, D. Schneider, Towards b as a high-level constraint modelling language, in: Y. Ait Ameur, K.-D. Schewe (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z, Vol. 8477 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 101–116. doi:10.1007/978-3-662-43652-3_8.
URL http://dx.doi.org/10.1007/978-3-662-43652-3_8

[20] M. Leuschel, T. Massart, Efficient approximate verification of B via symmetry markers, Annals of Mathematics and Artificial Intelligence 59 (1) (2010) 81–106.

[21] J. Bendisposto, Directed and parallel model checking of b specifications, Ph.D. thesis, University of Düsseldorf (January 2015).

[22] L. Lamport, The pluscal algorithm language, in: Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing, ICTAC '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 36–60. doi:10.1007/978-3-642-03466-4_2.
URL http://dx.doi.org/10.1007/978-3-642-03466-4_2

[23] O. Mosbahi, L. Jemni, J. Jaray, A formal approach for the development of automated systems, in: J. Filipe, B. Shishkov, M. Helfert (Eds.), ICSOFT (SE), INSTICC Press, 2007, pp. 304–310.

[24] I. Aït-Sadoune, Y. Aït-Ameur, Animating Event B models by formal data models, in: ISoLA, 2008, pp. 37–55.

[25] D. Plagge, M. Leuschel, I. Lopatkin, A. Romanovsky, SAL, Kodkod, and BDDs for Validation of B Models. Lessons and Outlook, Proceedings AFM 2009 (2009) 16–22.

[26] D. Plagge, M. Leuschel, Validating B, Z and TLA+ using ProB and Kodkod, in: D. Giannakopoulou, D. Méry (Eds.), Proceedings FM'2012, LNCS 7436, Springer, 2012, pp. 372–386.

[27] D. Déharbe, Automatic Verification for a Class of Proof Obligations with SMT-Solvers, in: Frappier et al. [31], pp. 217–230. doi:10.1007/978-3-642-11811-1.
URL http://dx.doi.org/10.1007/978-3-642-11811-1

[28] P. Malik, L. Groves, C. Lenihan, Translating z to alloy, in: Frappier et al. [31], pp. 377–390. doi:10.1007/978-3-642-11811-1.
URL http://dx.doi.org/10.1007/978-3-642-11811-1

[29] J. Derrick, S. North, A. J. H. Simons, Z2SAL: a translation-based model checker for Z, Formal Asp. Comput. 23 (1) (2011) 43–71. doi:10.1007/s00165-009-0126-7.
URL http://dx.doi.org/10.1007/s00165-009-0126-7

[30] D. Plagge, M. Leuschel, Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more, STTT 11 (2010) 9–21.

[31] M. Frappier, U. Glässer, S. Khurshid, R. Laleau, S. Reeves (Eds.), Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings, Vol. 5977 of Lecture Notes in Computer Science, Springer, 2010. doi:10.1007/978-3-642-11811-1.
URL http://dx.doi.org/10.1007/978-3-642-11811-1