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

Dominik Hansen and Michael Leuschel

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

Abstract. TLA$^+$ and B share the common base of predicate logic, arithmetic and set theory. However, there are still considerable differences, such as very different approaches to typing and modularization. There is also considerable difference in the available tool support. In this paper, we present a translation of the non-temporal part of TLA$^+$ to B, which makes it possible to feed TLA$^+$ specifications into existing tools for B. Part of this translation must include a type inference algorithm, in order to produce typed B specifications. There are many other tricky aspects, such as translating modules as well as LET/IN and IF/THEN/ELSE expressions. We also present an integration of our translation into ProB. ProB thus provides a complementary tool to the explicit state model checker TLC, with convenient animation and constraint solving for TLA$^+$. We also present a series of case studies, highlighting the complementarity to TLC. In particular, we highlight the sometimes dramatic difference in performance when it comes to solving complicated constraints in TLA$^+$.

Keywords: TLA, B-Method, Tool Support, Model Checking, Animation.

1 Introduction and Motivation

TLA$^+$ [5] and B [1] are both state-based formal methods rooted in predicate logic, combined with arithmetic and set theory. There are, however, considerable differences:

– TLA$^+$ is untyped, while B is strongly typed.
– The concepts of modularization are very different (as we will see later in the paper).
– TLA$^+$ and B both support sets and functions. However, functions in TLA$^+$ are total, while B supports relations, partial functions, injections, bijections, etc.
– TLA$^+$ has several constructs which are lacking in B, such as an IF/THEN/ELSE for expressions and predicates[1], a LET/IN construct or the CHOOSE operator. The latter enables one to define recursive functions over sets, which are akwkard to define in B.

* Part of this research has been sponsored by the EU funded FP7 projects 214158 (DEPLOY) and 287563 (ADVANCE).

[1] B only provides an IF/THEN/ELSE for substitutions.

J. Derrick et al. (Eds.): IFM 2012, LNCS 7321, pp. 24–38, 2012.
© Springer-Verlag Berlin Heidelberg 2012

- TLA+ allows to specify liveness properties while B is limited to invariance properties (temporal formulas such as liveness conditions will be excluded from our translation).

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 [2]. B has extensive proof support, e.g., in the form of the commercial product AtelierB [3]. The animator and model checker ProB [6] can also be applied to B specifications. Both AtelierB and ProB are being used by companies, mainly in the railway sector for safety critical control software. Some of the goals of our translation are

- to gain a better understanding of the common core and of the differences between TLA+ and B,
- to obtain an animator for TLA+,
- and to obtain a constraint solver for TLA+.

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 does not support fairness (yet) and is entirely RAM-based. The model checking core of ProB is less tuned than TLA+. However, ProB offers several features which are absent from TLC, notably an interactive animator with various visualization options. More importantly, the ProB kernel provides for constraint solving over predicate logic, set theory and arithmetic. ProB can also deal quite well with large data values. This has many applications, from constraint-based invariant or deadlock checking [4], over to test-case generation and on to improved animation because the user has to provide much less concrete values than with other tools. It also makes certain specifications "executable" which are beyond the reach of other tools such as TLC.

We suppose that the reader is familiar with either TLA+ or B. Indeed, we hope that through our translation, TLA+ constructs can be understood by B users and vice-versa. Below, in Sect. 2 we introduce the essentials of our translation on a simple example, while in Sect. 3 we present the translation more formally. In Sect. 4 we present case studies and experiments, and will also compare the tools ProB and TLC. We conclude with more related and future work in Sect. 5.

2 An Example Translation from TLA+ to B

To allow B users to become familiar with TLA+, we present a variation of the well known HourClock example from Chapter 2 of [5]. Figure 1 shows the *MyHourClock* module, which avoids the IF/THEN/ELSE expression of the original at this point. The specification describes the typical behavior of a digital clock displaying only hours. The module starts with the MODULE clause followed by the name of the specification. The analogous clause of a B machine is MACHINE or MODEL. At the beginning of the module body, arithmetic operators such as

MODULE *MyHourClock*

EXTENDS *Integers*
CONSTANTS c
ASSUME $c \in 1 \mathrel{..} 12$
VARIABLES hr
$Init \triangleq hr = c$
$add_1(p) \triangleq p + 1$
$Inc \triangleq hr < 12 \land hr' = add_1(hr)$
$Reset \triangleq hr = 12 \land hr' = 1$
$Next \triangleq Inc \lor Reset$

Fig. 1. Module MyHourClock

$+$ or ".." are loaded via EXTENDS from the standard module *Integers*. These operators are not built-in operators in TLA$^+$ and can either be defined by the user or imported with their usual meaning as here. The declaration of constants and variables is identical in both languages. The ASSUME clause in TLA$^+$ correspondes to the PROPERTIES clause in B.

To understand the meaning of the other definitions in the module we need some additional information.[2] For our translation we use a configuration file, as TLC also uses, telling us the initial state and the next-state relation of the module. For this example we suppose *Init* to be the initial state predicate and *Next* to be the next-state relation. *Init* indicates that the variable *hr* has the value of the constant c in the initial state. *Next* is separated into two actions by the disjunction operator. An action is a before-after predicate describing a transition to a next-state with the aid of the prime operator($'$). A primed variable represents the variable in the next-state. The use of the additional *add_1* operator may seem artificial here; its purpose is to demonstrate another aspect of our translation.

Figure 2 shows the translated B Machine of the *MyHourClock* example. We use the BECOMES/SUCH/THAT substitution under the INITIALISATION clause to initialize the variables of the B machine. It assigns a value to the variable such that the predicate in the brackets is satisfied. The TLA$^+$ actions *Inc* and *Reset* are translated as separate B operations. We represent the prime operator in B by adding a local auxiliary variable for every variable. The auxiliary variables (with suffix "_n") are generated in the ANY part of the ANY/WHERE/THEN substitution and get their value in the WHERE part. If the predicate in the WHERE part is not satisfiable the operation can not be executed. Finally, the values of the auxiliary variables are assigned to the corresponding global variables in the THEN part.

Operators such as *add_1* are translated using B definitions. B definitions are a kind of macro and help to write frequently used expressions. They are syntactic sugar and will be resolved in the the parsing phase. Furthermore, they can have parameters. Using B definitions avoids to replace an operator call by the

[2] "What those definitions represent [...] lies outside the scope of mathematics and therefore outside the scope of TLA$^+$" (see p. 21 of [5]).

```
MACHINE MyHourClock
DEFINITIONS   add_1(p) == p + 1
CONSTANTS c
PROPERTIES c ∈ 1..12
VARIABLES hr
INVARIANT hr ∈ ℤ
INITIALISATION hr :(hr = c)
OPERATIONS
  Inc_Op = ANY hr_n
              WHERE hr < 12 ∧ hr_n = add_1(hr)
              THEN hr := hr_n
              END
  Reset_Op = ANY hr_n
              WHERE hr = 12 ∧ hr_n = 1
              THEN hr := hr_n
              END
END
```

Fig. 2. Machine MyHourClock

definition of the operator. The arithmetic operators are translated with the use of B's built-in operators. Therefore, they do not appear in the DEFINITIONS clause. Finally, to obtain a correct B machine our translation has inferred and added the types of the variables in the INVARIANT clause.

3 The Translation from TLA$^+$ to B

3.1 Type System

The basis of our translation is a mapping of TLA$^+$ values to B values. Due to the strict type system of B, every B value has to be associated with a type. Below we list the translations of the TLA$^+$ values and the resulting restrictions:

- Numbers: In B real numbers are not supported. Thus, only integers can be translated. They get the B type $\mathbb{Z}$.
- The boolean values TRUE and FALSE are identical in both languages. They get the B Type BOOL.
- The concepts of strings are different in both languages. In TLA$^+$ a string is a sequence of characters and a single character can be accessed. However, a string in B is atomic and has the base type STRING. For the translation we currently reject strings if they are used as tuples.
- A model value is none of TLA$^+$'s own values but one of TLC's. But it is established to use TLA$^+$ together with TLC so we deliver a suitable translation. TLC allows to assign a model value or a set of model values to a constant in the configuration file. The equivalent of a model value is an element of an enumerated set in B. To make different model values comparable to each other we put them in the same enumerated set named: ENUM_i. However if two model values are never compared in the specification, we put

them in different sets (such as ENUM_1 and ENUM_2). The B type of a model value is the name of the enumerated set containing it.
- There is one main difference between sets in TLA$^+$ and B. In B all elements of a set must have the same type, i.e., a set has the B type $\mathbb{P}\tau$ where τ is the type of all its elements.
- In both languages functions are a mapping from a domain to a range. The B type of a function is $\mathbb{P}(\tau_1 \times \tau_2)$, where $\mathbb{P}\tau_1$ is the type of the domain and $\mathbb{P}\tau_2$ is the type of the range.
- In TLA$^+$ a record is a special case of a function whose domain is a set of strings (the field names). In B records have their own type $struct(h_1 : \tau_1, \ldots, h_n : \tau_n)$, where $h_1, \ldots, h_n$ are the names of the fields and $\tau_1, \ldots, \tau_n$ the corresponding field types.
- Likewise, tuples are based on functions in TLA$^+$. The domain is the interval from 1 to n, where n is the number of components. We translate tuples as sequences with the type $\mathbb{P}(\mathbb{Z} \times \tau)$. Thereby all components of a tuple must be of the same type τ.

In B only values of the same type are comparable to each other and variables as well as constants can only have one type. To verify these rules a type checking algorithm is required. Moreover we need a type inference algorithm to add missing type declarations to the translated B machine as shown in the example in Sect 2. Type checking and type inference are closely related and can be handled simultaneously.

We use an inference algorithm similar to [9], adapted to the B type system, where we add an extra type u representing an unspecified type. At the beginning each variable and constant have this type. The algorithm is based on the recursive method $eval(e, \epsilon)$, dealing with a TLA$^+$ expression e and an expected type ϵ. Evaluating an expression *eval* is applied recursively to its subexpressions. Moreover *eval* tries to unify the expected type with the type of the expression and returns the resulting type. The expected type of a subexpression is deduced from type informations of the operator calling this subexpression. Type informations of an operator arise from the translation. As an example the TLA$^+$ operator $+$ is translated by the B built-in operator $+$ and its operands are assumed to be integers. There are polymorphic operators such as $=$, which only require that both operands have the same type. In this case the expected type for both operands is u but the resulting types of both sides have to be unified. Due to unification, the *eval* method is only once applied to each (used) expression of the TLA$^+$ module. Moreover, declarations in the configuration file are also taken into account to infer the types of constants. The algorithm fails either if a unification of two types fails or if a variable or constant still has a variable type u (or a type constructor containing u such as $\mathbb{P}u$) at the end of algorithm.

3.2 Translation Rules

In this section we present translation rules for concepts which are different in TLA$^+$ and B, or even missing in B.

In contrast to TLA$^+$, B distinguishes between boolean values and predicates. The difference is already present at the syntactical level. Logical operators such as $\wedge$ or $\vee$ cannot be applied to boolean values. Similarly, variables or constants can not take a predicate as a value. Though, there is a way to convert from a predicate to boolean and vice versa. A predicate can be converted to a boolean value using the *bool* operator. The other way around, we can turn a boolean value into a predicate by comparing it with TRUE. The translation of the TLA$^+$ predicate

$$\text{TRUE} = (\text{TRUE} \vee \text{FALSE})$$

demonstrates both conversions:

$$\text{TRUE} = bool((\text{TRUE} = \text{TRUE}) \vee (\text{FALSE} = \text{TRUE}))$$

The IF/THEN/ELSE construct can be used in variety of ways in TLA$^+$. The two branches can consist of arbitrary expressions with or without primed variables. There is no general way to translate this construct with the IF/THEN/ELSE substitution of B. In order to make a translation to B possible, we first have to restrict both branches to the same type. In case the branches are predicates the construct

$$\text{IF } P \text{ THEN } e_1 \text{ ELSE } e_2$$

can be translated using two implications as

$$(P \Rightarrow e_1) \wedge (\neg(P) \Rightarrow e_2)$$

If e_1 and e_2 are expressions, we cannot use this scheme. Our solution is to create for both branches a lambda function, with respectively e_1 and e_2 as result expression. Moreover we choose TRUE as the sole dummy element of the domains. The "trick" is to add the condition P respectively its negation $\neg(P)$ to the corresponding function. As a consequence, one of the functions is always empty. As already mentioned, B functions are sets and we can apply $\cup$ to combine them (the result is still a function here). To get the value of the IF/THEN/ELSE construct, we just have to call the function with the value TRUE as argument:

$$(\lambda t.(t \in \{\text{TRUE}\} \wedge P|e_1) \cup \lambda t.(t \in \{\text{TRUE}\} \wedge \neg P|e_2))\,(\text{TRUE})$$

Compared to other possible translations, ours has the advantage that the expressions e_1 and e_2 are guarded by P and $\neg P$, i.e., the translation of IF $x = 0$ THEN 1 ELSE $1/x$ is well-defined in B. The translation of the CASE construct is based on the same principle. However, every case is treated as single branch and only one case can be true at the same time.

The *LET* $d \triangleq f$ *IN* e construct allows to define a "local" operator d which can only be used in the expression e. This operator is treated as an ordinary operator and translated with the aid of a B definition; conserving the scope of the operator. In TLA$^+$ operators within different LET/IN constructs could have the same name. We avoid name clashes by adding suffixes to multiply used names.

In TLA$^+$ the CHOOSE operator is used to choose an arbitrary value of a set. The operator works in a deterministic way and chooses always the same value for a given set. It is often combined with a recursive function such as determining the sum of a set. In B there is no way to express the general functionality of the CHOOSE operator[3] and recursive functions are still not (well) supported by B tools. Hence, we developed a way to handle frequently used operators which are based on the CHOOSE operator or on recursive functions. The principle is inspired by the way TLC overrides operators by its Java implementation: we create a new TLA$^+$ standard module (see Figure 3.2) with some useful operators, and during the translation these operators will be overridden by B built-in operators.

```
------------------------------ MODULE TLA2B ------------------------------
EXTENDS Integers, Sequences
MinOfSet(S) ≜ CHOOSE p ∈ S : ∀ n ∈ S : p ≤ n
MaxOfSet(S) ≜ CHOOSE p ∈ S : ∀ n ∈ S : p ≥ n
SetProduct(p) ≜
        LET prod[S ∈ SUBSET Int] ≜
                IF S = {} THEN 1
                 ELSE LET q ≜ CHOOSE pr ∈ S : TRUE
                          IN  q * prod[S \ {q}]
        IN  prod[p]
SetSummation(p) ≜
        LET sum[S ∈ SUBSET Int] ≜
                IF S = {} THEN 0
                 ELSE LET q ≜ CHOOSE pr ∈ S : TRUE
                          IN  q + sum[S \ {q}]
        IN  sum[p]
PermutedSequences(S) ≜
   LET perms[ss ∈ SUBSET S] ≜
        IF ss = {} THEN {⟨⟩}
         ELSE LET ps ≜ [x ∈ ss ↦
                          {Append(sq, x) : sq ∈ perms[ss \ {x}]}]
              IN   UNION {ps[x] : x ∈ ss}
   IN  perms[S]
==========================================================================
```

The concepts of modularization are different in TLA$^+$ and B. In B a machine is a closed system. Indeed, a machine can be included by another machine but variables can only be modified by its operations. As a result, a single machine of a compound system can be verified individually. A TLA$^+$ module does not need to satisfy this property. Hence, we translate a compound of TLA$^+$ modules as a single B machine:

– A module extending another module will be treated as a single module containing declarations and definitions (including local definitions of the

[3] Even though the operator does appear inside mathematical constructions of [1].

extended module) of both modules. Otherwise, there are no further differences in comparison to a translation of a single module.

– The statement

$$I \triangleq \mathit{INSTANCE}\ M\ \mathit{WITH}\ v_M \leftarrow v,\ c_M \leftarrow c$$

allows the specifier to use the definitions of the module M. Thereby, all variables and constants of M have to be overridden by variables and constants (or constant expressions) of the module instantiating M. A definition d_M of Module M can be accessed via $I!d_M$. Also multiple instantiations of the same module are possible. We translate every definition d_M of M as an ordinary definition by only renaming it to I_d_M and by overriding variables and constants as described in the statement.

In Sect. 2 we translated a TLA$^+$ action from Fig. 1 to a B operation in Fig. 2, but we did not exactly define what TLA$^+$ actions are and how they are extracted. An action is defined to be "an ordinary mathematical formula, except that it contains primed as well as unprimed variables"(see p. 16 of [5]). Following this definition we could handle the whole next state relation as a single action. However, this is not advisable, amongst others because of poor user feedback for animation, proof and model checking. Consequently we separate actions with the aid of the disjunction operator. If a disjunction of two actions occurs in a subdefinition of the next state relation, we also will treat them as separate actions unless the subdefinition has no parameter. Parameters indicate that a subdefinition can be used in different variations and multiple times; the translation should not dissolve this structure of a module. In this case the subdefinition is translated with a B definition. The mechanism splitting the next-state relation into separate actions is similar to what TLC does when it pre-processes TLA$^+$ specifications, except that TLC resolves definitions regardless if they have parameters.

A special translation is possible if an action contains an existential quantifier:

$$act \triangleq \exists x \in S : P(x)$$

The bounded variables of the quantification are handled as parameters of the resulting B operation:

$$act_op(x) = \text{ANY} \ldots \text{WHERE } x \in S \land P(x) \text{ THEN} \ldots \text{END}$$

The advantage is that a user can choose a possible value for the parameter x during the animation process (values for x satisfying $P(x)$ are generated by ProB).

4 Implementation and Experiments

The translator is implemented in Java and is called TLA2B. The frontend of TLA2B is based on SANY (cf., Chapter 12 of [5]) for parsing the module and

performing a semantic analysis. Likewise, SANY serves as the frontend of the modelchecker TLC. Moreover, we reuse the configuration file parser of TLC. But the semantic analysis of the configuration file ist different: TLC requires a value for every constant of the corresponding module. In our case, a constant only has to be given a value if the type of the constant cannot be inferred from the module. Otherwise, values of constants can be chosen at a later point in time (ProB infers values for a constant satisfying possible restrictions of the ASSUME clause). TLA2B can handle the clauses SPECIFICATION (temporal description of the specification), INVARIANT (an invariant holding in every state) and overriding of constants and definitions beside the already mentioned INIT (initial state) and NEXT (next state relation). Before inferring and checking types, we conduct a further analysis phase discarding the unused definitions of a module. As an example, temporal definitions are excluded from the translation. The remaining part of the TLA2B consists of implementations of the algorithms described in Sections 2 and 3. Finally, TLA2B creates a B machine file (.mch) containing the translated B machine.

TLA2B has been integrated into ProB as of version 1.3.5: opening a TLA$^+$ module ProB invokes TLA2B to translate the module. As can bee seen in Fig. 3, the TLA$^+$ module is displayed in the editor while ProB runs the translated B machine in the background. The editor offers syntax highlighting and gives an easy way to modify the module.

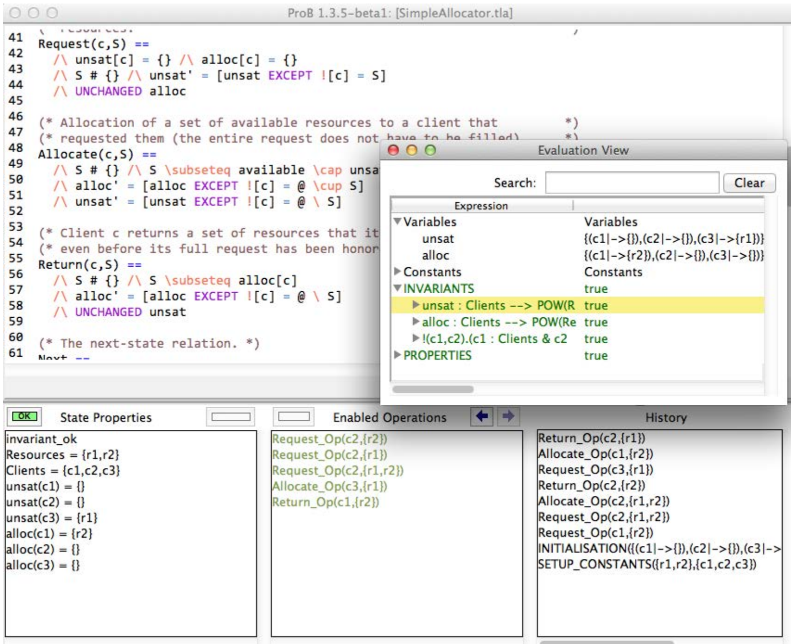

Fig. 3. ProB animator for the SimpleAllocator specification

The following examples show some fields of application of TLA2B in combination with ProB.[4] It is not our intention to present a complete comparison between ProB and TLC. The experiments were all run on a system with Intel Core2 Duo 2 GHz processor, running Windows Vista 32 Bit, TLC2 2.03 and ProB 1.3.5-beta1.

Note that both ProB and TLC support symmetry, but in different ways. In TLC symmetries are provided by the user (e.g., in a configuration file) and are not checked, ProB identifies symmetries over given sets automatically.

SimpleAllocator. As the first example we use the resource allocator case study from [8]. The purpose of the system is to manage a set of resources that are shared among a number of client processes. The first abstract specification of the system is the SimpleAllocator. TLA2B translates the module without the need for any modification (the TLA$^+$ module and the translated B machine are shown in Appendix A). Clients and resources are specified as sets of model values and allow TLC as well as ProB to use symmetry. Table 1 summarises the running times of model checking for TLC and ProB. Without symmetry TLC is superior to ProB, but for larger set sizes ProB's symmetry outperforms TLC. It seems that TLC's symmetry reduction cannot deal well with larger base set sizes and a lot of symmetrical states exist (incidentally, a situation where symmetry reduction could be particularly useful). This is actually to be expected, given the description of the symmetry reduction algorithm in [5]: when a state is added TLC checks for every permutation of it whether it already exists. This is expensive when there are many such permutations.

Table 1. SimpleAllocator: Runtimes of Model Checking (times in seconds)

Clients	Resources	TLC (no symmetry)	TLC (symmetry)	ProB (no symmetry)	ProB (symmetry)
3	2	<1	<1	<2	<1
4	3	28	2	678	8
5	3	450	29	-	28
6	3	>4200	573	-	90

Login. To specifically test this aspect of symmetry reduction, we have written the TLA specification Login which simply allows users to login and logout and deadlocks if all users have logged in. Here, for 9 Users, TLC without symmetry reduction takes 1 second to find the deadlock, but did not terminate within 105 minutes with symmetry enabled. ProB takes 0.73 seconds without symmetry, and 0.04 using hash symmetry reduction [7]. For 21 users, TLC requires 141 seconds to find the deadlock without symmetry, and with symmetry an error message is generated.[5] ProB with hash symmetry takes 0.29 seconds to find

[4] The source code of the examples are available in the technical report at: `http://www.stups.uni-duesseldorf.de/w/Special:Publication/HansenLeuschelTLA2012` .

[5] "Attempted to construct a set with too many elements (>1000000)". This error message already appears with 15 Users.

the deadlock for 21 users. The constraint-based deadlock checking algorithm [4] finds a deadlock in less than 0.01 seconds for 21 users.

SchedulingAllocator. This is an advanced version of the SimpleAllocator from [8]. However, this time a small modification is required to be able to validate the specification using our tool. Indeed, the SchedulingAllocator contains the definition PermSeqs(S) that is based on a recursive function and computes the set of permutation sequences of the set S. To translate this to the B built-in operator, we have to override PermSeqs with the PermutedSequences definition provided by our TLA2B module. For this, we simply have to create a new module MCSchedulingAllocator extending the SchedulingAllocator as well as the TLA2B module and then add the override statement `PermSeqs <- PermutedSequences` to the configuration file. The results of model checking are comparable to the SimpleAllocator specification, and can be found in Table 2.

Table 2. McSchedulingAllocator: Running times of Model Checking (times in seconds)

Clients	Resources	TLC (symmetry)	ProB (symmetry)
3	2	1	2
4	3	70	165
5	3	>3600	1579

Producer-Consumer. Another example is the specification of a multi-threaded program by Charpentier taken from `http://www.cs.unh.edu/%7Echarpov/Teaching/TLA/`. The specification describes a system of threads working on a buffer. In case of a critical ratio between consumer and producer threads, the system can deadlock. After translation ProB reproduces the various deadlocks by model checking. For example, for 11 producers and 10 consumers a deadlock can be reached after 431 steps. Using the AtelierB provers we have also managed to prove the invariant of that model, i.e., that the buffer capacity is never exceeded and that the waitSet only contains valid participants. This required 8 interactive proofs and 5 automatic ones.

Constraint Solving: GraphIso and N-Queens. One of the distinguishing features of ProB is its ability to solve complicated high-level constraints. For example, to find an isomorphism between two graphs (of out-degree exactly one and with nine vertices) , ProB requires less than a second to find all solutions, while TLC requires over two hours to find the first solution.

As another example, consider the well-known N-Queens puzzle. We have experimented with two encodings of the puzzle: one[6] where we use the model checker to search for all valid placements of N queens on an N×N chessboard and a more declarative encoding where we directly write a predicate describing

[6] The specification was written by S. Merz and is included in the TLA$^+$ Tools Distribution.

all valid solutions (i.e., all solutions are generated in single set-builder rather than through an iterative algorithm). As can be seen in Table 3, the model checking approach can only deal with very small values of N. In contrast, ProB can handle values of N up to 13 for the declarative version of N-Queens. Furthermore, when one is interested in only one solution, ProB can, e.g., find a solution for N=50 in less than a second. (Restricting to single solutions does not make much of a performance difference for TLC, however.)

Table 3. Finding *all* solutions for N-Queens (times in seconds)

N	Solutions	N-Queens (imperative)		N-Queens (declarative)	
		TLC	ProB	TLC	ProB
4	2	1	<2	<1	<1
5	10	>3600	>3600	<1	<1
6	4	-	-	1	<1
7	40	-	-	16	<1
8	92	-	-	375	<1
9	352	-	-	2970	<1
10	724	-	-	-	<1
11	2,680	-	-	-	<1
12	14,200	-	-	-	9
13	73,712	-	-	-	41

We have also successfully animated several other existing models from the literature, but several specifications are rejected by TLA2B due to type conflicts or unsupported concepts such as real numbers. In summary, the ProB constraint solving capabilities open up the way to animate and validate new kinds of specifications, which are outside the reach of TLC. TLC on the other hand is extremely valuable when it comes to explicit state model checking for large state spaces. However, ProB's symmetry reduction techniques seem to scale better than TLC's.

5 More Related Work, Discussion and Conclusion

The paper by Mokhtari and Merz [10] presents an animator and model checker for an executable subset of TLA$^+$. The article clearly outlines the needs for an animator for TLA$^+$; unfortunately the tool seems to be no longer to be available.

Mosbahi et al. [11] describe an approach of a translation from B to TLA$^+$. In contrast to our translation they have to deal with concepts which are missing in TLA$^+$ such as partial functions. Moreover their main intention is to let TLC verify liveness properties on the translated TLA$^+$ specification, to overcome the restriction of the B-Method to invariance properties. Otherwise, [12] presents a LTL model checker, implemented inside ProB, that can verify liveness properties. So far, this model checker does not support fairness conditions, but an extension would give us the possibility to enlarge TLA2B to support the temporal part of TLA$^+$.

In terms of features, we also plan to provide for TLA$^+$ the graphical visualization features of ProB already available for B, Z and Event-B. More work on translating various constructs effectively to B, such as the CHOOSE operator or recursive functions, is planned. Another important avenue of further work lies in improving our translation to B. In particular, we aim to generate various B style substitutions such as assignments or IF/THEN/ELSE constructs, rather than generic ANY substitutions. This makes the B translation more readable, but would also lead to noticeable performance improvements with ProB. E.g., in our experiments, this would lead, to a further 20 % runtime improvement for the SimpleAllocator example and up to a factor 2 for other examples.

We would also like to better exploit the symmetry reduction provided by ProB. While the SimpleAllocator, SchedulingAllocator and Login example worked well, the symmetry in the Producer-Consumer example could only be exploited by manually tweaking the B translation. We would like to automate this as much as possible, as it can lead to a considerable performance boost (after tweaking ProB with symmetry requires about a minute to find the 413 step counter example for the Producer-Consumer example; TLC requires more than three and a half hours to to find a deadlock[7]). We are also interested in the correctness of our translation. A formal correctness proof is probably not feasible, but we hope to be able to extensively validate our translation, e.g., by exporting the state space computed by ProB to TLC and use TLC to check that it conforms to the original specification.

In conclusion, we have presented a translation from TLA$^+$ to B, which makes use of a type inference algorithm and effectively translates a large subset of TLA$^+$ to B. The complicated aspects of the translation are linked to the different modularization concepts, as well as to various operators which are missing in B. The translation also identifies operations and parameters within the TLA$^+$ specification formula, in order to make the translation more readable as well as to enable effective application of B tools. In particular, by integrating our translation into the ProB validation tool, we obtain a new tool for TLA$^+$ specifications which is complementary to TLC, providing convenient animation, expression evaluation, constraint solving and improved symmetry reduction. As our experiments show, TLC remains more effective for brute-force explicit state model checking, at least for those specifications which do not require solving complicated constraints. As such it is very useful that both these tools can be applied to TLA$^+$ specifications. The translation itself is also human readable, and we hope that the paper also provides a bridge between the TLA$^+$ and B communities.

Acknowledgements. We are grateful to Daniel Plagge for various discussions and helpful comments that helped in developing the translator. We also would like to thank Leslie Lamport and Stephan Merz for very useful feedback concerning TLA$^+$ and TLC and for giving us access to various specifications. Finally, we are thankful to anonymous referees for their useful feedback.

[7] When trying to use symmetry, the same error message occurs as in the Login example.

References

1. Abrial, J.-R.: The B-Book. Cambridge University Press (1996)
2. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA$^+$ Proof System: Building a Heterogeneous Verification Platform. In: Cavalcanti, A., Deharbe, D., Gaudel, M.-C., Woodcock, J. (eds.) ICTAC 2010. LNCS, vol. 6255, p. 44. Springer, Heidelberg (2010)
3. ClearSy. Atelier B, User and Reference Manuals. Aix-en-Provence, France (2009), `http://www.atelierb.eu/`
4. Hallerstede, S., Leuschel, M.: Constraint-based deadlock checking of high-level specifications. TPLP 11(4-5), 767–782 (2011)
5. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
6. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
7. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. Annals of Mathematics and Artificial Intelligence 59(1), 81–106 (2010)
8. Merz, S.: TLA+ Case Study: A Resource Allocator. Technical Report A04-R-101, INRIA Lorraine - LORIA (2004), `http://hal.inria.fr/inria-00107809`
9. Merz, S., Vanzetto, H.: Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 289–303. Springer, Heidelberg (2012)
10. Mokhtari, Y., Merz, S.: Animating TLA Specifications. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) LPAR 1999. LNCS, vol. 1705, pp. 92–110. Springer, Heidelberg (1999)
11. Mosbahi, O., Jemni, L., Jaray, J.: A formal approach for the development of automated systems. In: Filipe, J., Shishkov, B., Helfert, M. (eds.) ICSOFT (SE), pp. 304–310. INSTICC Press (2007)
12. Plagge, D., Leuschel, M.: Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. STTT 11, 9–21 (2010)
13. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA$^+$ Specifications. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 54–66. Springer, Heidelberg (1999)

A A More Complicated Translation: SimpleAllocator

The configuration file for the model below is as follows:
INIT *Init* NEXT *Next* CONSTANTS $Clients = \{c1, c2, c3\}$ $Resources = \{r1, r2\}$

MODULE *SimpleAllocator*

EXTENDS *FiniteSets*, *TLC*
CONSTANTS *Clients*, *Resources*
ASSUME $IsFiniteSet(Resources)$
VARIABLES *unsat*, *alloc*
$TypeInvariant \triangleq \land\ unsat \in [Clients \rightarrow \text{SUBSET } Resources]$
$\land\ alloc \in [Clients \rightarrow \text{SUBSET } Resources]$
$available \triangleq Resources \setminus (\text{UNION } \{alloc[c] : c \in Clients\})$

$Init \triangleq \ \wedge unsat = [c \in Clients \mapsto \{\}] \wedge alloc = [c \in Clients \mapsto \{\}]$
$Request(c, S) \triangleq \ \wedge unsat[c] = \{\} \wedge alloc[c] = \{\}$
$\quad \wedge S \neq \{\} \wedge unsat' \ = [unsat \text{ EXCEPT } ![c] = S] \wedge \text{UNCHANGED } alloc$
$Allocate(c, S) \triangleq \ \wedge S \neq \{\} \wedge S \subseteq available \cap unsat[c]$
$\quad \wedge alloc' = [alloc \text{ EXCEPT } ![c] = @ \cup S] \wedge unsat' = [unsat \text{ EXCEPT } ![c] = @ \setminus S]$
$Return(c, S) \triangleq \ \wedge S \neq \{\} \wedge S \subseteq alloc[c]$
$\quad \wedge alloc' = [alloc \text{ EXCEPT } ![c] = @ \setminus S] \wedge \text{UNCHANGED } unsat$
$Next \triangleq \exists c \in Clients, S \in \text{SUBSET } Resources :$
$\quad Request(c, S) \vee Allocate(c, S) \vee Return(c, S)$

MACHINE SimpleAllocator
SETS $\text{ENUM}_1 = \{r1, r2\}$; $\text{ENUM}_2 = \{c1, c2, c3\}$
CONSTANTS Clients, Resources
PROPERTIES Clients = ENUM_2 ∧ Resources = ENUM_1
∧ ∃seq_.(seq_ ∈ seq(Resources) ∧ ∀s.(s ∈ Resources ⇒
∃n.(n ∈ 1 .. size(seq_) ∧ seq_(n) = s)))
DEFINITIONS
TypeInvariant == unsat ∈ Clients → ℙ(Resources)
∧ alloc ∈ Clients → ℙ(Resources);
available == Resources - union(t_|∃c.(c ∈ Clients ∧ t_ = alloc(c)));
Init == unsat = λc.(c ∈ Clients| {}) ∧ alloc = λc.(c ∈ Clients| {});
Request(c,S) == unsat(c) = {} ∧ alloc(c) = {}
∧ (S ≠ {} ∧ unsat_n = unsat ⩤ {c ↦ S});
Allocate(c,S) == S ≠ {} ∧ S ⊆ available ∩ unsat(c)
∧ alloc_n = alloc ⩤ {c ↦ (alloc(c) ∪ S)}
∧ unsat_n = unsat ⩤ {c ↦ (unsat(c) - S)};
Return(c,S) == S ≠ {} ∧ S ⊆ alloc(c) ∧ alloc_n = alloc ⩤ {c ↦ (alloc(c) - S)};
ResourceMutex == ∀c1,c2.(c1 ∈ Clients ∧ c2 ∈ Clients ⇒
(c1 ≠ c2 ⇒ alloc(c1) ∧ alloc(c2) = {}));
VARIABLES unsat, alloc
INVARIANT unsat ∈ $\mathbb{P}(\text{ENUM}_2 \times \mathbb{P}(\text{ENUM}_1))$
∧ alloc ∈ $\mathbb{P}(\text{ENUM}_2 \times \mathbb{P}(\text{ENUM}_1))$ ∧ TypeInvariant ∧ ResourceMutex
INITIALISATION unsat, alloc ∈ (Init)
OPERATIONS
Request_Op(c, S) = ANY unsat_n
WHERE c ∈ Clients ∧ S ∈ ℙ(Resources) ∧ Request(c, S)
THEN unsat := unsat_n END;
Allocate_Op(c, S) = ANY unsat_n, alloc_n
WHERE c ∈ Clients ∧ S ∈ ℙ(Resources) ∧ Allocate(c, S)
THEN unsat, alloc := unsat_n, alloc_n END;
Return_Op(c, S) = ANY alloc_n
WHERE c ∈ Clients ∧ S ∈ ℙ(Resources) ∧ Return(c, S)
THEN alloc := alloc_n END;
END