# Translating B to TLA$^+$ for Validation with TLC: There and Back Again

**(Technical Report)**

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.** The state-based formal methods B and TLA$^+$ share the common base of predicate logic, arithmetic and set theory. However, there are still considerable differences, such as the way to specify state transitions, the different approaches to typing, and the available tool support. In this paper, we present a translation from B to TLA$^+$ to validate B specifications using the model checker TLC. We provide translation rules for almost all constructs of B, in particular for those which are missing in TLA$^+$. The translation also includes many adaptions and optimizations to allow efficient checking by TLC. Moreover, we present a way to validate liveness properties for B specifications under fairness conditions. Our implemented translator, TLC4B, automatically translates a B specification to TLA$^+$, invokes the model checker TLC, and translates the results back to B. We use ProB to double check the counter examples produced by TLC and replay them in the ProB animator. We also present a series of case studies and benchmark tests comparing TLC4B and ProB.

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

## 1 Introduction and Motivation

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

- B is strongly typed, while TLA$^+$ is untyped. For the translation it is obviously easier to translate from a typed to an untyped language than vice versa.
- The concepts of modularization are quite different.
- Functions in TLA$^+$ are total, while B supports relations, partial functions, injections, bijections, etc.
- TLA$^+$ has several constructs which are absent in B, such as an IF/THEN/ELSE for expressions and predicates[1] or the CHOOSE operator.

[**] 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.

– B is limited to invariance properties, while TLA$^+$ also allows the specification of liveness properties.

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]. TLC has been used to validate a variety of distributed algorithms (e.g., [4]) and protocols. B has extensive proof support, e.g., in the form of the commercial product AtelierB [3] and the animator, constraint solver and model checker ProB [9, 10]. Both AtelierB and ProB are being used by companies, mainly in the railway sector for safety critical control software. In an earlier work [5] we have presented a translation from TLA$^+$ to B, which enabled applying the ProB tool to TLA$^+$ specifications. In this paper we present a translation from B to TLA$^+$, this time with the main goal of applying the model checker TLC to B specifications. Indeed, TLC is a very efficient model checker for TLA$^+$ with an efficient disk-based algorithm and support for fairness. ProB has an LTL model checker, but it does not support fairness (yet) and is entirely RAM-based. The model checking core of ProB is less tuned than TLA$^+$. On the other hand, ProB incorporates a constraint solver and offers several features which are absent from TLC, notably an interactive animator with various visualization options. Our approach is thus to replay the counter-examples produced by TLC within ProB to get access to those features and 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 the constraint solving capabilities of ProB lead to improved performance. The addition of a lower-level model checker thus opens up many new application possibilities.

## 2 Translation

The complete translation process from B to TLA$^+$ and back to B is illustrated in Fig. 1. 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 (e.g., about the parsing process) will be covered in Sect. 4.

### 2.1 Example

Below we use a specification (adapted from [10]) of a process scheduler (Fig. 2). The specified system allows at most one active process at the same time. 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 a (injective) sequence of processes. In the initial state no process is created and the queue is empty. Moreover, the specification has various operations to create (*new*), delete (*del*), or add a process to queue (*addToQueue*). Additionally,

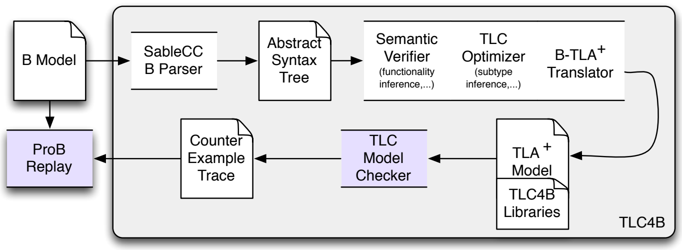


**Fig. 1.** The TLC4B Translation and Validation Process

there are two operations to load a process into the processor (*enter*) and to remove a process from the processor (*leave*). The specification contains two safety conditions beside the typing predicates of the variables:

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

The translated TLA$^+$- specification is shown in Fig. 3. (We will explain later how this translation is computed.) At the beginning of the module some standard modules are loaded via the *EXTEND* statement. These modules contain several operators used in this specification. The invariant of the B specification is divided into several definitions in the TLA$^+$ module. This enables TLC to provide better feedback about which part of the invariant is violated. A TLA$^+$ action is created for each B operation. Substitutions are translated as before-after predicates where a primed variable represents the variable in the next state. Unchanged variables must be explicitly specified. In TLA$^+$ an action is a conjunction of a precondition and before-after predicate. 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* which is a disjunction of all actions.

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.[2] In this case we assign a set[3] 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] We translate a constant as variable if it can have several values. All values will be enumerated in the initialization and the variable will be kept unchanged in all actions.
[3] The size of the set is a default number or can be specified by the user.

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

**Fig. 2.** MODEL Scheduler

## 2.2 Translating Data Values and Functionality Inference

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

A missing data type in TLA$^+$ Relations are[4], but TLA$^+$ provides all necessary data types to define relations based on the model of the B-Method. We represent a relation in TLA$^+$ as a set of tuples (e.g. $\{\langle 1, \mathit{TRUE}\rangle, \langle 1, \mathit{FALSE}\rangle\ \langle 2, \mathit{TRUE}\rangle\}$). The drawback of this approach is that in contrast to B, TLA$^+$'s own functions and sequences are not based on the relations defined is this way. As an example, we cannot specify a function as a set of pairs in TLA$^+$; in B it is usual to do this as well as to apply set operators (e.g. the union operator as in $r \cup \{2 \mapsto 3\}$) 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.

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

MODULE *Scheduler*

EXTENDS *Sequences*, *Relations*, *Functions*, *FunctionsAsRelations*, *SequencesExtended*
CONSTANTS *PROCESSES*, *idle*, *ready*, *active*
VARIABLES *state*, *queue*
$STATES \triangleq \{idle, ready, active\}$

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

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

$new(p) \triangleq p \notin RelDomain(state)$
$\quad \land state' = state \cup \{\langle p, idle\rangle\} \land \text{UNCHANGED } \langle queue\rangle$

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

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

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

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

$Next \triangleq \lor \exists p \in PROCESSES : new(p)$
$\quad \lor \exists p \in RelDomain(state) : del(p)$
$\quad \lor \exists p \in RelDomain(state) : addToQueue(p)$
$\quad \lor enter$
$\quad \lor \exists p \in RelDomain(state) : leave(p)$

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

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

**Fig. 3.** Module Scheduler

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

**Fig. 4.** Configuration file for module Scheduler

We use a type inference algorithm adapted to the extend 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 keeping a function type if they are applied to operands with a function type (e.g. *ran*, *first* or *tail*). For these operators we have to deliver two translation rules (functional vs relational).[5] 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 which can be included at the top of a specification.[6] 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 function types (they are not part of the B type system) which are missing in TLA$^+$. A function type is a combination of partial/total and injective/surjective/bijective. In TLA$^+$ we only have total functions. We group all missing functional operators together in an additional module (Fig. 6).

Some operators exists in both languages but their definitions differs 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* | $a \in \mathbb{N} \wedge b \in \mathbb{N}_1$ | $a \in \mathbb{Z} \wedge b \in \mathbb{N}_1$ |

To verify B's well-definedness condition for modulo we use TLC's ability to check assertions. The special operator $Assert(P, out)$ throws a runtime exception with the

[5] For various reasons we do not redefine TLA$^+$ built-in operators e.g. set operators.

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

MODULE *Relations*

$\textsc{extends}\ \mathit{FiniteSets},\ \mathit{Naturals},\ \mathit{TLC}$
$\mathit{Relation}(X,\ Y) \triangleq \textsc{subset}\ (X \times Y)$
$\mathit{RelDomain}(R) \triangleq \{x[1] : x \in R\}$
$\mathit{RelRange}(R) \triangleq \{x[2] : x \in R\}$
$\mathit{RelInverse}(R) \triangleq \{\langle x[2],\ x[1]\rangle : x \in R\}$
$\mathit{RelDomRes}(S,\ R) \triangleq \{x \in R : x[1] \in S\}$ Domain restriction
$\mathit{RelDomSub}(S,\ R) \triangleq \{x \in R : x[1] \notin S\}$ Domain subtraction
$\mathit{RelImage}(R,\ S) \triangleq \{y[2] : y \in \{x \in R : x[1] \in S\}\}$
$\mathit{RelOverride}(R1,\ R2) \triangleq \{x\ \in R : x[1] \notin \mathit{RelDomain}(R2)\} \cup R2$
$\mathit{RelComposition}(R1,\ R2) \triangleq \{\langle u[1][1],\ u[2][2]\rangle : u \in$
 $\{x \in \mathit{RelRanRes}(R1,\ \mathit{RelDomain}(R2)) \times \mathit{RelDomRes}(\mathit{RelRange}(R1),\ R2) :$
  $x[1][2] = x[2][1]\}\}$
$\vdots$

**Fig. 5.** Module Relations

MODULE *Functions*

$\textsc{extends}\ \mathit{FiniteSets}$
$\mathit{Range}(f) \triangleq \{f[x] : x \in \textsc{domain}\ f\}$
$\mathit{Image}(f,\ S) \triangleq \{f[x] : x \in S\}$
$\mathit{TotalInjFunc}(S,\ T) \triangleq \{f \in [S \rightarrow T] :$
 $\mathit{Cardinality}(\textsc{domain}\ f) = \mathit{Cardinality}(\mathit{Range}(f))\}$
$\mathit{ParFunc}(S,\ T) \triangleq \textsc{union}\ \{[x \rightarrow T] : x \in \textsc{subset}\ S\}$
$\mathit{ParInjFunc}(S,\ T) \triangleq \{f \in \mathit{ParFunc}(S,\ T) :$
 $\mathit{Cardinality}(\textsc{domain}\ f) = \mathit{Cardinality}(\mathit{Range}(f))\}$
$\vdots$

**Fig. 6.** Module Functions

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:

$$Modulo(a, b) \widehat{=} IF\ Assert(a \geq 0, "ERROR")\, THEN\ a\ \%\ b\ ELSE\ 0$$

The else clause will never reached because a runtime exception is thrown.

We also have to consider well-definedness conditions if we apply a function call to a relation as happened in the example translation (Sect. 2.1):

$$\begin{aligned}\mathit{RelCall}(r,\ x) \triangleq\ & \textsc{if}\ \mathit{Cardinality}(r) = \mathit{Cardinality}(\mathit{RelDom}(r)) \land x \in \mathit{RelDom}(r)\\ & \textsc{then}\ (\textsc{choose}\ y \in r : y[1] = x)[2]\\ & \textsc{else}\ \ \mathit{Assert}(\mathit{FALSE},\ \text{“ERROR”})\end{aligned}$$

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

– Relations (Sect. A.8)

– Functions (Sect. A.6)
– SequencesExtended (Sect. A.7)
– FunctionsAsRelations (Sect. A.9)
– SequencesAsRelations (Sect. A.10)
– BBuiltins (Sect. A.4)

### 2.4 Adaptions & Subtype Inference & TLC Optimizations

Firstly we will describe how TLC evaluates expressions: In general TLC evaluates an expression from left to right. Evaluating an expression containing a bounded 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 bounded 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 writing a existential quantification in B ($\exists x.(P)$). But we confine our translations to the subset of TLA$^+$ which is supported by TLC. Thus the translation is responsible for making all required adaptions to deliver an executable TLA$^+$ specification. For the existential quantification (or all other expressions containing bounded variables), we use the inferred type $\tau$ of the bounded variable as the associated set ($\exists x \in \tau_x : P$.) However, it is easy to see that in some cases it is not a good idea to enumerate over a type of a variable especially if the type is a infinite set. Alternatively, it is often possible to restrict the type of the bounded variable based on a static analyses of the corresponding (typing) predicate. We use a pattern matching algorithm to find the following kind of expressions where x is a bounded variable, $e$ is an expression, and S is ideally a subset of the type[7]: $x = e$ , $x \in S$, $x \subseteq S$ or $x \subset S$.

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

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

This reduces the number of times TLC has to evaluate the predicate $P$. [8]

Sometimes TLC can use heuristics to evaluate an expression. For example TLC can evaluate $\langle 1, 2, 1 \rangle \in \mathit{Seq}(\{1, 2\})$ 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 should we translate the set of all total functions ($S \rightarrow T$)? The easiest way is to convert each function to a relation in TLA$^+$:

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

[7] The B language description in [3] requires that each (bounded) variable must be typed by one these pattern before use.

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

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

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

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

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

In this case, TLC only checks if a function is a subset of the cartesian product (the whole Cartesian product will not be evaluated) and the conditions are checked only once. The advantage of the first definition is that it is faster to evaluate the whole set. 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).

## 3 Checking Temporal Formulas

One of the main advantages of TLA$^+$ is that temporal properties can be specified directly in the language itself. Moreover the model checker TLC can be used to verify such formulas. But before we show how to write temporal formulas for a B specification we first have to describe a main distinction between both formal methods. In contrast to B, TLA$^+$ allows stuttering steps at any time.[9] This means that a regular step of a TLA$^+$ specification is either a step satisfying one of the actions or an 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 $c$

$$Spec \mathrel{\widehat{=}} c = 1 \land \Box[c' = c + 1]_c$$

We would expect that the counter will eventually reach 10 ($\Diamond(c = 10)$). However TLC will report a counter-example, saying that at a certain state (before reaching 10), a infinite number of stuttering will occur and 10 will never reached. From the B site we do not want to care about these stuttering steps. TLA$^+$ allows the adding of fairness conditions to the specification to avoid infinite stuttering steps. Adding weak fairness for the next-state relation ($WF(Next)$) would prohibit

[9] $[Next]_{vars} \equiv Next \lor UNCHANGED\ vars$

a infinite number of stuttering steps if a step of the next-state relation is possible (i.e. *Next* is always enabled):

$$\begin{aligned} WF(A) \mathrel{\widehat{=}} & \lor \Box\Diamond(\langle A\rangle_{vars}) \\ & \lor \Box\Diamond(\neg ENABLED(A)) \end{aligned}$$

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

$$\langle A\rangle_{vars} \equiv A \land vars' \neq vars$$

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

$$\begin{aligned} VWF(A) \mathrel{\widehat{=}} & \lor \Box\Diamond(\langle A\rangle_{vars}) \\ & \lor \Box\Diamond(\neg ENABLED(A) \\ & \lor \Box\Diamond(ENABLED(A \land UNCHANGED\ vars)) \end{aligned}$$

The definition of VWF is identical to WF except for an additional third case allowing infinite stuttering steps if $A$ is a stuttering action ($A \land UNCHANGED\ vars$). [10] We define the resulting template of the translated TLA$^+$ specification as follows:

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

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)
– $\exists x.(P \land f)$ (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"[11])
– $\neg$, $\land$, $\lor$, $\Rightarrow$ (negation, conjunction, disjunction and implication)

[10] However VWF(A) only says that infinite "unchanging" A steps are possible. It does not say that they will occur infinitely often. In TLA$^+$ it is not possible to express that.

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

**Liveness conditions in B** Since temporal operators are not part of the B language we insert liveness conditions (represented in string format) as a B definitions into the specification. As an example we revisit the scheduler specification from Sect.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 these 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:

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

The translation of the liveness condition is almost simple:

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

All temporal operators can be directly mapped to the TLA$^+$ operators. Only 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 bounded 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. Finally, we have to add an entry in configuration file ($\mathit{PROPERTY}\ \mathit{ASSERT\_LTL\_1}$) telling TLC to verify this liveness property.

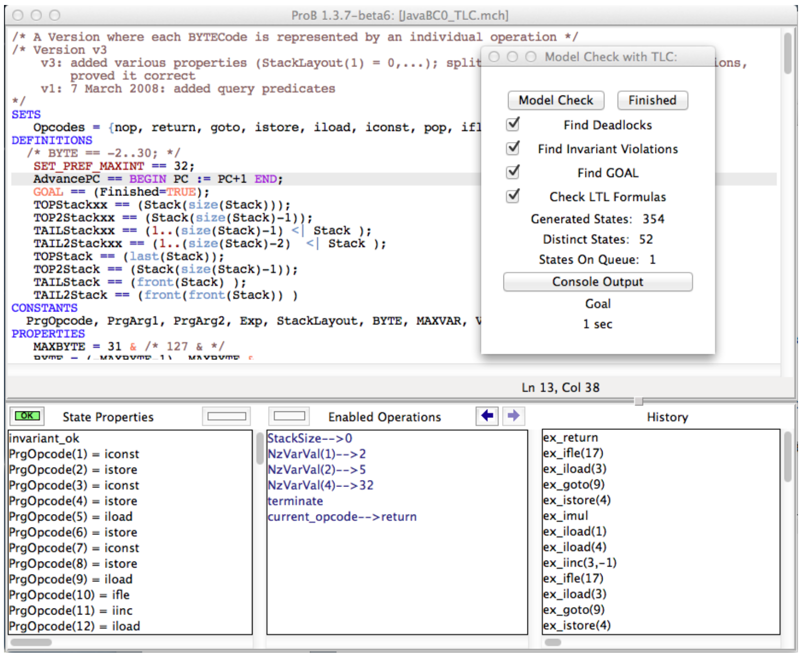
## 4 Implementation & Experiments

Our translator, called TLC4B, is implemented in Java and it took about six months to develop the initial version. Figure 1 in Sect. 2 shows the translation and validation process of TLC4B. After parsing the specification TLC4B performs some static analyses (e.g. type checking or checking the scope of the variables) verifying the semantic correctness of the B specification. Moreover, as explained in Sect. 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 a associated configuration file and invokes the model checker TLC. We expect TLC to find the following kinds of errors in the B specification:

– Deadlocks
– Invariant violations

– Assertion errors
– Goal found (a desired state is reached)
– Properties violations (i.e., axioms over the B constants are false)
– Well-definedness violations
– Temporal formulas violations.

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. In some cases, TLC reports a trace leading to the state where the error (e.g. deadlock or invariant violation) occur. A trace is a sequence of states where each state is a mapping from variables to values. TLC4B translates the trace back to B. TLC4B has been integrated into ProB as of version 1.3.7-beta: The user needs no knowledge of TLA$^+$ because the translation is completely hidden. Counter-examples found by TLC are automatically replayed in the ProB animator to give the user an optimal feedback. As shown in figure 7 counter-examples found by TLC are automatically replayed in the ProB animator (displayed in the history pane) to give the user an optimal feedback.



```
/* A Version where each BYTECode is represented by an individual operation */
/* Version v3
    v3: added various properties (StackLayout(1) = 0,...); split
        proved it correct
    v1: 7 March 2008: added query predicates
*/
SETS
   Opcodes = {nop, return, goto, istore, iload, iconst, pop, ifl
DEFINITIONS
  /* BYTE == -2..30; */
   SET_PREF_MAXINT == 32;
   AdvancePC == BEGIN PC := PC+1 END;
   GOAL == (Finished=TRUE);
   TOPStackxx == (Stack(size(Stack)));
   TOP2Stackxx == (Stack(size(Stack)-1));
   TAILStackxx == (1..(size(Stack)-1) <| Stack );
   TAIL2Stackxx == (1..(size(Stack)-2)  <| Stack );
   TOPStack == (last(Stack));
   TOP2Stack == (Stack(size(Stack)-1));
   TAILStack == (front(Stack) );
   TAIL2Stack == (front(front(Stack)) )
CONSTANTS
  PrgOpcode, PrgArg1, PrgArg2, Exp, StackLayout, BYTE, MAXVAR, V
PROPERTIES
   MAXBYTE = 31 & /* 127 & */
```

✓ Find Deadlocks
✓ Find Invariant Violations
✓ Find GOAL
✓ Check LTL Formulas
Generated States: 354
Distinct States: 52
States On Queue: 1
Goal
1 sec

Ln 13, Col 38

State Properties:

```
invariant_ok
PrgOpcode(1) = iconst
PrgOpcode(2) = istore
PrgOpcode(3) = iconst
PrgOpcode(4) = istore
PrgOpcode(5) = iload
PrgOpcode(6) = istore
PrgOpcode(7) = iconst
PrgOpcode(8) = istore
PrgOpcode(9) = iload
PrgOpcode(10) = ifle
PrgOpcode(11) = iinc
PrgOpcode(12) = iload
```

Enabled Operations:

```
StackSize-->0
NzVarVal(1)-->2
NzVarVal(2)-->5
NzVarVal(4)-->32
terminate
current_opcode-->return
```

History:

```
ex_return
ex_ifle(17)
ex_iload(3)
ex_goto(9)
ex_istore(4)
ex_imul
ex_iload(1)
ex_iload(4)
ex_iinc(3,-1)
ex_ifle(17)
ex_iload(3)
ex_goto(9)
ex_istore(4)
```

**Fig. 7.** ProB animator

The following examples show some fields of application of TLC4B. The experiments were all run on a Macbook Air with Intel Core i5 1,8 GHz processor, running

TLC Version 2.05 and Prob version 1.3.7-beta9. The full details about the examples can be found in the extended version of our paper [6].

**Can-Bus** As the first example we use a specification of the Can-Bus. The specification contains 314 lines B code, 18 variables and 21 operations. The specification is on a low level, i.e. the operations consists of simple assignments of concrete values to variables (no constraint solving is required). TLC4B needs 1.5 seconds[12] to translate the specification to TLA$^+$ and less than 6 seconds for the validation of the complete state space composed of 132,598 states. ProB needs 192 seconds to visit the same number of states. Both model checkers report no errors. For this specification TLC benefits from its efficient algorithm for storing big state spaces.

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

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

We have successfully validated several existing models from the literature (Fig. 8). In summary, ProB is substantially better than TLC4B when constraint solving is required (NQueens, SumAndProduct, GraphIsomorphism[13]) or when naive enumeration of operation arguments is inefficient (GardnerSwitchingPuzzle). For some specifications (not listened in the table) TLC was not able to validate the translated TLA$^+$ specification because TLC had to enumerate a infinite set (e.g. $\exists x \in \mathit{Int} : x + x = p$). On the other hand, TLC4B is substantially better than ProB for lower-level specifications with a large state space.

[12] Mainly the time is needed to start the JVM and to parse the B specification.

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

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

(1) Without Deadlock Check

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

## 5 Correctness of the Translation

There are several possible cases where our validation of B models using TLC could be unsound: there could be a bug in TLC, there could be a bug in our TLA$^+$ library for the B operators, there could be a bug in our implementation of the translation from B to TLA$^+$, there could be a fundamental flaw in our translation (e.g., related to subtle issues such as well-definedness).

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 does not eliminate the possibility that ProB has a bug which prevents detection of an unsound counter example, but this makes it very unlikely. Indeed, ProB, TLC, and our translator have been developed completely independently of each other and rely on different technology.

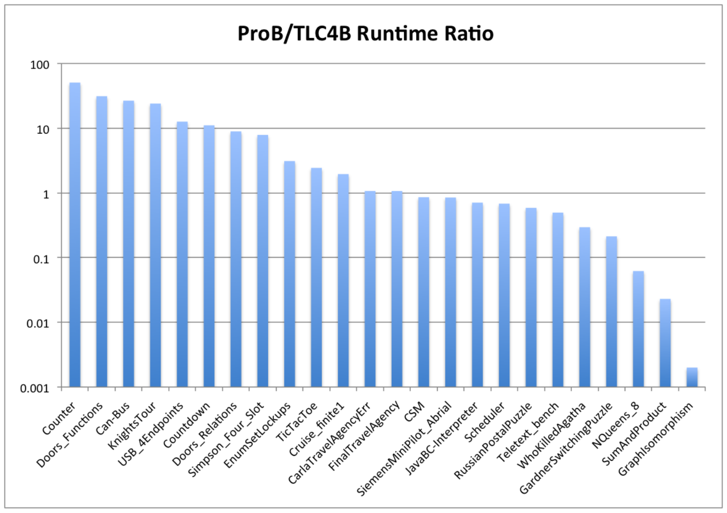


**Fig. 9.** Empirical Results: Ratio of running ProB vs TLC4B

Such an independent double chain is often state-of-the-art in industry for safety critical developments and is, for example, employed for code generation.

The more tricky case is when TLC finds no counter example and claims to have checked the full state space. Here we have the additional 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). Validating specifications containing mathematical laws have proven to be very useful to detect bugs in our translation and libraries (mainly bugs involving operator precedences). In addition, we have uncovered a bug in TLC relating to the cartesian product.[14] Moreover, we use a wide variety of benchmarks, checking that ProB and TLC producing the same result and generate the same number of states.

## 6 More Related Work, Discussion and Conclusion

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

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

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

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

– Providing better feedback to the user when TLC can not validate a translated specification (e.g. if TLC has to enumerate a infinite set).
– Extending our static analyses to make some specifications executable by TLC which are currently not supported.
– Supporting modularization and refinement techniques of B.[15]

The experimental results imply that it would be suitable to apply TLC4B to more low level refinement specifications. Normally, the state space increases and the needed constraint solving abilities decrease during a refinement process. We are also interested in further strategies to test the correctness of our translation. A formal correctness proof is probably not feasible, but a strong point of our approach is the replaying of counter examples using ProB. In addition, we plan to re-translate the TLA$^+$ specification back to B using our TLA2B translator [5] and comparing the state spaces with ProB. Indeed, we have now constructed a two-way bridge between TLA$^+$ and B, and also hope that this will bring both communities closer together.

In conclusion, by making TLC available to B models, we have closed a gap in the tool support and now have a range of complimentary 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.

**Acknowledgements** We are grateful to Ivaylo Dobrikov for various discussions and support. We also would like to thank Leslie Lamport and Stephan Merz for very useful feedback concerning TLA$^+$ and TLC.

[15] ProB is able to transform a compound of models to a single model which can be validated by TLC4B. However our approach is to support modularization independent from ProB.

## A Translation rules

### A.1 Operations

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| Op = Sub | $Op \triangleq Sub \land$ UNCHANGED $vars$ | $Op \triangleq Sub \land$ UNCHANGED $vars$ |
| Op(p) = Sub | - | $Op \triangleq \exists p \in \tau : Sub \land$ UNCHANGED $vars$ |
| q $\leftarrow$ Op = Sub | - | $Op \triangleq \exists q \in \tau : Sub \land$ UNCHANGED $vars$ |

### A.2 Substitutionen

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| BEGIN Sub END | $Sub$ | $Sub$ |
| SELECT P THEN Sub END | $P \land Sub$ | $P \land Sub$ |
| ANY t WHERE P THEN Sub END | $\exists t : P \land Sub$ | $\exists t \in \tau_p : P \land Sub$ |
| PRE P THEN Sub END | - | $P \land Sub$ |
| skip | - | UNCHANGED $vars$ |
| v := e | $v' = e$ | $v' = e$ |
| Sub$_1$ $\parallel$ Sub$_2$ | $Sub_1 \land Sub_2$ | $Sub_1 \land Sub_2$ |
| ASSERT P THEN Sub END | - | $P \land Sub$ |
| CHOICE Sub$_1$ OR Sub$_1$ END | - | $Sub_1 \lor Sub_2$ |
| IF P THEN Sub END | - | $P \land Sub$ |
| IF P THEN Sub$_1$ ELSE Sub$_2$ END | - | IF $P$ THEN $Sub_1$ ELSE $Sub_2$ |
| IF P$_1$ THEN Sub$_1$ <br> ELSIF P$_2$ THEN Sub$_2$ <br> ELSE Sub$_3$ | - | IF $P_1$ THEN $Sub_1$ <br> ELSE IF $P_2$ THEN $Sub_2$ <br> ELSE $Sub_3$ |
| CASE e OF <br> EITHER e$_{11}$,…,e$_{1n}$ THEN Sub$_1$ <br> OR e$_{21}$,…,e$_{2n}$ THEN Sub$_2$ <br> ELSE Sub$_3$ END END | - | CASE <br> $e_{11} = e \lor \ldots \lor e_{1n} = e \rightarrow Sub_1 \Box$ <br> $e_{21} = e \lor \ldots \lor e_{2n} = e \rightarrow Sub_2 \Box$ <br> OTHER $\rightarrow Sub_3$ |
| LET t$_1$,…, t$_n$ <br> BE t$_1$ = e$_1$, …, t$_n$ = e$_n$ <br> IN Sub END | - | $\exists\ t_1 \in \{e_1\}, \ldots, t_n \in \{e_n\} : Sub$ |
| v :$\in$ S | $v' \in S$ | $v' \in S$ |
| v:(P(v\$0, v)) | $P(v, v')$ | $P(v, v')$ |
| $f(e_1) := e_2$ | $f' = [f\ EXCEPT\ ![e_1] = e_2]$ | $f' = FuncAssign(f, e_1, e_2)$ [(1)] |

[(1)] Defined by the Functions module.

### A.3 Logic

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| $P \wedge Q$ | $P \wedge Q$ | $P \wedge Q$ |
| $P \vee Q$ | $P \vee Q$ | $P \vee Q$ |
| $P \Rightarrow Q$ | $P \Rightarrow Q$ | $P \Rightarrow Q$ |
| $P \Leftrightarrow Q$ | $P \Leftrightarrow Q$ | $P \Leftrightarrow Q$ |
| $\neg P$ | $\neg P$ | $\neg P$ |
| $bool(P)$ | - | $P$ |
| $\forall x.(P \Rightarrow Q)$ | $\forall x : P \Rightarrow Q$ | $\forall x \in \tau_x : P \Rightarrow Q$ |
| $\exists x.(P \wedge Q)$ | $\exists x : P \wedge Q$ | $\exists x \in \tau_x : P \wedge Q$ |

### A.4 Sets

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| $\{\}$ | $\{\}$ | $\{\}$ |
| $\{e_1, \ldots, e_n\}$ | $\{e_1, \ldots, e_n\}$ | $\{e_1, \ldots, e_n\}$ |
| $\{x \| P\}$ | - | $\{x \in \tau_x \| P\}$ |
| $\{x \| x : S \wedge P\}$ | $\{x \in S \| P\}$ | $\{x \in S \| P\}$ |
| $\mathbb{P}(S)$ | $SUBSET\ S$ | $SUBSET\ S$ |
| $\mathbb{P}_1(S)$ | - | $Pow1(S)$ (1) |
| $FIN(S)$ | - | $Fin(S)$ (1) |
| $FIN_1(S)$ | - | $Fin1(S)$ (1) |
| $card(S)$ | - | $Cardinality(S)$ |
| $S \times T$ | $S \times T$ | $S \times T$ |
| $S \cup T$ | - | $S \cup T$ |
| $S \cap T$ | - | $S \cup T$ |
| $S - T$ | - | $S \setminus T$ |
| $S \in T$ | - | $S \in T$ |
| $S \notin T$ | - | $S \notin T$ |
| $S \subseteq T$ | - | $S \subseteq T$ |
| $S \nsubseteq T$ | - | $NotStrictSubset(S, T)$ (1) |
| $S \subset T$ | - | $S \subset T$ (1) |
| $S \not\subset T$ | - | $NotSubset(S, T)$ (1) |
| $union(S)$ | UNION$(S)$ | UNION$(S)$ |
| $inter(S)$ | - | $Inter(S)$ (1) |
| $\bigcup x.(P \| S)$ | - | UNION $(\{S : x \in \{y \in \tau_x : P\})$ |
| $\bigcap x.(P \| S)$ | - | $Inter(\{S : x \in \{y \in \tau_x : P\})$ (1) |

(1) Defined by the BBuiltins module.

```
──────────────── MODULE BBuiltIns ────────────────
EXTENDS $Integers$, $FiniteSets$, $TLC$

$Max(S) \triangleq$ CHOOSE $x \in S : \forall p \in S : x \geq p$
  The largest element of the set $S$

$Min(S) \triangleq$ CHOOSE $x \in S : \forall p \in S : x \leq p$
  The smallest element of the set $S$

$succ[x \in Int] \triangleq x + 1$
  The successor function

$pred[x \in Int] \triangleq x - 1$
  The predecessor function

RECURSIVE $Sigma(\_)$
$Sigma(S) \triangleq$ LET $e \triangleq$ CHOOSE $e \in S :$ TRUE
                 IN  IF  $S = \{\}$ THEN 0 ELSE  $e[2] + Sigma(S \setminus \{e\})$
  The sum of all second components of pairs which are elements of $S$

RECURSIVE $Pi(\_)$
$Pi(S) \triangleq$ LET $e \triangleq$ CHOOSE $e \in S :$ TRUE
            IN  IF  $S = \{\}$ THEN 0 ELSE  $e[2] + Pi(S \setminus \{e\})$
  The product of all second components of pairs which are elements of $S$

$Pow1(S) \triangleq$ (SUBSET $S) \setminus \{\{\}\}$
  The set of non-empty subsets

$Fin(S) \triangleq \{x \in$ SUBSET $S : IsFiniteSet(x)\}$
  The set of all finite subsets.

$Fin1(S) \triangleq \{x \in$ SUBSET $S : IsFiniteSet(x) \land x \neq \{\}\}$
  The set of all non-empty finite subsets

$S \subset T \triangleq S \subseteq T \land S \neq T$
  The predicate becomes true if $S$ is a strict subset of $T$

$NotSubset(S, T) \triangleq \neg(S \subseteq T)$
  The predicate becomes true if $S$ is not a subset of $T$

$NotStrictSubset(S, T) \triangleq \neg(S \subset T)$
   The predicate becomes true if $S$ is not a strict subset of $T$

RECURSIVE $Inter(\_)$
$Inter(S) \triangleq$ IF $S = \{\}$
                THEN $Assert$(FALSE, "Error: Applied the inter operator to an empty set.")
                ELSE  LET $e \triangleq$ (CHOOSE $e \in S :$ TRUE)
                        IN  IF  $Cardinality(S) = 1$
                             THEN $e$
```

ELSE $e \cap \mathit{Inter}(S \setminus \{e\})$

The intersection of all elements of $S$.

## A.5 Numbers

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| $\mathit{NATURALS}$ | - | $\mathit{Nat}$ [(1)] |
| $\mathit{INTEGER}$ | - | $\mathit{Int}$ [(2)] |
| $\mathit{INT}$ | - | $\mathit{MinInt}..\mathit{MaxInt}$ [(3)] |
| $\mathit{NAT}$ | - | $0..\mathit{MaxInt}$ [(3)] |
| $\mathit{NAT}_1$ | - | $1..\mathit{MaxInt}$ [(3)] |
| $-m$ | - | $-m$ [(2)] |
| $m..n$ | - | $m..n$ [(1)] |
| $m > n$ | - | $m > n$ [(1)] |
| $m < n$ | - | $m < n$ [(1)] |
| $m \geq n$ | - | $m \geq n$ [(1)] |
| $m \leq n$ | - | $m \leq n$ [(1)] |
| $\mathit{min}(S)$ | - | $\mathit{Min}(S)$ [(4)] |
| $\mathit{max}(S)$ | - | $\mathit{Max}(S)$ [(4)] |
| $m + n$ | - | $m + n$ [(1)] |
| $m - n$ | - | $m - n$ [(1)] |
| $m * n$ | - | $m * n$ [(1)] |
| $m \div n$ | - | $m \div n$ [(1)] |
| $m^n$ | - | $m^n$ [(1)] |
| $m \ \mathit{mod} \ n$ | - | $m \ \% \ n$ [(1)] |
| $\prod x.(P \mid E)$ | - | $\mathit{Pi}(\{E : x \in \{y \in \tau_x : P\}\})$ [(4)] |
| $\sum x.(P \mid e)$ | - | $\mathit{Sigma}(\{\langle x, e\rangle : x \in \{y \in \tau_x : P\}\})$ [(4)] |
| $\mathit{succ}(x)$ | - | $\mathit{Succ}(x)$ [(4)] |
| $\mathit{pred}(x)$ | - | $\mathit{Pred}(x)$ [(4)] |

(1) Defined by the Naturals standard module.
(2) Defined by the Integers standard module.
(3) Default values are used for MinInt and MaxInt.
(4) Defined by the BBuiltins module.

## A.6 Functions

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| $f(e)$ | $f[e]$ | $f[e]$ |
| $\lambda x.(P \mid e)$ | - | $[x \in \{t \in \tau_x : P\} \mid e]$ |
| $\lambda x.(x \in S \mid e)$ | $[x \in S \mid e]$ | $[x \in S \mid e]$ |
| $dom(f)$ | $DOMAIN\ f$ | $DOMAIN\ f$ |
| $ran(f)$ | - | $Range(f)$ (1) |
| $f[S]$ | $\{f[x] : x \in S\}$ | $Image(f, S)$ (1) |
| $card(f)$ | - | $Card(f)$ (1) |
| $id(S)$ | - | $Id(S)$ (1) |
| $S \lhd f$ | - | $DomRes(S, r)$ (1) |
| $S \mathbin{\lhd\!\!\!-} f$ | - | $DomSub(S, r)$ (1) |
| $S \rhd f$ | - | $RanRes(S, r)$ (1) |
| $S \mathbin{\rhd\!\!\!-} f$ | - | $RanSub(S, r)$ (1) |
| $f^{-1}$ | - | $Inverse(r)$ (1) |
| $f_1 \mathbin{\lhd\!\!\!+} f_2$ | - | $Override(r_1, r_2)$ (1) |
| S $\rightarrow$ T | [S $\rightarrow$ T] | [S $\rightarrow$ T] |
| S $\twoheadrightarrow$ T | [S $\rightarrow$ T] | $TotalSurFunc(S, T)$ (1) |
| S $\rightarrowtail$ T | [S $\rightarrow$ T] | $TotalInjFunc(S, T)$ (1) |
| S $\rightarrowtail\!\!\!\!\!\twoheadrightarrow$ T | - | $BijFunc(S, T)$ (1) |
| S $\mathrel{+\!\!\!\!\rightarrow}$ T | [S $\rightarrow$ T $\cup$ undef] (2) | $ParFunc(S, T)$ (1) |
| S $\mathrel{\rightarrowtail\!\!\!\!\!\!+}$ T | [S $\rightarrow$ T $\cup$ undef] (2) | $ParInjFunc(S, T)$ (1) |
| S $\mathrel{+\!\!\!\!\twoheadrightarrow}$ T | [S $\rightarrow$ T $\cup$ undef] (2) | $ParSurFunc(S, T)$ (1) |

(1) Defined by the Functions module.
(2) Mosbahi et al. define partial functions as total functions and associate the value undef to the elements where the function is not defined.

──────── MODULE *Functions* ────────

EXTENDS *FiniteSets*

$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN}\ f\}$
  The range of the function $f$

$Image(f, S) \triangleq \{f[x] : x \in S \cap \text{DOMAIN}\ f\}$
  The image of the set $S$ for the function $f$

$Card(f) \triangleq Cardinality(\text{DOMAIN}\ f)$
  The *Cardinality* of the function $f$

$Id(S) \triangleq [x \in S \mapsto x]$
  The identity function on set $S$

$DomRes(S, f) \triangleq [x \in (S \cap \textsc{domain}\ f) \mapsto f[x]]$

The restriction on the domain of $f$ for set $S$

$DomSub(S, f) \triangleq [x \in \textsc{domain}\ f \setminus S \mapsto f[x]]$

The subtraction on the domain of $f$ for set $S$

$RanRes(f, S) \triangleq [x \in \{y \in \textsc{domain}\ f : f[y] \in S\} \mapsto f[x]]$

The restriction on the range of $f$ for set $S$

$RanSub(f, S) \triangleq [x \in \{y \in \textsc{domain}\ f : f[y] \notin S\} \mapsto f[x]]$

The subtraction on the range of $f$ for set $S$

$Inverse(f) \triangleq \{\langle f[x], x\rangle : x \in \textsc{domain}\ f\}$

The inverser relation of the function $f$

$Override(f, g) \triangleq [x \in (\textsc{domain}\ f) \cup \textsc{domain}\ g \mapsto \textsc{if}\ x \in \textsc{domain}\ g\ \textsc{then}\ g[x]\ \textsc{else}\ f[x]]$

Overwriting of the function $f$ with the function $g$

$FuncAssign(f, d, e) \triangleq Override(f, [x \in \{d\} \mapsto e])$

Overwriting the function $f$ at index $d$ with value $e$

$TotalInjFunc(S, T) \triangleq \{f \in [S \rightarrow T] :$
$\quad Cardinality(\textsc{domain}\ f) = Cardinality(Range(f))\}$

The set of all total injective functions

$TotalSurFunc(S, T) \triangleq \{f \in [S \rightarrow T] : T = Range(f)\}$

The set of all total surjective functions

$TotalBijFunc(S, T) \triangleq \{f \in [S \rightarrow T] : T = Range(f) \land$
$\quad Cardinality(\textsc{domain}\ f) = Cardinality(Range(f))\}$

The set of all total bijective functions

$ParFunc(S, T) \triangleq \textsc{union}\ \{[x \rightarrow T] : x \in \textsc{subset}\ S\}$

The set of all partial functions

$isEleOfParFunc(f, S, T) \triangleq \textsc{domain}\ f \subseteq S \land Range(f) \subseteq T$

Test if the function $f$ is a partial function

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

The set of all partial injective functions

$ParSurFunc(S, T) \triangleq \{f \in ParFunc(S, T) : T = Range(f)\}$

The set of all partial surjective function

$ParBijFunc(S, T) \triangleq \{f \in ParFunc(S, T) : T = Range(f) \land$
$\quad Cardinality(\textsc{domain}\ f) = Cardinality(Range(f))\}$

The set of all partial bijective functions

## A.7 Sequences

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| $[e_1, e_2, e_3]$ | - | $\langle e_1, e_2, e_3 \rangle$ |
| $[]$ | - | $\langle\rangle$ |
| $seq(S)$ | - | $Seq(S)$ [1] |
| $seq_1(S)$ | - | $Seq1(S)$ [2] |
| $iseq(S)$ | - | $ISeq(S)$ [2] $ISeqEleOf(S)$ [2] |
| $iseq1(S)$ | - | $ISeq1(S)$ [2] $ISeq1EleOf(S)$ [2] |
| $perm(S)$ | - | $Perm(S)$ [2] |
| $size(s)$ | - | $Len(s)$ [1] |
| $first(s)$ | - | $Head(s)$ [1] |
| $last(s)$ | - | $Last(s)$ [2] |
| $tail(s)$ | - | $Tail(s)$ [1] |
| $rev(s)$ | - | $Reverse(s)$ [2] |
| $s \frown t$ | - | $s \circ t$ [1] |
| $e \rightarrow s$ | - | $Prepend(e, s)$ [2] |
| $s \leftarrow e$ | - | $Append(s, e)$ [1] |
| $conc(s)$ | - | $Conc(s)$ [2] |
| $s \uparrow n$ | - | $TakeFirstElements(s, n)$ [2] |
| $s \downarrow n$ | - | $DropFirstElements(s, n)$ [2] |

(1) Defined by the Sequences standard module.
(2) Defined by the SequencesExtended module.

MODULE *SequencesExtended*

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

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

$Last(s) \triangleq s[Len(s)]$

The last element of the sequence $s$

$Front(s) \triangleq [i \in 1 \,..\, (Len(s) - 1) \mapsto s[i]]$

The sequence $s$ without its last element.

$Prepend(e, s) \triangleq [i \in 1 \,..\, (Len(s) + 1) \mapsto \text{IF } i = 1 \text{ THEN } e \text{ ELSE } s[i - 1]]$

The Sequence obtained by inserting $e$ at the front of sequence $s$

$Reverse(s) \triangleq \text{LET } l \triangleq Len(s)$

```
            IN   $[i \in 1 \,..\, l \mapsto s[l - i + 1]]$
```
Sequence $s$ in reverse order

$BoundedSeq(S, n) \triangleq \text{UNION } \{[1 \,..\, x \rightarrow S] : x \in 0 \,..\, n\}$

The set of sequences with maximum length $n$

$Seq1(S) \triangleq Seq(S) \setminus \{\langle\rangle\}$

The set of sequences without the empty sequence

```
$ISeq(S) \triangleq$
    UNION $\{\{x \in [(1 \,..\, n) \rightarrow S] : Cardinality(Range(x)) = Cardinality(\text{DOMAIN } x)\}$
                                              $: n \in 0 \,..\, Cardinality(S)\}$
```
The set of injective sequences

$ISeqEleOf(S) \triangleq \{x \in Seq(S) : Len(x) = Cardinality(Range(x))\}$

The set of injective sequences
optimized to test if a certain sequence is in this set.

$ISeq1(S) \triangleq ISeq(S) \setminus \{\langle\rangle\}$

The set of injective sequences without the empty sequence

$ISeq1EleOf(S) \triangleq \{x \in Seq(S) : x \neq \langle\rangle \land Len(x) = Cardinality(Range(x))\}$

The set of injective sequences without the empty sequence
optimized to test if a certain sequence is in this set.

```
RECURSIVE $Perm(\_)$
$Perm(S) \triangleq$ IF $S = \{\}$
              THEN $\{\langle\rangle\}$
              ELSE LET $ps \triangleq [x \in S \mapsto \{Append(s, x) : s \in Perm(S \setminus \{x\})\}]$
                   IN  UNION $\{ps[x] : x \in S\}$
```
The set of permutations (bijective sequences)

```
RECURSIVE $Conc(\_)$
$Conc(S) \triangleq$ IF $S = \langle\rangle$
              THEN $\langle\rangle$
              ELSE $Head(S) \circ Conc(Tail(S))$
```
The sequence obtained by concatenating all sequences
which are elements of the sequence $S$.

```
$TakeFirstElements(s, n) \triangleq$
    IF $n \in 0 \,..\, Len(s)$
     THEN $[i \in 1 \,..\, n \mapsto s[i]]$
     ELSE $Assert(n \in 0 \,..\, Cardinality(s),$
                “The second argument of the take-first-operator is an invalid number.”$)$
```
Taking the first $n$ elements of the sequence $s$.

```
$DropFirstElements(s, n) \triangleq$
    IF $n \in 0 \,..\, Len(s)$
    THEN  $[i \in 1 \,..\, (Len(s) - n) \mapsto s[n + i]]$
    ELSE  $Assert(n \in 0 \,..\, Cardinality(s),$
             "The second argument of the drop-first-operator is an invalid number.")
  Dropping the first $n$ elements of the sequence $s$
```

## A.8 Relations

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | Tlc4B |
| $e \mapsto f$ | - | $\langle e, f \rangle$ |
| $S \leftrightarrow T$ | - | $Relations(S, T)$ (1) |
| $dom(r)$ | - | $RelDomain(r)$ (1) |
| $ran(r)$ | - | $RelRange(r)$ (1) |
| $id(S)$ | - | $RelId(S)$ (1) |
| $S \lhd r$ | - | $RelDomRes(S, r)$ (1) |
| $S \unlhd r$ | - | $RelDomSub(S, r)$ (1) |
| $S \rhd r$ | - | $RelRanRes(S, r)$ (1) |
| $S \unrhd r$ | - | $RelRanSub(S, r)$ (1) |
| $r^{-1}$ | - | $RelInverse(r)$ (1) |
| $r[S]$ | - | $RelImage(r, S)$ (1) |
| $r_1 \Lleftarrow r_2$ | - | $RelOverride(r_1, r_2)$ (1) |
| $r_1 \otimes r_2$ | - | $RelDirectProduct(r_1, r_2)$ (1) |
| $r_1 ; r_2$ | - | $RelComposition(r_1, r_2)$ (1) |
| $r_1 \parallel r_2$ | - | $RelParallelProd(r_1, r_2)$ (1) |
| $prj_1(S, T)$ | - | $RelPrj_1(S, T)$ (1) |
| $prj_2(S, T)$ | - | $RelPrj_2(S, T)$ (1) |
| $r^+$ | - | $RelClosure1(r)$ (1) |
| $r^*$ | - | $RelClosure(r)$ (1) |
| $r^n$ | - | $RelIterate(r, n)$ (1) |
| $fnc(r)$ | - | $RelFnc(r)$ (1) |
| $rel(r)$ | - | $RelRel(r)$ (1) |

(1) Defined by the Relations module.

```
---------------------------- MODULE Relations ----------------------------
EXTENDS FiniteSets, Naturals, Sequences, TLC

Relations(S, T) ≜ SUBSET (S × T)
  The set of all relations

RelDomain(R) ≜ {x[1] : x ∈ R}
   The domain of the relation R
```

$RelRange(R) \triangleq \{x[2] : x \in R\}$
The range of the relation $R$

$RelId(S) \triangleq \{\langle x, x\rangle : x \in S\}$
The identity relation of set $S$

$RelDomRes(S, R) \triangleq \{x \in R : x[1] \in S\}$
The restriction on the domain of $R$ for set $S$

$RelDomSub(S, R) \triangleq \{x \in R : x[1] \notin S\}$
The subtraction on the domain of $R$ for set $S$

$RelRanRes(R, S) \triangleq \{x \in R : x[2] \in S\}$
The restriction on the range of $R$ for set $S$

$RelRanSub(R, S) \triangleq \{x \in R : x[2] \notin S\}$
The subtraction on the range of $R$ for set $S$

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

$RelImage(R, S) \triangleq \{y[2] : y \in \{x \in R : x[1] \in S\}\}$
The image of $R$ for set $S$

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

$$\begin{aligned}&RelComposition(R1, R2) \triangleq \{\langle u[1][1], u[2][2]\rangle : u \in \\ &\quad \{x \in RelRanRes(R1, RelDomain(R2)) \times RelDomRes(RelRange(R1), R2) : \\ &\qquad x[1][2] = x[2][1]\}\}\end{aligned}$$
The relational composition of $R1$ and $R2$

$$\begin{aligned}&RelDirectProduct(R1, R2) \triangleq \{\langle x, u\rangle \in RelDomain(R1) \times (RelRange(R1) \times RelRange(R2)) : \\ &\quad \land \langle x, u[1]\rangle \in R1 \\ &\quad \land \langle x, u[2]\rangle \in R2\}\end{aligned}$$
The direct product of relation $R1$ and $R2$

$$\begin{aligned}RelParallelProduct(R1, R2) \triangleq \{\langle a, b\rangle \in\ &(RelDomain(R1) \times RelDomain(R2)) \\ &\times (RelRange(R1) \times RelRange(R2)) \\ : &\langle a[1], b[1]\rangle \in R1 \land \langle a[2], b[2]\rangle \in R2\}\end{aligned}$$
The parallel product of $R1$ and $R2$

$RelPrj1(S, T) \triangleq \{\langle\langle a, b\rangle, a\rangle : a \in S, b \in T\}$
The first projection relation

$RelPrj2(S, T) \triangleq \{\langle\langle a, b\rangle, b\rangle : a \in S, b \in T\}$
The second projection relation

RECURSIVE $RelIterate(\_, \_)$

```
$RelIterate(R, n) \triangleq$ CASE $n < 0 \rightarrow Assert(\text{FALSE}, \text{``''})$
        $\Box n = 0 \quad \rightarrow RelId(RelDomain(R) \cup RelRange(R))$
        $\Box n = 1 \quad \rightarrow R$
        $\Box$OTHER $\rightarrow RelComposition(R, RelIterate(R, n-1))$
```

The relation $R$ iterated $n$ times in relation to the composition operator

```
$RelClosure1(R) \triangleq$  Warshall algorithm from *Leslie Lamport*'s *Hyperbook*
    LET $NR \triangleq \{r[1] : r \in R\} \cup \{r[2] : r \in R\}$
        RECURSIVE $W(\_)$
        $W(L) \triangleq$ IF $L = \{\}$
                   THEN $R$
                   ELSE LET $n \triangleq$ CHOOSE $node \in L$ : TRUE
                            $WM \triangleq W(L \setminus \{n\})$
                        IN  $TLCEval(WM \cup \{rs \in NR \times NR :$
                              $(\langle rs[1], n\rangle \in WM) \land (\langle n, rs[2]\rangle \in WM)\})$
    IN  $W(NR)$
```

The transitive closure of $R$

$RelClosure(R) \triangleq RelClosure1(R \cup \{\langle x[1], x[1]\rangle : x \in R\} \cup RelIterate(R, 0))$

The transitive and reflexive closure of $R$.

$RelFnc(R) \triangleq \{\langle x, RelImage(R, \{x\})\rangle : x \in RelDomain(R)\}$

The transformation of $R$ into a function

*e.g.* $RelFnc(\{(0, 1), (0, 2), (1, 1)\}) = \{(0, \{1, 2\}), (1, \{1\})\}$

```
RECURSIVE $RelRel(\_)$
$RelRel(Fct) \triangleq$ IF $Fct = \{\}$
              THEN $\{\}$
              ELSE LET $e \triangleq$ CHOOSE $x \in Fct$ : TRUE
                   IN  $\{\langle e[1], y\rangle : y \in e[2]\} \cup RelRel(Fct \setminus \{e\})$
```

The transformation of the function $Fct$ into a relation

*e.g.* $RelRel(\{(0, \{1, 2\}), (1, \{1\})\}) = \{(0, 1), (0, 2), (1, 1)\}$

## A.9 Functions as Relations

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| $f(e)$ | - | $RelCall(f, e)$ [1] |
| $\lambda x.(P\|e)$ | - | $\{\langle x, e\rangle : x \in \{t \in \tau_x : P\}$ |
| $\lambda x.(x \in S\|e)$ | - | $\{\langle x, e\rangle : x \in S\}$ |
| S $\rightarrow$ T | - | $RelTotalFunc(S, T)$ [1] $RelTotalFuncEleOf(S, T)$ [1] |
| S $\twoheadrightarrow$ T | - | $RelTotalSurFunc(S, T)$ [1] $RelTotalSurFuncEleOf(S, T)$ [1] |
| S $\rightarrowtail$ T | - | $RelTotalInjFunc(S, T)$ [1] $RelTotalInjFuncEleOf(S, T)$ [1] |
| S $\rightarrowtail\!\!\!\!\twoheadrightarrow$ T | - | $RelTotalBijFunc(S, T)$ [1] $RelTotalBijFuncEleOf(S, T)$ [1] |
| S $\nrightarrow$ T | - | $RelParFunc(S, T)$ [1] $RelParFuncEleOf(S, T)$ [1] |
| S $\rightarrowtail\!\!\!\!\nrightarrow$ T | - | $RelParInjFunc(S, T)$ [1] $RelParInjFuncEleOf(S, T)$ [1] |
| S $\nrightarrow\!\!\!\!\twoheadrightarrow$ T | - | $RelParSurFunc(S, T)$ [1] $RelParSurFuncEleOf(S, T)$ [1] |
| S $\nrightarrow\!\!\!\!\twoheadrightarrow$ T | - | $RelParBijFunc(S, T)$ [1] $RelParBijFuncEleOf(S, T)$ [1] |

[1] Defined by the FunctionsAsRelations module.

```
MODULE FunctionsAsRelations
EXTENDS FiniteSets, Functions, TLC, Sequences

LOCAL RelDom(f) ≜ {x[1] : x ∈ f}  The domain of the function
LOCAL RelRan(f) ≜ {x[2] : x ∈ f}  The range of the function
LOCAL MakeRel(f) ≜ {⟨x, f[x]⟩ : x ∈ DOMAIN f}
  Converting a TLA+ function to a set of pairs
LOCAL Rel(S, T) ≜ SUBSET (S × T)  The set of relations
LOCAL IsFunc(f) ≜ Cardinality(RelDom(f)) = Cardinality(f)
  Testing if f is a function
LOCAL IsTotal(f, dom) ≜ RelDom(f) = dom
  Testing if f is a total function
LOCAL IsInj(f) ≜ Cardinality(RelRan(f)) = Cardinality(f)
  Testing if f is a injective function
LOCAL IsSurj(f, ran) ≜ RelRan(f) = ran
  Testing if f is a surjective function

RelCall(f, x) ≜ IF Cardinality(f) = Cardinality(RelDom(f)) ∧ x ∈ RelDom(f)
```

```
    THEN (CHOOSE $y \in f : y[1] = x)[2]$
    ELSE $PrintT($"Error: Invalid function call to relation "
            $\circ\ ToString(f) \circ$ " and value " $\circ\ ToString(x) \circ$ ".")
        $\wedge\ Assert(Cardinality(f) \neq Cardinality(RelDom(f))$, "Applied a function call to a relation.")
        $\wedge\ Assert(x \notin RelDom(f)$, "Function applied outside domain.")
```
The function call applied to function $f$ and argument $x$

$RelTotalFunc(S,\ T) \triangleq \{MakeRel(f) : f \in [S \rightarrow T]\}$
The set of total function

$RelTotalFuncEleOf(S,\ T) \triangleq \{f \in Rel(S,\ T) : IsFunc(f) \wedge IsTotal(f,\ S)\}$
The set of total functions
(Optimized to test if a certain function is a element of this set.)

$RelTotalInjFunc(S,\ T) \triangleq \{MakeRel(f) : f \in TotalInjFunc(S,\ T)\}$
The set of total injective functions

$RelTotalInjFuncEleOf(S,\ T) \triangleq \{f \in Rel(S,\ T) : IsFunc(f) \wedge IsTotal(f,\ S) \wedge IsInj(f)\}$
The set of total injective functions
(Optimized to test if a certain function is a element of this set.)

$RelTotalSurFunc(S,\ T) \triangleq \{MakeRel(f) : f \in TotalSurFunc(S,\ T)\}$
The set of total surjective functions

$RelTotalSurFuncEleOf(S,\ T) \triangleq \{f \in Rel(S,\ T) :$
$\quad \wedge\ IsFunc(f) \wedge IsTotal(f,\ S) \wedge IsSurj(f,\ T)\}$
The set of total surjective functions
(Optimized to test if a certain function is a element of this set.)

$RelTotalBijFunc(S,\ T) \triangleq \{MakeRel(f) : f \in TotalBijFunc(S,\ T)\}$
The set of total bijective functions

$RelTotalBijFuncEleOf(S,\ T) \triangleq \{f \in (\text{SUBSET } (S \times T)) :$
$\quad \wedge\ IsFunc(f) \wedge IsTotal(f,\ S) \wedge IsInj(f) \wedge IsSurj(f,\ T)\}$
The set of total bijective functions
(Optimized to test if a certain function is a element of this set.)

$RelParFunc(S,\ T) \triangleq \{MakeRel(f) :\ f \in ParFunc(S,\ T)\}$
The set of partial functions

$RelParFuncEleOf(S,\ T) \triangleq \{f \in Rel(S,\ T) : IsFunc(f)\}$
The set of partial functions
(Optimized to test if a certain function is a element of this set.)

$RelParInjFunc(S,\ T) \triangleq \{MakeRel(f) :\ f \in ParInjFunc(S,\ T)\}$
The set of partial injective functions.

$RelParInjFuncEleOf(S,\ T) \triangleq \{f \in Rel(S,\ T) : IsFunc(f) \wedge IsInj(f)\}$

The set of partial injective functions
(Optimized to test if a certain function is a element of this set.)

$RelParSurFunc(S,\ T) \ \triangleq\ \{MakeRel(f):\ f \in ParSurFunc(S,\ T)\}$

The set of partial surjective functions

$RelParSurFuncEleOf(S,\ T) \ \triangleq\ \{f \in Rel(S,\ T) : IsFunc(f) \land IsSurj(f,\ T)\}$

The set of partial surjective functions.
(Optimized to test if a certain function is a element of this set.)

$RelParBijFunc(S,\ T) \ \triangleq\ \{MakeRel(f):\ f \in ParBijFunc(S,\ T)\}$

The set of partial bijective functions

$RelParBijFuncEleOf(S,\ T) \ \triangleq\ \{f \in Rel(S,\ T) : IsFunc(f) \land IsSurj(f,\ T) \land IsInj(f)\}$

The set of partial bijective functions
(Optimized to test if a certain function is a element of this set.)

## A.10 Sequences as Relations

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| $[e_1, e_2, e_3]$ | - | $\{\langle 1, e_1\rangle, \langle 2, e_2\rangle, \langle 3, e_3\rangle\}$ |
| [] | - | {} |
| s ∈ seq(T) | - | $\mathit{IsRelSeq}(s, T)$ (1) |
| s ∈ $\text{seq}_1$(S) | - | $\mathit{IsRelSeq1}(s, T)$ (1) |
| $\mathit{iseq}(S)$ | - | $\mathit{RelISeq}(S)$ (1) |
| $\mathit{iseq1}(S)$ | - | $\mathit{RelISeq1}(S)$ (1) |
| $\mathit{perm}(S)$ | - | $\mathit{RelSeqPerm}(S)$ (1) |
| $\mathit{size}(s)$ | - | $\mathit{Cardinality}(s)$ (2) |
| $\mathit{first}(s)$ | - | $\mathit{RelSeqFirst}(s)$ (1) |
| $\mathit{last}(s)$ | - | $\mathit{RelSeqLast}(s)$ (1) |
| $\mathit{tail}(s)$ | - | $\mathit{RelSeqTail}(s)$ (1) |
| $\mathit{rev}(s)$ | - | $\mathit{RelSeqReverse}(s)$ (1) |
| $s \hat{\ } t$ | - | $\mathit{ReSeqConcat}(s, t)$ (1) |
| $e \rightarrow s$ | - | $\mathit{RelSeqPrepend}(e, s)$ (1) |
| $s \leftarrow e$ | - | $\mathit{RelSeqAppend}(s, e)$ (1) |
| $\mathit{conc}(s)$ | - | $\mathit{RelSeqConc}(s)$ (1) |
| $s \uparrow n$ | - | $\mathit{RelSeqTakeFirstElements}(s, n)$ (1) |
| $s \downarrow n$ | - | $\mathit{RelSeqDropFirstElements}(s, n)$ (1) |

(1) Defined by the SequencesAsRelations module.
(2) Defined by the standard module FiniteSets module.

MODULE *SequencesAsRelations*

EXTENDS *FiniteSets*, *Naturals*, *Relations*, *FunctionsAsRelations*

$\mathit{IsRelSeq}(x, S) \triangleq \forall n \in 1 \,..\, \mathit{Cardinality}(x) : \mathit{RelCall}(x, n) \in S$

Testing if $x$ is a sequence with elements of the set $S$

$\mathit{RelSeqSet}(x, S) \triangleq \text{IF } \mathit{IsRelSeq}(x, S) \text{ THEN } \{x\} \text{ ELSE } \{\}$

$\mathit{IsRelSeq1}(x, S) \triangleq x \neq \{\} \land \mathit{IsRelSeq}(x, S)$

Testing if $x$ is a non-empty sequence with elements of the set $S$

$\mathit{RelSeqSet1}(x, S) \triangleq \text{IF } \mathit{IsRelSeq1}(x, S) \text{ THEN } \{x\} \text{ ELSE } \{\}$

$\text{LOCAL } \mathit{ISeq}(S) \triangleq \text{UNION } \{\{x \in [(1 \,..\, n) \rightarrow S] : \mathit{Cardinality}(\mathit{Range}(x)) = \mathit{Cardinality}(\text{DOMAIN } x)\}$
$: n \in 0 \,..\, \mathit{Cardinality}(S)\}$

$\mathit{RelISeq}(S) \triangleq \{\{\langle n, x[n]\rangle : n \in 1 \,..\, \mathit{Len}(x)\} : x \in \mathit{ISeq}(S)\}$

The set of all injective sequences with elements of $S$

$\mathit{RelISeq1}(S) \triangleq \mathit{RelISeq}(S) \setminus \{\{\}\}$

```
  The set of all non-empty injective sequences with elements of $S$

LOCAL $SeqTest(s) \triangleq RelDomain(s) = 1 \,..\, Cardinality(s)$
  Testing if $s$ is a sequence

$RelSeqFirst(s) \triangleq$ IF $SeqTest(s)$
    THEN $RelCall(s, 1)$
    ELSE $Assert($FALSE, "Error: The argument of the first-operator should be a sequence."$)$
  The head of the sequence

$RelSeqLast(s) \triangleq$ IF $SeqTest(s)$
        THEN $RelCall(s, Cardinality(s))$
        ELSE $Assert($FALSE, "Error: The argument of the last-operator should be a sequence."$)$
  The last element of the sequence

$RelSeqSize(s) \triangleq$ IF $SeqTest(s)$
        THEN $Cardinality(s)$
        ELSE $Assert($FALSE, "Error: The argument of the size-operator should be a sequence."$)$
  The size of the sequence $s$

$RelSeqTail(s) \triangleq$ IF $SeqTest(s)$
    THEN $\{\langle x[1] - 1, x[2] \rangle : x \in \{x \in s : x[1] \neq 1\}\}$
    ELSE $Assert($FALSE, "Error: The argument of the tail-operator should be a sequence."$)$
  The tail of the sequence $s$

$RelSeqConcat(s1, s2) \triangleq$ IF $SeqTest(s1) \land SeqTest(s2)$
    THEN $s1 \cup \{\langle x[1] + Cardinality(s1), x[2] \rangle : x \in s2\}$
    ELSE $Assert($FALSE, "Error: The arguments of the concatenation-operator should be sequences."$)$
  The concatenation of sequence $s1$ and sequence $s2$

$RelSeqPrepend(e, s) \triangleq$ IF $SeqTest(s)$
    THEN $\{\langle 1, e \rangle\} \cup \{\langle x[1] + 1, x[2] \rangle : x \in s\}$
    ELSE $Assert($FALSE, "Error: The second argument of the prepend-operator should be a sequence."$)$
  The sequence obtained by inserting $e$ at the front of sequence $s$.

$RelSeqAppend(s, e) \triangleq$ IF $SeqTest(s)$
    THEN $s \cup \{\langle Cardinality(s) + 1, e \rangle\}$
    ELSE $Assert($FALSE, "Error: The first argument of the append-operator should be a sequence."$)$
  The sequence obtained by appending $e$ to the end of sequence $s$.

$RelSeqReverse(s) \triangleq$ IF $SeqTest(s)$
    THEN $\{\langle Cardinality(s) - x[1] + 1, x[2] \rangle \ : x \in s\}$
    ELSE $Assert($FALSE, "Error: The argument of the reverse-operator should be a sequence."$)$
  The sequence obtained by reversing the order of the elements.

$RelSeqFront(s) \triangleq$ IF $SeqTest(s)$
    THEN $\{x \in s : x[1] \neq Cardinality(s)\}$
    ELSE $Assert($FALSE, "Error: The argument of the front-operator should be a sequence."$)$
```

The front of the sequence $s$ (all but last element)

```
RECURSIVE $RelSeqPerm(\_)$
$RelSeqPerm(S) \triangleq$ IF $S = \{\}$
                  THEN $\{\{\}\}$
                  ELSE LET $ps \triangleq [x \in S \mapsto \{RelSeqAppend(s, x) : s \in RelSeqPerm(S \setminus \{x\})\}]$
                       IN  UNION $\{ps[x] : x \in S\}$
```

The set of bijective sequences (permutations)
*e.g.* $\{\langle 1, 2, 3\rangle, \langle 2, 1, 3\rangle, \langle 2, 3, 1\rangle, \langle 3, 1, 2\rangle, \langle 3, 2, 1\rangle\}$ for $S = \{1, 2, 3\}$

```
RECURSIVE $RelSeqConc(\_)$
$RelSeqConc(S) \triangleq$
    IF $S = \{\}$
     THEN $\{\}$
     ELSE $RelSeqConcat(RelSeqFirst(S), RelSeqConc(RelSeqTail(S)))$
```

The sequence obtained by concatenating all sequences
which are elements of the sequence $S$.

```
$RelSeqTakeFirstElements(s, n) \triangleq$
    IF $SeqTest(s) \land n \in 0 \,..\, Cardinality(s)$
     THEN $\{x \in s : x[1] \leq n\}$
     ELSE $\land\ Assert(n \in 0 \,..\, Cardinality(s),$
                "The second argument of the take-first-operator is an invalid number.")
          $\land\ Assert($FALSE, "Error: The first argument of the take-first-operator should be a sequence.")
```

The first $n$ elements of $s$ as a sequence

```
$RelSeqDropFirstElements(s, n) \triangleq$
    IF $SeqTest(s) \land n \in 0 \,..\, Cardinality(s)$
     THEN $\{\langle x[1] - n, x[2]\rangle : x \in \{x \in s : x[1] > n\}\}$
     ELSE $\land\ Assert(n \in 0 \,..\, Cardinality(s),$
                "The second argument of the drop-first-operator is an invalid number.")
          $\land\ Assert($FALSE, "Error: The first argument of the drop-first-operator should be a sequence.")
```

The last $n$ elements of $s$ as a sequence

## A.11 Strings

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| "abc" | - | "abc" |
| *STRING* | - | *STRING* |

## A.12 Records

| B-Method | TLA$^+$ | |
|---|---|---|
| | Mosbahi et al. | TLC4B |
| $r'h$ | - | $r.h$ |
| $rec(h_1 : e_1, \ldots, h_n : e_n)$ | - | $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ |
| $struct(h_1 : S_1, \ldots, h_1 : S_n)$ | - | $[h_1 : S_1, \ldots, h_n : S_n]$ |

## References

1. J.-R. Abrial. *The B-Book.* Cambridge University Press, 1996.
2. K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. The TLA$^+$ proof system: Building a heterogeneous verification platform. In A. Cavalcanti, D. Déharbe, M.-C. Gaudel, and J. Woodcock, editors, *Proceedings ICTAC 2010*, LNCS 6255, page 44, 2010.
3. ClearSy. B language reference manual. `http://www.tools.clearsy.com/resources/Manrefb_en.pdf`. Accessed: 2013-11-10.
4. E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.
5. D. Hansen and M. Leuschel. Translating TLA+ to B for validation with ProB. In *Proceedings iFM'2012*, LNCS 7321, pages 24–38. Springer, 2012.
6. D. Hansen and M. Leuschel. Translating B to TLA+ for validation with TLC: There and back again. `http://www.stups.uni-duesseldorf.de/w/Special:Publication/HansenLeuschel_TLC4B_techreport`, 2013.
7. L. Lamport. The TLA$^+$ hyperbook. `http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html`. Accessed: 2013-10-30.
8. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.
9. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
10. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
11. O. Mosbahi, L. Jemni, and J. Jaray. A formal approach for the development of automated systems. In J. Filipe, B. Shishkov, and M. Helfert, editors, *ICSOFT (SE)*, pages 304–310. INSTICC Press, 2007.
12. M. Reynolds. Changing nothing is sometimes doing something. Technical Report TR-98-02, Department of Computer Science, King's College London, February 1998.
13. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA$^+$ specifications. In L. Pierre and T. Kropf, editors, *Proceedings CHARME'99*, LNCS 1703, pages 54–66. Springer-Verlag, 1999.