Encoding TLA$^+$ into Many-Sorted First-Order Logic

Stephan Merz[1,2] and Hernán Vanzetto[3]

[1] Inria, Villers-lès-Nancy, France
[2] CNRS, Université de Lorraine, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, France
[3] Yale University, New Haven, CT, United States

Abstract. This paper presents an encoding of a non-temporal fragment of the TLA$^+$ language, which includes untyped set theory, functions, arithmetic expressions, and Hilbert's ε operator, into many-sorted first-order logic, the input language of state-of-the-art SMT solvers. This translation, based on encoding techniques such as boolification, injection of unsorted expressions into sorted languages, term rewriting, and abstraction, is the core component of a back-end prover based on SMT solvers for the TLA$^+$ Proof System.

1 Introduction

The specification language TLA$^+$ [10] combines variants of Zermelo-Fraenkel set theory with choice (ZFC) and of linear-time temporal logic for modeling, respectively, the data manipulated by an algorithm, and its behavior. The TLA$^+$ Proof System (TLAPS) provides support for mechanized reasoning about TLA$^+$ specifications, integrating back-end provers for making automatic reasoners available to users of TLAPS. The work reported here is motivated by the development of an SMT backend, through which users of TLAPS interact with off-the-shelf SMT (satisfiability modulo theories) solvers for non-temporal reasoning. More specifically, TLAPS is built around a so-called Proof Manager [4] that interprets the TLA$^+$ proof language, generates corresponding proof obligations, and passes them to external automated verifiers, which are the back-end provers of TLAPS.

Previous to this work, three back-end provers with different capabilities were available for non-temporal reasoning: Isabelle/TLA$^+$, a faithful encoding of TLA$^+$ set theory in the Isabelle proof assistant, which provides automated proof methods based on first-order reasoning and rewriting; Zenon, a tableau prover for first-order logic with equality that includes extensions for reasoning about sets and functions; and a decision procedure for Presburger arithmetic called SimpleArithmetic (now deprecated). The Isabelle and Zenon backends have very limited support for arithmetic reasoning, while SimpleArithmetic handles only pure arithmetic formulas, requiring the user to manually decompose the proofs until the corresponding proof obligations fall within the respective fragments.

Beyond its integration as a semi-automatic backend, Isabelle/TLA$^+$ serves as the most trusted back-end prover. Accordingly, it is also intended for certifying proof scripts produced by other back-end provers. When possible, backends are expected to produce a detailed proof that can be checked by Isabelle/TLA$^+$. Currently, only the Zenon backend has an option for exporting proofs that can be certified in this way.

In this paper we describe the foundations of a back-end prover based on SMT solvers for non-temporal proof obligations arising in TLAPS.[4] When verifying distributed algorithms, proof obligations are usually "shallow", but they still require many details to be checked: interactive proofs can become quite large without powerful automated back-end provers that can cope with a significant fragment of the language. Sets and functions are at the core of modeling data in the TLA$^+$ language. Tuples and records, which occur very often in TLA$^+$ specifications, are defined as functions. Assertions mixing first-order logic (FOL) with sets, functions, and arithmetic expressions arise frequently in safety proofs of TLA$^+$ specifications. Accordingly, we do not aim at proofs of deep theorems of mathematical set theory but at good automation for obligations mixing elementary set expressions, functions, records, and (linear) integer arithmetic. Our main focus is on SMT solvers, although we have also used the techniques described here with FOL provers. The de-facto standard input language for SMT solvers is SMT-LIB [2], which is based on many-sorted FOL (MS-FOL [11]).[5]

In Section 3 we present the core of the SMT backend: a translation from TLA$^+$ to MS-FOL. Although some of our encoding techniques can be found in similar tools for other set-theoretic languages, the particularities of TLA$^+$ make the translation non-trivial:

- Since TLA$^+$ is untyped, "silly" expressions such as $3 \cup \text{TRUE}$ are legal; they denote some (unspecified) value. TLA$^+$ does not even distinguish between Boolean and non-Boolean expressions, hence Boolean values can be stored in data structures just like any other value.
- Functions, which are defined axiomatically, are total and have a domain. This means that a function applied to an element of its domain has the expected value but for any other argument, the value of the function application is unspecified. Similarly, the behavior of arithmetic operators is specified only for arguments that denote numbers.
- TLA$^+$ is equipped with a deterministic choice operator (Hilbert's ε operator), which has to be soundly encoded.

The first item is particularly challenging for our objectives: whereas an untyped language is very expressive and flexible for writing specifications, MS-FOL reasoners rely on types for good automation. In order to support TLA$^+$ expressions in a many-sorted environment, we introduce a "boolification" step for distinguishing between Boolean and non-Boolean expressions, and use a single sort for encoding non-Boolean TLA$^+$ expressions. We therefore call this translation the "untyped" encoding of TLA$^+$; it essentially delegates type inference of sorted expressions such as arithmetic to the solvers.

[4] Non-temporal reasoning is enough for proving safety properties and makes up the vast majority of proof steps in liveness proofs.

[5] In this paper we use the terms *type* and *sort* interchangeably.

The paper is structured as follows: Section 2 describes the underlying logic of TLA$^+$, Section 3 is the core of the paper and explains the encoding, Section 4 provides experimental results, Section 5 discusses related work, and Section 6 concludes and gives directions for future work.

2 A non-temporal fragment of TLA$^+$

In this section we describe a fragment of the language of proof obligations generated by the TLA$^+$ Proof System that is relevant for this paper. This language is a variant of FOL with equality, extended in particular by syntax for set, function and arithmetic expressions, and a construct for a deterministic choice operator. For a complete presentation of the TLA$^+$ language see [10, Sec. 16].

We assume given two non-empty, infinite, and disjoint collections $\mathcal{V}$ of *variable* symbols, and $\mathcal{O}$ of *operator* symbols,[6] each equipped with its arity. The only syntactical category in the language is the *expression*, but for presentational purposes we distinguish terms, formulas, set objects, *etc.* An expression e is inductively defined by the following grammar:

$$\begin{array}{rll}
e ::= & v \mid w(e,\ldots,e) & \text{(terms)}\\
& \mid \textsc{false} \mid e \Rightarrow e \mid \forall v\colon\ e \mid e = e \mid e \in e & \text{(formulas)}\\
& \mid \{\} \mid \{e,\ldots,e\} \mid \textsc{subset}\ e \mid \textsc{union}\ e & \\
& \mid \{v \in e : e\} \mid \{e : v \in e\} & \text{(sets)}\\
& \mid \textsc{choose}\ x\colon e & \text{(choice)}\\
& \mid e[e] \mid \textsc{domain}\ e \mid [v \in e \mapsto e] \mid [e \to e] & \text{(functions)}\\
& \mid 0 \mid 1 \mid 2 \mid \ldots \mid \mathit{Int} \mid -e \mid e+e \mid e<e \mid e \mathrel{..} e & \text{(arithmetic)}\\
& \mid \textsc{if}\ e\ \textsc{then}\ e\ \textsc{else}\ e & \text{(conditional)}
\end{array}$$

A *term* is a variable symbol v in $\mathcal{V}$ or an application of an operator symbol w in $\mathcal{O}$ to expressions. *Formulas* are built from FALSE, implication and universal quantification, and from the binary operators $=$ and $\in$. From these formulas, we can define the constant TRUE, the unary $\neg$ and the binary connectives $\wedge$, $\vee$, $\Leftrightarrow$, and the existential quantifier $\exists$. Also, $\forall x \in S\colon e$ is defined as $\forall x\colon x \in S \Rightarrow e$.

In contrast to standard set theory, TLA$^+$ has explicit syntax for *set objects* (empty set, enumeration, power set, generalized union, and two forms of set comprehension derived from the standard axiom schema of replacement), whose semantics are defined by the following axioms:

$$\begin{array}{lcr}
\text{(extensionality)} & (\forall x\colon x \in S \Leftrightarrow x \in T) \Rightarrow S = T & (2.1)\\
\text{(empty set)} & x \in \{\} \Leftrightarrow \textsc{false} & (2.2)\\
\text{(enumeration)} & x \in \{e_1,\ldots,e_n\} \Leftrightarrow x = e_1 \vee \ldots \vee x = e_n & (2.3)\\
\text{(power set)} & S \in \textsc{subset}\ T \Leftrightarrow \forall x \in S\colon x \in T & (2.4)\\
\text{(union)} & x \in \textsc{union}\ S \Leftrightarrow \exists T \in S\colon x \in T & (2.5)
\end{array}$$

[6] TLA$^+$ operator symbols correspond to the standard function and predicate symbols of first-order logic but we reserve the term “function” for TLA$^+$ functional values.

$$\text{(comprehension}_1\text{)} \qquad x \in \{y \in S \colon P(y)\} \Leftrightarrow x \in S \land P(x) \tag{2.6}$$

$$\text{(comprehension}_2\text{)} \qquad x \in \{e(y) \colon y \in S\} \Leftrightarrow \exists y \in S \colon x = e(y) \tag{2.7}$$

We consider that the free variables in these formulas are universally closed, except for P and e in the comprehension axioms that are schematic variables, meaning that they can be instantiated by countably infinite expressions.[7]

Another primitive construct of TLA$^+$ is Hilbert's choice operator ε, written CHOOSE $x \colon P(x)$, that denotes an arbitrary but fixed value x such that $P(x)$ is true, provided that such a value exists. Otherwise the value of CHOOSE $x \colon P(x)$ is some fixed, but unspecified value. The semantics of CHOOSE is expressed by the following axiom schemas. The first one gives an alternative way of defining quantifiers, and the second one expresses that CHOOSE is deterministic.

$$\big(\exists x \colon P(x)\big) \;\Leftrightarrow\; P\big(\textsc{choose}\ x \colon P(x)\big) \tag{2.8}$$

$$\big(\forall x \colon P(x) \Leftrightarrow Q(x)\big) \;\Rightarrow\; \big(\textsc{choose}\ x \colon P(x)\big) = \big(\textsc{choose}\ x \colon Q(x)\big) \tag{2.9}$$

From axiom (2.9) note that if there is no value satisfying some predicate P, then (CHOOSE $x \colon P(x)$) = (CHOOSE $x \colon$ FALSE). Consequently, the expression CHOOSE $x \colon$ FALSE and all its equivalent forms represent a unique value.

Certain TLA$^+$ values are *functions*. Unlike standard ZFC set theory, TLA$^+$ functions are not identified with sets of pairs, but TLA$^+$ provides primitive syntax associated with functions. The expression $f[e]$ denotes the result of applying function f to e, DOMAIN f denotes the domain of f, and $[x \in S \mapsto e]$ denotes the function g with domain S such that $g[x] = e$, for any $x \in S$. For $x \notin S$, the value of $g[x]$ is unspecified. A TLA$^+$ value f is a function if and only if it satisfies the predicate $\mathit{IsAFcn}(f)$ defined as $f = [x \in \textsc{domain}\ f \mapsto f[x]]$. The fundamental law governing TLA$^+$ functions is

$$f = [x \in S \mapsto e] \;\Leftrightarrow\; \mathit{IsAFcn}(f) \land \textsc{domain}\ f = S \land \forall x \in S \colon f[x] = e \tag{2.10}$$

Natural numbers $0, 1, 2, \ldots$ are primitive symbols of TLA$^+$. Standard modules of TLA$^+$ define *Int* to denote the set of integer numbers, the operators $+$ and $<$ are interpreted in the standard way when their arguments are integers, and the interval $a \,..\, b$ is defined as $\{n \in \mathit{Int} : a \leq n \land n \leq b\}$.

As a set theoretic language, every TLA$^+$ expression—including formulas, functions, and numbers—denotes a set.

[7] Both axioms (2.6) and (2.7) for set comprehension objects are instances of the standard axiom schema of replacement: taking the two single-valued predicates $\phi_1(x, y) \triangleq x = y \land P(y)$ and $\phi_2(x, y) \triangleq x = e(y)$, we can define $\{y \in S : P(y)\} \triangleq \mathcal{R}(S, \phi_1)$ and $\{e(y) : y \in S\} \triangleq \mathcal{R}(S, \phi_2)$. The replacement axiom says that, given an expression S and a binary predicate ϕ, such that ϕ is *single-valued* for any x in S, that is, $\forall x \in S \colon \forall y, z \colon \phi(x, y) \land \phi(x, z) \;\Rightarrow\; y = z$, then there exists a set object $\mathcal{R}(S, \phi)$, and that $x \in \mathcal{R}(S, \phi) \Leftrightarrow \exists y \in S : \phi(x, y)$.

3 Untyped encoding into many-sorted first-order logic

The translation from TLA$^+$ to MS-FOL is as follows: given a TLA$^+$ proof obligation, we generate a collection of equi-satisfiable SMT-LIB formulas (restricted to the AUFLIA logic) whose proof can be attempted by SMT solvers.

First, all expressions having a truth value are mapped to the sort Bool, and we declare a new sort U (for TLA$^+$ universe) for all non-Boolean expressions, including sets, functions, and numbers (§3.1). Then we proceed in two main steps. A preprocessing phase applies satisfiability-preserving transformations in order to remove expressions not supported by the target language (§3.2). The result is an intermediate *basic* TLA$^+$ formula, *i.e.*, a TLA$^+$ expression that has an obvious counterpart in SMT-LIB. We define basic TLA$^+$ as a subset of TLA$^+$ consisting of terms, formulas, equality and set membership relations, primitive arithmetic operators, and IF-THEN-ELSE expressions. The second step is a shallow embedding of basic expressions into MS-FOL (§3.3). Finally, we explain how the encoding of functions (§3.4) and CHOOSE expressions (§3.5) fit in the translation.

3.1 Boolification

Since TLA$^+$ has no syntactic distinction between Boolean and non-Boolean expressions, we first need to determine which expressions are used as propositions. TLAPS adopts the so-called liberal interpretation of TLA$^+$ Boolean expressions [10, Sec. 16.1.3] where any expression with a top-level connective among logical operators, $=$, and $\in$ has a Boolean value.[8] Moreover, the result of any expression with a top-level logical connective agrees with the result of the expression obtained by replacing every argument e of that connective with $e = \text{TRUE}$.

For example, consider the expression $\forall x : (\neg\neg x) = x$, which is not a theorem. Indeed, x need not be Boolean, whereas $\neg\neg x$ is necessarily Boolean, hence we may not conclude that the expression is valid. However, $\forall x : (\neg\neg x) \Leftrightarrow x$ is valid because it is interpreted as $\forall x : (\neg\neg(x = \text{TRUE})) \Leftrightarrow (x = \text{TRUE})$. Observe that the value of $x = \text{TRUE}$ is a Boolean for any x, although the value is unspecified if x is non-Boolean.

In order to identify the expressions used as propositions we use a simple algorithm that recursively traverses an expression searching for sub-expressions that should be treated as formulas. Expressions e that are used as Booleans, *i.e.*, that could equivalently be replaced by $e = \text{TRUE}$, are marked as e^b, whose definition can be thought of as $e^b \triangleq e = \text{TRUE}$. This only applies if e is a term, a function application, or a CHOOSE expression. If an expression which is

[8] The standard semantics of TLA$^+$ offers three alternatives to interpret expressions [10, Sec. 16.1.3]. In the liberal interpretation, an expression like $42 \Rightarrow \{\}$ always has a truth value, but it is not specified if that value is true or false. In the conservative and moderate interpretations, the value of $42 \Rightarrow \{\}$ is completely unspecified. Only in the moderate and liberal interpretation, the expression $\text{FALSE} \Rightarrow \{\}$ has a Boolean value, and that value is true. In the liberal interpretation, all the ordinary laws of logic, such as commutativity of $\wedge$, are valid, even for non-Boolean arguments.

known to be non-Boolean by its syntax, such as a set or a function, is attempted to be boolified, meaning that a formula is expected in its place, the algorithm aborts with a "type" error. In SMT-LIB we encode x^b as $\mathsf{boolify}(x)$, with $\mathsf{boolify} : \mathsf{U} \to \mathsf{Bool}$. The above examples are translated as $\forall x^{\mathsf{U}} \colon (\neg\neg\mathsf{boolify}(x)) = x$ and $\forall x^{\mathsf{Bool}} \colon (\neg\neg x) \Leftrightarrow x$, revealing their (in)validity.

3.2 Preprocessing

Though a series of transformations to a boolified TLA$^+$ proof obligation, we obtain an equi-satisfiable formula that can be straightforwardly passed to the solvers using the direct encoding of basic expressions described below. The main motivation is to get rid of those TLA$^+$ expressions that cannot be expressed in first-order logic. Namely, they are $\{x \in S : P\}$, $\{e : x \in S\}$, $\textsc{choose}\ x \colon P$, and $[x \in S \mapsto e]$, where the predicate P and the expression e, both of which may have x as free variable, become second-order variables when quantified.

3.2.1 Normalization by rewriting. We define a rewriting process that systematically expands definitions of non-basic operators. Instead of letting the solver find instances of the background axioms introduced in Section 2, it applies the "obvious" instances of those axioms during the translation. In most cases, we can eliminate all non-basic operators. For instance, the axioms (2.5) for the UNION operator and (2.6) for the first form of comprehension yield, respectively, the rewriting rules

$$\begin{aligned} x \in \textsc{union}\ S &\longrightarrow \exists T \in S \colon x \in T, \text{ and} \\ x \in \{y \in S \colon P\} &\longrightarrow x \in S \wedge P. \end{aligned}$$

The other cases not covered by rewriting are left to the abstraction mechanism in the next subsection.

All rewriting rules defined in this paper apply equivalence-preserving transformations. To ensure soundness, we derive each rewriting rule from a theorem already proved in Isabelle/TLA$^+$. This is comparable to how rules are obtained in Isabelle's rewrite system, though in a manual way. More specifically, the theorem corresponding to a rule $a \longrightarrow b$ is $\forall \mathbf{x} : a \Leftrightarrow b$ when a and b are Boolean expressions, and $\forall \mathbf{x} : a = b$ otherwise, where $\mathbf{x}$ denotes all free variables in the rule. Most of these theorems exist already in Isabelle/TLA$^+$'s library.

The standard ZF extensionality axiom for sets (2.1) is unwieldy because it introduces an unbounded quantifier, which can be instantiated by any value of sort U. We therefore decided not to include it in the default background theory. Instead, we instantiate the extensionality property for expressions $x = y$ whenever x or y has a top-level operator that constructs a set. In these cases, we say that we *expand* equality. For each set expression T we derive rewriting rules for equations $x = T$ and $T = x$. For instance, the rules

$$\begin{aligned} x = \textsc{union}\ S &\longrightarrow \forall z \colon z \in x \Leftrightarrow \exists T \in S \colon z \in T, \text{ and} \\ x = \{z \in S : P\} &\longrightarrow \forall z \colon z \in x \Leftrightarrow z \in S \wedge P \end{aligned}$$

are derived from set extensionality (2.1) and the axioms of UNION (2.5) and of bounded set comprehension (2.6).

By not including general extensionality, the translation becomes incomplete. Even if we assume that the automated theorem provers are semantically complete, it may happen that the translation of a semantically valid TLA$^+$ formula becomes invalid when encoded. In these cases, the user will need to explicitly add the extensionality axiom as a hypothesis to the TLA$^+$ proof.

We also need to include a rule for the *contraction* of set extensionality:

$$(\forall z\colon z \in x \Leftrightarrow z \in y) \longrightarrow x = y,$$

which we apply with higher priority than the expansion rules.

All rules of the form $a \longrightarrow b$, including those introduced below for functions and CHOOSE expressions, define a term rewriting system $(\text{TLA}^+, \longrightarrow)$, where $\longrightarrow$ is a binary relation over well-formed TLA$^+$ expressions.

Theorem 1. $(\text{TLA}^+, \longrightarrow)$ *terminates and is confluent.*

Proof (sketch). Termination is simply proved by embedding $(\text{TLA}^+, \longrightarrow)$ into another reduction system that is known to terminate, typically $(\mathbb{N}, >)$ [1]. The embedding is through an ad-hoc monotone mapping μ such that $\mu(a) > \mu(b)$ for every rule $a \longrightarrow b$. We define it in such a way that every rule instance strictly decreases the number of non-basic and complex expressions such as quantifiers. Confluence is proved by Newman's lemma [1], thus it suffices to prove that all critical pairs are joinable. By enumerating all combinations of rewriting rules, we can find all critical pairs $\langle e_1, e_2 \rangle$ between them. Then we just need to prove that e_1 and e_2 are joinable for each such pair. In particular, the contraction rule is necessary to obtain a strongly normalizing system. □

3.2.2 Abstraction. Applying rewriting rules does not always suffice for obtaining formulas in basic normal form. As a toy example, consider the valid proof obligation $\forall x\colon P(\{x\} \cup \{x\}) \Leftrightarrow P(\{x\})$. The non-basic sub-expressions $\{x\} \cup \{x\}$ and $\{x\}$ do not occur in the form of a left-hand side of any rewriting rule, so they must first be transformed into a form suitable for rewriting.

We call the technique described here *abstraction* of non-basic expressions. After applying rewriting, some non-basic expression ψ may remain in the proof obligation. For all occurrences of ψ with free variables $x_1, \ldots, x_n$, we introduce in their place a fresh term $k(x_1, \ldots, x_n)$, and add the formula $k(x_1, \ldots, x_n) = \psi$ as an assumption in the appropriate context. The new term acts as an *abbreviation* for the non-basic expression, and the equality acts as its *definition*, paving the way for a transformation to a basic expression using normalization. Note that we replace non-basic expressions occurring more than once by the same symbol.

In our example the expressions $\{x\} \cup \{x\}$ and $\{x\}$ are replaced by fresh constant symbols $k_1(x)$ and $k_2(x)$. Then, the abstracted formula is

$$\begin{array}{l} \land\ \forall x\colon k_1(x) = \{x\} \cup \{x\} \\ \land\ \forall x\colon k_2(x) = \{x\} \\ \Rightarrow \forall x\colon P(k_1(x)) \Leftrightarrow P(k_2(x)). \end{array}$$

which is now in a form where it is possible to apply the instances of extensionality to the equalities in the newly introduced definitions. In order to preserve satisfiability of the proof obligation, we have to add as hypotheses instances of extensionality contraction for every pair of definitions where extensionality expansion was applied. The final equi-satisfiable formula in basic normal form is

$$\begin{array}{l}
\land\ \forall x, z \colon z \in k_1(x) \Leftrightarrow z = x \lor z = x \\
\land\ \forall x, z \colon z \in k_2(x) \Leftrightarrow z = x \\
\land\ \forall x, y \colon (\forall z \colon z \in k_1(x) \Leftrightarrow z \in k_2(y)) \Rightarrow k_1(x) = k_2(y) \\
\Rightarrow \forall x \colon P(k_1(x)) \Leftrightarrow P(k_2(x)).
\end{array}$$

3.2.3 Eliminating definitions. To improve the encoding, we introduce a procedure that eliminates definitions, having the opposite effect of the abstraction method where definitions are introduced and afterwards expanded to basic expressions. This process collects all definitions of the form $x = \psi$, and then simply applies the rewriting rules $x \longrightarrow \psi$ to substitute every occurrence of the term x by the non-basic expression ψ in the rest of the context. The definitions we want to eliminate typically occur in the original proof obligation, that is, they do not result from the abstraction step.

This transformation produces expressions that can eventually be normalized to their basic form. To avoid rewriting loops and ensure termination, it can only be applied if x does not occur in ψ. For instance, the two equations $x = y$ and $y = x + 1$ will be transformed into $y = y + 1$, which cannot further be rewritten. After applying the substitution, we can safely discard from the resulting formula the definition $x = \psi$, when x is a variable. However, we must keep the definition if x is a complex expression. Suppose we discard an assumption $\textsc{domain}\ f = S$, where the conclusion is $f \in [S \to T]$. Only after applying the rewriting rules, the conclusion will be expanded to an expression containing $\textsc{domain}\ f$, but the discarded fact required to simplify it to S will be missing.

3.2.4 Preprocessing algorithm. Now we can put together boolification and the encoding techniques described above in a single algorithm called *Preprocess*.

$$\begin{array}{llll}
\mathit{Preprocess}(\phi) \triangleq & \phi & \qquad \mathit{Reduce}(\phi) \triangleq & \phi \\
& \triangleright\ \mathit{Boolify} & & \triangleright\ \textsc{Fix}\ (\mathit{Eliminate} \circ \mathit{Rewrite}) \\
& \triangleright\ \textsc{Fix}\ \mathit{Reduce} & & \triangleright\ \textsc{Fix}\ (\mathit{Abstract} \circ \mathit{Rewrite})
\end{array}$$

Here, $\textsc{Fix}\ \mathcal{A}$ means that step $\mathcal{A}$ is executed until reaching a fixed point, the combinator $\triangleright$, used to chain actions on a formula ϕ, is defined as $\phi \triangleright f \triangleq f(\phi)$, and function composition $\circ$ is defined as $f \circ g \triangleq \lambda\phi.\, g(f(\phi))$.

Given a TLA$^+$ formula ϕ, the algorithm boolifies it and then applies repeatedly the step called *Reduce* to obtain its basic normal form. Only then the resulting formula is ready to be translated to the target language using the embedding of Section 3.3. In turn, *Reduce* first eliminates the definitions in the given formula (Sect. 3.2.3), applies the rewriting rules (Sect. 3.2.1) repeatedly, and then applies abstraction (Sect. 3.2.2) followed by rewriting repeatedly.

The *Preprocess* algorithm is sound, because it is composed of sound sub-steps, and terminates, meaning that it will always compute a basic normal formula.

Theorem 2. *The Preprocess algorithm terminates.*

Proof (idea). Observe that the elimination step is in some sense opposite to the abstraction step: the first one eliminates every definition $x = \psi$ by using it as the rewriting rule $x \longrightarrow \psi$, while the latter introduces a new symbol x in the place of an expression ψ and asserts $x = \psi$, where ψ is non-basic in both cases. That is why we apply elimination before abstraction, and why each of those is followed by rewriting. We have to be careful that *Abstract* and *Eliminate* do not repeatedly act on the same expression. *Eliminate* does not produce non-basic expressions, but *Abstract* generates definitions that can be processed by *Eliminate*, reducing them again to the original non-basic expression. That is the reason for *Rewrite* to be applied after every application of *Abstract*: the new definitions are rewritten, usually by an extensionality expansion rule. In short, termination depends on the existence of extensionality rewriting rules for each kind of non-basic expression that *Abstract* may catch. Then, for any TLA$^+$ expression there exists an equi-satisfiable basic expression in normal form that the algorithm will compute. □

3.3 Direct embedding

The preprocessing phase outputs a boolified basic TLA$^+$ expression that we will encode essentially using FOL and uninterpreted functions, without substantially changing its structure. In short, our encoding maps the given basic expression to corresponding formulas in the target language in an (almost) verbatim way.

For first-order TLA$^+$ expressions it suffices to apply a shallow embedding into first-order MS-FOL formulas. Non-logical TLA$^+$ operators are declared as function or predicate symbols with U-sorted arguments. For instance, the primitive relation $\in$ is encoded in SMT-LIB as the function $\mathsf{in} : \mathsf{U} \times \mathsf{U} \rightarrow \mathsf{Bool}$. This is the only set theoretic operator that can appear in a basic formula. Expressions like IF c THEN t ELSE u can be conveniently mapped verbatim using SMT-LIB's conditional operator to $\mathsf{ite}(c, t, u)$, where c is of sort Bool (or boolified), and t and u have the same sort.

In order to reason about the theory of arithmetic, an automated prover requires type information, either generated internally, or provided explicitly in the input language. The operators and formulas that we have presented so far are expressed in FOL over uninterpreted function symbols over the sorts U and Bool. Because we want to benefit from the prover's native capabilities for arithmetic reasoning, we declare an injective function $\mathsf{i2u} : \mathsf{Int} \rightarrow \mathsf{U}$ that embeds built-in integers into the sort U.[9] Integer literals k are simply encoded as $\mathsf{i2u}(k)$. For example, the formula $3 \in \mathit{Int}$ is translated as $\mathsf{in}(\mathsf{i2u}(3), \mathsf{tla\_Int})$, for which we have

[9] The typical injectivity axiom $\forall m^{\mathsf{Int}}, n^{\mathsf{Int}} \colon \mathsf{i2u}(m) = \mathsf{i2u}(n) \Rightarrow m = n$ generates instantiation patterns for every pair of occurrences of $\mathsf{i2u}$. Noting that $\mathsf{i2u}$ is injective iff it has a partial inverse $\mathsf{u2i}$, we use instead the axiom $\forall n^{\mathsf{Int}} \colon \mathsf{u2i}(\mathsf{i2u}(n)) = n$, which generates a linear number of $\mathsf{i2u}(n)$ instances, where $\mathsf{u2i} : \mathsf{U} \rightarrow \mathsf{Int}$ is unspecified.

to declare $\mathsf{tla\_Int} : \mathsf{U}$ and add to the translation the axiom for *Int*

$$\forall x^{\mathsf{U}} \colon \mathsf{in}(x, \mathsf{tla\_Int}) \Leftrightarrow \exists n^{\mathsf{Int}} \colon x = \mathsf{i2u}(n).$$

Observe that this axiom introduces two quantifiers to the translation. We can avoid the universal quantifier by encoding expressions of the form $x \in \mathit{Int}$ directly into $\exists n^{\mathsf{Int}} \colon x = \mathsf{i2u}(n)$, but the existential quantifier remains. Arithmetic operators over TLA$^+$ values are defined homomorphically over the image of $\mathsf{i2u}$ by axioms such as

$$\forall m^{\mathsf{Int}}, n^{\mathsf{Int}} \colon \mathsf{plus}(\mathsf{i2u}(m), \mathsf{i2u}(n)) = \mathsf{i2u}(m + n),$$

where $+$ denotes the built-in addition over integers. For other arithmetic operators we define analogous axioms.

As a result, type inference in all these cases is, in some sense, delegated to the back-end prover. The link between built-in operations and their TLA$^+$ counterparts is effectively defined only for values in the range of the function $\mathsf{i2u}$.

TLA$^+$ strings are encoded using the same technique: for every string literal that occurs in a proof obligation, we declare it as a constant of a newly declared sort Str, and assert that these constants are different from each other. Then, we use an injective function $\mathsf{str2u} : \mathsf{Str} \to \mathsf{U}$ to lift string expressions.[10]

If we call $\mathit{BasicEncode}(\phi)$ to the embedding of a basic TLA$^+$ formula ϕ into MS-FOL, we can define the whole process of encoding TLA$^+$ into MS-FOL as:

$$\begin{aligned} \mathit{Tla2MsFol}(\phi) \triangleq\ & \phi \\ & \triangleright\ \mathit{Preprocess} \\ & \triangleright\ \mathit{BasicEncode} \end{aligned}$$

3.4 Encoding functions

A TLA$^+$ function $[x \in S \mapsto e(x)]$ is akin to a "bounded" λ-abstraction: the function application $[x \in S \mapsto e(x)][y]$ reduces to the expected value $e(y)$ if the argument y is an element of S, as stated by the axiom (2.10). As a consequence, e.g., the formula

$$f = [x \in \{1, 2, 3\} \mapsto x * x] \Rightarrow f[0] < f[0] + 1, \tag{3.1}$$

although syntactically well-formed, should not be provable. Indeed, since 0 is not in the domain of f, we cannot even deduce that $f[0]$ is an integer.

We represent the application of an expression f to another expression x by two distinct first-order terms depending on whether the *domain condition* $x \in \textsc{domain}\ f$ holds or not: we introduce binary operators α and ω defined as

$$x \in \textsc{domain}\ f \Rightarrow \alpha(f, x) = f[x] \quad \text{and} \quad x \notin \textsc{domain}\ f \Rightarrow \omega(f, x) = f[x].$$

[10] This encoding does not allow us to implement the standard TLA$^+$ interpretation of strings, which are considered as tuples of characters. Fortunately, characters are hardly used in practice.

From these conditional definitions, we can derive the theorem

$$f[x] = \text{IF } x \in \text{DOMAIN } f \text{ THEN } \alpha(f,x) \text{ ELSE } \omega(f,x) \tag{3.2}$$

that gives a new defining equation for function application. In this way, functions are just expressions that are conditionally related to their argument by α and ω.

Using theorem (3.2), the expression $f[0]$ in the above example (3.1) is encoded as $\text{IF } 0 \in \text{DOMAIN } f \text{ THEN } \alpha(f,0) \text{ ELSE } \omega(f,0)$. The solver would have to use the hypothesis to deduce that $\text{DOMAIN } f = \{1,2,3\}$, reducing the condition $0 \in \text{DOMAIN } f$ to false. The conclusion can then be simplified to the formula $\omega(f,0) < \omega(f,0)+1$, which cannot be proved, as expected. Another example is $f[x] = f[y]$ in a context where $x = y$ holds: the formula is valid irrespective of whether the domain conditions hold or not.

Whenever possible, we try to avoid the encoding of function application as in the definition (3.2). From (2.10) and (3.2), we deduce the rewriting rule

$$[x \in S \mapsto e][a] \longrightarrow \text{IF } a \in S \text{ THEN } e[x \leftarrow a] \text{ ELSE } \omega([x \in S \mapsto e], a)$$

where $e[x \leftarrow a]$ denotes e with a substituted for x. This rule replaces two non-basic operators (function application and the function expression) in the left-hand side by only one non-basic operator in the right-hand side (the first argument of ω), which is required for termination of $(\text{TLA}^+, \longrightarrow)$ (Theorem 1).

In sorted languages like MS-FOL, functions have no notion of function domain other than the types of their arguments. Because explicit functions $[x \in S \mapsto e]$ cannot be mapped directly to first-order expressions, we treat them as any other non-basic expression. The following rewriting rule derived from axiom (2.10) replaces the function construct by a formula containing only basic operators:

$$f = [x \in S \mapsto e] \longrightarrow \mathit{IsAFcn}(f) \land \text{DOMAIN } f = S \land \forall x \in S : \alpha(f,x) = e$$

Observe that we have simplified $f[x]$ to $\alpha(f,x)$, because $x \in \text{DOMAIN } f$.

In order to prove that two functions are equal, we need to add a background axiom that expresses the extensionality property for functions:

$$\begin{aligned} \forall f, g :\ & \land\ \mathit{IsAFcn}(f) \land \mathit{IsAFcn}(g) \\ & \land\ \text{DOMAIN } f = \text{DOMAIN } g \\ & \land\ \forall x \in \text{DOMAIN } g : \alpha(f,x) = \alpha(g,x) \\ & \Rightarrow f = g \end{aligned}$$

Again, note that $f[x]$ and $g[x]$ were simplified using α. Unlike set extensionality, this formula is guarded by *IsAFcn*, avoiding the instantiation by expressions that are not considered functions. To prove that $\text{DOMAIN } f = \text{DOMAIN } g$, we still need to add to the translation the set extensionality axiom, which we abstain from. Instead, reasoning about the equality of domains can be solved by adding to the translation an instance of set extensionality for DOMAIN expressions only:

$$\begin{aligned} \forall f, g :\ & \land\ \mathit{IsAFcn}(f) \land \mathit{IsAFcn}(g) \\ & \land\ \forall x : x \in \text{DOMAIN } f \Leftrightarrow x \in \text{DOMAIN } g \\ & \Rightarrow \text{DOMAIN } f = \text{DOMAIN } g \end{aligned}$$

TLA$^+$ defines n-tuples as functions with domain $1\,..\,n$ and records as functions whose domain is a fixed finite set of strings. By treating them as non-basic expressions, we just need to add suitable rewriting rules to $(\text{TLA}^+, \longrightarrow)$, in particular those for extensionality expansion. For instance, a tuple $\langle e_1, e_2, \ldots, e_n \rangle$ is defined as the function

$$[i \in 1\,..\,n \mapsto \text{IF } i = 1 \text{ THEN } e_1 \text{ ELSE } (\text{IF } i = 2 \text{ THEN } e_2 \text{ ELSE } (\cdots \text{ ELSE } e_n))],$$

so that $\langle e_1, \ldots, e_n \rangle[i] = e_i$ when $i \in 1\,..\,n$. The following rule is derived from these definitions and from the axioms of extensionality (2.1) and functions (2.10):

$$\begin{aligned} t = \langle e_1, \ldots, e_n \rangle \longrightarrow\ & \land\ \mathit{IsAFcn}(t) \\ & \land\ \text{DOMAIN } t = 1\,..\,n \\ & \land\ \textstyle\bigwedge_{e_i:\mathsf{U}} \alpha(t, i) = e_i \\ & \land\ \textstyle\bigwedge_{e_i:\mathsf{Bool}} \alpha(t, i)^b \Leftrightarrow e_i \end{aligned}$$

In order to preserve the satisfiability of expressions considered as terms from those considered as formulas, we treat differently the tuple elements e_i that are Booleans (noted $e_i : \mathsf{Bool}$) from those that are not (noted $e_i : \mathsf{U}$).

3.5 Encoding CHOOSE

The CHOOSE operator is notoriously difficult for automatic provers to reason about. Nevertheless, we can exploit CHOOSE expressions by using the axioms that define them. By introducing a definition for CHOOSE $x\colon P(x)$, we obtain the theorem

$$\big(y = \text{CHOOSE } x\colon P(x)\big) \Rightarrow \big((\exists x\colon P(x)) \Leftrightarrow P(y)\big),$$

where y is some fresh symbol. This theorem can be conveniently used as a rewriting rule after abstraction of CHOOSE expressions, and for CHOOSE expressions that occur negatively, in particular, as hypotheses of proof obligations.

For determinism of choice (axiom (2.9)), suppose an arbitrary pair of CHOOSE expressions $\phi_1 \triangleq \text{CHOOSE } x\colon P(x)$ and $\phi_2 \triangleq \text{CHOOSE } x\colon Q(x)$ where the free variables of ϕ_1 are $x_1, \ldots, x_n$ (noted $\mathbf{x}$) and those of ϕ_2 are $y_1, \ldots, y_m$ (noted $\mathbf{y}$). We need to check whether formulas P and Q are equivalent for every pair of expressions ϕ_1 and ϕ_2 occurring in a proof obligation. By abstraction of ϕ_1 and ϕ_2, we obtain the axiomatic definitions $\forall \mathbf{x}\colon f_1(\mathbf{x}) = \text{CHOOSE } x\colon P(x)$ and $\forall \mathbf{y}\colon f_2(\mathbf{y}) = \text{CHOOSE } x\colon Q(x)$, where f_1 and f_2 are fresh operator symbols of suitable arity. Then, we just need to state the extensionality property for the pair f_1 and f_2 as the axiom $\forall \mathbf{x}, \mathbf{y}\colon \big(\forall x\colon P(x) \Leftrightarrow Q(x)\big) \Rightarrow f_1(\mathbf{x}) = f_2(\mathbf{y})$.

4 Evaluation

In order to validate our approach we reproved several test cases that had been proved interactively using the previously available TLAPS backend provers Zenon,

	size	ZIP	CVC4	Z3
Peterson	3	-	0.41	0.34
Peterson	10	5.69	0.78	0.80
Bakery	19	-	36.86	15.20
Bakery	223	52.74		
Memoir-T	1	-	-	1.99
Memoir-T	12	-	3.11	3.21
Memoir-T	424	7.31		
Memoir-I	8	-	3.84	9.35
Memoir-I	61	8.20		
Memoir-A	27	-	11.31	11.46
Memoir-A	126	19.10		

Finite Sets	ZIP		Zenon+SMT	
	size	time	size	time
CardZero	11	5.42	5	0.48
CardPlusOne	39	5.35	3	0.49
CardOne	6	5.36	1	0.35
CardOneConv	9	0.63	2	0.35
FiniteSubset	62	7.16	21	5.94
PigeonHole	42	7.07	20	7.01
CardMinusOne	11	5.44	5	0.75

Table 1. Evaluation benchmarks results. An entry with the symbol "-" means that the solver has reached the timeout without finding the proof for at least one of the proof obligations. The backends were executed with a timeout of 300 seconds.

Isabelle/TLA$^+$ and the decision procedure for Presburger arithmetic. We will refer to the combination of those three backends as ZIP for short.

For each benchmark, we compare two dimensions of an interactive proof: size and time. We define the *size* of an interactive proof as the number of non-trivial proof obligations generated by the Proof Manager, which is proportional to the number of interactive steps and therefore represents the user effort for making TLAPS check the proof. The *time* is the number of seconds required by the Proof Manager to verify those proofs on a 2.2GHz Intel Core i7 with 8GB of memory.

Table 1 presents the results for four case studies: type correctness and mutual exclusion of the Peterson and Bakery algorithms, type correctness (T) and refinement proofs (I, A) of the Memoir security architecture [7], and proofs of theorems about finite sets and cardinalities. We compare how proofs of different sizes are handled by the backends. Each line corresponds to an interactive proof of a given size. Columns correspond to the running times for a given SMT solver, where each prover is executed on all generated proof obligations. For our tests we have used off-the-shelf SMT solvers CVC4 v1.3 and Z3 v4.3.2.

In all cases, the use of the new SMT backend leads to significant reductions in proof sizes and running times compared to the original interactive proofs. In particular, the "shallow" proofs of the first three case studies required only minimal interaction. For instance, in the Peterson case, SMT solvers can cope with a proof that generates 3 obligations, while the ZIP backends time out in at least one of them. Instead, ZIP requires a more fine-grained proof of size 10. In the Finite Sets benchmarks, some proof obligations generated from big structural high-level formulas can be proved only by Zenon. Beyond these benchmark problems, the SMT backend has become the default backend of TLAPS.

5 Related work

In previous publications [13, 14] we presented a primitive encoding of TLA$^+$ into SMT-LIB where boolification, normalization and abstraction were not made explicit in the translation, and CHOOSE expressions were not fully supported. This paper supersedes them. Some of our encoding techniques (Section 3) were already presented before (injection of unsorted expressions [14]) or are simply folklore (*e.g.*, abstraction), but to our knowledge they have not been combined and studied in this way. Moreover, the idiosyncrasies of TLA$^+$ render their applicability non-trivial. For instance, axiomatized TLA$^+$ functions with domains, including tuples and records, are deeply rooted in the language.

The B and Z languages are also based on ZF set theory, although in a somewhat weaker version, because terms and functions have (monomorphic) types in the style of MS-FOL, thus greatly simplifying the translations to SMT languages. Another difference is that functions are defined as binary relations, as is typical in set theory. There are two SMT plugins for the Rodin tool set for Event-B. The SMT solvers plugin [5] directly encodes simple sets (*i.e.*, excluding set of sets) as polymorphic λ-expressions, which are non-standard and are only handled by the parser of the veriT SMT solver. The ppTrans plugin [9] generates different SMT sorts for each combination of simple sets, power sets and cartesian products found in the proof obligation. Therefore, there is one membership operator for every declared set sort, with the advantage that it further partitions the proof search space, although this requires that the type of every term be known beforehand. (In TLA$^+$, this can only be achieved through *type synthesis*; see [15].) Additionally, when ppTrans detects the absence of set of sets, the translation is further simplified by encoding sets by their characteristic predicates.

Similarly, Atelier-B discharges proof obligations to different SMT solvers based on Why3 [12], with sets encoded using polymorphic types. ProB includes a translation between TLA$^+$ and B [8], allowing TLA$^+$ users to use ProB tools. It relies on Kodkod, the Alloy Analyzer's backend, to do constraint solving over the first-order fragment of the language, and on the ProB kernel for the rest [16].

More recently, Delahaye et al. [6] proposed a different approach to reason about set theory, instead of a direct encoding into FOL. The theory of deduction modulo is an extension of predicate calculus that includes rewriting of terms and propositions. It is well suited for proof search in axiomatic theories such as Peano arithmetic or Zermelo set thery, as it turns axioms into rewrite rules.

MPTP [18] translates Mizar to the unsorted first-order format TPTP/FOF [17]. The Mizar language, targeted at formalized mathematics, provides second-order predicate variables and abstract terms derived from replacement and comprehension, such as the set $\{n - m \ \mathsf{where}\ m, n \ \mathsf{is}\ \mathit{Integer} : n < m\}$. During preprocessing, MPTP replaces them by fresh symbols, with their definitions at the top level. Similar to our abstraction technique, it resembles Skolemization.

6 Conclusions

We have presented a sound and effective way of discharging TLA$^+$ proof obligations using automated theorem provers based on many-sorted first-order logic. This encoding forms the core of a back-end prover that integrates external SMT solvers as oracles to the TLA$^+$ Proof System (TLAPS). The main component of the backend is a generic translation framework that makes available to TLAPS any SMT solver that supports the de facto standard format SMT-LIB/AUFLIA. Within the same framework, we have also integrated automated theorem provers based on unsorted FOL [17], such as those based on the superposition calculus.

Our translation enables the backend to successfully handle a useful fragment of the TLA$^+$ language. The untyped universe of TLA$^+$ is represented as a universal sort in MS-FOL. Purely set-theoretic expressions are mapped to formulas over uninterpreted symbols, together with relevant background axioms. The built-in integer sort and arithmetic operators are homomorphically embedded into the universal sort, and type inference is in essence delegated to the solver. Functions, tuples, records, and the CHOOSE operator (Hilbert's choice) are encoded using a preprocessing mechanism that combines term rewriting with abstraction. The soundness of the encoding is immediate: all rewriting rules and axioms about sets, functions, records, tuples, *etc.* are theorems in the background theory of TLA$^+$ that exist in the Isabelle encoding.

Encouraging results show that SMT solvers significantly reduce the effort of interactive reasoning for verifying "shallow" TLA$^+$ proof obligations, as well as some more involved formulas including linear arithmetic expressions. Both the time required to find automatic proofs and, more importantly, the size of the interactive proof, which reflects the number of user interactions, can be remarkably reduced with the new back-end prover.

The translation presented here forms the basis for further optimizations. In [15] we have explored the use of (incomplete) type synthesis for TLA$^+$ expressions, based on a type system with dependent and refinement types. Extensions for reasoning about real arithmetic and finite sequences would be useful. What is more important, we rely on the soundness of external provers, temporarily including them as part of TLAPS's trusted base. In future work we intend to reconstruct within Isabelle/TLA$^+$ (along the lines presented in [3]) the proof objects that many SMT solvers can produce. Such a reconstruction would have to take into account not only the proofs generated by the solvers, but also all the steps performed during the translation, including rewriting and abstraction.

References

1. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1999.
2. C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
3. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *Journal of Automated Reasoning*, 51(1):109–128, 2013.

4. D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto. TLA$^+$ Proofs. In D. Giannakopoulou and D. Méry, editors, *18th International Symposium on Formal Methods*, volume 7436 of *LNCS*, pages 147–154, Paris, France, 2012. Springer.
5. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT solvers for Rodin. In *3rd Intl. Conf. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*, volume 7316 of *LNCS*, pages 194–207, Pisa, Italy, 2012. Springer.
6. D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise Using Deduction Modulo. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *LPAR-19: Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, Stellenbosch, South Africa*, pages 274–290, Berlin, Heidelberg, December 2013. Springer.
7. J. R. Douceur, J. R. Lorch, B. Parno, J. Mickens, and J. M. McCune. Memoir—Formal Specs and Correctness Proofs. Technical Report MSR-TR-2011-19, Microsoft Research, 2011.
8. D. Hansen and M. Leuschel. Translating TLA$^+$ to B for Validation with ProB. In *Proceedings of the 9th International Conference on Integrated Formal Methods*, IFM'12, pages 24–38, Berlin, Heidelberg, 2012. Springer-Verlag.
9. M. Konrad and L. Voisin. Translation from set-theory to predicate calculus. Technical report, ETH Zurich, 2012.
10. L. Lamport. *Specifying Systems: The* TLA$^+$ *Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Boston, Mass., 2002.
11. M. Manzano. *Extensions of First-Order Logic.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2nd edition, 2005.
12. D. Mentré, C. Marché, J.-C. Filliâtre, and M. Asuka. Discharging Proof Obligations from Atelier B Using Multiple Automated Provers. In *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, pages 238–251, Berlin, Heidelberg, 2012. Springer-Verlag.
13. S. Merz and H. Vanzetto. Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers. In *Proceedings of the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'12, pages 289–303, Berlin, Heidelberg, 2012. Springer-Verlag.
14. S. Merz and H. Vanzetto. Harnessing SMT Solvers for TLA$^+$ Proofs. *Electronic Comm. of the European Assoc. of Software Science and Technology*, Vol. 53, 2012.
15. S. Merz and H. Vanzetto. Refinement Types for TLA$^+$. In J. M. Badger and K. Y. Rozier, editors, *NASA Formal Methods: 6th International Symposium, NFM 2014, Houston, TX, USA*, pages 143–157. Springer International, April-May 2014.
16. D. Plagge and M. Leuschel. Validating B,Z and TLA$^+$ using ProB and Kodkod. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods: 18th International Symposium, Paris, France*, pages 372–386, Berlin, Heidelberg, August 2012. Springer.
17. G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. *J. Autom. Reason.*, 43(4):337–362, Dec. 2009.
18. J. Urban. Translating Mizar for first-order theorem provers. In A. Asperti, B. Buchberger, and J. Davenport, editors, *Mathematical Knowledge Management*, volume 2594 of *LNCS*, pages 203–215. Springer, 2003.