Harnessing SMT Solvers for TLA+ Proofs

Stephan Merz, Hernán Vanzetto

Electronic Communications of the EASST
Volume 53 (2012)

Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)

Harnessing SMT Solvers for TLA$^+$ Proofs

Stephan Merz and Hernán Vanzetto

15 pages

Guest Editors: Gerald Lttgen, Stephan Merz
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Harnessing SMT Solvers for TLA$^+$ Proofs

Stephan Merz[1] **and Hernán Vanzetto**[2]

[1] stephan.merz@loria.fr
Inria Nancy Grand-Est & LORIA, Villers-lès-Nancy, France

[2] hernan.vanzetto@inria.fr
Inria Nancy Grand-Est, Villers-lès-Nancy, France
Microsoft Research-INRIA Joint Lab, Saclay, France

Abstract: TLA$^+$ is a language based on Zermelo-Fraenkel set theory and linear temporal logic designed for specifying and verifying concurrent and distributed algorithms and systems. The TLA$^+$ proof system TLAPS allows users to interactively verify safety properties of these systems. At the core of TLAPS, a *proof manager* interprets the proof language, generates corresponding proof obligations and passes them to backend provers. We recently developed a backend that relies on a typing discipline to encode (untyped) TLA$^+$ formulas into multi-sorted first-order logic for SMT solvers. In this paper we present a different encoding of TLA$^+$ formulas that does not require explicit type inference for TLA$^+$ expressions. We also present a number of techniques based on rewriting in order to simplify the resulting formulas.

Keywords: Interactive Proof, TLA+, Set Theory, SMT, Translation, Type Inference

1 Introduction

Over the past years there have been several efforts to integrate interactive and automatic theorem provers for first-order logic. SMT (satisfiability modulo theories) solvers have attracted particular interest because they combine first-order reasoning with decision procedures for theories relevant to verification such as decidable fragments of arithmetic or arrays. For example, Sledgehammer [BBP11] makes different automatic theorem provers, including SMT solvers, usable from the higher-order logic of Isabelle/HOL. Déharbe et al. [DFGV12] and Mentré et al. [MMFA12] propose the use of automatic theorem provers and SMT solvers for discharging proof obligations generated from the B method.

TLA$^+$ [Lam02] is a language for specifying and verifying systems, in particular concurrent and distributed algorithms. It is based on Zermelo-Fraenkel (ZF) set theory with the axiom of choice for specifying the data structures, and on the Temporal Logic of Actions (TLA) for describing the dynamic system behavior. Recently, TLA$^+$ has been extended by a notation for writing hierarchical proofs. The TLA$^+$ proof system TLAPS [CDLM10] is an interactive proof environment in which users can deductively verify safety properties of TLA$^+$ specifications. TLAPS is built around a *proof manager*, which interprets the proofs, expands the necessary module and operator definitions, generates corresponding proof obligations, and passes them to backend verifiers, as illustrated in Figure 1. While TLAPS is a proof assistant that relies on users for guiding the proof effort, it integrates powerful backend provers for achieving a satisfactory level of automation.

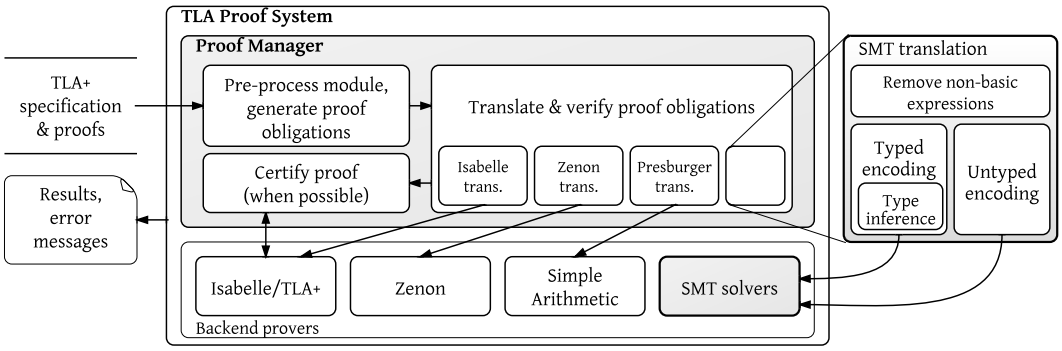

Figure 1: General architecture of TLAPS.

The main backends of the current version of TLAPS are Zenon [BDD07], a tableau prover for first-order logic with equality that includes extensions for TLA+ for reasoning about sets and functions, and Isabelle/TLA+, a faithful encoding of TLA+ in the Isabelle proof assistant, which provides automated proof methods based on first-order reasoning and rewriting. Beyond its use as a semi-automatic backend, Isabelle/TLA+ can also be used for certifying proof scripts produced by Zenon. The SimpleArithmetic backend implements a decision procedure for Presburger arithmetic. The current release of TLAPS also provides a backend [MV12] that calls upon SMT solvers for discharging "shallow" proof obligations mixing set theory, functions, and integer arithmetic: combinations of these theories are central for TLA+ specifications and proofs. In this contribution, we briefly review the SMT translation underlying the current backend, and then present an alternative SMT encoding of the untyped set theory underlying TLA+.

The input languages of state-of-the-art SMT solvers are based on many-sorted first-order logic. This allows us to design a generic translation from TLA+ expressions to an intermediate language, from which the translation to the actual input languages of particular SMT solvers is straightforward. Our backends translate to SMT-LIB [BST10], the *de facto* standard input format for SMT solvers, as well as to an extension of SMT-LIB for the solver Z3 [dB08], and to the native input language of the solver Yices [Dd06]. We outline two approaches for encoding (non-temporal) TLA+ proof obligations that encompass set theory, functions, arithmetic, records, and tuples into SMT input languages. In a first phase of both translations, the proof obligation is pre-processed to eliminate expressions that are not directly available in SMT, such as set operators or function expressions. The resulting formula will contain only TLA+ expressions that have a direct representation in the first-order logic of SMTs, including logical and arithmetic operators and conditional expressions. These formulas are translated to quantified first-order formulas over the theory of linear integer arithmetic, extended with free sort and function symbols. In particular, we make heavy use of uninterpreted functions and quantified formulas.

Since TLA+ is untyped whereas SMT input languages are sorted, a first challenge consists in assigning SMT sorts to the subexpressions of a TLA+ proof obligation. The current backend relies on the presence of typing hypotheses for constants, variables, and operators for performing type inference. The work presented in this paper, based on a suggestion by McMillan [McM11], does not require typing hypotheses, but introduces even more quantifiers in the SMT input.

In the following section we describe the formal semantics of TLA$^+$, from which we will derive the translation rules of the typed and the untyped encodings, described in Sections 3 and 4, respectively. In Section 5 we describe some techniques for simplifying the SMT input and in particular reducing the number of quantifiers. Some experimental results and conclusions appear in Sections 6 and 7.

2 Relevant constructs of TLA$^+$

The current version of TLAPS and its backends handle non-temporal TLA$^+$ expressions, which are at the heart of system verification. In the following we introduce the constructs that are relevant for the SMT backends. A more detailed presentation of the formal semantics of TLA$^+$ appears in [Lam02, Sect. 16]. TLA$^+$ is based on a variant of ZF set theory, in which the following constructs are primitive:

- The set membership operator $\in$ is taken as an uninterpreted binary predicate. Due to set extensionality, the value of any set S is determined by knowing for which values x the predicate $x \in S$ is true. TLA$^+$ assumes the standard axiom schemas of ZFC set theory.
- The standard connectives of first-order logic with equality, including Hilbert's ε operator, written as CHOOSE $x : P(x)$. The latter expression denotes an arbitrary, but fixed value v such that $P(v)$ is true if such a value exists. The choice operator is deterministic, as expressed by the rule

$$P(x) \Leftrightarrow Q(x) \;\vdash\; (\text{CHOOSE } x : P(x)) = (\text{CHOOSE } x : Q(x)) \tag{1}$$

  (where x is a fresh variable in the hypothesis), and the standard quantifiers can be defined in terms of CHOOSE, e.g.

$$(\exists x : P(x)) \;\equiv\; P(\text{CHOOSE } x : P(x)) \tag{2}$$

- The construct $f[e]$ for function application, the expression $[x \in S \mapsto e]$ that represents the function f with domain S such that $f[x] = e$ for $x \in S$, and DOMAIN f for function domain. These constructs are related by the axiom

$$f = [x \in S \mapsto e(x)] \;\Leftrightarrow\; \begin{array}{l} \wedge\ \text{DOMAIN } f = S \\ \wedge\ \forall x \in S : f[x] = e(x) \end{array} \tag{3}$$

We now show definitions of additional constructs in terms of these primitive ones.

Sets. Standard set operators are introduced by the following definitions.

$$x \in S \cup T \;\equiv\; x \in S \vee x \in T \tag{4}$$
$$x \in S \cap T \;\equiv\; x \in S \wedge x \in T \tag{5}$$
$$x \in S \setminus T \;\equiv\; x \in S \wedge x \notin T \tag{6}$$
$$S \in \text{SUBSET } T \;\equiv\; \forall x :\ x \in S \Rightarrow x \in T \tag{7}$$
$$S \subseteq T \;\equiv\; \forall x :\ x \in S \Rightarrow x \in T \tag{8}$$
$$x \in \text{UNION } S \;\equiv\; \exists T :\ T \in S \wedge x \in T \tag{9}$$

The following formulas correspond to set comprehension constructs. The empty set is a special case of (10) where $n = 0$; the empty disjunction is interpreted as FALSE. The set BOOLEAN is defined as $\{\text{TRUE}, \text{FALSE}\}$.

$$x \in \{e_1, \ldots, e_n\} \equiv x = e_1 \vee \ldots \vee x = e_n \tag{10}$$

$$x \in \{e(y_1, \ldots, y_n) : y_1 \in S_1, \ldots, y_n \in S_n\} \equiv \exists y_1, \ldots, y_n : \begin{array}{l} \wedge\ y_1 \in S_1 \\ \wedge\ \ldots \\ \wedge\ y_n \in S_n \\ \wedge\ x = e(y_1, \ldots, y_n) \end{array} \tag{11}$$

$$x \in \{y \in S : p(y)\} \equiv x \in S \wedge p(x) \tag{12}$$

Functions. By function extensionality, two functions are equal iff they have the same domain and assign the same values to each element of their domain.

$$f = g \Leftrightarrow \begin{array}{l} \wedge\ \text{DOMAIN}\ f = \text{DOMAIN}\ g \\ \wedge\ \forall x \in \text{DOMAIN}\ f : f[x] = g[x] \end{array} \tag{13}$$

The construct $[S \to T]$ denotes the set of all functions whose domain is S and whose range is a subset of T. It can be defined as follows:

$$f \in [S \to T] \equiv \begin{array}{l} \wedge\ \text{DOMAIN}\ f = S \\ \wedge\ \forall x \in S : f[x] \in T \end{array} \tag{14}$$

The construct $[f \text{ EXCEPT } ![d] = e]$ denotes the function $\hat{f}$ equal to f except that $\hat{f}[d] = e$.

$$[f \text{ EXCEPT } ![d] = e] \equiv [y \in \text{DOMAIN}\ f \mapsto \text{IF}\ y = d\ \text{THEN}\ e\ \text{ELSE}\ f[y]] \tag{15}$$

Conditionals. The semantics of the IF-THEN-ELSE construct for conditional expressions is defined in terms of CHOOSE. However, SMT also provides an analogous ite construct, which we can use directly in our translation. CASE expressions are also defined in terms of CHOOSE, by the following rules.

$$\begin{array}{l} \text{CASE}\ p_1 \to e_1 \,\square\, \ldots \,\square\, p_n \to e_n \equiv \\ \quad \text{CHOOSE}\ v : (p_1 \wedge (v = e_1)) \vee \ldots \vee (p_n \wedge (v = e_n)) \end{array} \tag{16}$$

$$\begin{array}{l} \text{CASE}\ p_1 \to e_1 \,\square\, \ldots \,\square\, p_n \to e_n \,\square\, \text{OTHER} \to e \equiv \\ \quad \text{CASE}\ p_1 \to e_1 \,\square\, \ldots \,\square\, p_n \to e_n \,\square\, \neg(p_1 \vee \ldots \vee p_n) \to e \end{array} \tag{17}$$

Tuples and Records. In TLA+, tuples are functions whose domain is an interval of numbers from 1 to n, for some natural number n.

$$\langle e_1, \ldots, e_n \rangle \equiv [i \in \{j \in \mathit{Nat} : 1 \leq j \wedge j \leq n\} \mapsto e_i] \tag{18}$$

$$S_1 \times \ldots \times S_n \equiv \{\langle y_1, \ldots, y_n \rangle : y_1 \in S_1, \ldots, y_n \in S_n\} \tag{19}$$

Records are functions whose domain is a set of strings, representing the record fields. The following three rules correspond to record selection, explicit record construct and the set of records construct, respectively.

$$e.h \equiv e[\text{“h”}] \tag{20}$$

$$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] \equiv [y \in \{\text{“h}_1\text{”}, \ldots, \text{“h}_n\text{”}\} \mapsto \text{CASE } (y = \text{“h}_1\text{”}) \to e_1 \,\square\, \ldots \,\square\, (y = \text{“h}_n\text{”}) \to e_n] \tag{21}$$

$$[h_1 : S_1, \ldots, h_n : S_n] \equiv \{[h_1 \mapsto y_1, \ldots, h_n \mapsto y_n] : y_1 \in S_1, \ldots, y_n \in S_n\} \tag{22}$$

Arithmetic. The usual arithmetic operators $+, -, *, \hat{}, \div, \%, <, \leq, >, \geq$ are defined in the standard TLA$^+$ modules *Naturals* and *Integers*, which also introduce the constants *Nat* and *Int* denoting the sets of all natural and integer numbers, respectively. The semantics of these arithmetic operators is the same as that of their counterparts in the SMT languages. The construct $m..n$ represents the set of integers k such that $m \leq k$ and $k \leq n$.

Basic TLA$^+$ expressions

The core logic of state-of-the-art SMT languages contains predefined functions and sorts, including equality, propositional operators and quantifiers, the ite (if-then-else) function and the sort Bool. We also make use of arithmetic theories that define the standard arithmetic operators and the sort Int. Those TLA$^+$ operators or expressions that are in a one-to-one correspondence with the predefined SMT operators are called *basic* TLA$^+$ operators or expressions. Namely, they are the logical and arithmetic operators and the IF-THEN-ELSE construct. The above definitions help us in reducing proof obligations to basic TLA$^+$ expressions. However, they do not eliminate CHOOSE expressions, and we will discuss in Section 5 how many of them can be abstracted.

3 Encoding based on a TLA$^+$ typing discipline

Since SMT solvers are based on multi-sorted first-order logic while TLA$^+$ is untyped, our first approach [MV12] relied on assigning types to TLA$^+$ expressions. The subsequent translation of the proof obligation makes use of this type assignment. For example, if it is known from the context that $x \in Nat$ and $y \in Nat$ then the TLA$^+$ formula $x + 42 \leq y$ can be translated to the corresponding formula over integers for the SMT solver. Otherwise $+$ and $\leq$ should be translated to uninterpreted function and predicate symbols over a sort representing unspecified TLA$^+$ values. Type inference may fail because not every set-theoretic expression is typable in a discipline compatible with SMT input languages. For instance, the expression $f[2]$ is rejected unless f is known to be a function with integer domain. In such cases the backend aborts.

If type inference succeeds, derived set and function operators are replaced according to the rules shown in Section 2. Also, equations $S = T$ for set expressions S and T are replaced by $\forall x.\, x \in S \Leftrightarrow x \in T$. Finally, formulas $x \in Int$ and $x \in Nat$ are replaced by the predicates TRUE and $x \geq 0$ (where x is an SMT expression of integer sort). These rewriting steps eliminate all occurrences of non-atomic set expressions: for every remaining subformula $exp \in S$, the expression S is either a simple identifier or of the form $Op(\vec{y})$ where Op is a user-defined operator

whose definition has not been expanded for the current proof obligation. These remaining set expressions are converted to characteristic predicates, e.g. $exp \in S$ becomes $S(exp)$, where the TLA+ identifier S is represented as an uninterpreted predicate symbol in the SMT input.

In this approach, correct type assignments are relevant for soundness: a proof obligation that is unprovable according to the semantics of untyped TLA+ must not become provable due to incorrect type annotation. As a trivial example, consider the formula $x+0=x$, which should be provable only if x is known to be of an arithmetic sort. For ensuring soundness, the type inference algorithm requires the presence of hypotheses that ascertain the types of symbols (variables or operators). These *typing hypotheses* are of the forms $x \approx exp$ and $\forall \vec{y} \in \vec{S} : f(\vec{y}) \approx exp$, where $\approx \in \{=, \in, \subseteq\}$ and *exp* is an expression whose type can already be inferred. In the above example, the formula $x+0=x$ requires a hypothesis such as $x \in Int$ in order to be typable.

Whereas it is common in TLA+ to assert typing hypotheses for state variables, typing hypotheses are less natural for operator symbols. For example, the standard induction principle for natural numbers is written as follows in TLA+:

$$\begin{array}{lll} \text{THEOREM } \mathit{NatInduction} \equiv & \text{ASSUME NEW} & P(\_), \\ & & P(0),\ \forall n \in \mathit{Nat} : P(n) \Rightarrow P(n+1) \\ & \text{PROVE} & \forall n \in \mathit{Nat} : P(n) \end{array}$$

Now assume that a users wants to prove theorem *GeneralNatInduction*, which expresses course-of-value induction, as follows:

THEOREM *GeneralNatInduction* $\equiv$
  ASSUME NEW $P(\_)$,
    $\forall n \in \mathit{Nat} : (\forall m \in 0..(n-1) : P(m)) \Rightarrow P(n)$
  PROVE $\forall n \in \mathit{Nat} : P(n)$
PROOF

$\langle 1 \rangle$. DEFINE $Q(n) \equiv \forall m \in 0..n : P(m)$	
$\langle 1 \rangle 1$. $Q(0)$	BY *SMT*
$\langle 1 \rangle 2$. $\forall n \in \mathit{Nat} : Q(n) \Rightarrow Q(n+1)$	BY *SMT*
$\langle 1 \rangle 3$. $\forall n \in \mathit{Nat} : Q(n)$	BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, *NatInduction*, *SMT*
$\langle 1 \rangle 4$. QED	BY $\langle 1 \rangle 3$, *SMT*

This proof is not accepted by the SMT backend of [MV12] because no typing hypotheses is provided for the parameter $P(\_)$. We need the extra assumption $\forall n \in \mathit{Nat} : P(n) \in \text{BOOLEAN}$ in order for type inference (and then the proof) to succeed. This additional hypothesis is unnecessary according to the semantics underlying TLAPS.

4 Untyped encoding of TLA+ formulas

McMillan [McM11] suggested an alternative encoding of TLA+ formulas for SMT solvers, which essentially delegates type inference to the SMT solver. In this approach, the universe of TLA+ values is represented by a single SMT sort U, and TLA+ operators are represented as uninterpreted function or predicate symbols whose arguments are of sort U. For example, the binary operator $\cup$ will have the SMT sort $U \times U \to U$ while the predicate $\in$ will be of sort

$U \times U \to Bool$. Where needed, we will distinguish the name of TLA$^+$ operators from their corresponding SMT counterpart by a subscript U, as in the SMT function $+_U : U \times U \to U$ which corresponds to the TLA$^+$ operator $+$. The semantics of these operators is defined axiomatically. We distinguish between operators whose definitions follow those given in Section 2, and arithmetic operators, which have native counterparts in SMT solvers.

Axiomatized operators. Instead of representing sets by their characteristic predicates (which makes it impossible to represent sets of sets or to quantify over sets), we introduce an uninterpreted binary predicate symbol $\in$ for set membership. Derived set operators are defined in terms of $\in$ and the built-in SMT theories of first-order logic and uninterpreted functions, which are common to all SMT solvers, using the definitions (4)–(11) of Section 2. For example, the axiom for $\cup$ that corresponds to the formula (4), is declared in the SMT input file as

$$\forall x, S, T : U.\ (x \in S \cup T) \Leftrightarrow (x \in S \vee x \in T).$$

Note that sets are just values of sort U, and it is therefore possible to represent sets of sets and to quantify over sets. The construct $\{e_1, \ldots, e_n\}$ for set enumeration is an n-ary expression, with $n \geq 0$. We declare separate uninterpreted functions for the arities that occur in the proof obligation, together with the corresponding axioms according to formula (10). We will discuss in Section 5 how CHOOSE expressions are handled.

Function application is encoded as a binary uninterpreted function $\mathsf{tla\_apply} : U \times U \to U$, that takes as arguments a (unary) function and its argument. In this way, functions are just elements of sort U that are related to its argument by the function tla_apply. SMT does not provide a construct corresponding to the TLA$^+$ expression $[x \in S \mapsto e(x)]$, which is similar to a λ-abstraction. Instead, we introduce a new variable $\hat{f}$ for any such expression and assert in the appropriate context the equality $\hat{f} = [x \in S \mapsto e(x)]$, rewritten according to rule (3). The resulting formula will contain only basic operators. This mechanism is the same as the abstraction method for non-basic operators described in Section 5.5.

The characteristic axiom for the operator $[S \to T]$ is derived from the formula (14). The two following axioms for the EXCEPT construct follow from (15).

$$\forall f, x, e, y : U.\quad [f \text{ EXCEPT } ![x] = e][y] \;=\; \text{IF } x = y \wedge x \in \text{DOMAIN } f \text{ THEN } e \text{ ELSE } f[y]$$
$$\forall f, x, e : U.\quad \text{DOMAIN } [f \text{ EXCEPT } ![x] = e] = \text{DOMAIN } f$$

The encoding of tuples and records is similar as in the typed encoding, but over the single sorted universe. A record $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ is represented as $\mathsf{record}_{h_1,\ldots,h_n}(e_1, \ldots, e_n)$, where $\mathsf{record}_{\vec{h}} : U \times \ldots \times U \to U$ is an n-ary record constructor. Record selection $r.h$ is encoded $(\mathsf{tla\_dot}\ r\ h)$ where h has a special sort F for fields and $\mathsf{tla\_dot} : U \times F \to U$ is an uninterpreted function. The encoding of tuples is analogous: the expression $\langle e_1, \ldots, e_n \rangle$ is represented as $\mathsf{tuple}_n(e_1, \ldots, e_n)$ for the n-ary function $\mathsf{tuple}_n : U \times \ldots \times U \to U$. The axioms for record and tuple selectors are defined as (the notation ϕ_{Int} is explained below):

$$\forall x_1, \ldots, x_n : U.\ \mathsf{record}_{h_1,\ldots,h_n}(x_1, \ldots, x_n).h_i = x_i \quad (i \in 1..n)$$
$$\forall x_1, \ldots, x_n : U.\ \mathsf{tuple}_n(x_1, \ldots, x_n)[\phi_{Int}(i)] = x_i \quad (i \in 1..n)$$

In order to prove equations between sets, such as

$$R \setminus (S \cup T) = (R \setminus S) \cap (R \setminus T)$$

we add the following extensionality axiom:

$$\forall S, T : U.\ (\forall x : U.\ x \in S \Leftrightarrow x \in T) \Rightarrow S = T.$$

Arithmetic. The axioms that we have presented so far rely on first-order logic over uninterpreted function and predicate symbols. For arithmetic reasoning, we want to benefit from the native capabilities of the SMT solvers. For this, we declare an uninterpreted, injective function $\phi_{Int} : Int \to U$ that embeds SMT integers into the sort representing TLA+ values. Integer literals k are translated as $\phi_{Int}(k)$. Similarly, predicates $exp \in Int$ and $exp \in Nat$ are translated to

$$\exists n : Int.\ exp = \phi_{Int}(n) \qquad \text{and} \qquad \exists n : Int.\ exp = \phi_{Int}(n) \wedge 0 \leq n \tag{23}$$

respectively. Note that these translations introduce existential quantifiers, which can be difficult to handle by SMT solvers, as we will see later.

Arithmetic operators over TLA+ values are defined homomorphically over the image of ϕ_{Int} by axioms such as

$$\forall m, n : Int.\ \phi_{Int}(m) +_U \phi_{Int}(n) = \phi_{Int}(m+n) \tag{24}$$

where $+$ on the right-hand side denotes the built-in addition over SMT integers. Analogous axioms are defined for the other arithmetic operators. For example, the axioms for the inequality $\leq$ and for the interval operator are, respectively:

$$\forall m, n : Int.\ \phi_{Int}(m) \leq_U \phi_{Int}(n) \Leftrightarrow m \leq n \tag{25}$$

$$\forall m, n : Int, x : U.\ x \in \phi_{Int}(m) \mathinner{..} \phi_{Int}(n) \Leftrightarrow \exists\, k : Int.\ x = \phi_{Int}(k) \wedge m \leq k \wedge k \leq n \tag{26}$$

In this way, the link between SMT operations and their TLA+ counterparts is effectively defined only for values in the range of the function ϕ_{Int}, and type inference is performed by the SMT solver during the proof attempt. This approach can be extended to other useful theories that are natively supported by some SMT solvers, such as arrays or sequences, although we have not yet done so.

Let us illustrate the interplay of these axioms on a concrete example. Consider the TLA+ proof obligation $\forall x \in Int : x + 0 = x$, which is translated as

$$\forall x : U.\ (\exists n : Int.\ x = \phi_{Int}(n)) \Rightarrow x +_U \phi_{Int}(0) = x.$$

By Skolemization, the solver introduces a new constant, say n, of sort Int, such that $x = \phi_{Int}(n)$. It can then reason as follows:

$$\begin{aligned}
x +_U \phi_{Int}(0) &= \phi_{Int}(n) +_U \phi_{Int}(0) && (x = \phi_{Int}(n)) \\
&= \phi_{Int}(n+0) && \text{(by axiom 24)} \\
&= \phi_{Int}(n) && \text{(by the SMT arithmetic decision procedure)} \\
&= x && (x = \phi_{Int}(n)).
\end{aligned}$$

Using this encoding, the proof of theorem *GeneralNatInduction* shown in Section 3 is accepted as-is, without the need of a typing hypothesis for predicate P.

Soundness. Whereas the soundness of the typed encoding relies on the correctness of type inference, which is non-trivial [MV12], soundness of the untyped encoding is immediate: all the axioms about sets, functions, records, and tuples are theorems in the background theory of TLA$^+$ that exist in the Isabelle encoding. The "lifting" axioms for the encoding of arithmetic assert that TLA$^+$ arithmetic coincides with SMT arithmetic over integers.

On the other hand, the untyped encoding introduces additional quantifiers. Reconsidering the above example for arithmetic reasoning, it is translated in the typed approach simply as

$$\forall x : \mathit{Int}.\ x + 0 = x$$

which is proved directly by the arithmetic decision procedure.

5 Refining proof obligations

5.1 Overview

The untyped encoding has simple translation rules and is easy to implement but, in practice, SMT solvers are unable to prove even the simplest proof obligations. Consider for instance the TLA$^+$ formula $2 \in 0 \mathinner{..} 3$ that is translated as $\phi_{\mathit{Int}}(2) \in \phi_{\mathit{Int}}(0) \mathinner{..} \phi_{\mathit{Int}}(3)$. This formula is obviously provable using axiom (26) about integer intervals, but SMT solvers fail to find suitable instances of the axiom formula and do not terminate. State-of-the-art SMT solvers provide *instantiation patterns* to control the potential explosion in the number of ground terms generated for instantiating quantified variables, but we have not been able to come up with patterns to attach to the axiom formulas that would significantly improve the performance.

Instead, we perform several rewriting steps to reduce the number of derived TLA$^+$ operators that occur in a proof obligation, essentially applying the "obvious" instances of the background axioms of Section 4 during the translation to SMT input format. In most cases, we can eliminate all non-basic operators, and therefore the SMT solver does not have to find suitable axiom instances. Most of our rewriting steps preserve the equivalence of formulas, with respect to the TLA$^+$ semantics.

Besides *grounding* the TLA$^+$ expressions occurring in a proof obligation to basic ones, auxiliary steps transform the obligation into a form where grounding is possible. The algorithm can be presented as follows:

1. Loop until reaching a fix-point:
   1.1. Term rewriting of top-level equalities (see Section 5.4).
   1.2. Grounding of non-basic expressions by application of rewriting rules based on the operator semantics (Section 5.2) and on the rules to disambiguate equalities (Section 5.3).
2. Abstraction of remaining non-basic operators (Section 5.5):
   2.1. Replace every non-basic expression e in the proof obligation by a new variable s_e.
   2.2. Let eq be the result of grounding the formula $s_e = e$.

2.3. Add the resulting assertion eq to the proper context.

2.4. If eq still contains non-basic operators, apply step 2 on eq.

5.2 Grounding of expressions: rewriting rules based on operator semantics

This main pre-processing step removes the non-basic operators appearing in the proof obligation. It mainly applies suitable instances of the axioms that we presented earlier, instead of letting the solver find those instances. The rewriting rules are described by a recursive operator $[\![\cdot]\!]$ that transforms a TLA+ expression to another one. For all basic constructs, they are just applied recursively to their arguments, as in

$$[\![e_1 \in e_2]\!] \equiv [\![e_1]\!] \in [\![e_2]\!] \qquad \text{or} \qquad [\![e_1 \vee e_2]\!] \equiv [\![e_1]\!] \vee [\![e_2]\!]$$

for any expressions e_1 and e_2. The definitions of non-primitive operator give rise to rewriting rules. For example, formula (4) yields

$$[\![x \in e_1 \cup e_2]\!] \equiv [\![x \in e_1 \vee x \in e_2]\!].$$

We also add some extra rules not derived from the semantics. The following rule avoids the declaration of a new variable corresponding to the construct $[x \in S \mapsto e]$ and introduced by the method described in Section 4.

$$[\![[x \in S \mapsto e(x)][a]]\!] \equiv [\![\text{IF } a \in S \text{ THEN } e(a) \text{ ELSE } \hat{y}]\!]$$

where $\hat{y}$ is a new unspecified variable that represents an unknown value.

When available, we derive rewriting rules from TLA+ definitions even for expressions that involve arithmetic. For example, the following rule defines integer intervals:

$$[\![x \in e_1..e_2]\!] \equiv [\![x \in Int \wedge e_1 \leq_U x \wedge x \leq_U e_2]\!]$$

Using these rewriting rules, the above example $2 \in 0..3$ is eventually translated to $0 \leq 2 \wedge 2 \leq 3$, avoiding an existential quantifier compared to the use of axiom (26). However, most arithmetic operators are treated as primitive and remain in the proof obligation.

5.3 Disambiguation of equalities by inferred kinds

Knowing the kind of an expression allows us to disambiguate the translation of equality, rather than lifting equality on SMT types representing sets, functions, tuples or records. The following rewriting rules for equality are derived from the corresponding extensionality axioms defined above and are added to the above grounding rules.

$$\begin{array}{lll}
[\![S = T]\!] & \equiv \forall x : [\![x \in S \Leftrightarrow x \in T]\!] & \text{if } S, T \text{ are sets} \\
[\![f = g]\!] & \equiv \wedge\, [\![\text{DOMAIN } f = \text{DOMAIN } g]\!] & \text{if } f, g \text{ are functions} \\
 & \quad \wedge \forall x : [\![x \in \text{DOMAIN } f \Rightarrow f[x] = g[x]]\!] & \\
[\![t_1 = t_2]\!] & \equiv [\![t_1[1] = t_2[1] \wedge \ldots \wedge t_1[n] = t_2[n]]\!] & \text{if } t_1, t_2 \text{ are } n\text{-tuples} \\
[\![r_1 = r_2]\!] & \equiv [\![r_1.h_1 = r_2.h_1 \wedge \ldots \wedge r_1.h_n = r_2.h_n]\!] & \text{if } r_1, r_2 \text{ are records of same shape}
\end{array}$$

The kind of expressions is determined from the context they occur in. For example, the rule for set equality will be applied whenever S or T is a complex set expression such as a union or intersection. Indeed, there is no point in rewriting equalities between two atomic symbols, which are handled natively by the SMT solver.

The CHOOSE operator of TLA$^+$ is notoriously difficult for automatic provers to reason about. Nevertheless, we can partly exploit CHOOSE expressions, in particular using the TLA$^+$ theorem

$$y = (\text{CHOOSE } x : P(x)) \;\Rightarrow\; \big((\exists x : P(x)) \Rightarrow P(y)\big). \tag{27}$$

The theorem states that if there exists some x satisfying $P(x)$, then CHOOSE $x : P(x)$ also satisfies P. We can use this to rewrite equations $y = \text{CHOOSE } x : P(x)$ that occur negatively, in particular, as hypotheses of proof obligations.

We express determinism of CHOOSE (cf. (1)) by adding the following axiom, for every pair of expressions CHOOSE $x : P(x)$ and CHOOSE $x : Q(x)$ that appear in the proof obligation:

$$(\forall x : P(x) \Leftrightarrow Q(x)) \Rightarrow (\text{CHOOSE } x : P(x)) = (\text{CHOOSE } x : Q(x)) \tag{28}$$

5.4 Term rewriting of top-level equalities

This process takes equations of the form $x = exp$ that appear as hypotheses at the top-level of a given context, where x is an identifier and exp is some expression that does not contain x. It replaces all the occurrences of x by exp in the rest of the context. Usually, exp is a complex, possibly a non-basic, expression. The purpose is to generate expressions that can potentially be grounded later. The restriction that x does not occur in exp avoids rewriting loops and ensures termination of this rewriting step. For example, the two equations $x = y$ and $y = x$ will be transformed into $y = y$, which can no longer be transformed.

As a concrete example, consider the proof obligation $T = \{1,2,3\} \Rightarrow T \subseteq Nat$. After replacing T by $\{1,2,3\}$ in the conclusion, we obtain the proof obligation $\{1,2,3\} \subseteq Nat$. This formula can be grounded, reducing it to the following formula containing only basic expressions:

$$\forall x : (x = 1 \vee x = 2 \vee x = 3) \Rightarrow (x \in Int \wedge 0 \leq x).$$

Note that if the original proof obligation were grounded before rewriting the equality, it would result in the following equisatisfiable formula with an extra quantifier:

$$\big(\forall x : x \in T \Leftrightarrow (x = 1 \vee x = 2 \vee x = 3)\big) \Rightarrow \big(\forall x : x \in T \Rightarrow (x \in Int \wedge 0 \leq x)\big).$$

5.5 Abstraction of non-basic operators

The previous rewriting steps significantly reduce the number of non-basic operators that occur in the proof obligation. However, some non-basic expressions may remain in the proof obligation because they do not occur in forms that appear as left-hand sides of rewriting rules. In those cases, we abstract any remaining non-basic expression exp by introducing a new variable s, adding the assumption $s = exp$, and replacing exp by s throughout the original formula. The equalities introduced in this way can then be rewritten to basic expressions using the above rules.

For example, consider the proof obligation

$$\forall a : P(\{a\} \cup \{\}) \Leftrightarrow P(\{a\})$$

The non-basic sub-expressions $\{a\} \cup \{\}$ and $\{a\}$ cannot be removed because they appear as arguments of the operator P and not in the forms $x \in \{a\} \cup \{\}$ and $x \in \{a\}$. New variables s_1 and s_2 are added to the appropriate context, which in this example is the scope of the quantifier $\forall a$, and the following proof obligation is obtained:

$$\begin{array}{lll} \forall a, s_1, s_2 : & \wedge & s_1 = \{a\} \cup \{\} \\ & \wedge & s_2 = \{a\} \\ & \Rightarrow & P(s_1) \Leftrightarrow P(s_2) \end{array}$$

The previous rewriting rules can now be applied to obtain the basic proof obligation

$$\begin{array}{lll} \forall a, s_1, s_2 : & \wedge & \forall x : x \in s_1 \Leftrightarrow x = a \vee \text{FALSE} \\ & \wedge & \forall x : x \in s_2 \Leftrightarrow x = a \\ & \Rightarrow & P(s_1) \Leftrightarrow P(s_2) \end{array}$$

This step is in some sense opposite to the transformation shown in Section 5.4, and should only be applied after that step. The abstraction step is frequently applied to CHOOSE expressions, since it allows us to replace an expression CHOOSE $x : P(x)$ by a new variable s, for which the equality $s =$ CHOOSE $x : P(x)$ is asserted, preparing an application of rule (27). Also, the axioms (28) for determinism of CHOOSE can be rewritten to

$$(\forall x : P(x) \Leftrightarrow Q(x)) \Rightarrow y_1 = y_2$$

for any pair of expressions $y_1 =$ CHOOSE $x : P(x)$ and $y_2 =$ CHOOSE $x : Q(x)$.

Systematically applying these rewrite rules allows us to remove all non-basic TLA+ operators, except for arithmetic. After an initial pass using the steps of Sections 5.2–5.4, abstraction is applied when non-basic operators remain in the formula, and the process is repeated until a fixed point is reached. Only then do we translate the formula to SMT input.

6 Experimental results

We have used both SMT encodings with good success on several examples that had previously been proved interactively using TLAPS and have observed significant reductions of proof sizes and running times.

In particular, the following table shows results for three case studies. The first one corresponds to a specification (assuming atomic memory access) of the N-process Bakery algorithm [Lam74] for mutual exclusion. The second one consists of invariant proofs for the Memoir security architecture [PLD+11]. The proof obligations of the Bakery example contain some basic arithmetic reasoning, while the Memoir specification makes heavy use of records. The third test case is a TLA+ module containing proofs about the cardinality of finite sets.

For each benchmark, we record the *size* of the proof, i.e. the number of non-trivial proof obligations generated by the proof manager, and the *time* in seconds required to verify those

proofs on a standard laptop. TLAPS uses short timeouts for automatic backends, hence running times are not very significant for comparison. However, the proof size corresponds to the number of leaf steps that are passed to the backend provers. It is proportional to the number of interactive steps and therefore represents the user effort for making TLAPS check the proof. We compare these figures for the original proof using the Zenon, Isabelle and SimpleArithmetic backends and for the corresponding proofs using the two SMT backends, with Z3 as the SMT solver.

	Original		Typed-SMT		Untyped-SMT	
	size	time	size	time	size	time
Bakery	120	15.66	3	2.76	4	0.67
Memoir	424	7.31	14	5.08	14	1.11
Cardinality	185	2.12	-	-	54	0.88

In all three cases, the use of the SMT backends leads to significant reductions in proof sizes compared to the original interactive proofs. In particular, the "shallow" proofs of the first two case studies required only minimal interaction.

The Bakery proof using the untyped encoding requires one more proof obligation than that of the typed encoding (3 instead of 4). The additional step states a lemma about the set of processes, whose proof is trivial. Instead of proving this lemma, one could just expand the corresponding definition in the proof of the main theorem. Unfortunately, this generates a proof obligation with 51 existential quantifiers instead of only 5 such quantifiers when the definition is left unexpanded. As a result, the SMT solvers are not able to discharge this obligation. Most of the proofs in the Cardinality benchmark contain expressions involving set of sets, which are not handled by the typed encoding because they would give rise to second-order characteristic predicates that are out of the scope of SMT solvers.

The success of the SMT backends for these and similar benchmarks are mostly due to the fact that they can handle obligations that mix set theory, functions, and arithmetic. The original Isabelle and Zenon backends have very limited support for arithmetic reasoning, while SimpleArithmetic handles only pure arithmetic formulas, requiring the user to decompose proof obligations until they fall within the respective fragments. Moreover, different users have different styles of writing interactive proofs, and the original proofs could likely have been compressed somewhat further. Nevertheless, we noticed that on several obligations, Z3 used with the untyped SMT encoding was faster than Zenon even for formulas that Zenon could handle. Comparing the two SMT encodings, the untyped encoding is more expressive – for example, it accepts obligations involving sets of sets. We were pleasantly surprised that the untyped SMT encoding was competitive with the typed encoding even for those obligations that fall into the scope of the latter, once we had implemented the optimizations described in Section 5. These are currently not implemented for the typed encoding, which probably accounts for some of the time differences in the above table.

7 Conclusions

We have presented two different approaches to translating TLA$^+$ formulas to the input formats of state-of-the-art SMT solvers. The first one, described in detail in [MV12], is based on a typing

discipline. A new approach that delegates type inference to the solvers was introduced, together with preprocessing methods to improve the translations.

Encouraging results show that automation can be significantly improved by using SMT solvers for the verification of "shallow" TLA+ proof obligations. Both the size and the processing time of the proofs can be reduced with the SMT backends. We consider the reduction in proof size to be more important, as it reflects the number of user interactions.

The backends can handle a useful fragment of TLA+, including first-order logic, sets, functions, linear arithmetic, records and tuples. Due to the translation to characteristic predicates, the typed encoding does not currently handle sets of sets. Our preprocessing techniques enable the backends to successfully handle some proof obligations involving the CHOOSE operator (Hilbert's choice), in the first-order logic of SMTs.

Comparing the two encodings, the translation based on type inference can sometimes be more efficient when it is applicable, because it generates fewer quantifiers. However, type inference may sometimes fail, and logically valid obligations cannot be proved without adding extra typing hypotheses that are unnatural in the untyped framework of TLA+. The second encoding delegates type inference to the SMT solver, based on a homomorphic embedding of arithmetic (and potentially other theories supported by the SMT solver). It is more widely applicable, and once we implemented the simplifications described in Section 5, we found its efficiency to be on a par, or even better, than that of the typed encoding. Given that it is more expressive and that is avoids TLA+ users having to provide extra typing hypotheses, it is likely that the new encoding will be the basis for the SMT backend included in the next release of TLAPS.

In future work, we intend to reconstruct proofs (along the lines presented in [AFG+11]) that many SMT solvers can produce, within Isabelle/TLA+, the trusted encoding of TLA+ as an Isabelle object logic. This would allow us to check the results of these solvers, as well as of the translation from TLA+ into SMT input, and would further raise our confidence in the SMT backend, just as currently TLAPS can direct Isabelle/TLA+ to check proofs produced by Zenon.

Acknowledgements: Damien Doligez, Leslie Lamport, and Tom Rodeheffer provided useful feedback on the SMT backends. The suggestions of the anonymous referees helped us improve the presentation, and they are gratefully acknowledged.

Bibliography

[AFG+11] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jouannaud and Shao (eds.), *1st Intl. Conf. Certified Programs and Proofs (CPP 2011)*. LNCS 7086, pp. 135–150. Springer, Kenting, Taiwan, 2011.

[BBP11] J. C. Blanchette, S. Böhme, L. C. Paulson. Extending Sledgehammer with SMT solvers. In Bjørner and Sofronie-Stokkermans (eds.), *23rd Intl. Conf. Automated Deduction*. LNCS 6803, pp. 116–130. Springer, Wroclaw, Poland, 2011.

[BDD07] R. Bonichon, D. Delahaye, D. Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In Dershowitz and Voronkov (eds.), *14th Intl.*

Conf. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2007). LNCS 4790, pp. 151–165. Springer, Yerevan, Armenia, 2007.

[BST10] C. Barrett, A. Stump, C. Tinelli. The SMT-LIB Standard: Version 2.0. In Gupta and Kroening (eds.), *Satisfiability Modulo Theories (SMT 2010)*. Edinburgh, UK, 2010. http://www.SMT-LIB.org.

[CDLM10] K. Chaudhuri, D. Doligez, L. Lamport, S. Merz. Verifying Safety Properties with the TLA$^+$ Proof System. In Giesl and Hähnle (eds.), *5th Intl. Joint Conf. Automated Reasoning (IJCAR 2010)*. Lecture Notes in Computer Science 6173, pp. 142–148. Springer, Edinburgh, UK, 2010. http://www.msr-inria.inria.fr/~doligez/tlaps/.

[dB08] L. de Moura, N. Bjørner. Z3: An Efficient SMT Solver. In Ramakrishnan and Rehof (eds.), *14th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. LNCS 4963, pp. 337–340. Springer, Budapest, Hungary, 2008.

[Dd06] B. Dutertre, L. de Moura. The Yices SMT Solver. 2006. Tool paper at http://yices.csl.sri.com/tool-paper.pdf.

[DFGV12] D. Déharbe, P. Fontaine, Y. Guyot, L. Voisin. SMT Solvers for Rodin. In *3rd Intl. Conf. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*. LNCS 7316, pp. 194–207. Springer, Pisa, Italy, 2012.

[Lam74] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM* 17(8):453–454, 1974.

[Lam02] L. Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, Mass., 2002.

[McM11] K. McMillan. A Proposal for Translating TLA$^+$ to SMT. 2011. Personal communication.

[MMFA12] D. Mentré, C. Marché, J.-C. Filliâtre, M. Asukaa. SMT Solvers for Rodin. In *3rd Intl. Conf. Abstract State Machines, Alloy, B, VDM, and Z (ABZ 2012)*. LNCS 7316, pp. 238–251. Springer, Pisa, Italy, 2012.

[MV12] S. Merz, H. Vanzetto. Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers. In Bjørner and Voronkov (eds.), *18th Intl. Conf. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-18)*. Lecture Notes in Computer Science 7180, pp. 289–303. Springer, 2012.

[PLD+11] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, J. M. McCune. Memoir: Practical State Continuity for Protected Modules. In *IEEE Symp. Security and Privacy*. IEEE Computer Society, Berkeley, California, U.S.A., 2011. Formal Specifications and Correctness Proofs: Tech. Report, Microsoft Research, Feb. 2011.