Automatic Verification Of TLA$^+$ Proof Obligations With SMT Solvers

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

[1] INRIA Nancy Grand-Est & LORIA, Nancy, France
[2] Microsoft Research-INRIA Joint Centre, Saclay, France.

Abstract. TLA$^+$ is a formal specification language that is based on ZF set theory and the Temporal Logic of Actions TLA. The TLA$^+$ proof system TLAPS assists users in deductively verifying safety properties of TLA$^+$ specifications. TLAPS is built around a *proof manager*, which interprets the TLA$^+$ proof language, generates corresponding proof obligations, and passes them to backend verifiers. In this paper we present a new backend for use with SMT solvers that supports elementary set theory, functions, arithmetic, tuples, and records. Type information required by the solvers is provided by a typing discipline for TLA$^+$ proof obligations, which helps us disambiguate the translation of expressions of (untyped) TLA$^+$, while ensuring its soundness. Preliminary results show that the backend can help to significantly increase the degree of automation of certain interactive proofs.

1 Introduction

TLA$^+$ [10] is a language for specifying and verifying systems, in particular concurrent and distributed algorithms. It is based on a variant of Zermelo-Fraenkel (ZF) set theory for specifying the data structures, and on the Temporal Logic of Actions (TLA) for describing the dynamic system behavior. Recently, a first version of the TLA$^+$ proof system TLAPS [5] has been developed, in which users can deductively verify safety properties of TLA$^+$ specifications. TLA$^+$ contains a declarative language for writing hierarchical proofs, and TLAPS is built around a *proof manager*, which interprets this proof language, expands the necessary module and operator definitions, generates corresponding proof obligations (POs), and passes them to backend verifiers, as illustrated in Figure 1. While TLAPS is an interactive proof environment that relies on users guiding the proof effort, it integrates automatic backends to discharge proof obligations that users consider trivial.

The two main backends of the current version of TLAPS are Zenon [4], a tableau prover for first-order logic and set theory, and Isabelle/TLA$^+$, a faithful encoding of TLA$^+$ in the Isabelle [13] proof assistant, which provides automated proof methods based on first-order reasoning and rewriting. The backends available prior to the work presented here also included a generic translation to the input language of SMT solvers that focused on quantifier-free formulas of linear arithmetic (not shown in Fig. 1). This SMT backend was occasionally useful

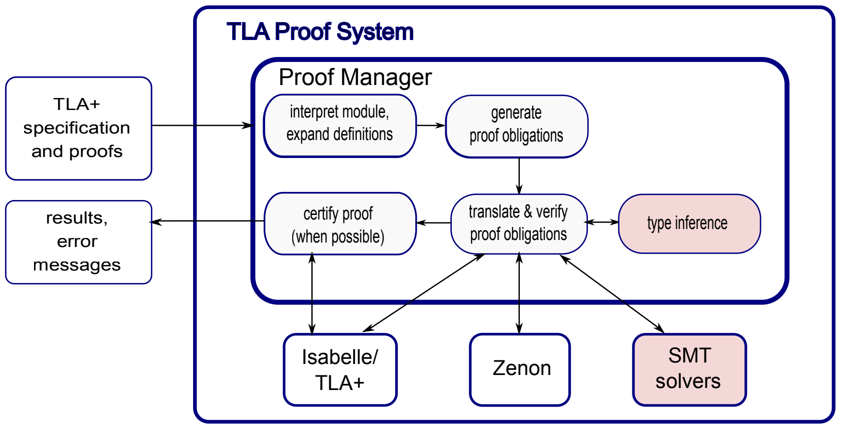
TLA Proof System
Proof Manager
TLA+ specification and proofs
results, error messages
interpret module, expand definitions
generate proof obligations
certify proof (when possible)
translate & verify proof obligations
type inference
Isabelle/ TLA+
Zenon
SMT solvers

Fig. 1. General architecture of TLAPS.

because the other backends perform quite poorly on obligations involving arithmetic reasoning. However, it covered a rather limited fragment of TLA$^+$, which heavily relies on modeling data using sets and functions. Assertions mixing arithmetic, sets and functions arise frequently in TLA$^+$ proofs.

In the work reported here we present a new SMT-based backend for (non-temporal) TLA$^+$ formulas that encompasses set-theoretic expressions, functions, arithmetic, records, and tuples. By evaluating the performance of the backend over several existing TLA$^+$ proofs we show that it achieves good coverage for "trivial" proof obligations. The new modules comprising our backend appear shaded in Figure 1.

State of the art and context. Over the last years there have been several efforts to integrate interactive and automatic theorem provers (ATPs). ATP systems are satisfiability solvers for first-order logic, while the Satisfiability Modulo Theories (SMT) approach combines first-order reasoning with decision procedures for theories such as equality, integer and real arithmetic, arrays and bit-vectors. For example, Sledgehammer [3] integrates Isabelle/HOL, the encoding of polymorphic higher-order logic, with ATPs and SMT solvers. The translation to SMT is allowed to be unsound, since proof scripts produced by the solvers are reconstructed and verified in the trusted kernel of Isabelle/HOL.

TLA$^+$ is an untyped language, which makes it very expressive and flexible, but also makes automated reasoning quite challenging [11]. Since TLA$^+$ variables can assume *any* value, it is customary to start any verification project by proving a so-called *type invariant* that associates every variable of the specification with the set of values that variable may assume. Most subsequent correctness proofs rely on the type invariant. It should be noted that TLA$^+$ type invariants frequently express more sophisticated properties than what could be ensured by a decidable type system.

The input languages of state-of-the-art SMT solvers are based on many-sorted first-order logic. This allows us to design a unique translation from TLA$^+$ expressions to an intermediate language, from which the translation to the actual target languages of particular SMT solvers is straightforward. The considered languages are: (i) SMT-LIB [1], the *de facto* standard input format for SMT solvers (in our experiments we use the CVC3 solver [2] as a "baseline"), (ii) an extension of SMT-LIB for the solver Z3 [6], and (iii) the native input language of the solver Yices [8]. Using SMT-LIB as the target of our translation, TLAPS can be independent of any particular solver. Z3 adds support for datatypes, that allows us to easily encode tuples and records. On the other hand, the Yices language provides useful concepts such as sub-typing or a direct representation of tuples, records and λ-terms. The considered TLA$^+$ 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 we do not restrict ourselves to quantifier-free formulas.

The first challenge is therefore to design a typing discipline that is compatible with the logics of SMT solvers but accommodates typical TLA$^+$ specifications. In a first step of the translation, a suitable type is assigned to every expression that appears in the proof obligation. We make use of this type assignment during the translation of expressions. For example, equality between integer expressions will be translated differently from equality between sets or functions.

Type inference may fail because not every set-theoretic expression is typable according to our typing discipline, and in this case the backend aborts. Otherwise, the proof obligation is translated to SMT formulas. Observe that type inference is relevant for the soundness of the SMT backend: 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 arithmetic sort. Type inference essentially relies on assumptions that are present in the proof obligation and that constrain the values of symbols (variables or operators).

Paper outline. A brief introduction to TLA$^+$ and the input languages of SMT solvers appear in the next section. A type system for TLA$^+$ together with its inference algorithm is described in Section 3, and the translation rules in Section 4. Results for some case studies and conclusions are given in Sections 5 and 6.

2 TLA$^+$ and the SMT languages

2.1 The non-temporal fragment of TLA$^+$

Our backend handles non-temporal TLA$^+$ expressions, which make up the vast majority of proof obligations that arise in TLA$^+$ developments. For the purposes of this paper we fix a core subset of the language to illustrate just the main challenges, where we only include main primitive operators and constructs. This fragment of the language, named ξ, is defined and described below by the following simplified grammar, which defines TLA$^+$ expressions ϕ, where *Id* is an identifier name for a constant, variable, record field or operator with possible arguments. Other symbols include strings and integer numbers.

$$
\begin{array}{rll}
\phi ::= & \mathit{Id} \mid \mathit{Id}(\phi, \ldots, \phi) \mid \mathit{String} \mid \mathit{Number} \mid (\phi) & \\
& \mid \textsc{true} \mid \textsc{false} \mid \textsc{boolean} \mid \mathrm{Nat} \mid \mathrm{Int} & \text{(atomic expressions)} \\
& \mid \textsc{if}\ \phi\ \textsc{then}\ \phi\ \textsc{else}\ \phi \mid \neg\phi & \text{(conditional, negation)} \\
& \mid \phi\ [\ \wedge \mid \in \mid \cup \mid \subseteq \mid = \mid + \mid <\ ]\ \phi & \text{(infix operators)} \\
& \mid \forall\ \mathit{Id} : \phi \mid \exists\ \mathit{Id} : \phi & \text{(quantifiers)} \\
& \mid \{\} \mid \{\phi, \ldots, \phi\} & \text{(enumerated sets)} \\
& \mid [\mathit{Id} \in \phi \mapsto \phi] \mid [\phi \to \phi] \mid \phi[\phi] \mid \textsc{domain}\ \phi & \text{(function expressions)} \\
& \mid [\mathit{Id} \mapsto \phi, \ldots, \mathit{Id} \mapsto \phi] \mid \phi.\mathit{Id} \mid \langle \phi, \ldots, \phi \rangle & \text{(records and tuples)}
\end{array}
$$

- The basic set operators consist of $\in$, $\cup$, and $\subseteq$. In TLA$^+$, equality is also an operator of set theory, since it formally means equality of sets.
- Operators on functions include function application $f[e]$, DOMAIN f (domain of function f), $[x \in S \mapsto e]$ (the function f such that $f[x] = e$ for $x \in S$), and $[S \to T]$ (the set of functions f with domain S and $f[x] \in T$ for $x \in S$).
- TLA$^+$ provides the usual operators of propositional logic; our restricted fragment includes $\wedge$ and $\neg$. BOOLEAN denotes the set $\{\textsc{true}, \textsc{false}\}$. Quantified formulas are of the form $Qx : e$, where $Q \in \{\forall, \exists\}$.
- A record $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ is a function whose domain is the finite set of strings $\{“h_1”, \ldots, “h_n”\}$. Access to record fields is written $r.h$, abbreviating $r[“h”]$, thus $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n].h_i = e_i$. Similarly, an n-tuple $\langle e_1, \ldots, e_n \rangle$ is a function whose domain is $\{1, \ldots, n\}$ and $\langle e_1, \ldots, e_n \rangle[i] = e_i$, for $1 \leq i \leq n$.
- A TLA$^+$ operator is a symbol of arity 0 or higher. Operators are associated with definitions whose expansion is controlled by the user. Expansion of defined operators is handled by the proof manager: definitions for operators that occur in proof obligations passed to backends are hidden. Operators by themselves are not expressions (they correspond to class functions in set theory), and they cannot be quantified over.
- Finally, the arithmetic operators include $+$ and $<$. Nat and Int are the sets of natural and integer numbers, respectively. We also include the construct IF/THEN/ELSE for conditional expressions.

We omit several features from this simplified description that can be introduced as syntactic sugar and that are handled by our backend, such as EXCEPT constructs for functions, and set comprehension. The extension to a larger subset of the language is straightforward following the TLA$^+$ semantics. A notable TLA$^+$ construct that is not handled by our backend is the CHOOSE primitive, known as Hilbert's ε operator. A detailed description of the full TLA$^+$ syntax and semantics can be found in [10, Chap. 16].

2.2 Input languages of SMT solvers

The input languages of SMT solvers are based on a many-sorted first-order logic. Accordingly, each well-formed expression has a unique sort. The languages provide syntax and commands for declaring new sort and function symbols, and for asserting formulas over the resulting signature. With each function symbol are associated the sorts of its arguments and its result sort. Terms and formulas are written in a Lisp-like language.

In particular, the SMT-LIB [1] initiative provides a common input format, as well as a repository of benchmarks, for SMT solvers. In general, an SMT input file is a sequence of declarations of sorts, functions, assumptions, and a goal. It is related with a logic, identified by a pre-established name, to which are associated sort and function declarations, and possibly syntactic and semantic restrictions. Our backend produces formulas for the SMT-LIB logic AUFLIA, which supports quantified formulas over the theory of linear integer arithmetic extended with free sort and function symbols. In this logic the predefined sorts are Bool and Int. Set theory is not currently supported natively by any pre-defined logic in SMT-LIB.

Simplified grammars for SMT-LIB sorts and terms can be given as follows:

$$\begin{aligned}\sigma &::= s \mid (s\ \sigma^+)\\ t &::= x \mid \mathit{Number} \mid (f\ t^+) \mid ([\mathsf{forall}|\mathsf{exists}]\ (((x\ \sigma))^+)\ t)\end{aligned}$$

where σ is a sort, s is a sort identifier, t is a term, x is a variable symbol, and f is a function symbol. A sort constructor is defined by its name and the argument sorts. A function declaration is composed of the function symbol, a list of argument sorts, and the sort of the result. Constants are simply functions with no arguments. SMT-LIB provides by default a Boolean sort for terms and the standard functions and, or, not, true and false. The logic AUFLIA provides a sort for integer numbers and the arithmetic functions $+$, $-$, $*$, $/$, $<$, $<=$, $>=$ and $>$, to which we add an extra sort of arity 0 to represent the universe of TLA$^+$ constants of unspecified sort[3].

The Z3 input format is an extension of the one defined above, in particular adding algebraic datatypes that we use for representing tuples and records. The structure and syntax of the Yices input format are similar to SMT-LIB, and supports lambda expressions, tuples, and records. Boolean and integer sorts are also pre-defined, as well as a sort for natural numbers.

3 Type inference for TLA$^+$

We define a type system for TLA$^+$ expressions that underlies our SMT translation. We consider types τ according to the following grammar:

$$\tau ::= \bot \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Nat} \mid \mathsf{Int} \mid \mathbf{P}\,\tau \mid \tau \to \tau \mid \mathsf{Rec}\,\{h_i \mapsto \tau_i\} \mid \mathsf{Tup}\,[\tau_i].$$

[3] The logic AUFLIA also provides a theory of arrays, that we do not make use of.

The atomic types are $\bot$ (terms of unspecified type), Bool (propositions), strings, and natural and integer numbers. Complex types are sets (of base type τ), functions, records (defined by a mapping from field names h_i to types) and tuples (as a fixed-size list of types). A partial order $\leq$ on types, with $\bot$ as the smallest element, is defined as the least reflexive and transitive relation that satisfies

$$\begin{array}{ll}
\bot \leq \tau & \text{for any type } \tau \\
\mathbf{P}\,\tau_1 \leq \mathbf{P}\,\tau_2 & \text{if } \tau_1 \leq \tau_2 \\
\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' & \text{if } \tau_1 \leq \tau_1' \text{ and } \tau_2 \leq \tau_2' \\
\mathsf{Rec}\,\{h_i \mapsto \tau_i\}_{i \in 1..n} \leq \mathsf{Rec}\,\{h_i \mapsto \tau_i'\}_{i \in 1..n'} & \text{if } n \leq n' \text{ and } \tau_i \leq \tau_i' \text{ for } 1 \leq i \leq n \\
\mathsf{Tup}\,[\tau_i]_{i \in 1..n} \leq \mathsf{Tup}\,[\tau_i']_{i \in 1..n} & \text{if } \tau_i \leq \tau_i' \text{ for } 1 \leq i \leq n \\
\mathsf{Nat} \leq \mathsf{Int} &
\end{array}$$

We define an inference algorithm for this type system that is based on an operator $[\![e, \varepsilon]\!]_I$ whose arguments are a TLA$^+$ expression e and an expected lower bound ε for the type of e according to the partial order. The computation either returns the inferred type or fails. The operator recurses over the structure of TLA$^+$ expressions, gathering information in a typing environment *type*, that maps each TLA$^+$ symbol to its type. Therefore, $\mathit{type}(x)$ is the type of symbol x.

Initially, we consider that every symbol has the unspecified type $\bot$. Recursive calls to the operator $[\![\cdot]\!]_I$ may update the type of symbols as recorded in *type* by new types that are larger than the previous ones. A type assignment is definitive only when types for all expressions in the proof obligation have been successfully inferred. For example, consider a proof obligation including two hypotheses $S = \{\}$ and $S \subseteq \mathsf{Int}$. After evaluating the first one, S will have type $\mathbf{P}\,\bot$, but it will be updated to $\mathbf{P}\,\mathsf{Int}$ when the second hypothesis is processed.

The rules of $[\![\cdot]\!]_I$ are defined in Figures 2 and 3. Before describing them, we introduce some preliminary definitions and notations. The rules are defined operationally: for example, we write $[\![\ldots]\!]_I \equiv f; g$ to indicate that f is evaluated first and the result of the overall rule is the result computed by g. We also use *if* and *case* with their usual meanings, *let* that performs pattern matching, and $:=$ for variable assignment. The base function $\mathbf{b}\,\tau$ is the dual of $\mathbf{P}\,\tau$, and is defined by $\mathbf{b}\,\mathbf{P}\,\tau = \tau$, whereas $\mathbf{b}\,\tau$ fails if τ is not a set type. The function $\mathsf{ch}(c)$ fails when condition c is false. The function $\mathsf{ret}(c(\tau))$ returns the type τ if $c(\tau)$ is satisfied, otherwise fails. For example, rule (3.6) first checks that the minimum type ε is $\bot$ or Bool, it tries to "equalize" the type of both subexpressions, and then the resulting type τ is checked to be Bool before returning it. The function max returns the greater of two comparable types as defined by

$$\max(\tau_1, \tau_2) \equiv \text{if } \tau_1 \leq \tau_2 \text{ then } \tau_2 \text{ else (if } \tau_2 \leq \tau_1 \text{ then } \tau_1 \text{ else fail).}$$

The typing environment is updated only when evaluating symbols (rule 3.1), where $\mathit{type} \oplus s$ denotes the typing environment *type*, updated with the mapping s. The rules (3.8) and (3.21) introducing bound variables silently rename the variables in order to avoid any clashes with symbols already introduced. Just as any other symbol, these fresh variables are initially assigned type $\bot$. Upon recursive calls to $[\![\cdot]\!]_I$, appropriate return types are passed on to subexpressions,

Symbols and other constructs

$$[\![x,\varepsilon]\!]_I \equiv \alpha := \max(\mathit{type}(x),\varepsilon);\, \mathit{type} := \mathit{type} \oplus \{x \mapsto \alpha\};\, \alpha \tag{3.1}$$

$$[\![\text{TRUE},\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Bool});\mathsf{Bool} \qquad [\![\text{FALSE},\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Bool});\mathsf{Bool} \tag{3.2}$$

$$[\![\text{BOOLEAN},\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathbf{P}\,\mathsf{Bool});\mathbf{P}\,\mathsf{Bool} \qquad [\![\text{“...”},\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Str});\mathsf{Str} \tag{3.3}$$

$$[\![e(e_1,\dots,e_n),\varepsilon]\!]_I \equiv \text{let } \alpha_1 = [\![e_1,\bot]\!]_I,\dots,\alpha_n = [\![e_n,\bot]\!]_I \text{ in}$$
$$\text{let } (\_ \to \dots \to \_ \to \alpha_{n+1}) = [\![e,\alpha_1 \to \dots \to \alpha_n \to \varepsilon]\!]_I \text{ in } \alpha_{n+1} \tag{3.4}$$

$$[\![\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2,\varepsilon]\!]_I \equiv \mathsf{ch}([\![p,\mathsf{Bool}]\!]_I = \mathsf{Bool});\mathsf{eq}([e_1,e_2],\varepsilon) \tag{3.5}$$

Logic

$$[\![e_1 \wedge e_2,\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Bool});\mathsf{eq}([e_1,e_2],\mathsf{Bool});\mathsf{Bool} \tag{3.6}$$

$$[\![\neg\, e,\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Bool});[\![e,\mathsf{Bool}]\!]_I;\mathsf{Bool} \tag{3.7}$$

$$[\![Q\ x : e,\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Bool});[\![e,\mathsf{Bool}]\!]_I;\mathsf{Bool} \quad \text{for } Q \in \{\forall,\exists\} \tag{3.8}$$

Arithmetic

$$[\![e_1 + e_2,\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Int});\alpha := \mathsf{eq}([e_1,e_2],\varepsilon);\mathsf{ret}(\alpha \in \{\mathsf{Nat},\mathsf{Int}\}) \tag{3.9}$$

$$[\![e_1 < e_2,\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Bool});\mathsf{ch}(\mathsf{eq}([e_1,e_2],\mathsf{Nat}) \le \mathsf{Int});\mathsf{Bool} \tag{3.10}$$

$$[\![n,\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Int});\mathsf{ret}(\mathsf{Nat} \le \varepsilon) \quad \text{(where } n \text{ is a number)} \tag{3.11}$$

$$[\![\mathsf{Nat},\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathbf{P}\,\mathsf{Int});\mathsf{ret}(\mathbf{P}\,\mathsf{Nat} \le \varepsilon) \qquad [\![\mathsf{Int},\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathbf{P}\,\mathsf{Int});\mathbf{P}\,\mathsf{Int} \tag{3.12}$$

Sets

$$[\![e_1 = e_2,\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Bool});\mathsf{eq}([e_1,e_2],\bot);\mathsf{Bool} \tag{3.13}$$

$$[\![S \subseteq T,\varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \le \mathsf{Bool});\mathsf{eq}([S,T],\mathbf{P}\,\bot);\mathsf{Bool} \tag{3.14}$$

$$[\![e_1 \in e_2,\varepsilon]\!]_I \equiv [\![\{e_1\} \subseteq e_2,\varepsilon]\!]_I \tag{3.15}$$

$$[\![S \cup T,\varepsilon]\!]_I \equiv \max(\mathbf{P}\,\bot,\mathsf{eq}([S,T],\varepsilon)) \tag{3.16}$$

$$[\![\{\},\varepsilon]\!]_I \equiv \max(\mathbf{P}\,\bot,\varepsilon) \tag{3.17}$$

$$[\![\{e_1,\dots,e_n\},\varepsilon]\!]_I \equiv \mathbf{P}\,\mathsf{eq}([e_1,\dots,e_n],\mathbf{b}\varepsilon) \tag{3.18}$$

Functions

$$[\![f[e],\varepsilon]\!]_I \equiv \alpha := [\![e,\bot]\!]_I;\text{let } (\alpha' \to \beta) = [\![f,\alpha \to \varepsilon]\!]_I \text{ in } (\mathsf{ch}(\alpha = \alpha');\beta) \tag{3.19}$$

$$[\![\text{DOMAIN } f,\varepsilon]\!]_I \equiv \text{let } (\alpha \to \_) = [\![f,\mathbf{b}\varepsilon \to \bot]\!]_I \text{ in } \mathbf{P}\,\alpha \tag{3.20}$$

$$[\![[x \in S \mapsto e],\varepsilon]\!]_I \equiv \text{case } \varepsilon \text{ of } \mid \alpha \to \beta : \ \mathbf{b}[\![S,\mathbf{P}\,\alpha]\!]_I \to [\![e,\beta]\!]_I$$
$$\mid \bot : \ \mathbf{b}[\![S,\mathbf{P}\,\bot]\!]_I \to [\![e,\bot]\!]_I \tag{3.21}$$

$$[\![[S \to T],\varepsilon]\!]_I \equiv \text{case } \varepsilon \text{ of } \mid \mathbf{P}\,(\alpha \to \beta) : \ \mathbf{P}\,(\mathbf{b}[\![S,\mathbf{P}\,\alpha]\!]_I \to \mathbf{b}[\![T,\mathbf{P}\,\beta]\!]_I)$$
$$\mid \bot : \ \mathbf{P}\,(\mathbf{b}[\![S,\mathbf{P}\,\bot]\!]_I \to \mathbf{b}[\![T,\mathbf{P}\,\bot]\!]_I) \tag{3.22}$$

Fig. 2. Rules for the type inference operator $[\![\cdot]\!]_I$.

Records and Tuples

$$\llbracket r.h, \varepsilon \rrbracket_I \;\equiv\; \text{let } \mathsf{Rec}\,\{\ldots, h \mapsto \alpha, \ldots\} = \llbracket r, \mathsf{Rec}\,\{h \mapsto \varepsilon\} \rrbracket_I \text{ in } \alpha \tag{3.23}$$

$$\llbracket t[i], \varepsilon \rrbracket_I \;\equiv\; \text{let } \mathsf{Tup}\,[\ldots, \alpha_i, \ldots] = \llbracket t, \bot \rrbracket_I \text{ in } \max(\alpha_i, \varepsilon) \tag{3.24}$$

$$\llbracket [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n], \varepsilon \rrbracket_I \equiv \text{case } \varepsilon \text{ of } \begin{array}{ll} \mid \mathsf{Rec}\,\{h_i \mapsto \varepsilon_i\} : & \mathsf{Rec}\,\{h_i \mapsto \llbracket e_i, \varepsilon_i \rrbracket_I\} \\ \mid \bot : & \mathsf{Rec}\,\{h_i \mapsto \llbracket e_i, \bot \rrbracket_I\} \end{array} \tag{3.25}$$

$$\llbracket \langle e_1, \ldots, e_n \rangle, \varepsilon \rrbracket_I \equiv \text{case } \varepsilon \text{ of } \begin{array}{ll} \mid \mathsf{Tup}\,[\varepsilon_i] : & \mathsf{Tup}\,[\llbracket e_i, \varepsilon_i \rrbracket_I] \\ \mid \bot : & \mathsf{Tup}\,[\llbracket e_i, \bot \rrbracket_I] \end{array} \tag{3.26}$$

Fig. 3. Rules for the type inference operator $\llbracket \cdot \rrbracket_I$ (continued).

propagating the type information through the formula. The type information associated with a symbol x is updated to a larger type when so required by the expected minimum type ε.

The operator $\llbracket \cdot \rrbracket_I$ assigns types to complex expressions based on the types of their constituents. Although expressions such as $\langle a \rangle \cup 0$ or $3 + \text{TRUE}$ appear silly, they are allowed in TLA$^+$, yet their meaning is unknown. Our fragment rules out such expressions by enforcing a typing discipline that requires subexpressions to have types compatible with the larger expression.

When type inference succeeds on a proof obligation, the typing environment *type* will contain the resulting final type assignments. There are two reasons why the inference algorithm may fail: (1) The expected type ε or the type obtained from subexpressions can be incompatible with the type associated with primitive operators, as in the examples given above. The inference rules check for this kind of mismatch using the operators ch and ret. (2) Type inference can fail to solve a constraint stating that two or more expressions need to be of the same type, as we discuss next.

The sorting discipline of SMT solvers requires in several cases that subexpressions of a TLA$^+$ expressions be assigned the same type. This is in particular true for the expressions e_1 and e_2 in $e_1 = e_2$, $e_1 \subseteq e_2$, IF p THEN e_1 ELSE e_2 (rules 3.13, 3.14, 3.5); the second of these expressions moreover requires e_1 and e_2 to be of set type. Arithmetic expressions (rules 3.9-3.10) and set operators (3.16 and 3.18) pose similar constraints. The expression $e_1 \in e_2$ requires e_2 to be of type $\mathbf{P}\,\alpha$ and e_1 of type α (rule 3.15), and $f[e]$ requires f to be of type $\alpha \to \alpha'$, and e of type α (rule 3.19), for some types α, α'. Similarly, we do not allow different applications of the same function or operator symbol to return values of different types. For those cases, the type inference rules make use of the function $\mathsf{eq}([e_1, \ldots, e_n], \varepsilon) : \tau$, that given a list of expressions $e_1, \ldots, e_n$ and an expected type ε, returns the common type of all expressions e_i (bounded below by ε), or fails if no such type can be assigned.

Type inference proceeds in three steps, that all rely on (variants of) the operator $\llbracket \cdot \rrbracket_I$. We will explain the algorithm using a proof obligation whose hypotheses are $x \in S$ and $S \subseteq \mathsf{Nat}$ and whose conclusions are $x + 0 = x$ and $y \cup \{\} = y$. In the first step, the algorithm computes an approximate type assignment for the proof obligation by applying the operator $[\![\cdot]\!]_I^{safe}$, which differs from $[\![\cdot]\!]_I$ by restricting types to *safe* types defined by the grammar $\tau_s ::= \bot \mid \mathbf{P}\,\tau_s$ (the rules of Figs. 2 and 3 are adapted accordingly). In other words, this step only distinguishes between elementary values and sets, and ensures that all symbols that appear in the proof obligation are used consistently according to these categories. In our running example, it infers types $\bot$ for the symbol x, and $\mathbf{P}\,\bot$ for y and S.

The second step refines this type assignment by running the operator $[\![\cdot]\!]_I$ on "typing hypotheses", i.e. available facts of the forms

$$x \otimes e \qquad \text{and} \qquad \forall a_1 \in S_1, \ldots, a_n \in S_n : x(a_1, \ldots, a_n) \otimes e,$$

for $\otimes \in \{=, \in, \subseteq\}$, starting from the typing environment computed during the first step. In these expressions, x is a constant, variable or operator, and e is an expression whose type can already be inferred.[4] These typing hypotheses are obtained by decomposing the assumptions present in a proof obligation by elementary heuristics. In our example, we have the typing hypotheses $x \in S$ and $S \subseteq \mathsf{Nat}$, and the previously inferred types for x and S will be refined to Nat and $\mathbf{P}\,\mathsf{Nat}$. The reason to perform this step on a restricted set of facts is to avoid the evaluation of $[\![\cdot]\!]_I$ on hypotheses such as $z \notin Nat$, which would incorrectly assign type Nat to z.

The third step ensures that the entire proof obligation can be typed using the typing environment computed in the first two steps. It does so by applying the operator $[\![o]\!]_I^{check}$, which differs from $[\![\cdot]\!]_I$ in that the rule 3.1 for symbols is defined as $[\![x, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathit{type}(x))$. In particular, the typing environment is not updated. This step will succeed for our running example, given the previously assumed typing hypotheses for x. (No typing hypothesis is needed for y.) However, it would fail without an appropriate typing hypothesis because the subexpression $x + 0$ requires x to be of arithmetic type, not $\bot$.

Similarly, consider the proof obligation $(\neg\neg P) = P$. While we may infer that $\neg\neg P$ is Boolean, the typing rule for equality requires that the expressions on both sides must be of equal type. However, the type of P inferred during the first step is just $\bot$. If we allowed the algorithm to assign Bool as the type of P the above proof obligation could be proved without any hypotheses – but its instance $(\neg\neg 42) = 42$ should not be provable in TLA$^+$. The soundness of the type inference algorithm is asserted by the following proposition.

Proposition 1. *Assume given a* TLA$^+$ *proof obligation o for which type inference succeeds. Then for any expression e occurring in the obligation, to which the algorithm assigns a non-safe type τ, we have $\Gamma \vdash e \in [\![\tau]\!]_{\mathrm{TLA}^+}$ where Γ denotes the set of typing hypotheses of o and $[\![\tau]\!]_{\mathrm{TLA}^+}$ denotes the* TLA$^+$ *expression representing the set of values of type τ.*

Proof (idea). The first step of the algorithm assigns only safe types, so there is nothing to prove. Note that semantically, safe types correspond to the universe

[4] For variables, facts of this kind usually come from the type invariant. Our backend requires similar type-correctness lemmas for operators.

of all TLA$^+$ values, so that step cannot introduce any unsoundness. However, if it fails, the proof obligation cannot be represented in the multi-sorted logic of SMT solvers. The assertion for the second and third steps is proved by induction on the expression e, using the rules of Figs. 2 and 3. QED

4 From TLA$^+$ to SMT

Once a type assignment is determined for the symbols in a TLA$^+$ proof obligation, it can be translated to the input languages of SMT solvers. This is done in two steps. In a first phase, 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, namely, the logical and arithmetic operators and the IF/THEN/ELSE construct. These are called *basic* expressions in the language ξ_b. The translation to our target languages – SMT-LIB, Yices and Z3 – is then just a syntactic rewriting.

The operator $[\![\cdot]\!]_B : \xi \to \xi_b$ transforms TLA$^+$ expressions to basic expressions, using the type information gathered previously. During this transformation, we temporarily introduce λ-terms to represent non-basic expressions such as set or function operators. We will prove that all λ-terms introduced during the translation of a well-typed TLA$^+$ expression can be β-reduced to an expression in ξ_b, which no longer contains λ-expressions.

Sets are encoded by their characteristic predicate, allowing for the direct translation of the set membership relation. Set of sets are not considered for this translation, in order to stay within the realm of first-order logic. Any hypotheses of a proof obligation that fall outside this class are discarded. For example, hypotheses of the form $S \in T$ where T is of type $\mathbf{P}\mathbf{P}\,\tau$ are useful during type inference in order to determine the type of S but are then dropped during the translation.

A similar translation for sets in Event-B was given by Déharbe [7], who also considers alternative representations of sets, such as via arrays or using a finite axiomatization of ZF set theory. Hence, if S is an expression of type $\mathbf{P}\,\tau$ then $[\![S]\!]_B$ is a λ-abstraction, and $[\![e \in S]\!]_B \equiv [\![S]\!]_B([\![e]\!]_B)$. Elementary sets are represented as uninterpreted functions. The following rules indicate the translation of more complex set expressions; we simplify the presentation of the rules by omitting the type annotations.

$$[\![S \cup T]\!]_B \equiv \lambda x.\ [\![x \in S \vee x \in T]\!]_B \tag{4.1}$$

$$[\![S \subseteq T]\!]_B \equiv [\![\forall x : x \in S \Rightarrow x \in T]\!]_B \tag{4.2}$$

$$[\![\{\}]\!]_B \equiv \lambda x.\ \text{FALSE} \tag{4.3}$$

$$[\![\{e_1, \ldots, e_n\}]\!]_B \equiv \lambda x.\ [\![x = e_1 \vee \ldots \vee x = e_n]\!]_B \tag{4.4}$$

$$[\![\mathsf{Nat}]\!]_B \equiv \lambda x.\ x \geq 0 \tag{4.5}$$

$$[\![\mathsf{Int}]\!]_B \equiv \lambda x.\ \text{TRUE} \tag{4.6}$$

Translation of equality depends on the type of the two sub-expressions, which must be equal because of typing rule 3.13.

$$\begin{aligned}
[\![e_1 = e_2]\!]_B \equiv\ & \text{case } [\![e_1, \bot]\!]_I \text{ of} \\
\mid \mathbf{P}\,\_ :\ & [\![\forall x : x \in e_1 \Leftrightarrow x \in e_2]\!]_B \\
\mid \_ \to \_ :\ & [\![\textsc{domain}\ e_1 = \textsc{domain}\ e_2 \\
& \quad \wedge \forall x : x \in \textsc{domain}\ e_1 \Rightarrow e_1[x] = e_2[x]]\!]_B \\
\mid \_ :\ & [\![e_1]\!]_B = [\![e_2]\!]_B
\end{aligned} \tag{4.7}$$

Similarly to set membership, function application reduces to λ-aplication (rule 4.8). A function $[x \in S \mapsto e]$ is translated to $\lambda y.\ [\![e(x \leftarrow y)]\!]_B$ (rule 4.9), where x is replaced by y in the expression e (the domain S is represented separately, as explained later).

$$[\![f[e]]\!]_B \equiv [\![f]\!]_B([\![e]\!]_B) \qquad (\lambda\text{-application}) \tag{4.8}$$

$$[\![[x \in S \mapsto e]]\!]_B \equiv \lambda y.\ [\![e(x \leftarrow y)]\!]_B \tag{4.9}$$

$$[\![[S \to T]]\!]_B \equiv \lambda f.\ [\![S = \textsc{domain}\ f \wedge \forall x \in S : f[x] \in T]\!]_B \tag{4.10}$$

$$[\![\textsc{domain}\ f]\!]_B \equiv [\![dom(f)]\!]_B \qquad (\text{when } f \text{ is a symbol})$$

$$[\![\textsc{domain}\ [x \in S \mapsto e]]\!]_B \equiv [\![S]\!]_B \tag{4.11}$$

The translation of function or operator symbols is guided by their types. In case of atomic type, they are simply represented by symbols of appropriate type declared in the SMT output. An n-ary operator or function that returns a set of individuals is represented as an $(n+1)$-ary characteristic predicate. For example, a function symbol $f : \mathsf{Int} \to \mathbf{P}\,\mathsf{Int}$ will be encoded by a binary predicate f over integers.

Because SMT functions have no notion of function domain other than their argument type(s), we associate with each function or operator symbol f a set $\textsc{domain}\ f$. We maintain a mapping $dom : Id \mapsto \xi$ that associates symbols with their domains. Domains of operators are extracted from the corresponding typing hypotheses. For every function or operator application that occurs in the proof obligation, we check that the argument values are in the domain: otherwise the value of the application would be unspecified. To this end, we define an auxiliary operator $[\![\cdot]\!]_F : \xi \to \xi$ that computes corresponding proof obligations. It maintains the structure of the original proof obligation, preserving the quantified variables and conditionals, and collects all function or operator applications that occur in the formula. In particular, we define the following rules.

$$\begin{aligned}
[\![f[e]]\!]_F &\equiv [\![f]\!]_F \wedge [\![e]\!]_F \wedge e \in \textsc{domain}\ f \\
[\![f(e)]\!]_F &\equiv [\![f]\!]_F \wedge [\![e]\!]_F \wedge e \in dom(f) \\
[\![\forall x : e]\!]_F &\equiv \forall x : [\![e]\!]_F \qquad [\![e_1 \wedge e_2]\!]_F \equiv [\![e_1]\!]_F \wedge [\![e_2]\!]_F \quad \text{etc.} \\
[\![[x \in S \mapsto e]]\!]_F &\equiv \forall x : [\![x \in S \Rightarrow e]\!]_F
\end{aligned}$$

For compound expressions other than logical formulas, the operator $[\![\cdot]\!]_F$ recurses on all subexpressions. For example, $[\![[e_1 \subseteq e_2]]\!]_F \equiv [\![e_1]\!]_F \wedge [\![e_2]\!]_F$. For atomic expressions x, we define $[\![x]\!]_F \equiv \textsc{true}$.

It can be shown that given a well-typed TLA$^+$ expression e from the fragment ξ, all λ-terms that occur in its translation $[\![e]\!]_B$ can be β-reduced, as stated by the following proposition.

Proposition 2. *Given a well-typed* TLA$^+$ *expression from the fragment ξ that contains only sets of individuals and functions whose arguments are atomic types, then the β-normal form of $[\![e]\!]_B$ does not contain λ-terms.*

Proof (idea). The translation of expressions $e \in S$ and $f[e]$ introduces function applications that, due to the assumption on the types of TLA$^+$ expressions that appear in the input, remove any λ's introduced during the translation. The only atomic formulas that directly involve set or function types are $S \subseteq T$, $S = T$, and $f = g$, for sets S and T and functions f and g, and in these cases the translation introduces explicit quantifiers that provide the required function arguments for β-reduction. QED

After transforming a proof obligation o in ξ to a basic expression, $[\![o]\!]_B$ is ready to be translated to the input format of SMT solvers. Purely arithmetic and first-order expressions are translated to the corresponding built-in operators of the target languages. For example, the basic expression $e_1 + e_2$ (where e_1 and e_2 must be of arithmetic type because of type checking) is translated to SMT-LIB as $(+\ e_1\ e_2)$ and $\forall x : e$ as $(\mathsf{forall}\ ((\mathsf{x}\ [\![\tau_x]\!]_S))\ e)$, where x is a fresh identifier, τ_x is the type of x as determined by type inference, and where $[\![\cdot]\!]_S$ translates a type to an SMT sort.

The SMT-LIB, Yices, and Z3 backends mainly differ in the encoding of tuples and records. SMT-LIB currently does not have a pre-defined theory for these types, whereas Yices supports them natively, and the Z3 extension of SMT-LIB provides algebraic data types. These kinds of expressions are therefore translated differently for each particular solver format. Currently, only constituents of tuples and records of atomic types are allowed.

In the Yices format, the encoding of records and tuples is almost verbatim. For example, the TLA$^+$ record $[h_1 \mapsto e_1, h_2 \mapsto e_2]$ is translated to $(\mathsf{mk\text{-}record}\ h_1{::}e_1\ h_2{::}e_2)$ and the expression $r.h$ corresponds to $(\mathsf{select}\ r\ h)$. The type $\mathsf{Rec}\,\{h_1 \mapsto \tau_1, h_2 \mapsto \tau_2\}$ is represented as the sort $(\mathsf{record}\ h_1 :: \tau_1\ h_2 :: \tau_2)$. The translation of tuples is analogous, with indexes taking the place of record field names.

For every record type $r = \mathsf{Rec}\,\{h_1 \mapsto \tau_1, \ldots, h_n \mapsto \tau_n\}$, the Z3 backend declares the data type

$$(\mathsf{record\text{-}sort}_r\ (\mathsf{mk\text{-}record}_r\ (\mathsf{h}_1\ \tau_1)\ \ldots\ (\mathsf{h}_n\ \tau_n)))$$

that introduces the new sort identifier $\mathsf{record\text{-}sort}_r$, the datatype constructor $\mathsf{mk\text{-}record}_r$, and the selector function h_i with their corresponding types. Record construction and selection are then translated according to the following rules (the operator $[\![\cdot]\!]_T : \xi \to \mathrm{SMT}$ represents the translation function to SMT format.)

$$[\![[h_1 \mapsto e_1, h_2 \mapsto e_2]]\!]_T \equiv \lambda r.[\![r = \mathsf{mk\text{-}record}_r(e_1, e_2)]\!]_T \tag{4.12}$$

$$[\![r.h]\!]_T \equiv [\![h(r)]\!]_T \tag{4.13}$$

In the SMT-LIB backend, records are axiomatized as follows. For each record sort r that occurs in the proof obligation, we declare a new sort $\mathsf{record\text{-}sort}_r$ of arity 0. The record constructor and the selector functions are declared separately as uninterpreted functions with the appropriate sorts. The translation rules are the same as above (4.12 and 4.13). The logical connection between the constituents with their function selectors and the constructor are asserted for each new declared datatype by the axioms

$$\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ x_i = \mathsf{h}_i(\mathsf{mk\text{-}record}_r\ x_1\ \ldots\ x_n) \qquad \text{for } 1 \leq i \leq n.$$

5 Experimental results

We have used our new backend with good success on several examples that had previously been proved interactively using TLAPS. In particular, we show the results for two cases in the following table. For each benchmark, we indicate the size (number of lines) of the interactive proof, the time (in seconds) required to verify that proof on a standard laptop, as well as the corresponding figures when parts of the proof are performed using the SMT backend, in its three flavors.

	Original		SMT-LIB/CVC3		Yices		Z3	
	size	time	size	time	size	time	size	time
Bakery	398	24	7	33	76	11	7	5
Memoir	2381	53	208	7	208	5	208	7

The first example concerns the invariant proof for (an atomic version of) the well-known N-process Bakery algorithm [9], which mainly uses set theory, functions and arithmetic over the natural numbers. It could be reduced from almost 400 lines of interactive proof to a completely automatic proof. The resulting obligation generates SMT formulas containing 105 quantifiers (many of them nested), which could be proved by the CVC3 SMT solver in around 33 seconds and by Z3 in 5 seconds. On the other hand, Yices could not handle the entire proof obligation at once, and it was necessary to split the theorem into separate cases per subaction; it then takes about 11 seconds to prove the resulting obligations.

More interestingly, the backend could handle significant parts of the type and safety invariant proofs of the Memoir security architecture [12], a generic framework for executing modules of code in a protected environment. The proofs were almost fully automated, except for three sub-proofs that required manual Skolemization of second-order quantifiers. In terms of lines of proof, they were reduced to around 10% of the original size. In particular, the original 2381 lines of proof for the complete type invariant theorems were reduced to 208 lines. Our three solvers took between 5 and 7 seconds to prove them.

These encouraging results show that significant automation can be gained by using SMT solver for the verification of standard TLA$^+$ models, without adapting these models to the SMT backend. There are, however, certain proof obligations that cannot be translated and on which the backend fails. Such examples typically involve the use of advanced set-theoretic constructs, or even just sets of sets, which cannot be encoded in first-order logic using characteristic predicates. For example, our backend cannot prove

$$\forall S \in T : S \neq \{\} \wedge (\forall x \in S : P(x)) \Rightarrow \exists x \in S : P(x).$$

In simple cases such as this one, it suffices to Skolemize the outermost quantifier: the backend will then discard the irrelevant hypothesis $S \in T$ that cannot be translated.

Another source of failures is the use of TLA$^+$ operators that accept arguments of different SMT sorts and that cannot be type checked according to our typing discipline. Fortunately, such cases appear rarely in actual specifications.

6 Conclusions

We defined a translation of certain TLA$^+$ proof obligations to the input language of state-of-the-art SMT solvers. The translation relies on imposing a typing discipline on the untyped specification language TLA$^+$, and is based on a corresponding type inference algorithm. This discipline restricts the class of TLA$^+$ expressions that can be translated. Nevertheless, a significant fragment of the source language can be handled. In particular, we support first-order logic, elementary set theory, functions, integer and real arithmetic, records and tuples. Sets and functions are represented as lambda-abstractions, which works quite efficiently but excludes handling second-order expressions involving, for example, sets of sets. The translation of records and tuples relies on an axiomatization for SMT-LIB, and on appropriate native constructs of Yices and Z3. Our type inference and translation algorithms provide the formal basis for the implementation of an SMT-based backend prover for TLAPS. Universal set quantifiers that occur at the outermost level can easily be removed by the user of TLAPS, by introducing Skolem constants. An automatic pre-processing of such terms would further improve the backend.

In future work, we intend to study the question of interpreting proofs that many SMT solvers can produce for reconstructing them (as well as the type assignment) in the trusted object logic of Isabelle/TLA$^+$. This would allow us to check the results of these solvers, as well as of the translation from TLA$^+$ into SMT input, and would raise the confidence in the SMT backend, just as currently TLAPS can check proofs produced by Zenon.

We also envisage extending our translation to support λ-abstractions (for functions as basic terms) using, for example, combinators, and to support some more advanced set-theoretic constructions, perhaps using a different representation of sets.

Acknowledgements. Denis Cousineau, Damien Doligez, and Leslie Lamport provided constructive feedback on the design and implementation of the SMT backend. Helpful comments from the anonymous referees are gratefully acknowledged.

References

1. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Satisfiability Modulo Theories (SMT 2010)*, Edinburgh, UK, 2010. `http://www.SMT-LIB.org`.
2. C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *19th Intl. Conf. Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, pages 298–302, Berlin, Germany, 2007. Springer.
3. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In N. Bjørner and V. Sofronie-Stokkermans, editors, *23rd Intl. Conf. Automated Deduction*, volume 6803 of *LNCS*, pages 116–130, Wroclaw, Poland, 2011. Springer.
4. R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *14th Intl. Conf. Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2007)*, volume 4790 of *LNCS*, pages 151–165, Yerevan, Armenia, 2007. Springer.
5. K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA$^+$ proof system. In J. Giesl and R. Hähnle, editors, *5th Intl. Joint Conf. Automated Reasoning (IJCAR 2010)*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148, Edinburgh, UK, 2010. Springer.
6. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *14th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340, Budapest, Hungary, 2008. Springer.
7. D. Déharbe. Automatic verification for a class of proof obligations with SMT-solvers. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z (ASM 2010)*, volume 5977 of *LNCS*, pages 217–230, Orford, Canada, 2010. Springer.
8. B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.
9. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–454, 1974.
10. L. Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Boston, Mass., 2002.
11. L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Trans. Prog. Lang. Syst.*, 21(3):502–526, 1999.
12. B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *IEEE Symp. Security and Privacy*, Berkeley, California, U.S.A., 2011. IEEE Computer Society. Formal Specifications and Correctness Proofs: Tech. Report, Microsoft Research, Feb. 2011.
13. M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle framework. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *21st Intl. Conf. Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 of *LNCS*, pages 33–38, Montreal, Canada, 2008. Springer.