Automatic Verification of TLA$^+$ proof obligations with SMT solvers

Stephan Merz and Hernán Vanzetto

LPAR-18, Mérida, Venezuela

March 12th, 2012

The TLA$^+$ language

- Specification and verification language
  for (concurrent and distributed) systems and algorithms
  (Designed by Leslie Lamport, 1999)
- Based on
  - ZF set theory
  - Temporal Logic of Actions (TLA)
    (about 95% of the specs is not-temporal)
- Includes also FO logic, functions, arithmetic, records, tuples, ...
- ...and a proof language:
  - hierarchical proof structure (tree)
  - top-down development: refine assertions until they are "obvious"
  - leaf: invoke proof method, citing necessary assumptions and facts

TLA$^+$ (toy) proof example

MODULE *AbsoluteValue*

VARIABLES n, abs

THEOREM ASSUME $n \in Int$,

$abs = [x \in Int \mapsto \text{IF } x \geq 0 \text{ THEN } x \text{ ELSE } -x]$

PROVE $abs[n] \in Nat$

TLA$^+$ (toy) proof example

MODULE *AbsoluteValue*

VARIABLES n, abs

THEOREM ASSUME $n \in Int$,
$abs = [x \in Int \mapsto \text{IF } x \geq 0 \text{ THEN } x \text{ ELSE } -x]$
PROVE $abs[n] \in Nat$

$\langle 1 \rangle 1.$ CASE $n \geq 0$

$\langle 1 \rangle 2.$ CASE $n < 0$

$\langle 1 \rangle 3.$ $n \in Int \Rightarrow (n \geq 0 \lor n < 0)$
BY SimpleArithmetic

$\langle 1 \rangle 4.$ QED
BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$

TLA$^+$ (toy) proof example

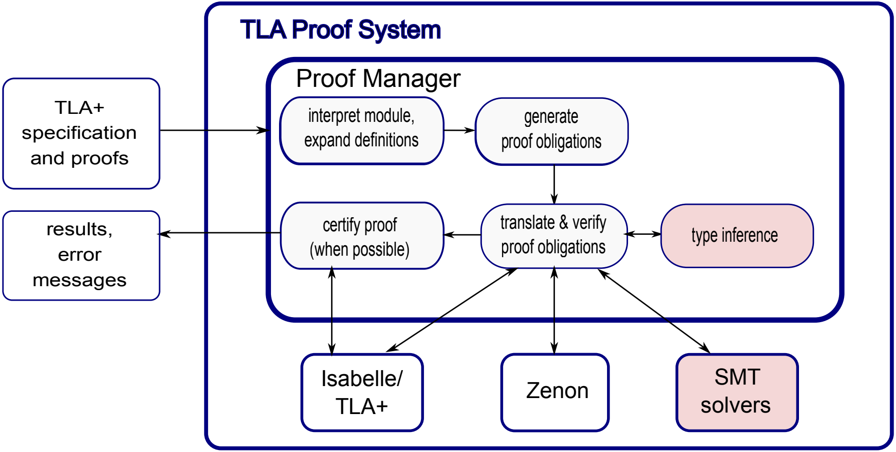
MODULE *AbsoluteValue*

VARIABLES n, abs

THEOREM ASSUME $n \in Int$,
$abs = [x \in Int \mapsto \text{IF } x \geq 0 \text{ THEN } x \text{ ELSE } -x]$
PROVE $abs[n] \in Nat$

$\langle 1 \rangle 1.$ CASE $n \geq 0$
  $\langle 2 \rangle 1.$ $n \leq 0 \Rightarrow n \in Nat$
    BY SimpleArithmetic
  $\langle 2 \rangle 2.$ QED
    BY $\langle 2 \rangle 1$
$\langle 1 \rangle 2.$ CASE $n < 0$
$\langle 1 \rangle 3.$ $n \in Int \Rightarrow (n \geq 0 \vee n < 0)$
  BY SimpleArithmetic
$\langle 1 \rangle 4.$ QED
  BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$

The TLA$^+$ Proof System



- Isabelle/TLA$^+$ = faithful encoding of TLA$^+$ over Isabelle/Pure.
- Zenon = tableau prover for FOL and Set Theory. Outputs Isar.

Goal

MODULE *AbsoluteValue*

VARIABLES n, abs

THEOREM ASSUME $n \in Int$,
$abs = [x \in Int \mapsto \text{IF } x \geq 0 \text{ THEN } x \text{ ELSE } -x]$
PROVE $abs[n] \in Nat$

BY **SMT**

Goal

MODULE *AbsoluteValue*

VARIABLES n, abs

THEOREM ASSUME $n \in Int$,
$abs = [x \in Int \mapsto \text{IF } x \geq 0 \text{ THEN } x \text{ ELSE } -x]$
PROVE $abs[n] \in Nat$

BY **SMT**

TLA$^+$ PO

$\rightsquigarrow$ $\langle 1 \rangle$ Type inference

$\rightsquigarrow$ $\langle 2 \rangle$ Translation to SMT

$\rightsquigarrow$ (Proof reconstruction in Isabelle/TLA$^+$)

Dealing with an untyped language

TLA$^+$ is an **untyped** language[1].

Why do we need to know the TLA$^+$ symbols' types?

1. the SMT input languages are sorted
2. the translation of some operators depends on the type of their arguments, e.g. *equality*:

$$x : Int \vdash \quad x = 3 \quad \leadsto \quad x = 3$$

$$S, T : \mathrm{P}\, Int \vdash \quad S = T \quad \leadsto \quad \forall x \in Int : \ x \in S \Leftrightarrow x \in T$$

[1] *Should your specification language be typed?* (L. Lamport & L. Paulson, 1999)

Dealing with an untyped language

TLA$^+$ is an **untyped** language[1].

Why do we need to know the TLA$^+$ symbols' types?

1. the SMT input languages are sorted
2. the translation of some operators depends on the type of their arguments, e.g. *equality*:

$$
\begin{array}{llcr}
x : Int \vdash & x = 3 & \leadsto & x = 3 \\
S, T : \mathrm{P}\, Int \vdash & S = T & \leadsto & \forall x \in Int : \ x \in S \Leftrightarrow x \in T
\end{array}
$$

Example:

- THEOREM $x \in Nat \Rightarrow x + 0 = x$ ✓
- THEOREM $x + 0 = x$ ✗

[1] *Should your specification language be typed?* (L. Lamport & L. Paulson, 1999)

Typing discipline for TLA$^+$

Ad-hoc type system

$$\tau ::= \bot \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{Nat} \mid \mathit{Int} \mid \qquad \text{(atomic types)}$$

$$\mathrm{P}\ \tau \mid \tau \rightarrow \tau \mid \mathit{Rec}\ \{\mathit{field}_i, \tau_i\} \mid \mathit{Tup}\ [\tau_i] \qquad \text{(complex types)}$$

- Partial order $\leq$ on types is defined.

  For example: $\bot \leq \tau$
  $\mathrm{P}\ \tau_1 \leq \mathrm{P}\ \tau_2 \qquad \text{if } \tau_1 \leq \tau_2$
  $\mathit{Nat} \leq \mathit{Int}$

Type inference algorithm

- Initially, all symbols have type $\perp$
- Type operator: $[\![exp, \varepsilon]\!]_I : \tau$ (ε is the *least* type of *exp*)

  Typing variable: $type : symbol \mapsto \tau$
- Types are updated while recursing over the structure of the PO
- $[\![e]\!]_I$ fails when:
  1. A symbol does not have an assigned type ($x + 0 = x$)
  2. Cannot equate expressions that need to be of the same type, i.e. $=$, $+$, $<$, $\subseteq$, IF-THEN-ELSE

$$
\begin{aligned}
[\![e_1 = e_2, \varepsilon]\!]_I &\equiv \mathcal{S}([e_1, e_2], \varepsilon) \; ; Bool \\
[\![e_1 < e_2, \varepsilon]\!]_I &\equiv \mathcal{S}([e_1, e_2], Nat) \; ; Bool
\end{aligned}
$$

Type inference algorithm

Inference rules according to TLA$^+$ semantics for operators

- Logical: *always return Boolean values*.

$$\llbracket e_1 \land e_2, \varepsilon \rrbracket_I \equiv \text{if } \varepsilon \leq \mathit{Bool} \\ \text{then } \llbracket e_1, \mathit{Bool} \rrbracket_I ; \llbracket e_2, \mathit{Bool} \rrbracket_I ; \mathit{Bool} \text{ else } \mathit{fail}$$

- Arithmetic: *arguments should be in an arithmetic domain*.

$$\llbracket e_1 + e_2, \varepsilon \rrbracket_I \equiv \text{let } \gamma = \mathcal{S}([e_1, e_2], \varepsilon) \text{ in} \\ \text{if } \gamma \in \{\mathit{Nat}, \mathit{Int}, \mathit{Real}\} \text{ then } \gamma \text{ else } \mathit{fail}$$

- Sets: *always return a set (that depends on the arguments' type)*

$$\llbracket S \cup T, \mathsf{P}\,\varepsilon \rrbracket_I \equiv \text{let } \mathsf{P}\,\tau_1 = \llbracket S, \mathsf{P}\,\varepsilon \rrbracket_I, \quad \mathsf{P}\,\tau_2 = \llbracket T, \mathsf{P}\,\varepsilon \rrbracket_I \text{ in} \\ \text{if } \tau_1 = \tau_2 \text{ then } \mathsf{P}\,\tau_1 \text{ else } \mathsf{P}\,\bot$$

Type inference algorithm

- If x is a symbol, then

$$(\neg\neg x) = x \qquad \text{✗}$$

cannot be proved!

In fact, if $x \equiv 42$ then $(\neg\neg 42) = 42$.

Type inference algorithm

- If x is a symbol, then

$$(\neg\neg x) = x \qquad \text{✗}$$

  cannot be proved!

  In fact, if $x \equiv 42$ then $(\neg\neg 42) = 42$.

- Rule: Infer types only from available facts of the forms
  - $x \approx exp$
  - $\forall y \in S : x(y) \approx exp$

  where $\approx\ \in \{=, \in, \subseteq\}$, x is a symbol and exp any expression.

Type inference algorithm

- If x is a symbol, then

$$(\neg\neg x) = x \qquad ✗$$

cannot be proved!

In fact, if $x \equiv 42$ then $(\neg\neg 42) = 42$.

- Rule: Infer types only from available facts of the forms
  - $x \approx exp$
  - $\forall y \in S : x(y) \approx exp$

  where $\approx \ \in \{=, \ \in \ , \subseteq\}$, x is a symbol and *exp* any expression.

- These facts are usually provided by **type invariants** in the specification.
- Drawback: now "$S = \{\} \Rightarrow S \subseteq Nat$" cannot be proved.

The target language: SMTLIB

SMTLIB grammar:

$$
\begin{array}{lll}
(\textit{sorts}) & \sigma ::= & s \mid \texttt{(}s\ \sigma^{+}\texttt{)} \\
(\textit{terms}) & t ::= & \mathit{Var} \mid \mathit{Number} \mid \texttt{(}f\ t^{+}\texttt{)} \mid \texttt{(= } t\ t\texttt{)} \mid \texttt{(ite } c\ t\ t\texttt{)} \\
 & & \mid \texttt{(and } t\ t\texttt{)} \mid \texttt{(or } t\ t\texttt{)} \mid \texttt{(not } t\texttt{)} \\
 & & \mid \texttt{(}[\texttt{forall} | \texttt{exists}]\ \texttt{(}(\texttt{(}x\ \sigma\texttt{)})^{+}\texttt{))}\ t
\end{array}
$$

where s is a sort identifier, and f is a function symbol.
(Yices native input format is similar to SMTLIB)

- Each well-formed expression has a **unique sort**.
- We use the AUFLIRA logic.
  - quantified formulas over the theory of linear integer and real arithmetic (and arrays)

From TLA$^+$ to SMT formats

Translation operator $[\![exp]\!]_T : SMT^*$.

- $SMT^* =$ SMT input format $+$ λ-terms
- Type discipline ensures that all λ-abs are β-reduced

From TLA$^+$ to SMT formats

Translation operator $[\![exp]\!]_T : SMT^*$.

- SMT^* = SMT input format + λ-terms
- Type discipline ensures that all λ-abs are β-reduced

Translation rules:

- Arithmetic

$$
\begin{array}{lcl}
[\![e_1 + e_2]\!]_T & \equiv & \texttt{(+ } [\![e_1]\!]_T \ [\![e_2]\!]_T\texttt{)} \\
[\![e_1 < e_2]\!]_T & \equiv & \texttt{(< } [\![e_1]\!]_T \ [\![e_2]\!]_T\texttt{)}
\end{array}
$$

- Logic

$$
\begin{array}{lcl}
[\![e_1 \wedge e_2]\!]_T & \equiv & \texttt{(and } [\![e_1]\!]_T \ [\![e_2]\!]_T\texttt{)} \\
[\![\forall x : e]\!]_T & \equiv & type \oplus (x \mapsto \bot) \vdash [\![e, Bool]\!]_I \ ; \\
 & & \quad \texttt{(forall ((}[\![x]\!]_T \ [\![type(x)]\!]_S\texttt{))} \ [\![e]\!]_T\texttt{)}
\end{array}
$$

From TLA$^+$ to SMT formats

Sets and functions are encoded as uninterpreted functions

- $[\![S]\!]_T$ represents the *characteristic predicate* of set S
- Only simple sets are allowed

$$
\begin{array}{lcll}
[\![x]\!]_T & \equiv & \text{case } \mathit{type}(x) \text{ of} & \\
 & & \mid (\_ \rightarrow \mathrm{P}\,\_) : & \lambda y, z.\,(\mathtt{x}\ \ y\ z) \\
 & & \mid (\_ \rightarrow \_) \mid (\mathrm{P}\,\_) : & \lambda y.\,(\mathtt{x}\ \ y) \\
 & & \mid \_ : & \mathtt{x} \\
[\![e \in S]\!]_T & \equiv & [\![S]\!]_T\ [\![e]\!]_T & (\lambda\text{-application}) \\
[\![f[e]]\!]_T & \equiv & [\![f]\!]_T\ [\![e]\!]_T & \\
[\![[x \in S \mapsto e(x)]]\!]_T & \equiv & \lambda y.\ [\![e(x \leftarrow y)]\!]_T &
\end{array}
$$

From TLA$^+$ to SMT formats

Problem: function domains are not directly translated.

- $$\begin{aligned} [\![\phi]\!]_T &\leadsto [\![f = [x \in 1..5 \mapsto x+1] \Rightarrow f[0] = 0]\!]_T \\ &\leadsto \quad [\![\forall x : f[x] = x + 1 \Rightarrow f[0] = 0]\!]_T \qquad ✗ \end{aligned}$$

From TLA$^+$ to SMT formats

Problem: function domains are not directly translated.

- $[\![\phi]\!]_T \leadsto [\![f = [x \in 1..5 \mapsto x+1] \Rightarrow f[0] = 0]\!]_T$
  $\leadsto \quad [\![\forall x : f[x] = x+1 \Rightarrow f[0] = 0]\!]_T$ ✗
- Instead, we want to prove also that the argument is in the domain: $\leadsto \quad [\![\forall x : f[x] = x+1 \Rightarrow f[0] = 0 \land 0 \in 1..5]\!]_T$

From TLA$^+$ to SMT formats

<u>Problem</u>: function domains are not directly translated.

- $[\![\phi]\!]_T \leadsto [\![f = [x \in 1..5 \mapsto x + 1] \Rightarrow f[0] = 0]\!]_T$
  $\leadsto \quad [\![\forall x : f[x] = x + 1 \Rightarrow f[0] = 0]\!]_T$ ✗
- Instead, we want to prove also that the argument is in the domain: $\leadsto \quad [\![\forall x : f[x] = x + 1 \Rightarrow f[0] = 0 \wedge 0 \in 1..5]\!]_T$
- $[\![\cdot]\!]_F$ computes function arguments belonging to their domain:

$$\begin{aligned} [\![f[e]]\!]_F &\equiv [\![f]\!]_F \wedge [\![e]\!]_F \wedge e \in \text{DOMAIN } f \\ [\![\forall\, x \in S : e]\!]_F &\equiv \forall x \in S : [\![e]\!]_F \end{aligned}$$

  The rest of expressions are computed as TRUE or conjunctions.
- $[\![\phi]\!]_F \quad \leadsto \quad \text{TRUE} \wedge 0 \in \text{DOMAIN } f \wedge \text{TRUE} \quad \leadsto \quad 0 \in 1..5$

Translation example

MODULE *AbsoluteValue*

VARIABLES n, abs

THEOREM ASSUME $n \in Int,$
$abs = [x \in Int \mapsto \text{IF } x \geq 0 \text{ THEN } x \text{ ELSE } -x]$
PROVE $abs[n] \in Nat$

BY SMT

Translation example

MODULE *AbsoluteValue*

VARIABLES n, abs

THEOREM ASSUME $n \in Int$,
$abs = [x \in Int \mapsto \text{IF } x \geq 0 \text{ THEN } x \text{ ELSE } -x]$
PROVE $abs[n] \in Nat$
BY SMT

```
(declare-fun n () Int)
(declare-fun abs (Int) Int)

(assert (forall ((?x Int))
            (= (abs ?x) (ite (>= ?x 0) ?x (- ?x)))))
(assert (not (and (>= (abs n) 0))))
```

Experimental results

- Bakery algorithm (N-process mutual exclusion)
  - 105 (nested) quantifiers
  - from 320 to 1 line of proof
  - Yices: split by cases
- Memoir system (security architecture/generic framework for executing modules of code in a protected environment)
  - only type invariant and main part of safety invariant
  - manual Skolemization in 3 out of 11 subcases

	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

Conclusions and Future work

- Type system and inference algorithm for (untyped) TLA$^+$.
- Handles a useful fragment of TLA$^+$:
  - FOL, elementary sets, functions, arithmetic, records, tuples
- Translates to CVC3 (SMT-LIB), Yices and Z3.
- Interactive proof size could be reduced significantly.
- This method replaced Cooper's algorithm.

Future work:

- Try untyped encoding (ie., types handled by the solver).
- Interpret SMT solvers output and certify it with Isabelle/TLA$^+$.
- Automatic Skolemization of second-order quantifiers.