The SMT-LIB 2 Standard: Overview and Proposed New Theories

Philipp Rümmer

Oxford University Computing Laboratory

philr@comlab.ox.ac.uk

Third Workshop on
Formal and Automated Theorem Proving and Applications
Belgrade, Serbia
29 January 2010

Outline

Overview of SMT-LIB 2, comparison with version 1

- Joint work by somebody else

Set-theoretic datatypes for the SMT-LIB

- Finite sets, lists, maps, relations
- Joint work with Daniel Kroening, Georg Weissenbacher

Floating-point arithmetic for the SMT-LIB

- Joint work with Thomas Wahl

The SMT-LIB Standard

SMT → **S**atisfiability **M**odulo **T**heories

SMT-LIB is . . .

- a standardised input format for SMT solvers (since 2003)
- a standardised format for exchanging SMT problems
- a library of more than 90 000 SMT benchmarks
- the basis for the annual SMT competition
  (this year: on FLoC)

Relevant for verification + program analysis tool:

- Krakatoa, Caduceus, ESC/Java2, Spec#, VCC, Havoc, Pex, CBMC, F7, . . .

Example in SMT-LIB Format (Version 1)

```
(benchmark Ensures_Q_noinfer_2
:source { Boogie/Spec# benchmarks. }
:logic AUFLIA
[...]
:extrapreds (( InRange Int Int ))
:extrafuns (( this  Int ))
:extrafuns (( intAtLeast Int Int Int ))
[...]
:assumption
 (forall (?t Int) (?u Int) (?v Int)
   (implies (and (subtypes ?t ?u) (subtypes ?u ?v)) (subtypes ?t ?v))
   :pat (subtypes ?t ?u) (subtypes ?u ?v))
[...]
:formula
  (not (implies (implies (implies (implies
  (and
   (forall (?o Int) (?F Int)
    (implies (and (= ?o this) (= ?F X)) (= (select2 H ?o ?F) 5)))
   (implies
    (forall (?o Int) (?F Int)
     (implies (and (= ?o this) (= ?F X)) (= (select2 H ?o ?F) 5)))
    (implies true true)))
   (= ReallyLastGeneratedExit_correct Smt.true))
  (= ReallyLastGeneratedExit_correct Smt.true))
 (= start_correct Smt.true))
(= start_correct Smt.true))))
```

Example in SMT-LIB Format (Version 1)

```
(benchmark Ensures_Q_noinfer_2
:source
:logic          Preamble + problem logic/category
[...]
:extrapreds
:extrafuns      Problem signature: sorts, functions, predicates
:extrafuns
[...]
:assumption

                Premises + axioms

[...]
:formula

                Verification condition



                              )
```

Versions of SMT-LIB

Latest "stable" version 1.2

- Introduced 2006
- Supported by virtually all SMT solvers
- Theories: arrays, bit-vectors, integers, reals

Upcoming version 2.0

- Proposed July 2009[1]
- Improvements + simplifications over 1.2 . . . *next slides*
- More flexible w.r.t. combination of theories
- But: semantics similar to 1.2

[1] *Working group:* Clark Barrett, Sylvain Conchon, Bruno Dutertre, Jim Grundy, Leonardo de Moura, Albert Oliveras, Aaron Stump, Cesare Tinelli

The Brave New World

(of SMT-LIB 2)

1. Sort Constructors

SMT-LIB 1

Only nullary sort constructors:

```
:sorts (Int)
[...]
:extrasorts (U T)
```

Types are atomic:

```
:extrafuns
    ((f T T))
```

SMT-LIB 2

Sort constructors of any arity:

```
:sorts ((Array 2))
[...]
:extrasorts ((List 1)
             U T)
```

Types can be compound:

```
:extrafuns
    ((f T (Array U T)))
```

SMT-LIB 1

Theories are monomorphic:

```
(theory Int_Arrays
:sorts (Int Array)
:funs
((select Array Int Int)
 (store Array Int Int
        Array)
[...]
))
```

SMT-LIB 2

Parametric polymorphism in theories:

```
(theory Array
:sorts ((Array 2))
:funs
((par (X Y)
   (select
     (Array X Y) X Y))
 (par (X Y)
   (store
     (Array X Y) X Y
     (Array X Y)))
[...]
))
```

3. Symbol Overloading

SMT-LIB 1

Unique operator names:

```
:sorts (Int)
:funs ((~ Int Int)
       (- Int Int Int)
       (+ Int Int Int))
[...]
:sorts (BitVec)
:funs
((bvneg BitVec BitVec))
```

SMT-LIB 2

Symbol overloading:

```
:sorts (Int)
:funs ((- Int Int)
       (- Int Int Int)
       (+ Int Int Int))
[...]
:sorts (BitVec)
:funs ((- BitVec BitVec))
```

4. No Formula/Term Distinction

SMT-LIB 1

Formulae $\neq$ terms,
predicates $\neq$ functions:

```
:extrapreds
  ((divides Int Int))
:extrafuns
  ((succ Int Int))
```

Only terms can be function/predicate arguments

Work-arounds:
reflection, `ite` operator

SMT-LIB 2

`Bool` is simply a sort:

```
:extrafuns
  ((divides Int Int Bool)
   (prime (Array Int Bool)))
```

`and`, `or`, `=`, ...
are just functions

5. Standardised Command Language

Text-based interface to SMT solvers:

```
> (set-logic AUFLIA)
> (declare-fun a () Int)
> (declare-fun b () Int)
> (assert (= (* 8 a) (* 4 b)))
> (push)
> (assert (forall ((x Int))
              (not (= b (* 2 x))))))
> (check-sat)
unsat
> (pop)
[...]
```

- Apparently:
  Interface will replace the old benchmark file format

Proposals for Additional SMT-LIB 2 Theories

Theories of Set-Theoretic Datatypes

We propose to add datatypes inspired by VDM-SL

- Tuples
- Lists
- (Finite) Sets
- (Finite) Partial Maps

Main applications for us:

- Bounded Model Checking for C, C++ (CBMC)
- Model-based test-case generation
  (UML/OCL, Simulink/Stateflow, Lustre)
- Analysis of requirements + architecture specifications
- System development in Event-B, VDM

SMT-LIB 2 Theory Schemas

Tuples	**Sets**	**Lists**	**Maps**
`(Tuple`$_n$ `T`$_1$ `...` `T`$_n$`)`	`(Set T)`	`(List T)`	`(Map S T)`
`tuple`	`emptySet` $\emptyset$	`nil` $[\,]$	`emptyMap` $\emptyset$
$(x_1, \ldots, x_n)$	`insert`	`cons` $x :: L$	`apply` $f(x)$
`project`	$M \cup \{x\}$	`head`	`overwrite`
x_k	`in` $\in$	`tail`	$\mathbin{<\!\!+}$
`product`	`subset` $\subseteq$	`append` $\frown$	`domain`
$M_1 \times \cdots \times M_n$	`union` $\cup$	`length` $\lvert l \rvert$	`range`
	`inter` $\cap$	`nth` l_k	`restrict` $\lhd$
	`setminus` $\setminus$	`inds`	`subtract` $⩤$
	`card` $\lvert M \rvert$	$\{1, \ldots, \lvert l \rvert\}$	
		`elems`	
		$\{l_1, \ldots, l_{\lvert l \rvert}\}$	

Example: Verification Cond. Generated by VDMTools

In VDM-SL notation:

$$\forall l : \mathbb{L}(\mathbb{Z}), i : \mathbb{N}.\ \big(i \in \mathsf{inds}(l) \Rightarrow \forall j \in \mathsf{inds}(l) \setminus \{i\}.\ j \in \mathsf{inds}(l)\big)$$

In SMT-LIB notation:

```
(forall ((l (List Int)) (i Int))
  (implies
     (and (>= i 0) (in i (inds l)))
     (forall (j Int)
       (implies
          (in j (setminus (inds l)
                          (insert i emptySet)))
          (in j (inds l)))))))
```

Status of the Proposal

- Syntax + Semantics of theories is defined
  $\Rightarrow$ In collaboration with Cesare Tinelli
- Parser + type checker + converter to SMT-LIB 1 available (using a rather naive axiomatisation of the datatypes)
- Meaningful sublogics still to be identified
- We have a small initial collection of benchmarks
  $\Rightarrow$ More to be converted from Event-B VCs
  $\Rightarrow$ *Further benchmarks would be welcome*

http://www.cprover.org/SMT-LIB-LSM/

Floating-Point Arithmetic (FPA)

Binary floating-point numbers (IEEE 754-2008)

$$\begin{aligned} \mathbb{F} &= \{(-1)^s \cdot m \cdot 2^e \mid (m, e) \in E, s \in \{0, 1\}\} \\ &= \{\text{NaN}, +\infty, -\infty, 0^-, \ldots\} \end{aligned}$$

where:

s … sign
m … mantissa/significand
e … exponent

- Standard mathematical operations + rounding (defined more or less ambiguously in IEEE 754-2008)
- Important for embedded software, control software, etc.

A Theory of Floating-Point Arithmetic (FPA)

So far: no SMT solvers with FPA support

Correct reasoning about FPA is hard

- Precise encoding: hard for automatic solvers
  (but works for interactive proof assistants)
- Interval arithmetic: sound but imprecise, no models
  (bad for test cases)
- Rational arithmetic: only an approximation
  (unsound in certain settings)

Main applications for us:

- Bounded model checking for Simulink/Stateflow
- Test-case generation

Abstraction for Floating-Point Arithmetic [FMCAD'09]

New reasoning approach:

- Precise SAT encoding combined with mixed over/under-approximation
- Outperforms naive SAT encoding + can generate models
- Prototypical implementation as part of CBMC
- Planned: move implementation to an SMT solver
  $\Rightarrow$ *SMT-LIB interface is needed!*

An SMT-LIB Theory of FPA (work in progress)

Goals

- Model FPA core that is relevant for reasoning + verification
  Not considered:
  Exact error handling, bit-precise encoding, ...
- Precise + concise definition of FPA semantics
- Useable syntax

http://www.cprover.org/SMT-LIB-Float/

Example: FPA Problem in SMT-LIB

```
:extrafuns ((x (ind FP 11 53))
            (y (ind FP 11 53)))
:problem
  (exists ((z (ind FP 11 53)))
          (= (+ roundTowardZero x z) y))
```

- 64-bit floating-point arithmetic (double precision)
  $\Rightarrow$ 11 bit exponent, 53 bit significand
- `ind` notation is used for indexed types
  $\Rightarrow$ `(ind FP 11 53)` means $\mathtt{FP}_{11,53}$
- $+$ is ternary: first argument is rounding mode

Conclusion

- Overview of SMT-LIB 2
- Datatypes of sets, lists, maps, relations
- Floating-point arithmetic

Trade-off when defining theories:

- Generality $\rightarrow$ good for users
- Implementation complexity
- Decidability

$\rightarrow$ good for tool writers

$\Rightarrow$ We hope that we have found a good compromise
$\Rightarrow$ Feedback is welcome!

Thanks for your attention!

Don't forget about . . . Ad