The SMT-LIB Initiative

Cesare Tinelli, The University of Iowa
David Cok, GrammaTech

SAT/SMT Solver Summer School
Boston, MA - June 2011
1. Introduction and overview of SMT-LIB
2. Brief history
3. The SMT-LIB 2 language and library
4. Resources and tools
5. Demos
6. Future directions
What is SMT-LIB

- International initiative
  - Aimed at facilitating R&D in SMT
  - Backed by research groups worldwide

- People involved
  - 3 coordinators
  - 90+ contributors
  - many more users
The SMT-LIB Initiative

- Concrete goals

1. Provide standard rigorous descriptions of background theories used in SMT systems
2. Develop and promote common I/O languages for SMT solvers
3. Collect and make available an extensive benchmarks library
The SMT-LIB Initiative

- Sister Initiatives
  - SMT-COMP, annual solver competition
  - SMT-EXEC, public solver execution service

- Funding
  - NSF, SRC, Intel, MSR, UIowa
Credits

- Founders
  - S. Ranise, C. Tinelli

- Current/past coordinators
  - C. Barrett, S. Ranise, A. Stump, C. Tinelli

- Major contributors
  - D. Cok, C. Conway, M. Deters, L. de Moura, A. Oliveras
Credits

Other contributors*


(*) Apologies for any omissions

SAT/SMT Solver Summer School

Boston, 2011
A Brief History of SMT-LIB
SMT Beginnings (late 90s)

- **Substrate**
  - Early work on decision procedures

- **Catalyst:**
  - Spectacular advances in SAT

- **New ideas:**
  - *eager* encodings of SMT problems into SAT [Bryant, Velev, Strichman, Lahiri, Seisha,..., -'02]
  - *lazy* encodings into SAT + decision procedures [Armando et al.'00, Audemard et al.'02, Ruess & de Moura'02, Barrett et al.'02]
Several SMT solvers

- based on different variants of FOL
- working with different theories
- dealing with different classes of formulas
- having different interfaces and input formats
Many different solvers

Solver's theory often unclear

Arduous to assess the relative merits of techniques or solvers

Difficult even to evaluate a single solver

Each solver good on its own benchmarks
FroCoS’02: a Call for Arms

- Excitement about the promise of SMT
- Frustration about lack of standard benchmarks
- Chair A. Armando calls for the creation of a common benchmark library
- S. Ranise and C. Tinelli agree to lead the initiative
- Several participants promise assistance and contributions
R & T soon realize that a common library would first need to fix a standard:
1. underlying logic,
2. catalog of rigorously defined theories,
3. specification of relevant fragments of these theories,
4. concrete syntax for benchmarks

This becomes the blueprint for SMT-LIB
Three main components:

1. **Theory declarations**, semi-formal specifications of theories of interest (e.g., integers, reals, arrays, bit vectors, …)

2. **Logic declarations**, semi-formal specifications of fragments of (combinations of) theories (e.g., linear real arithmetic, integer difference constraints, …)

3. **Benchmarks**, formulas to be checked for satisfiability (Version 1), or scripts (Version 2)
The SMT-LIB Repository

Three main components:

1. Catalog of theory declarations
2. Catalog of logic declarations
3. Library of benchmarks

External components:

1. Utility tools (parsers, checkers, converters, ...)
2. Additional resources
SMT-LIB Today

- 95,000+ benchmarks in online database
- 20+ logics in online catalog
- SMT-LIB format (V. 1.2) adopted by all major SMT solvers (12+)
- major new version (V. 2.0) of format and library released in 2010
- SMT-COMP’10-11 run with Version 2.0
The SMT-LIB 2 Language
The SMT-LIB 2 Language

- Textual, command-based I/O format for SMT solvers
- Intended mostly for machine processing
  - Easy to generate automatically
  - Easy to parse
  - Human-readable, but with minimal syntactic sugar
- Specifically designed for on-line integration of SMT solvers into other tools
The SMT-LIB 2 Language

- Typical usage:
  - Asserting a series of logical statements, in the context of a given logic
  - Checking their satisfiability in the logic
  - Exploring resulting models (if sat) or proofs (if unsat)
- Logical statements expressed in a sorted (typed) first-order predicate language
Language Highlights

- Concrete syntax
  - Sublanguage of Common Lisp \textit{S-expressions}
  - Few syntactic categories

- Versatile underlying logic
  - Many-sorted FOL with (pseudo-)parametric sorts
  - Function symbol overloading

- Command language
  - Allows sophisticated interaction with solvers
  - Stack-based, assert-and-query execution model
  - Benchmarks are command \textit{scripts}
Concrete Syntax

- Proper subset of Common Lisp S-expressions

\[
\langle\text{literal}\rangle ::= \langle\text{numeral}\rangle | \langle\text{decimal}\rangle | \langle\text{hexadecimal}\rangle | \langle\text{binary}\rangle | \langle\text{string}\rangle
\]

\[
\langle\text{s_expr}\rangle ::= \langle\text{literal}\rangle | \langle\text{symbol}\rangle | (\langle\text{s_expr}\rangle^*)
\]

- Some reserved words

exists  forall  let  par  as  _  !

NUMERAL  DECIMAL  STRING
Concrete Syntax

- **Literals**
  - Numerals: 0, 12, 832
  - Decimals: 0.1, 123.0, 6.01
  - Hexadecimal: #x0, #xFF0A, #xdad
  - Binary: #b0, #b11, #b010101
  - Strings: ", "abef", "\"Hi\"

- **Symbols**
  - `true`, `a`, `<`, `a<>`, `b._?`, `$abc`
  - `:pat`, `|single symbol|`, `|a {} %$2|`
Concrete Syntax

- S-expressions

\[
\text{(assert}
\begin{align*}
&\text{(forall } ( (l1 \ (\text{List} \ \text{Int})) \ (l2 \ (\text{List} \ \text{Int})) \ ) \\
&\quad (\text{ite } (\text{append} \ l1 \ l2) \\
&\quad \text{12} \\
&\quad (\text{let } ((\text{h1} \ (\text{head} \ l1)) \\
&\quad \quad (\text{t1} \ (\text{tail} \ l1))) \\
&\quad \quad (\text{insert} \ h1 \ (\text{append} \ t1 \ l2))))))
\end{align*}
\text{)}
\]

\[
\text{(set-option :print-success true)}
\]
Essentially, many-sorted (i.e., simply typed) first-order logic with equality

Main differences:

1. Sorts denoted by (first-order) sort terms

Ex:

```
Bool    Int    Elem
(Array Int Elem)
(Set (Array Int Real))
```
Base Logic

- Essentially, **many-sorted** (i.e., simply typed) **first-order logic** with equality

- Main differences:

  1. Sorts denoted by (first-order) **sort terms**
  2. **No distinction** between
     - function, predicate symbols, and logical connectives
     - terms and formulas
     - *e.g.* `not` is a function from `Bool` to `Bool`
Base Logic

- Essentially, many-sorted (i.e., simply typed) first-order logic with equality

- Main differences:
  1. Sorts denoted by (first-order) sort terms
  2. No distinction between
     - function and predicate symbols
     - terms and formulas
  3. Overloading and parametric polymorphism

  e.g.  
  
  + can have type \( \text{Int} \times \text{Int} \rightarrow \text{Int} \) and \( \text{Real} \times \text{Real} \rightarrow \text{Real} \)

  = has type \( \sigma \times \sigma \rightarrow \text{Bool} \) for every sort \( \sigma \)
Base Logic

- Essentially, many-sorted first-order logic with eq.
- Only logical symbols:
  - quantifiers ($\forall, \exists$)
  - let binder
- Sort and function symbols, and their type, declared in
  - predefined theories, or
  - user scripts
- Meaning of theory symbols specified in a theory declaration
Theory Declarations

Theories in the SMT-LIB catalog are defined with theory declaration schemas

- Semi-formal
  - Formally specified: signature (sort & function symbols)
  - Informally specified: semantics

- Parametric
  - Provide some advantages of parametric types
  - But maintain classical many-sorted semantics
Example: Core Theory

(\texttt{theory Core} \\
:sorts ( (\texttt{Bool 0}) ) \\
:funs ( (\texttt{true Bool}) (\texttt{false Bool}) (\texttt{not Bool Bool}) \\
(\texttt{and Bool Bool Bool} :left\_assoc) \\
(\texttt{or Bool Bool Bool} :left\_assoc) \\
(\texttt{xor Bool Bool Bool} :left\_assoc) \\
(\texttt{=> Bool Bool Bool} :right\_assoc) \\
(par (A) (= A A Bool :chainable)) \\
(par (A) (distinct A A Bool :pairwise)) \\
(par (A) (ite Bool A A A)) \\
) \\
:definition \\
"\texttt{Bool} is the two-element domain of Boolean values. For any sort} s, \\
\texttt{- (= s s Bool) is the identity relation over the domain denoted by} s. \\
..."
Example: Lists with Length

```
(theory ListsWithLength
  :sorts ((List 1) (Int 0))
  :funs ((par (X) (nil (List X)))
      (par (X) (cons X (List X) (List X)))
      (par (X) (head (List X) X))
      (par (X) (length (List X) Int)) )
...
)
```

**Sorts:** `Bool, Int, (List Bool), (List Int), (List (List Bool)), (List (List Int)), ...`

**Function symbols:** `(nil (List Int))), (nil (List Bool))),
  (nil (List (List Int))), ...
  (cons Int (List Int) (List Int))),
  (cons Bool (List Bool) (List Bool))), ...`
Current Theories

ArraysEx  Functional arrays with extensionality
Fixed_Size_BitVectors  Bit vectors of all sizes
Core  Core theory, basic Boolean operators
Ints  Integer numbers
Reals  Real numbers
Reals_Ints  Real and integer numbers
SMT-LIB Logics

- For efficiency, SMT typically fix
  - a background theory they reason about
  - a class of formulas they accept as input

- In SMT-LIB, this is reflected in the notion of a (sub)logic, a fragment of the SMT-LIB base logic
SMT-LIB Logics

( theories; free symbols; syntax restrictions )

Ex:

QF_UF = ( Core; free sort and function symbols;
  no quantifiers )

QF_LIA = ( Ints; free constant symbols;
  no quantifiers, only linear terms )

AUFLIA = ( ArraysEx, Ints; free sort and function symbols;
  only linear terms, only arrays of sort
  (Array Int Int))

Several of the logics define a decidable fragment of FOL
Example: QF_IDL

(logic QF_IDL
  :smt-lib-version 2.0
  :written_by "Cesare Tinelli"
  :date "2010-04-30"
  :theories ( Ints )
  :language
  "Closed quantifier-free formulas with atoms of the form:
  - q
  - (op (- x y) n),
  - (op (- x y) (- n)), or
  - (op x y)
  where
  - q is a variable or free constant symbol of sort Bool,
  - op is <, <=, >, >=, =, or distinct,
  - x, y are free constant symbols of sort Int,
  - n is a numeral."
)
Commands

- Fed to solver’s standard input channel or stored in a file
- Look like Lisp function calls: (⟨com_name⟩⟨arg⟩*)
- Operate on a stack of assertion sets
- Cause solver to output an S-expression to standard output or standard diagnostic channel

Four categories:
- assertion-set commands, modify the assertion set stack
- post-check commands, query about the assertion sets
- option setting commands, set solver parameters
- diagnostic commands, get solver diagnostics
Assertion Sets

Assertion: a formula, a symbol declaration, or a symbol definition

Assertion set: a set of assertions

Assertion stack: a stack of assertion sets (stack frames)

- Theory symbols are implicitly declared in initial, empty stack frame
- Each stack frame defines a lexical scope for (new) symbols declared/defined in it
- Popping a frame retracts all assertions in it
Assertion-Set Commands

(set-logic $s$)

Ex.: (set-logic QF_LRA)

Effect: establishes the logic to be used

(push $n$)

Ex.: (push 1)

Effect: pushes $n > 0$ empty frames into the stack

(pop $n$)

Ex.: (pop 1)

Effect: pops the most recent $n > 0$ frames from the stack
Assertion-Set Commands

(declare-sort $s$ $n$)

**Ex.:**
(declare-sort Elem 0)
(declare-sort Set 1)

**Effect:** declares sort symbol $s$ with arity $n$ and allows the use of sorts such as Elem, (Set Elem), (Set (Set Elem)), ...

(define-sort $s$ $(u_1 \ldots u_n) \sigma$)

**Ex.:**
(define-sort MyArray (u) (Array Int u))

**Effect:** allows use of, e.g., (MyArray Real) as a shorthand for (Array Int Real)
Assertion-Set Commands

(declare-fun \( f (\sigma_1 \ldots \sigma_n) \sigma \))

Ex.:  (declare-fun a () Int)
       (declare-fun even (Int) Bool)
       (declare-fun nth ((List Real) Int) Real)

Effect: declares \( f \) with type \( \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma \)

(define-fun \( f ((x_1 \sigma_1) \ldots (x_n \sigma_n)) \sigma t \))

Ex.:  (define-fun a () Int 4)
       (define-fun sq ((x Int)) Int (* x x))

Effect: declares \( f \) with type \( \sigma_1 \times \ldots \times \sigma_n \rightarrow \sigma \) and asserts

\[ (\forall ((x_1 \sigma_1) \ldots (x_n \sigma_n)) \Rightarrow (f x_1 \ldots x_n)) = (f x_1 \ldots x_n) t) \]
Assertion-Set Commands

(assert \( t \))

Ex.:  
ASSERT: (assert \( \Rightarrow P \ O \))

(assert (or (> \( x \) 1) (= \( x \) \( y \))))

(assert (forall ((\( x \) \( A \))))

(exists ((\( y \) \( B \))) (p \( x \) \( y \))))

(assert (\( \Rightarrow \) \( P \) (! (and \( Q \) \( R \)) : named F))))

Effect:  adds \( t \) of sort Bool to the current frame

(check-sat)

Effect:  checks if all asserted formulas are satisfiable in the specified logic

Returns sat, unsat or unknown
Post-Check Commands

\( (\text{get-value} \ (t_1 \ldots t_n)) \)

**Ex.:**  
(\text{get-value} \ (x \ (+ \ y \ z) \ y ))  
(\text{get-value} \ (\text{(select} \ a \ n)) )

**Effect:** returns the value of quantifier-free terms \( t_1 \ldots t_n \) in the current model  
Output has the form \( ((t_1 \ v_1) \ldots (t_n \ v_n)) \)

\( (\text{get-unsat-core}) \)

**Effect:** computes an *unsatisfiable core* of the asserted formulas  
Output is restricted to labels \( l \) of formulas \( t \) asserted with \( (\text{assert} \ (t : \text{named} \ l)) \)

See SMT-LIB 2 reference document for the full command list
SMT-LIB 2 Language Demo
Resources and Tools
Resources and Tools

- **Documents**
  - Official V. 2 reference (Barrett, Stump & Tinelli)
  - Tutorial (Cok)

- **Scripts**
  - Benchmark library (Barrett & Deters)
  - Validation suite (Cok)

- **SMT-EXEC** (Deters & Stump)
Resources and Tools

- (Partially) Conformant SMT solvers
  - AProVE
  - CVC3
  - CVC4
  - MathSAT 5
  - MiniSmt
  - OpenSMT
  - SimplifyingSTP
  - SONOLAR
  - veriT
  - Yices
  - Z3
  - ...

SAT/SMT Solver Summer School

Boston, 2011
Resources and Tools

- **Parsers and type checkers in**
  - C99: (Griggio)
  - Haskell: (Hawkins)
  - Java: (Cok)
  - OCaml: (Krchak & Stump)

- **Converters and adapters**
  - jSMTLIB (Cok)
Resources and Tools

- **Java API** for programmatic interaction and user extension
  - jSMTLIB

- **Eclipse plug-in**
  - jSMTLIB

[www.smt-lib.org](http://www.smt-lib.org)
SMT-LIB Repository Demo
SMT-EXEC Demo
Future Directions
Future Directions

- More
  - theories and logics (Inductive Data Types, Finite Sets, Finite Maps, Partial Orders, FP Arithmetic, Strings, ...)
  - benchmarks
  - commands
- Standard formats for
  - proofs
  - runtime statistics
- StarExec: mega execution service for logical systems (not just SMT)
How You Can Contribute

- Provide feedback on the standards: language, theories, logics, commands
- Use the SMT-LIB 2 language to communicate with compliant solvers
- Submit your benchmarks to the repository
  - if they do not fit in the existing logics, we’ll create a new one!
- Write a compliant SMT-LIB 2 solver and participate to SMT-COMP
- Write and share utility tools (parsers, converters, editor modes, ...)