DRAFT: The SMT-LIB Command Language

Aaron Stump
CS Department
The University of Iowa
http://www.cs.uiowa.edu/~astump

January 11, 2010

1 Motivation

To enable finer-grained interaction of SMT solvers with other tools, such as verification tools, which wish to call them, this document proposes a Command Language for SMT-LIB. The calling tool issues commands in the format of the Command Language to the SMT solver via a standard textual input channel. The SMT solver then replies over a standard textual output channel, in the format stipulated by the Command Language. Briefly, commands are included for:

- managing a stack of *assertion sets*
- declaring extra sorts, functions, and predicates in the current assertion set
- defining variables to be formulas or terms in the current assertion set
- adding assumptions to the current assertion set
- checking satisfiability of a formula with respect to all assertions in the assertion-set stack
- setting solver options and getting additional information from the solver
- requesting refutation proofs for unsatisfiability.

This format relies on the SMT-LIB Formula Language, documented in "The SMT-LIB Standard", for the format of the assumptions and the formulas queried for satisfiability [2]. While version 2.0 of the specification of the Formula Language is nearing completion, this document, developed in parallel with version 2.0 of the Formula Language, assumes version 1.2. We will update the document for version 2.0 shortly. Also, a standard approach to models which uses commands to query a satisfying model of the formula is still being devised. The syntax and informal semantics of commands are presented in Section 2. Also, changes to the version 1.2 Formula Language's format for benchmarks are proposed in Section 3.11.

2 Commands

Commands have the concrete syntax given by the grammar of Figure 1. Here, we reference syntactic categories from the SMT-LIB standard (version 1.2, as explained above), Section 7, and standard grammar

$$
\begin{array}{lcl}
\langle \textit{command} \rangle & ::= & \texttt{(\ set-logic}\ \langle \textit{logic\_name} \rangle\ \texttt{)} \\
& | & \texttt{(\ declare-sorts\ (}\langle \textit{sort\_symb} \rangle^{+}\ \texttt{))} \\
& | & \texttt{(\ declare-funs\ (}\langle \textit{fun\_symb\_decl} \rangle^{+}\ \texttt{))} \\
& | & \texttt{(\ declare-preds\ (}\langle \textit{pred\_symb\_decl} \rangle^{+}\ \texttt{))} \\
& | & \texttt{(\ define}\ \langle \textit{fvar} \rangle\ \langle \textit{an\_formula} \rangle\ \texttt{)} \\
& | & \texttt{(\ define}\ \langle \textit{var} \rangle\ \langle \textit{an\_term} \rangle\ \texttt{)} \\
& | & \texttt{(\ push}\ \langle \textit{numeral} \rangle\ \texttt{)} \\
& | & \texttt{(\ pop}\ \langle \textit{numeral} \rangle\ \texttt{)} \\
& | & \texttt{(\ assert}\ \langle \textit{an\_formula} \rangle\ \texttt{)} \\
& | & \texttt{(\ check-sat\ )} \\
& | & \texttt{(\ get-assertions}\ \langle \textit{string} \rangle\ \texttt{)} \\
& | & \texttt{(\ get-sat-assertions}\ \langle \textit{string} \rangle\ \texttt{)} \\
& | & \texttt{(\ keep-sat-assertions\ )} \\
& | & \texttt{(\ get-unsat-core}\ \langle \textit{string} \rangle\ \texttt{)} \\
& | & \texttt{(\ get-proof}\ \langle \textit{string} \rangle\ \texttt{)} \\
& | & \texttt{(\ set-option}\ \langle \textit{string} \rangle\ \langle \textit{string} \rangle^{+}\ \texttt{)} \\
& | & \texttt{(\ get-info}\ \langle \textit{string} \rangle\ \texttt{)} \\
& | & \texttt{(\ set-info}\ \langle \textit{string} \rangle\ \langle \textit{string} \rangle\ \texttt{)} \\
& | & \texttt{(\ exit\ )} \\
\\
\langle \textit{script} \rangle & ::= & \langle \textit{command} \rangle^{*}
\end{array}
$$

Figure 1: Syntax for SMT-LIB Commands and Scripts

meta-syntax from Section 3 [2]. The reader not fluent in this standard is requested to consult particularly Section 7 as necessary to follow the present syntax. Standard SMT-LIB whitespace and comments are allowed between all terminals and non-terminals of the grammar. A script is just a sequence of commands. The commands `push`, `pop`, `declare-`$\star$, `define`, `assert`, and `check-sat` are called *assertion-set commands*, because they operate on the assertion set stack (explained further below).

3 Informal Semantics

The informal semantics of scripts for an SMT solver is given in this Section. After a few preliminary considerations, we describe how solvers conforming to this specification should respond to the commands related to the assertion-set stack, declarations and definitions, asserting and checking satisfiability, getting evidence, setting options, and reporting additional information.

3.1 Filenames

Several commands interpret strings they are given as file names. The string `"stdout"` must be interpreted as referring specially to the standard output of the solver (not a file called `stdout` on disk), and similarly for `"stderr"`.

3.2 Responses and Errors

When a solver completes its processing in response to a command, by default it should print to its standard output channel a ⟨*response*⟩:

$$\langle response \rangle \quad ::= \quad \texttt{success} \mid \langle error \rangle$$

$$\langle error \rangle \quad ::= \quad \texttt{(}\ \texttt{error}\ \langle string \rangle\ \texttt{)}$$

This default format applies to commands discussed below, unless otherwise noted. The string given to `error` may be empty (but in that case should still be present as `""`) or else an otherwise unspecified message describing the problem encountered. Tools communicating with an SMT solver thus can always determine when the solver has completed its processing in response to a command. Several options described in Section 3.7 below affect the printing of responses, in particular by suppressing the printing of "`success`", and by redirecting the standard output.

Errors and solver state. This standard gives solvers two options when encountering errors. They may either print an error message in the above format and then immediately exit with a non-zero exit status; or else leave the state of the solver unmodified (by the command which encountered an error), and continue accepting commands. For the second option, the solver's state should remain unmodified by the error-generating command, except possible for timing and diagnostic information). In particular, the assertion-set stack, discussed just below (Section 3.4) is not modified. The motivation for specifying these two modes in this document is that the first mode (exiting immediately when an error occurs) may be simpler to implement, while the latter may be more useful for applications, though it might be more burdensome to support the semantics of leaving the state unmodified by the erroneous command. The standard "`error-behavior`" name can be used with the `get-info` command to check which error behavior the tool supports (see Section 3.8 below).

3.3 Setting the Logic

The `set-logic` command tells the solver what logic (in the sense of the SMT-LIB specification, Section 4.1 [2]) is being used. For simplicity, it is an error for more than one `set-logic` command to be issued to a single running instance of the solver, and it is an error for the ⟨*logic_name*⟩ given not to correspond to an SMT-LIB logic. The logic must be set before any of the commands for declaring symbols, or `assert` or `check-sat` are used.

3.4 The Assertion-Set Stack

As mentioned above, the solver maintains a stack of sets, containing some locally scoped information: formulas asserted to be true, declarations, and definitions. Some terminology related to this data structure is needed:

- **assertion-set stack:** the single global stack of sets.
- **assertion sets:** the sets which are the elements on the stack.
- **set of current assertions:** the union of all the assertion sets currently on the assertion-set stack.

The `(push N)` command pushes N empty assertion sets onto this stack (typically, N will be 1). The command `(pop N)` pops the top `N` assertion sets from the stack. If `N` is greater than the current stack depth, an error results. If `N` is `0`, no assertion sets are popped.

Note that, following lengthy discussion in the working group, we have decided to make declarations and definitions part of assert sets. This means that popping an assertion set will remove any declarations and definitions that were added to it. This alternative has several advantages over the main competing alternative, which would be to have explicit commands for undefining and undeclaring symbols. One advantage, for example, is that it is no longer necessary to specify (as part of this document) how to print symbols that have been undefined or undeclared. Furthermore, the scenarios we know of in which it is desired to undefine (or undeclare) a symbol are ones in which the symbol is being undefined so that it can be redefined (or redeclared). This functionality is easily supported by pushing an assertion set, adding the declaration or definition to it, and then popping the assertion set later, when it is time to redeclare or redefine the symbol. As a final note, some members of the SMT-API working group have suggested selective retraction (of assertions, declarations, or definitions) as a desirable feature. Unfortunately, this seems likely to complicate solver implementation significantly, so this standard stays with the more conservative stack discipline.

3.5 Declarations and Definitions

As just explained, the `declare-sorts`, `declare-funs`, and `declare-preds` commands declare new symbols of the appropriate syntactic kind, in the current assertion set. These symbols are local, in the sense that popping assertion sets removes the corresponding declarations of sort, function, and predicate symbols. It is an error for a declared symbol to be shadowed by any other declared or defined symbols.

Variables may be defined to equal formulas or terms using `define`. Following the current SMT-LIB standard (version 1.2) for formulas, we distinguish term-level and formula-level variables. (This distinction is eliminated in version 2.0 of the standard.) Like declarations, definitions have local scope. Also like declarations, defined variables may not be shadowed. Note that some solvers may be able to deallocate memory associated with definitions that have been popped. This functionality is not required, however, for solvers to conform to this standard.

Printing defined symbols. All output from a solver compliant with this specification should print defined symbols just as they are, without replacing them by the expression they are defined to equal. This approach generally keeps output from solvers much more compact than with definitions expanded. An option is included below (Section 3.7), however, to expand all definitions in solver output.

3.6 Assertions and Checking Satisfiability

The `assert` command adds an asserted formula to the assertion set on the top of the assertion-set stack. The `get-assertions` command causes the solver to print the set of current assertions in a parenthesis-delimited list. That is:

$$\texttt{(}\ \langle \textit{formula} \rangle^{*}\ \texttt{)}$$

This list is printed to the file whose name is the sole argument to the command. The `check-sat` command instructs the solver to check whether or not the conjunction of the current assertions is satisfiable. When it has finished attempting to do this, the solver should reply on `stdout` with a $\langle \textit{status} \rangle$ (either `sat`, `unsat`, or `unknown`). The `get-proof` command can be issued only following a `check-sat` command which reports unsatisfiability, without intervening assertion-set commands. In response to a `get-proof` command in this situation, the solver should write a refutation proof to the file whose name is given as a string

input to the command. As mentioned above, there is, as yet, no standard SMT-LIB proof format, so this proof will necessarily be in a solver-specific format. Solvers that do not support proof production should output "unsupported" to the indicated file.

If the solver reports "sat" or, optionally, if it reports "unknown", it should respond to the commands get-sat-assertions and keep-sat-assertions. It is permitted for a push command to intervene between a check-sat command and these commands. For the first, the solver should output (to the named file) a parenthesis-delimited list of formulas whose conjunction is equivalent to the partial assignment of truth values to atomic formulas with which the solver ended its search. The command keep-sat-assertions requests the solver to assert all of those formulas.

If the solver reports "unsat", it should respond to the unsat-core command, by printing a parenthesis-delimited list of a subset of the set of current assertions, which the solver has determined is inconsistent. Notice that solvers conforming to this standard can simply print the entire set of current assertions in response to the command. But some solvers may provide functionality for discovering a proper inconsistent subset of the set of current assertions. This information can be very useful for applications, and so this standard includes a command for exporting it from solvers.

A check-sat command may be followed by other assert and check-sat commands, without an intervening pop. In that case, the semantics is that subsequent assert commands are just extending the current assertion set, and check-sat commands are checking satisfiability of the resulting set of current assertions. So subsequent check-sat commands are never handled with respect to a partial assignment found by an earlier check-sat command (unless that assignment is explicitly asserted with assert or keep-sat-assertions by the calling environment). This restriction is for ease of solver implementation. For applications like enumerating all the models of a formula, see the Examples section below.

The exit command instructs the solver to exit. The interface to the set-option and get-info commands is described in detail next.

3.7 Formats for `set-option`

Solvers options may be set and solver information queried using set-option and get-info commands, respectively. The name of the option or the piece of queried information is given by the first string argument to the command. Solver-specific names are allowed and indeed expected. A set of standard names is catalogued below and in the next section. This Command Language specification requires solvers to recognize and reply in a standard way to a few of these names. The majority, however, solvers need not support, although in that case they should reply "unsupported". These sets of names are likely to be expanded or otherwise revised as further desirable common options and kinds of information across tools are identified. For set-option commands, the following format is required for replies, for both solver-specific and standard options:

⟨*set-option-response*⟩ ::= unsupported | success

The current list of standard option names is given next, together with default values and whether or not the option must be supported (by solvers conforming to this specification).

print-success, default "true", required. Setting this to "false" causes the solver to suppress the printing of "success" in all responses to commands. Other output remains unchanged.

`random-seed`, no default value, optional. The argument is a positive numeral (as a ⟨*string*⟩) for the solver to use as a random seed, in case the solver uses (pseudo-)randomization.

`timeout`, default value is "-1" (meaning no timeout), optional. The argument is either "-1" or a positive numeral (as a ⟨*string*⟩) for the solver to use as the number of seconds before timing out, if the solver supports limiting its own runtime.

`expand-definitions`, default "false", optional. If the solver supports this option, setting it to "true" causes all subsequent output from the solver to be printed with all definitions fully expanded. That is, subsequent output should contain no defined symbols at all, only the (full expansions of the) expressions they are defined to equal.

`verbosity`, default "0", optional. The argument is a non-negative numeral controlling the level of diagnostic output written by the solver. All such output should be written to a secondary output channel, called the diagnostic output channel, to avoid confusion with the responses to commands which are written to standard output. These channels can be changed via the "output-channel" and "diagnostic-output-channel" options below. An argument of 0 requests that no such output be produced. Higher values then request more verbose output.

`output-channel`, default `stdout`, required. The argument should be a filename to use subsequently for the output channel.

`diagnostic-output-channel`, default `stdout`, required. The argument should be a filename to use subsequently for the verbose output channel.

3.8 Formats for `get-info`

For `get-info` commands, the following format is required for replies, for both solver-specific and standard information names:

⟨*get-info-response*⟩ ::= `unsupported` | `(` ⟨*key-value-pair*⟩$^+$ `)`

⟨*key-value-pair*⟩ ::= `(` ⟨*string*⟩ ⟨*string*⟩ `)`

The different information names and more specific formats of key-value pairs given in responses are given next. First we discuss statistics, then some additional pieces of information.

3.9 Standard Names for Statistics for `get-info`

The `statistics` name may be given to `get-info` to get various solver statistics. Supporting it is optional. Solvers reply with a parenthesis-delimited list of key-value pairs giving statistics on the most recent `check-sat` command. It is required that this `check-sat` command is one without any intervening assertion-set commands. Each statistic's name below may be given as an information name in its own right to `get-info`. The format for the reply to `statistics` is just the aggregation of the replies for each individual statistic. A solver may reply to `statistics` giving possibly just some of the statistics listed next, and possibly some additional solver-specific ones.

decisions: optional. This, like many other kinds of information, is what we shall call a *singleton*: solvers reply with just one key-value pair, where the key is again the name of the piece of information. Note that the reply must still conform to the above response format, so this key-value pair must be in its own list, like: `(("decisions" "3"))`. For `decisions`, the value is a string representation of a non-negative numeral giving the number of decisions made during search.

`conflicts`, optional. This is a singleton, where the value is the number of conflicts encountered during search.

`restarts`, optional. This is a singleton, where the value is the number of restarts performed during search.

`time`, optional. This is a singleton, where the value is the running time taken during search.

`memory`, optional. This is a singleton, where the value is the memory allocated during search.

3.10 Additional Standard Names for `get-info`

Some further standard names for `get-info` are given here. A few of these may be set by the `set-info` command. Setting other standard names with `set-info` results in an error. Support for setting values for other names with `set-info` is optional.

`error-behavior`, required. The response is either `"immediate-exit"` or `"continued-execution"`. In the former case, the solver will exit immediately when an error is encountered. In the latter, the solver will leave the state unmodified by the erroneous command, and continue accepting and executing new commands. See Section 3.2 above for more on the motivation for these two error behaviors.

`name`, required. A singleton, where the value is the name of the solver.

`version`, required. A singleton, where the value is the version number of the solver (e.g., "1.2").

`authors`, required. The response is a list of key-value pairs where the key is `author` and the value is the name of one of the tool's authors.

`status`, required (may be set with `set-info`). The status of the most recent `check-sat` command or the value most recently set for `status` with `set-info`, whichever is more current.

`reason-unknown`, optional. If the status of the most recent `check-sat` command is `unknown`, this gives a short reason why the solver could not successfully check satisfiability, from the following options: `timeout` (for exceeding a time limit, if one is currently set), `memout` (for out of memory), or `incomplete` (if the solver knows it is incomplete for the class of formulas containing the most recent query).

`notes`, required (maybe be set with `set-info`). Any notes about the current (or, if set by `set-info`, future) set of asserted formulas.

3.11 Benchmarks and the `set-info` Command

The SMT-LIB Formula Language version 1.2 includes a format for benchmarks. The SMT-LIB initiative has collected a large number of benchmarks in this format, for a variety of different logics. These benchmarks are used in the SMT research community for standardized comparison of solvers, as well as for the SMT Competition (SMT-COMP) [1]. Since the current format for benchmarks overlaps considerably with the proposed Command Language, it is here proposed that they be consolidated, as follows. Benchmarks will become scripts, possibly making use of the "`set-info`" command to include some declarative information. There are also the following additional requirements. Some of these are expected well-formedness conditions, which we list in summarized form (see Sections 5 and 7 of "The SMT-LIB Standard" for a more complete description).

- The (single) `set-logic` command setting the benchmark's logic is the first command.
- There is exactly one `check-sat` command.
- There is at most one `set-info` command for `status`.
- The formulas in the script belong to the benchmark's logic, with any free symbols declared in the script.
- Extra symbols are declared exactly once before any use, and are part of the allowed signature expansion for the logic.
- The commands `push`, `pop`, `get-info`, and `set-option` are not used.

It is anticipated that an additional category of benchmarks will be developed where some of these restrictions will be relaxed (e.g., benchmarks which may contain `push` and `pop` commands, and/or multiple `check-sat` commands), but that remains for future consideration. Note that the above requirements leave open the possibility that benchmarks could contain `get-proof` commands. Of course, a solver that does not have support for these is required to output "`unsupported`" in place of a proof.

3.12 Examples

We demonstrate some allowed behavior of an imagined solver in response to several example scripts. Each command is followed by example legal output from the solver in a comment, if there is any. The script in Figure 2 makes two background assertions, and then conducts two independent queries. The `get-info` command requests information on the search.

The script in Figure 3 enumerates all the models of the (very simple) queried formula. Doing this automatically would require that the tool issuing commands to the SMT solver is able to parse the sets of assertions printed in response to `get-sat-assertions`. Also, we set the `print-success` option to `"false"` to suppress printing "`success`".

4 Acknowledgments

Thanks to the members of the SMT-API working group for considering this proposal, and particularly the following people who contributed to the discussion by email: Clark Barrett, Sascha Boehme, David Cok, David Deharbe, Albert Oliveras, Michal Moskal, Leonardo de Moura, Roberto Sebastiani, and Cesare

Tinelli. This work is supported in part by NSF award 0551697 “CRI: Collaborative Research: SMT-LIB, A Common Library and Infrastructure for Satisfiability Modulo Theories”.

References

[1] C. Barrett, L. de Moura, and A. Stump. Design and Results of the 2nd Annual Satisfiability Modulo Theories competition (SMT-COMP 2006). *Formal Methods in System Design*, 31(3):221–239, 2007.

[2] S. Ranise and C. Tinelli. The SMT-LIB Standard, Version 1.2, 2006. Available from the ”Documents” section of http://combination.cs.uiowa.edu/smtlib.

```
(set-logic QF_LIA)
; (success)

(declare-funs ((w Int) (x Int) (y Int) (z Int))
; (success)

(assert (> x y))
; (success)

(assert (> y z))
; (success)

(push)
; (success)

(assert (> z x))
; (success)

(check-sat)
; unsat

(get-info "statistics")
; (("decisions" "0") ("restarts" "0"))

(pop)
; (success)

(push)
; (success)

(check-sat (= x w))
; sat

(get-sat-assertions "stdout")
; ( (> x y) (> y z) (= x w) )

(exit)
; (success)
```

Figure 2: Example Script with Solver Responses in Comments

```
(set-logic QF_UF)
; success

(set-option "print-success" "false")
;

(declare-preds ((p) (q)))
;

(assert (implies p q))
;

(push)
;

(check-sat)
; sat

(get-sat-assertions)
; ( (not p) )

(pop)
;

(assert p)
;

(push)
;

(check-sat)
; sat

(get-sat-assertions)
; ( p q )

(pop)
;

(assert (not q))
;

(push)
;

(check-sat)
; unsat

(get-proof "stdout")
; unsupported

(pop)
;
```

Figure 3: Interaction Enumerating All Models of a Simple Formula