ArgoSMTExpression: an SMT-LIB 2.0 compliant expression library

Milan Banković
milan@matf.bg.ac.rs

Faculty of Mathematics, University of Belgrade

Abstract. In this paper, we describe our library for handling first order expressions that is fully compliant with the SMT-LIB 2.0 standard. It enables efficient representation of first order expressions and their sorts (types). It supports signature combination, well-sortedness checking and sort inference. The library includes declarations of all standard SMT theories and logics, and also permits declaring new ones. Finally, it provides an API which enables working with the SMT expressions and integrating the library with the solvers and tools in a simple and straightforward way. The library is written in the C++ programming language, and includes a Lex/Yacc generated parser. It is meant to be extensible, flexible and easy to use. It is a free software, released under the GNU/GPL license.

Keywords: SMT-LIB 2.0, expression library, SMT solver API, well-sortedness checking

1 Introduction

Satisfiability modulo theories (SMT) is the problem of deciding, given a first order formula F and a background theory $\mathcal{T}$, whether F is satisfiable with respect to $\mathcal{T}$. In recent years, SMT has become a significant research area, since plenty of practical problems can be represented and solved as SMT problems. The applications include hardware and software verification, scheduling problems, model checking, etc. Because of the applicability, a great number of research groups work on development and implementation of SMT solvers and tools.[1] More information about SMT can be found in [1].

Because of the growing popularity of SMT solving, there is an increased need for standardization and unification of the language and interface used by different SMT solvers. In 2003, SMT-LIB initiative[2] was started with a goal to standardize theories and logics commonly used in SMT solving, to develop a standardized input and output language for SMT solvers, and to establish a large library of benchmarks for testing and comparing SMT solvers. The current

[1] Some of the most significant state-of-the-art solvers are listed at: `http://goedel.cs.uiowa.edu/smtlib/solvers.html`.

[2] For more information about SMT-LIB, see the official web page: `http://goedel.cs.uiowa.edu/smtlib/`.

version 2.0 ([2]) of the standard was released in 2010, and currently most of the leading SMT solvers are in the process of adopting the new standard.

One of the first issues in developing an SMT solver is to implement efficient representation of first order expressions used by the solver. Since this part of the solver is the closest to the user input, it is also the part that is the most dependent on the input language, specifications of used theories, and other features required by the standard. In this paper, we describe a C++ library for representing first order expressions, compliant with the SMT-LIB 2.0 standard. Its most prominent features include: efficient implementation of expressions and their sorts, full well-sortedness checking, sort inference, signature combining and expanding, simple way to define new theories and logics, and standard-compliant application programming interface (API), accompanied with a Lex/Yacc generated parser. The library is designed in a multi-layered fashion, permitting different ways of use, from low level operation, giving the user full control and flexibility, to high level API driven operation, in which case using the library is simple and straightforward. The library is developed in a way that permits extensions and new features, as the standard itself evolves in the future. Finally, the library is a free software, and the author hopes that it will be a good starting point for other researchers who want to develop new tools and solvers.

Motivation. Historically, each SMT solver has developed its own input language and its own expression library. These languages and libraries in different solvers are quite similar, but still incompatible. This has raised many issues, regarding the cooperation within the SMT community. Some of these issues were resolved when SMT-LIB was introduced, but some of them still remain:

- expression libraries in existing SMT solvers typically do not support all the features required by SMT-LIB standard. Thus, standardization of the input language alone does not mean much, since simple translation from SMT-LIB standard input to solver specific representation is not always possible, without loss of information.
- intermixing codebases of different SMT solvers is not possible, since their representations of expressions are incompatible. For instance, using the same implementation of some theory decision procedure with different SMT solvers is impossible without special adapters that transform between different expression representations.
- developing a new SMT solver requires implementing the expression library from scratch, since there is no standard SMT expression library (contrary to many other areas of computer programming, where providing a standard implementation of common software parts is considered as a natural step).
- developing other tools that depend on SMT solvers as backends can be a tedious work, if a tool should be able to work with different SMT solvers. In that case, if a tool generates an SMT formula to be solved, it must support exporting the formula to different expression implementations, since it must be able to communicate with the API's of different solvers.

A natural solution to all these issues is adoption of one standard expression library, compliant with the SMT-LIB standard. So, in the authors opinion, the

standard itself is just one step in achieving the goals of the SMT-LIB initiative – the standard expression library is what would enable full cooperation within the SMT community.

Related work. As far as we know, there is no open source SMT expression library that implements all the features of the current SMT-LIB standard. OpenSMT ([3]) solver includes an open source implementation of expressions, but still does not fully support the standard. Other related work include several tools that act as parsers and translators to other solver specific representations, but these tools do not include expression libraries. There is a Java SMT-LIB 2.0 compliant suite of tools ([4]) that acts as a front-end to several different (non-conforming) SMT solvers. An open-source C99 parser ([5]) also exists, and it provides an abstract interface that enables integration with an SMT solver. There are also parsers available in Haskell ([6]) and Ocaml ([7]) languages.

2 The library description

In this section we provide a description of the library. We adopt notation, concepts and definitions from [2]. Due to the lack of space, no usage examples are provided in the text. Detailed documentation and usage examples (along with the source code of the library itself) are available at the project's web page: `http://www.matf.bg.ac.rs/~milan/software/argosmte/`.

2.1 Low level: signatures, expressions and sorts

Basic types. The library provides classes that represent basic lexical and syntactical elements defined by the standard ([2]) (symbols, keywords, identifiers, variables, etc.). All these basic types are internally represented as strings. The attribute type is defined as a `(keyword, value)` pair. Attribute value is represented by an abstract class that can contain virtually any kind of data. This enables versatile usage of attributes during the solving process (not just for purposes currently defined by the standard).

Expressions. Expressions (or *terms* in SMT-LIB's terminology, [2]) are represented using the *common subexpression sharing* technique. Expressions are maintained within the *expression factory*, which is responsible for creation and destruction of the expressions. Each expression is represented by a unique *expression node*. The factory keeps all created expression nodes in a hash table, and returns a *shared pointer* to a node, when requested. Common subexpressions are shared between different nodes – each node keeps only shared pointers to its subnodes, which saves the memory. Shared pointers are used as handles for manipulating the expressions in the factory. Expressions are equipped with a rich interface that allows convenient work with them (extracting subexpressions, free variables, expanding definitions, etc.).

Sorts. Since sorts in SMT-LIB 2.0 can have term-like structure ([2]), they are represented in the similar way as the expressions. All sort nodes are created and maintained within the *sort factory*, and a shared pointer is returned to the user

as a handle for the created sort. As for the expressions, a similar interface is provided for convenient work with sorts.

Signatures. The purpose of the `Signature` class is to hold the information about declared sort symbols, function symbols and special constants (or special constant types, e.g. `NUMERAL`s). Different forms of declarations are permitted – from simple declarations (for instance, when function symbol is declared with given rank and attributes), to complex abstract declarations (for instance, when function symbols are declared in a free form language, with complex constraints). In the latter case, the user can add an abstract *checker* that can accept arbitrary user-defined symbols and check the symbols against arbitrary constraints.

Signature hierarchy. Notice that SMT-LIB 2.0 standard ([2]) defines two different types of relations between signatures: *expansion* and *combination*. Expansion means that one signature contains other signature. Combination means the union of two or more signatures. Our library supports both types of relations. Combination is typically used in logic declarations, when signatures of different theories are combined into one logic's signature. Expansion is useful when the user wants to temporarily add some symbols to the signature, and then remove them when they are out of scope (for instance, when `push` and `pop` API commands are invoked, [2]). In that case, one can attach a signature expansion to the existing signature, and add these temporary symbols to that expansion. When expansion is not needed any more, it is simply destroyed.

Well-sortedness. After a signature is created, an expression factory can be created over the signature. When expressions (sorts) are created, expression (sort) factory consults the signature to check for function (sort) symbols or special constants. This well-sortedness checking can be turned off, for efficiency. During the well-sortedness checking, sort inference is performed – each expression is assigned a sort, retrieved from the corresponding rank of the top function symbol. If well-sortedness checking is turned off, no sort inference is performed and sorts of all expressions are set to *undefined* (except in case when the sort is explicitly given, for instance by `as` clause, [2]). In that case, sort inference can be explicitly invoked later, if needed.

2.2 Intermediate level: theory and logic declarations

When the library is used at the low level, one must manually create and populate signatures with appropriate sort and function symbols. This gives power and flexibility (especially when custom theories and logics are declared), but can be troublesome and error-prone. Easier way is to define a signature for some theory or logic once, and then create an instance of it when needed. Abstract classes `TheoryDeclaration` and `LogicDeclaration` act like *virtual constructors* – their subclasses should define one virtual method `createSignature` that creates and populates an appropriate signature and returns a pointer to it. The library includes such subclasses for each standard SMT-LIB 2.0 theory and logic. In order to create the signature of some standard theory or logic, an instance to appropriate subclass should be created (for instance, `IntsTheoryDeclaration` for `Ints`

theory), and its `createSignature` method should be invoked. The signature is then created and ready to use.

2.3 High level: SMT-LIB 2.0 API

The current standard of SMT-LIB ([2]) defines *scripts* as a sequence of *commands*. The set of allowed commands and their syntax and semantics is also defined by the standard. In general, commands are dedicated to the SMT solver which should respond to them by appropriate action. However, while certain commands *are* dependent on the solver, many commands do not involve any actual SMT solving for their execution. For instance, commands `check-sat`, `get-unsat-core`, `get-values` and `get-assignment` require that the SMT solver first checks for satisfiability of the current set of all assertions ([2]). On the other side, commands `declare-sort`, `declare-fun`, `push` and `pop` just affect the current signature, and do not require any solving (the same is with commands `set-option`, `set-info`, `get-option` and `get-info`). Our idea is to separate and fully implement the commands that do not need the solver for their execution, and provide the standard interface to the solver for those commands that require actual solving. The class `SMTLibAPI` provides API for all commands defined by the standard. The class includes internal database of logic declarations that can be activated by the `set-logic` command ([2]). All standard logic declarations are included by default, and new ones can be easily registered. When `set-logic` command is invoked, the corresponding `LogicDeclaration` instance creates the signature for that logic. It also creates an expression factory over that signature. After that, sorts and expressions can be created in a usual fashion.

Assertions. The class `SMTLibAPI` also implements the *assertion set stack* ([2]). When the `push` command ([2]) is invoked, a new assertion set is initialized and pushed to the assertion set stack, and also a new expansion signature is attached to the current signature and pushed to the *signature stack*. All sort and function symbols declared and defined subsequently are added to that expansion signature. All expression subsequently asserted are added to the current assertion set. When the `pop` command ([2]) is invoked, the current assertion set is poped from the assertion set stack, and the top expansion signature is poped from the signature stack and destroyed, together with all symbols declared in it.

Symbol definitions. Function and sort symbol definitions are also supported. When commands `define-sort` or `define-fun` ([2]) are invoked, the corresponding symbol is added to the top expansion signature, and the symbol's definition is attached to it as a value of `:definition` attribute. Symbol definitions are used in sort and expression expansion, also fully implemented in the library.

Communication with the solver. The state of the `SMTLibAPI` class instance can be easily read by the SMT solver, using the interface of the class. For those API commands that depend on the SMT solver, the library provides an abstract class `SolverInterface` that should implement virtual methods `checkSat`, `getValue`, `getProof`, and so on. The library user should define a subclass of this class that should either implement the SMT solver itself, or implement an adapter to the SMT solver's code. The instance of that subclass is then assigned

to the `SMTLibAPI` instance, and further operation can be driven completely using the interface provided by the `SMTLibAPI` class. Commands that require SMT solver are then just delegated to the solver through the `SolverInterface` class.
Parser. The class `SMTLibAPI` also includes the method `parseInput` that implements a Lex/Yacc generated parser. Therefore, instead of invoking the commands programmatically (through API), the library can parse an input file with some SMT-LIB 2.0 script.

3 Conclusions and Further Work

In this paper, we have described a free and open source C++ program library that implements first order expressions, compliant with the SMT-LIB 2.0 standard. It enables easy development of conforming tools and solvers, and provides different ways to work with expressions, from low level manipulation, to high level API driven operation. It is developed in a way that permits further extensions, as the standard itself evolves. Contrary to the existing libraries and utilities, our library fully implements first order expressions – it is not just an SMT-LIB 2.0 parser and syntax checker that works as a wrapper or an interface to existing SMT solvers.

The author tried to make the library as complete as possible, but is quite sure that some features might be missing. Efficiency improvements and profiling may also be required, in order to prepare the library for production use. Further work also includes ensuring thread-safety, to enable the usage of the library in a multi-threaded environment.

Acknowledgements. This work was partially supported by the Serbian Ministry of Science grant 174021 and by the SNF grant SCOPES IZ73Z0_127979/1. The author is grateful to Filip Marić for very careful reading of the text and providing detailed and useful comments and remarks.

References

1. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability. IOS Press (2009) 825–885
2. Barrett, C., Stamp, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. (2010)
3. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT Solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 6015., Paphos, Cyprus, Springer, Springer (2010) 150–153
4. Cok, D.R.: The jSMTLIB User Guide. `http://www.grammatech.com/resources/smt/jSMTLIBUserGuide.pdf` (2011)
5. Griggio, A.: A Generic Parser for SMT-LIB v2. `https://es.fbk.eu/people/griggio/misc/smtlib2parser.html` (2011)
6. Hawkins, T.: A library for reading and writing SMT-LIB 2 scripts via Haskell. `http://hackage.haskell.org/package/smt-lib` (2010)
7. Krchak, K., Stump, A.: OCaml lexer and parser for SMT 2.0 scripts. `http://www.cs.uiowa.edu/~astump/software/ocaml-smt2.zip` (2010)