The SMT-LIB Standard – Version 2.0

Clark Barrett[1] Aaron Stump[2]
Cesare Tinelli[2]

[1]New York University, barrett@cs.nyu.edu
[2]University of Iowa, astump|tinelli@cs.uiowa.edu

Abstract

The SMT-LIB initiative is an international effort, supported by research groups worldwide, with the two-fold goal of producing an extensive on-line library of benchmarks and promoting the adoption of common languages and interfaces for SMT solvers. This paper introduces Version 2 of the SMT-LIB Standard. This is a major upgrade of the previous Version 1.2 which, in addition to simplifying and extending the languages of that version, includes a new command language for interfacing with SMT solvers.

1 Introduction

Satisfiability Modulo Theories (SMT) is a growing area of automated deduction with many important applications, especially in static analysis and system verification [2]. The main problem in SMT is determining whether a first-order formula is satisfiable in a model that also satisfies one or more fixed background theories. Typical background theories of interest include formalizations of arithmetic, arrays, bit-vectors, inductive data types, equality with uninterpreted functions, and various combinations of these.

SMT-LIB is an international initiative, coordinated currently by the authors and endorsed by a large number of research groups world-wide, aimed at facilitating research and development in SMT [4]. Since its inception in 2003, the initiative has pursued these aims by focusing on the following concrete goals: provide standard rigorous descriptions of background theories used in SMT systems; develop and promote common input and output languages for SMT solvers; establish and make available to the research community a large library of benchmarks for SMT solvers.

The main motivation of the SMT-LIB initiative was the expectation that the availability of common standards and of a library of benchmarks would greatly facilitate the evaluation and the comparison of these systems, and advance the state of the art in the field, in the same way as,

for instance, the TPTP library [7] has done for theorem proving, or the SATLIB library [5] for propositional satisfiability. These expectations have been largely met, thanks in no small part to extensive benchmark contributions from the research community and to an annual SMT solver competition, SMT-COMP [1], based on benchmarks from the library. At the time of this writing, the library contains more than 85,000 benchmarks and continues to grow. Formulas in SMT-LIB format are now accepted by the great majority of current SMT solvers. Moreover, most published experimental work in SMT relies heavily on SMT-LIB benchmarks.

Experience with the previous version (1.2) of the SMT-LIB standard helped identify ways in which the standard could be simplified, extended, and generally improved. In this paper, we give a high-level and selective overview of the four main components of the new standard: the underlying logic and expression language; a language for specifying theories; a language for specifying (sub)logics; and a command-based interface language for SMT solvers. For complete details on the new standard, we refer the reader to the SMT-LIB Version 2.0 Reference Manual [3]. For space reasons, this paper must assume familiarity with SMT, including some acquaintance with the previous version of the SMT-LIB standard. A more tutorial introduction to the new standard from the user's perspective is planned, but outside the scope of this paper.

1.1 Acknowledgments

The SMT community contributed extensively to the creation of this new version of the standard. The first drafts of the main components of Version 2.0 were developed with the input of three international work groups consisting of developers and users of SMT tools: the SMT-API work group, led by A. Stump, the SMT-LOGIC work group, led by C. Tinelli, the SMT-MODELS work group, led by C. Barrett. These groups worked since Fall 2008 into Spring 2010. The groups went through numerous drafts, and exchanged more than 200 emails in the course of discussing them. In early 2010, the various components of the standard were compiled into a single reference document, which was then circulated among the broader SMT community for comments.

Particular thanks are due to the following work group members, who contributed numerous suggestions and helpful constructive criticism in person or in email discussions: Nikolaj Bjørner, Sascha Boehme, David Cok, David Deharbe, Bruno Dutertre, Pascal Fontaine, Vijay Ganesh, Alberto Griggio, Jim Grundy, Paul Jackson, Albert Oliveras, Sava Krstić, Michał Moskal, Leonardo de Moura, Philipp Rümmer, Roberto Sebastiani, and Johannes Waldmann. Many thanks also to Anders Franzén, Amit Goel, and Tjark Weber for additional feedback, and to David Cok and Philipp Rümmer for their careful proof-reading of the entire draft reference manual.

2 Main Logic and Language

As in Version 1.2, the underlying logic for Version 2.0 is many-sorted first-order logic with equality. However, the new version borrows several ideas from higher-order logic. First, there is no longer a distinction between terms and formulas. The latter are simply terms of a distinguished sort `Bool`. This simplifies the concrete syntax, for instance by eliminating the distinction between function and predicate symbols. Second, Version 2.0 allows sort symbols of arity greater than 0, for forming *sort terms* like `(Array Int Int)`. With this feature, parametric types and their theories can be more conveniently represented, while keeping the logic first-order.[1] Overloaded function symbols are also allowed (discussed below).

The concrete syntax used in Version 2.0 is very close to the LISP-based syntax used in Version 1.2, although version 2.0 goes a step further by making every expression a legal *S-expression* of Common Lisp [6]. This means existing parsers and other tools that can already process S-expressions can be used for SMT.

When defining certain SMT-LIB theories, such as bit-vectors, it is convenient to have indexed symbols as identifiers. Instead of having a special token syntax for that (as in `concat[2:3]`), indexed identifiers are defined more systematically as the application of the reserved symbol `_` to a symbol and one or more *indices*, given by numerals (as in `(_ concat 2 3)`).

Several syntactic categories in the language contain *attributes*. These are pairs consisting of an attribute name and an associated value, or just attributes with no value. Attribute names are preceded by the character `:`. Attribute values are S-expressions, although most predefined attributes use a more restricted category.

Well-sorted terms are a subset of the set of all terms. The latter are constructed out of constant symbols (numerals, rationals, strings, etc.), variables, function symbols, a distinguished symbol for equality and one for disequality (respectively `=` and `distinct`), three kinds of binders, and an *annotation operator* (`!`), explained below.

A variable can be any symbol,[2] while a function symbol can be either a symbol or an indexed symbol. Every function symbol f is separately associated with one or more *ranks*, each specifying the sort of f's arguments and result; multi-arity symbols are supported with associativity declarations (see Figure 1 for examples). To simplify sort checking, every ambiguous

[1] In particular, contrary to higher-order typed logics, there is no distinguished binary sort symbol for creating function types. (We use *type* and *sort* as synonyms in the context of the standard.)

[2] A symbol is either a non-empty sequence of letters, digits and the characters ~!@$%^&*_-+=<>.?/ that does not start with a digit, or a sequence of any printable ASCII characters that starts and ends with | and does not otherwise contain |. This significantly extends the class of acceptable symbols from version 1.2 of the standard.

function symbol in a term must be annotated with one of its result sorts σ. Such an annotated function symbol is a *qualified identifier* of the form `(as` f σ`)`.

Binders include `let`, `forall` and `exists`. The `forall` and `exists` binders correspond to the usual existential and universal quantifiers of first-order logic, except that the variables they quantify are sorted. A `let` binder introduces and defines one or more local variables in parallel. A simple example of a formula is the following array axiom:

```
(forall ((a (Array Int Real)) (i Int) (e Real))
  (let ((a1 (store a i e))
    (= (select a1 i) e)))
```

Every term t can be optionally annotated with attributes $\alpha_1, \ldots, \alpha_n$ using the wrapper expression `(!` t α_1 $\cdots$ α_n`)`. These are a convenient mechanism for adding meta-logical information for SMT solvers, for example to help guide quantifier instantiation. Semantically, `(!` t α_1 $\cdots$ α_n`)` is equivalent to t. Currently there is only one predefined term attribute, called `:named` that takes a single symbol as its value. This attribute can be used in scripts to give a closed term a symbolic name, which can be then used as a proxy for the term (as in `(! (- (+ a b) c) :named a_1)`). These definitions are global, in contrast with lexically scoped local definitions introduced by `let`.

3 Specifying Theories

The set of SMT-LIB theories is defined by a catalog of *theory declarations*.[3] While these have a formal syntax and thus could be processed by tools, the primary intention is to provide concise declarative specifications of theories, for reference by solver implementors and users. Theories like `ArraysEx`, for arrays with extensionality, and `Fixed_Size_BitVectors`, for bitvectors of arbitrary but fixed size, have been ported from 1.2.

In Version 1.2, a theory declaration defined a many-sorted *signature*, i.e., a collection of sorts and sorted function symbols, and a theory with that signature. In this version instead, theory declarations can declare entire families of overloaded function symbols by using *sort parameters*, locally scoped sort symbols of arity 0. A theory declaration generally defines a *class* of similar theories. The syntax of theory declarations follows an attribute-based format. A theory declaration consists of a theory name and a list of attributes. Theory attributes with the following predefined names have a prescribed usage and semantics: `:definition`, `:funs`, `:funs-description`, `:notes`, `:sorts`, `:sorts-description`, and `:values`.

[3] Available on the SMT-LIB web site.

```
(theory Core
 :sorts ((Bool 0))
 :funs ((true Bool)  (false Bool)  (not Bool Bool)
        (=> Bool Bool Bool :right-assoc)  (and Bool Bool Bool :left-assoc)
        (or Bool Bool Bool :left-assoc)  (xor Bool Bool Bool :left-assoc)
        (par (A) (= A A Bool))  (par (A) (ite Bool A A))
       )
 :definition
 "For every expanded signature Sigma, the instance of Core with that signature
  is the theory consisting of all Sigma-models in which:
  - the sort Bool denotes the set {true, false} of Boolean values;
  - for all sorts s in Sigma, (= s s Bool) denotes the function that
    returns true iff its two arguments are identical;
  - for all sorts s in Sigma, (ite Bool s s) denotes the function that
    returns its second argument or its third depending on whether
    its first argument is true or not;
  - the other function symbols of Core denote the standard Boolean operators
    as expected.
  "
  :values "The Bool values are the terms true and false."
)

(theory ArraysEx
 :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)))
       )
 :notes
 "A schematic version of the theory of functional arrays with extensionality."
 :definition
 "For every expanded signature Sigma, the instance of ArraysEx with that
  signature is the theory consisting of all Sigma-models that satisfy all
  axioms of the form below, for all sorts s1, s2 in Sigma:
  - (forall ((a (Array s1 s2)) (i s1) (e s2))
      (= (select (store a i e) i) e))
  - (forall ((a (Array s1 s2)) (i s1) (j s1) (e s2))
      (implies (distinct i j) (= (select (store a i e) j) (select a j))))
  - (forall ((a (Array s1 s2)) (b (Array s1 s2)))
      (implies
        (forall ((i s1)) (= (select a i) (select b i))) (= a b)))
  "
)
```

Figure 1: Examples of theory declaration schemas.

Theory attributes can be *formal* or *informal* depending on whether or not their value has a formal semantics. The value of an informal attribute is free text, in the form of a string value. For instance, the `:funs` and `:sorts` attributes (specifying the function and sort symbols of the theory respectively) are formal in the sense above, while `:funs-description` and `:sorts-description` are informal alternatives for theories that would otherwise require an infinite number of formal declarations.

The `:definition` attribute is meant to contain a natural language definition of the theory. While this definition is expected to be as rigorous as possible, it does not have to be a formal one. The `:values` attribute is used to identify for each sort σ in a certain class of sorts, a particular set of ground terms of sort σ that are to be considered as *values for* σ. Intuitively, given an instance theory containing a sort σ, a set of values for σ is a set of terms (of sort σ) that denotes, in each model of the theory, all the elements of that sort. The `:notes` attribute is meant to contain documentation information on the theory declaration such as authors, date, version, references, etc., although this information can also be provided with more specific, user-defined attributes.

A theory declaration (`theory` $T\ \alpha_1\ \cdots\ \alpha_n$) specifies a *theory schema* with name T and attributes $\alpha_1, \ldots, \alpha_n$. Each instance of the schema is a theory $\mathcal{T}_\Sigma$ with an *expanded* signature Σ, containing (zero or more) additional sort and function symbols with respect to those in the declaration. Figure 1 contains two examples of theory declaration. The first, `Core`, specifies a special basic theory which defines the sort `Bool` and the Boolean connectives, and is implicitly included in every SMT-LIB theory. Note the use of `par` to declare a family of equality and of if-then-else operators. The second declaration, `ArrayEx`, specifies a parametric theory of extensional arrays.

The simplest way to obtain an instance of a theory schema is to provide a possibly empty set Q of sort symbols (this set is specified in a separate sublogic declaration described in the next section). The instance theory then contains the set S of all (parameter-free) sort terms over $Q \cup R$, where R is the set of sort symbols declared in the schema itself. Note that S is non-empty because R always contains the 0-arity sort symbol `Bool`. For example, instantiating the theory schema `ArrayEx` above with $Q = \{\texttt{Int}\}$, we get a theory $Array_Q$ with a set S of sorts defined inductively as follows: (i) $\texttt{Bool}, \texttt{Int} \in S$; (ii) for all $\sigma_1, \sigma_2 \in S$, (`Array` $\sigma_1\ \sigma_2$) $\in S$.[4] That theory contains infinitely-many `select` symbols, each with *rank* of the form (`(Array` $\sigma_1\ \sigma_2$`)` $\sigma_1\ \sigma_2$) for some $\sigma_1, \sigma_2 \in S$ (similarly for `store`).

Schematic theory declarations are a meta-level approximation of true parametric types and polymorphic functions. Their main advantage versus extending the logic with type variables and true parametric types is that

[4]Note that logic declarations, described in the next section, can be used to restrict this set of sorts; for example, to exclude a sort like `Array Bool Bool`, if desired.

```
(logic QF_IntIntArrays
 :theories (ArraysEx Ints)
 :language
 "Quantifier-free formulas possibly with free constant symbols but
  with terms exclusively of sort Bool, Int, or (Array Int Int).
  All terms of sort Int are linear."
)
```

Figure 2: Example logic declaration.

the semantics of the logic and, as a consequence, its associated inference apparatus, remains essentially the same as in Version 1.2. Syntactically, the only difference is that sorts are now named by ground terms such as `(Array Int Real)` instead of constants such as `IntRealArray`. Another advantage is that we can get, with one instantiation of the schema, a *single* theory with arbitrarily nested sorts. This is crucial in providing a simple mechanism for theory combination (see [3] for more details). The main limitation is that users cannot define (new) polymorphic function symbols in a benchmark because sort parameters can occur only in theory declarations. While this is a true limitation, it can be overcome by defining finitely many monomorphic versions of the polymorphic symbol, as needed.

4 Logic Declarations

The SMT-LIB format allows the explicit definition of sublogics of its main logic that restrict both the main logic's syntax and semantics. A new sublogic, or simply logic, is defined in the SMT-LIB language by a *logic declaration*. Logic declarations have a similar format to theory declarations, although they are not parametric and they have mostly informal attributes.

Attributes with the following predefined keywords have a prescribed usage and semantics in logic declarations: `:theories`, `:language`, `:notes`, `:extensions`, and `:values`.

When the value of the `:theories` attribute is $(T_1 \ \cdots \ T_n)$, with $n > 0$, the logic refers to a combination $\mathcal{T}$ of specific instances $\mathcal{T}_1, \ldots, \mathcal{T}_n$ of the theory declarations $T_1, \ldots, T_n$.[5] The effect of this attribute is to declare that the logic's sort and function symbols consist of those of the combined theory $\mathcal{T}$, and that the logic's semantics is restricted to the models of $\mathcal{T}$.

The `:language` attribute describes in free text the logic's *language*, a specific class of SMT-LIB formulas. This information is useful for tailoring SMT solvers to the specific sublanguage of formulas used in an input script. The formulas in the logic's language are built over (a subset of) the signature of the associated theory $\mathcal{T}$, as specified in this attribute.

[5] The combination operator that formally yields $\mathcal{T}$ from $\mathcal{T}_1, \ldots, \mathcal{T}_n$ is described in detail in the Version 2.0 reference document [3].

⟨*command*⟩ ::= `(` `set-logic` ⟨*symbol*⟩ `)`
| `(` `set-option` ⟨*option*⟩ `)` | `(` `set-info` ⟨*attribute*⟩ `)`
| `(` `declare-sort` ⟨*symbol*⟩ ⟨*numeral*⟩ `)`
| `(` `define-sort` ⟨*symbol*⟩ `(` ⟨*symbol*⟩* `)` ⟨*sort*⟩ `)`
| `(` `declare-fun` ⟨*symbol*⟩ `(` ⟨*sort*⟩* `)` ⟨*sort*⟩ `)`
| `(` `define-fun` ⟨*symbol*⟩ `(` ⟨*sorted_var*⟩* `)` ⟨*sort*⟩ ⟨*term*⟩ `)`
| `(` `push` ⟨*numeral*⟩ `)` | `(` `pop` ⟨*numeral*⟩ `)`
| `(` `assert` ⟨*term*⟩ `)` | `(` `check-sat` `)`
| `(` `get-assertions` `)` | `(` `get-proof` `)`
| `(` `get-unsat-core` `)` | `(` `get-assignment` `)`
| `(` `get-option` ⟨*keyword*⟩ `)` | `(` `get-info` ⟨*info_flag*⟩ `)`
| `(` `get-value` `(` ⟨*term*⟩$^+$ `)` `)` | `(` `exit` `)`

⟨*script*⟩ ::= ⟨*command*⟩*

Figure 3: Syntax of commands

The optional `:extensions` attribute is meant to document any notational conventions, or syntactic sugar, allowed in the concrete syntax of formulas in this logic. The `:values` attribute has the same use as in theory declarations but it refers to the specific theories and sorts of the logic. It is meant to complement the `:values` attribute specified in the theory declarations referred to in the `:theories` attribute. The textual `:notes` attribute serves the same purpose as in theory declarations. An small example of a logic declaration is shown in Figure 2.

5 Scripts

Most SMT solvers, in addition to reading the SMT-LIB input format, provide an API for using the solver as a library or interactively. A major new feature of Version 2.0 is a standard command language for such APIs. The command language consists of a set of standard *S-expression* commands that can be issued by users, together with a set of standard reponses that can come from the solver. The syntax is given in Figure 3. Commands include (i) *assertion-set commands*: `declare-sort` and `define-sort`, `declare-fun` and `define-fun`, `assert`, `check-sat`, `push`, `pop`, `get-assertions`; (ii) *post-check commands*: `get-value`, `get-assignment`, `get-proof`, `get-unsat-core`; and (iii) *option and diagnostic commands*: `set-logic`, `set-option`, `set-info`, `get-option`, `get-info`. Benchmarks in Version 2.0 are just scripts in this command language.

The assertion-set commands operate on a data structure called the *assertion-set stack*. This is a single global stack, consisting of *assertion sets*, which are sets of assertions, definitions, and declarations. The union of all the assertion sets is called the *set of all assertions*. The command `get-assertions` returns all the assertions (but not the definitions and declarations) in this

set. The `check-sat` command checks the satisfiability of the set of all assertions. The command `push` n pushes n empty assertion sets onto the top of the assertion-set stack, and `pop` n pops the top n assertion sets off the stack (or gives an error if the size of the stack is less than n).[6] The *current assertion set* is the assertion set (if any) currently on the top of the stack.

The `define`/`declare` commands add a definition or declaration, respectively, to the current assertion set. They allow a user to declare or define new sort and function symbols, and can also be used to create aliases for larger sorts and terms. This provides a global mechanism for expression sharing that was missing in Version 1.2. The `assert` command adds a formula to the current assertion set.

The post-check commands can be used immediately following a call to `check-sat` to obtain more information from the solver. Calls to `check-sat` return one of `unsat`, `sat`, or `unknown`. If `unsat` is returned, the optional commands `get-unsat-core` and `get-proof` are enabled. The first returns an unsatisfiable subset of the set of assertions checked—there is no minimality requirement, but the expectation is that solvers will attempt to approximate it. The second returns a proof that the formula is unsatisfiable in some solver-specific format (there is not yet a standard format for proofs). If a check returns `sat` or `unknown`,[7] then the other commands are enabled.

Many applications using SMT solvers need concrete values for terms occurring in the problem, for example to produce a trace showing a violation of a property during program verification. The optional `get-value` command requires the solver to produce internally a model for the set of checked assertions and return values from the model's domain for one or more ground terms given as arguments. The values returned by the solver should be from the set described in a `:values` attribute of either the current logic being used or one of the theories referenced by that logic. Finally, option and diagnostic commands are also available.

Figure 4 shows a sample interaction using the command language. A short example of a Version 1.2 SMT-LIB benchmark converted into a Version 2.0 script in provided in the appendix.

6 Conclusion and Further Work

Version 2.0 of the SMT-LIB standard was developed with the input of many SMT researchers to sustain and expand the role of the standard in facilitating research and development in SMT. The new version increases the ex-

[6] The n here is for convenience; a more minimalistic formulation would omit n, and simply push or pop a single assertion-set.

[7] An `unknown` response is typically given when a solver has computed a satisfying assignment but does not know whether this assignment is consistent due to some incompleteness in its method. The standard encourages solver implementers to make the same information available after an `unknown` response as after a `sat` response.

```
> (set-logic QF_LIA)
success
> (declare-fun x () Int)
success
> (declare-fun y () Int)
success
> (assert (or (> x y) (> y x))
success
> (set-option :print-success false)
> (push 1)
> (assert (= x y))
> (check-sat)
unsat
> (pop 1)
> (check-sat)
sat
> (get-value ((> x y) x y))
(((> x y) true)
 (x 1)
 (y 0)
)
> (exit)
```

Figure 4: Sample interaction using command language. The meta-symbol > indicates inputs.

pressiveness of SMT-LIB theory declarations to include parametric theories and theory combinations. It provides a simplified syntax, and a new command language for more dynamic and sophisticated interaction—whether by a human or another tool—with SMT solvers. We believe that with the continued support of the SMT community, this standard will help the field of SMT reach the next level of utility and power for its growing range of applications. A number of useful features and commands that were discussed by the work groups did not make it in this version because they needed further discussion or greater consensus. We plan to continue working on them with the aim of introducing them in Version 2.1.

References

[1] C. Barrett, M. Deters, A. Oliveras, and A. Stump. Design and results of the 3^{rd} annual satisfiability modulo theories competition (SMT-COMP 2007). *International Journal on Artificial Intelligence Tools (IJAIT)*, 17(4):569–606, Aug. 2008.

[2] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009.

[3] C. Barrett, A. Stump, and C. Tinelli. *The SMT-LIB Standard Version 2.0 Reference Manual*, Jan. 2010. (Available at `www.SMT-LIB.org`).

[4] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[5] H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In *SAT2000: Highlights of Satisfiability Research in the year 2000*, pages 283–292. Kluwer Academic, 2000.

[6] G. L. Steele. *Common Lisp the Language*. Digital Press, 1990.

[7] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

A Sample Benchmark in New and Old Format

```
(set-logic QF_LIA)
(set-info :source | Mathsat benchmarks available from http://mathsat.itc.it |)
(set-info :smt-lib-version 2.0)
(set-info :category "industrial")
(set-info :status sat)
(declare-fun arg1 () Int)
(declare-fun ARG2_LSBRCK_0_RSBRCK_ () Bool)
...
(assert
(let ((?v_0 (not ARG2_LSBRCK_0_RSBRCK_)) (?v_1 (not ARG2_LSBRCK_1_RSBRCK_)))
 (and (and
 (= (- (- arg2 arg2_LSBRCK_0_RSBRCK_) (* 2 arg2_LSBRCK_1_RSBRCK_)) 0))
 (>= arg2_LSBRCK_0_RSBRCK_ 0))
... ))
(check-sat)
(exit)
```

Figure 5: Part of `CIRC/MULTIPLIER_PRIME_2.msat.smt2`, Version 2.0.

```
(benchmark MULTIPLIER_PRIME_2.msat.smt
  :source { Mathsat benchmarks available from http://mathsat.itc.it }
  :status sat
:category { industrial }
:difficulty { 0 }
  :logic QF_LIA
  :extrafuns ((arg1 Int))
  :extrapreds ((ARG2_LSBRCK_0_RSBRCK_))
  ...
  :formula
(flet ($cvcl_0 (not ARG2_LSBRCK_0_RSBRCK_))
(flet ($cvcl_1 (not ARG2_LSBRCK_1_RSBRCK_))
 (and (and
 (= (- (- arg2 arg2_LSBRCK_0_RSBRCK_) (* 2 arg2_LSBRCK_1_RSBRCK_)) 0))
 (>= arg2_LSBRCK_0_RSBRCK_ 0))
 ...))
)
```

Figure 6: Part of `CIRC/MULTIPLIER_PRIME_2.msat.smt`, Version 1.2.