The SMT-LIB Format: An Initial Proposal

Silvio Ranise[1] and Cesare Tinelli[2]

[1] LORIA & INRIA-Lorraine
Nancy, France
ranise@loria.fr
[2] The University of Iowa
Iowa City, IA, USA
tinelli@cs.uiowa.edu

1 Introduction

This paper is a first proposal for a common format for the Satisfiability Modulo Theories Library, or SMT-LIB for short. The main goal of the SMT-LIB initiative [2], coordinated by these authors and supported by a growing number of researchers world-wide, is to produce a on-line library of benchmarks for *satisfiability modulo theories*. By benchmark we mean a logical formula to be checked for satisfiability modulo (combinations of) background theories of interest. Examples of background theories typically used in computer science are real and integer arithmetic and the theories of various data structures such as lists, arrays, bit vectors and so on.

A lot of work has been done in the last few years by several research groups on building systems for satisfiability modulo theories. We believe that having a library of benchmarks will 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 [3] has done for theorem proving, or the SATLIB library [1] has done for propositional satisfiability.

2 The Satisfiability Modulo Theories Library

We envision a library consisting of two main sections: one containing the specification of several background theories, and another containing benchmark sets, grouped under a number of indexes such as their corresponding background theory, the class of formulas they belong to, the type of problem they originate from and so on.

For the library to be viable and useful it should adopt a common standard for expressing the benchmarks, and for defining the background theories in a rigorous way—so that there is no doubt on which theories are intended. In this respect, some natural questions arise.

1. Is it sufficient to use a first-order language for the benchmarks?
   (After all, most of the research in decision procedure has been done in a first-order setting.)

2. If so, should the library be limited to ground (i.e. quantifier-free) satisfiability problems? Or should it also consider problems with quantifiers?
3. Should the library adopt a sorted or an unsorted input language?
4. How should the background theories and their various combinations be defined and specified?
5. Which concrete syntax should we use for the benchmarks?

In the rest of the paper we discuss some possible answers to these questions and put forward an initial proposal. The proposal described here has been prepared by unifying a number of ideas contributed by the members of the SMT-LIB interest group.

3 Basic Assumptions and Proposals

We start by fixing the terminology and making some basic assumptions.

We assume that input problems, i.e. logical formulas, are to be checked for satisfiability, not validity.[3] In particular, given a theory T and a formula φ, we are interested in whether φ is *satisfiable in* T, or is *satisfiable modulo* T, that is, whether there is a model of φ that satisfies (the existential closure of) φ.

Informally speaking, let us call a *satisfiability procedure* any procedure for satisfiability modulo some given theory. With satisfiability procedures one can distinguish among

1. the procedure's *underlying logic* (first-order, many-sorted, modal, intuitionistic, higher order, etc.),
2. the procedure's *background theory*, the theory against which satisfiability is checked (typically a set of closed formulas in the underlying logic's syntax or a set of models in the logic's semantics), and
3. the procedure's *input language*, that is, the class of formulas the procedure accepts as input (ground, CNF, first-order, temporal, etc.).

For instance, the underlying logic of a typical solver for linear arithmetic is first-order logic with equality, the background theory is the theory of real numbers, and the input language is the class of conjunctions of linear equations and inequations.

We believe that to ease organization and classification it would be helpful for SMT-LIB to follow this distinction as well. This entails that

> *SMT-LIB should (i) adopt at least one underlying logic, (ii) define a number of background theories, and (iii) specify a general syntax for the various benchmarks, providing a way to indicate to which class of formulas a benchmark belongs.*

We briefly discuss these three points in the following subsections, together with our proposal for each.

[3] The difference matters only for those classes of problems that are not closed under logical negation.

3.1 The Logic

There is unanimous agreement within the SMT-LIB interest group that SMT-LIB should concentrate on a single underlying logic for its benchmarks and that this logic should be first-order logic with equality ($FOL^{=}$) or some variations of it. There is less agreement on whether it should be classical unsorted logic or a multi-sorted version.

An argument against supporting a multi-sorted version of $FOL^{=}$ is that it is a more complicated framework than classical first-order logic, and that almost all theoretical results in satisfiability modulo theories (e.g. on combining satisfiability procedures) are given in the context of unsorted $FOL^{=}$ only.

The issue of sorts, however, is very important in this field, because a large number of background theories and input problems are more naturally formulated in a sorted framework. As a matter of fact, some of the existing solvers actually have a sorted input language. Therefore, we feel that SMT-LIB should provide some kind of support for sorts.

In an attempt to balance the pros and cons of a multi-sorted framework, we propose here a compromise solution that seeks to combine the simplicity and familiarity of unsorted logic with the convenience of a sorted language.

> *We propose for SMT-LIB to adopt unsorted $FOL^{=}$ as the sole underlying logic, but to allow benchmarks and theories to be specified in a many-sorted language whose semantics is provided by a translation into $FOL^{=}$.*

The proposed language and its translation into $FOL^{=}$ are discussed in the next sections.

3.2 The Background Theories

One of the goals of the SMT-LIB initiative is to clearly define a catalog of background theories, starting with a small number of popular ones, and adding new ones as solvers for them are developed. Theories will be specified in SMT-LIB independently of any benchmarks or solvers. Each set of benchmarks then will contain a reference to its own background theory.

We distinguish between *basic* (or *component*) theories and *combined* theories. By combined theory we mean a theory that is defined as some kind of combination of basic theories. Example of basic theories include the theory of real numbers, the theory of arrays, the theory of lists and so on.

> *We propose that basic theories be specified informally albeit as rigorously as possible. We do not prescribe any specific way to specify a theory.*

The issue of specifying combined theories modularly in terms of their component theories is a bit tricky, in particular in the presence of sorts. We leave its discussion to a later version of this paper. Although this is not ideal, for the time being we can specify a combined theory in SMT as if it was a basic one, that is, disregarding the fact that it is the combination of other theories.

3.3 The Input Language

We propose to adopt a general first-order (sorted) language in which to write all SMT-LIB benchmarks.

We realize, however, that many benchmarks are typically expressed in a some fragment of the language of first-order logic. The particular fragment of $FOL^{=}$ considered does matter because one can often write a solver specialized on that fragment that is a lot more efficient than a solver meant for a larger fragment.[4]

An extreme case of this situation occurs when satisfiability modulo a given theory T is decidable for a certain fragment (quantifier-free, say) but undecidable for a larger one (full first-order, say), as for instance happens with the theory of arrays specified in Section 7.2. But it is also true when the decidability of the satisfiability problem is preserved across various fragments. For instance, if T is the theory of real numbers, the satisfiability in T of full-first order formulas is decidable. However, one can build increasingly faster solvers by restricting the language respectively to quantifier-free formula, linear equations and inequations, difference equations, inequations between variables, and so on. As a consequence,

it is useful for a benchmark to specify which specific fragment of the general first-order language it belongs to. The attribute-based format we propose for writing benchmark sets provides an attribute to specify just that.

4 The SMT-LIB Logic and Language

We argued earlier for adopting classical (unsorted) first-order logic with equality as the sole underlying logic for SMT-LIB.

For specifying benchmarks and possibly theory axioms, however, we propose to adopt a many-sorted language with subsorts. As a typed language, the proposed language is intentionally limited in expressive power. In essence,

the language allows one only to declare sorts (types) only by means of sort symbols, to define a preordering on the sorts, to specify the interface of function and predicate symbols in terms of the declared sorts, and to specify the sort of quantified variables.

In type theory terms, the language has no type constructors, no type quantifiers, no provisions for parametric polymorphism, and so on. The only form of polymorphism it allows is subsort polymorphism, akin to subtype polymorphism in object-oriented languages. Also, explicit (ad-hoc) overloading of function or predicate symbols—by which a symbol could be explicitly given more than one interface—is not allowed. The idea is to provide, at least at the beginning of this

[4] By efficiency here we do not necessary refer to worst-case time complexity, but to efficiency "in practice".

project, just enough expressive power to represent typical benchmarks without getting bogged down in the complexity (and higher-orderness) of type theory.

An abstract syntax for expressing formulas in the proposed language is provided in the following. In order for us to provide already actual examples of benchmarks at this stage, the abstract syntax is actually not as abstract as it could be. We wrote it so that an actual, concrete syntax is immediately derivable from it.

The proposed syntax is attribute-based and Lisp-like. In designing it, we followed the recommendation of several members of the SMT-LIB interest group that the benchmarks be easily parsable. Preferring ease of parsing over human readability is reasonable in this context because we expect not only that benchmarks will be typically read by solvers but also that, by and large, they will be produced in the first place by automated tools like verification condition generators or translators from other formats.

> *The syntax of formulas in the SMT-LIB language extends the standard abstract syntax of FOL$^{=}$ with the following additional constructs:*
> - *a construct for declaring the sort of quantified variables,*
> - *an* if-then-else-*like logical connective,*
> - *a* let *construct for terms,*
> - *a* let *construct for formulas, and*
> - *a* distinct *construct for declaring a number of values as pairwise distinct.*

Except for the first extension, dictated by our goal of supporting sorts, the other extensions are provided for greater convenience. We discuss each of them in turn.

The *if-then-else* construct This construct is very common in benchmarks coming for instance from hardware verification. Although it can be defined in terms of more basic constructs such as conjunctions and implications, it provides important structural information that a solver can use to speed up its computation. Since such information is lost if one imposes a previous translation into the more basic constructs, it seems important to support an *if-then-else* construct natively in the SMT-LIB language.

We point out that conceptually there are two kinds of *if-then-else*-like constructs: a logical connective of the form

$$\textbf{if } \varphi \textbf{ then } \varphi_1 \textbf{ else } \varphi_2$$

where $\varphi, \varphi_1, \varphi_2$ are formulas, and function symbol of the form

$$\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2$$

where t, t_1, t_2 are terms—with t being typically, but not necessarily, a Boolean term. We propose to support only the first type, which has a straightforward general semantics and provides already enough flexibility. The second type is

problematic because its semantics is domain dependent as it is based on the type of t.[5] The complications introduced by the second *if-then-else* construct can be easily avoided because its effect can be achieved (with no substantial loss of structural information) by means of the first *if-then-else* construct and fresh constants. In fact, every atomic formula of the form $\varphi(\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2)$, that is, containing $\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2$ as a subterm, can be equivalently rewritten as

$$(\textbf{if } t = v \textbf{ then } c = t_1 \textbf{ else } c = t_2) \wedge \varphi(c)$$

where c is a fresh constant and v is the value t must evaluate to for $\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2$ to evaluate to t_1's value.

The *let* construct for terms This is a construct of the form

$$\textbf{let } x = t \textbf{ in } \varphi$$

where x is a variable, t is a term and φ is a formula. This construct is convenient for benchmark compactness as it allows one to replace multiple occurrences of the same term by a variable. It is of course also useful for a solver because it saves the solver the effort to recognize the various occurrences of the same term as such.

We propose to include this kind of construct, but with the restriction that t be a ground term, that is, a term with no variables. This restriction is perhaps overly strong. With it, however, the interpretation of $\textbf{let } x = t \textbf{ in } \varphi$ is straightforward and unproblematic: it stands for the formula obtained from φ by replacing each unbound occurrence of x in φ by t.[6] With t ground we do not have to worry about the problem of "variable capturing", which occurs when a variable that is originally free in t becomes bound after the substitution of t for x in φ.

The *let* construct for formulas This is a construct of the form

$$\textbf{let } p \equiv \varphi \textbf{ in } \psi$$

where p is a propositional variable, φ is a formula with no free variables, and ψ is any formula. The rationale for this construct and the restriction on φ to be a closed formula is entirely similar to that for the previous *let* construct.

To simplify parsing, the proposed language will distinguish syntactically between the two *let* constructs.

[5] In the literature, the second *if-then-else* evaluates to the value of t_1, as opposed to the value of t_2 if t evaluates to some value v which, depending on problem domain, is chosen to be *true*, 0, a positive number, an empty list, a non-empty one, and so on.

[6] By unbound we mean not in the scope of another *let* or of a quantifier in φ.

The *distinct* construct This construct is also added for conciseness. It has the form

$$distinct(t_1, \ldots, t_n)$$

for a variable $n \geq 2$, where $t_1, \ldots, t_n$ are terms, and it stands for all pairwise disequations between the t_i's.

4.1 The language of formulas

The proposed language of formulas is the *well-sorted* subset of the language generated by the non-terminal symbol *let_formula* in the grammar below. Derivation rules defining the well-sorted formulas and terms of the language follow later.

The grammar is given as a set of BNF-style production rules. In these rules, we use boldface text to denote terminal symbols, the $(\_)^*$ operator for denoting zero or more repetitions of the operand, the $(\_)^+$ operator for one or more repetitions, and the $[\_]$ operator for zero or one repetitions.

let_formula ::= **(** :**let** *var term let_formula* **)**
| **(** :**flet** *pred_sym formula let_formula* **)**
| *formula*

formula ::= **(** *quant_symb* **(** *var sort_symb* **)** *formula* **)**
| **(** *connective formula** **)**
| *atomic_formula*

quant_symb ::= :**exists** | :**forall**
var ::= **?***identifier*
sort_symb ::= *identifier*
connective ::= :**not** | :**impl** | :**iff** | :**ite** | :**and** | :**or** | :**xor**

atomic_formula ::= :**true** | :**false** | *pred_symb* | **(** *pred_symb term*$^+$ **)**
| **(** = *term term*$^+$ **)** | **(** :**distinct** *term term*$^+$ **)**

term ::= *var* | *numeral* | *fun_symb* | **(** *fun_symb term*$^+$ **)**

pred_symb ::= *identifier* | *operator*
fun_symb ::= *identifier* | *operator*

numeral ::= a sequence of digits
identifier ::= a sequence of letters, digits and underscores (_),
starting with a letter
operator ::= a "math operator" symbol such as $+, *, <, <=, \&$, etc.

Here follow some salient features of the grammar that we think deserve discussion.

The *let* declarations generated by *let_formula* are in a sense global: although they can be nested, they must all come before the "main formula". In other words, *let* expressions are not allowed to appear below quantifiers or logical connectives. Consistently with this restriction, in expressions of the form **(**:**flet** $p\ \varphi\ \psi$**)**

the formula φ cannot itself contain a *let* expression. These two restrictions as well are dictated by simplicity concerns. However, they are only tentative at the moment, and might be relaxed in the future if needed.

In this grammar we do not distinguish between constant and function symbols (they are all defined as *fun_symb*), and between propositional variables and predicate symbols (they are all defined as *pred_symb*). These distinctions are really a matter of arity, which is taken care of later by the well-sortedness rules. A similar observation applies to the logical connectives (the members of *connective*) and the number of arguments they are allowed take.

Finally, we do not enforce here any disjointness constraints between the sets of sort, function, and predicate symbols. Such constraints will be enforced later at the level of specific theories and benchmark sets.

In the rest of the paper, we will often treat a non-terminal symbols of the grammar above as the set of expressions it generates.

4.2 Well-sorted Formulas

The SMT-LIB language of formula is the largest set of well-sorted formulas contained in the language generated by the previous grammar. Well-sorted formulas are defined below by means of a set of sorting rules, similar in format and spirit to the kind of typing rules found in the programming languages literature.

We first provide a set of subsorting rules that define the subsort relation as a preorder on sort symbols. Then we provide a set of rules defining well-sorted terms, and another set defining well-sorted formulas.

To specify these rules we need to presuppose the existence of a sorted signature Σ, defined formally below. Strictly speaking then, the SMT-LIB language is a family of languages parametrized by Σ. As we will see later, for each benchmark ψ and theory T, the specific signature is going to be jointly defined by the specification of of T and that of the benchmark set containing ψ.

Definition 4.1. An *order-sorted signature* Σ is a tuple consisting of:

- a non-empty subset Σ^{S} of *sort_symb*, a subset Σ^{F} of *fun_symb*, a subset Σ^{P} of *pred_symb*, all pairwise disjoint,
- a binary relation $\Sigma^{<:}$ over Σ^{S},
- a mapping of the elements of *var* to elements of Σ^{S},
- a mapping of the elements of Σ^{F} to a non-empty sequence of elements of Σ^{S}, and
- a mapping of the elements of Σ^{P} to a possibly empty sequence of elements of Σ^{S}.

Note that the absence of overloading of function or predicate symbols in the SMT-LIB language is enforced by the last two mappings in the definition of signature.

To simplify the notation in this section we use the following meta-variables, possibly with subscripts: S, U and T ranging over *sort_symb*, x ranging over *var*, a and f ranging over *fun_symb*, p ranging over *pred_symb*, t ranging over *term*,

k ranging over *connective*, φ ranging over *formula*, ψ ranging over *let_formula*, and n ranging over the non-negative integers.

We also use a distinguished sort symbol, denoted by F and assumed not to be in *sort_symb*, as the sort of well-sorted formulas.

In the following we will fix a signature Σ. Then, we will write $x : S \in \Sigma$ to mean that Σ maps the variable x to the sort S. We will write $f : S_1 \cdots S_{n+1} \in \Sigma$ to mean that $f \in \Sigma^{\mathrm{F}}$ and Σ maps f to the sort sequence $S_1 \cdots S_{n+1}$ (and similarly for predicate symbols). We will write $\Sigma, x : S$ to denote the signature that coincides with Σ except (possibly) that it maps x to S. Similarly, we will write $\Sigma, p : \epsilon$ to denote the signature that coincides with Σ except (possibly) that it contains the predicate symbol p and maps p to the empty string of sorts (in other words, declares p as a propositional variable).

Subsorting rules

$$\frac{S \in \Sigma^{\mathrm{S}}}{\Sigma \vdash S <: S} \qquad \frac{(S, T) \in \Sigma^{<:}}{\Sigma \vdash S <: T} \qquad \frac{\Sigma \vdash S <: U \qquad \Sigma \vdash U <: T}{\Sigma \vdash S <: T}$$

We say that a sort S is a *subsort* of a sort T (according to Σ) if $\Sigma \vdash S <: T$ is derivable by the rules above.

Note that the subsorting relation $<:$ induced by Σ coincides with the reflexive-transitive closure of $\Sigma^{<:}$.

Well-sortedness rules for terms

$$\frac{x : S \in \Sigma}{\Sigma \vdash x : S} \qquad \frac{f : S_1 \cdots S_{n+1} \in \Sigma}{\Sigma \vdash f : S_1 \cdots S_{n+1}} \qquad \frac{\Sigma \vdash t : S \quad \Sigma \vdash S <: T}{\Sigma \vdash t : T}$$

$$\frac{\Sigma \vdash f : S_1 \cdots S_{n+1} \quad \Sigma \vdash t_1 : S_1 \quad \cdots \quad \Sigma \vdash t_n : S_n}{\Sigma \vdash (f\ t_1\ \cdots\ t_n) : S_{n+1}}$$

We call an element t of *term well-sorted* (with respect to Σ) if $\Sigma \vdash t : S$ is derivable by the sort rules above for some sort S. In that case, we also say that t *is of sort* S.

The rules are pretty standard and self-explanatory. The only interesting rule is perhaps the third one which basically says that any term of sort S is also of sort T for any supersort T of S. Subsort polymorphism is allowed in the language be virtue of this rule.

Well-sortedness rules for formulas

$$\frac{}{\Sigma \vdash :\mathbf{true} : \mathsf{F}} \qquad \frac{}{\Sigma \vdash :\mathbf{false} : \mathsf{F}} \qquad \frac{p : S_1 \cdots S_n \in \Sigma}{\Sigma \vdash p : S_1 \cdots S_n\ \mathsf{F}}$$

$$\frac{\Sigma \vdash t_1 : S_1 \quad \cdots \quad \Sigma \vdash t_{n+2} : S_{n+2}}{\Sigma \vdash (=\ t_1\ \cdots\ t_{n+2}) : \mathsf{F}} \qquad \frac{\Sigma \vdash t_1 : S_1 \quad \cdots \quad \Sigma \vdash t_{n+2} : S_{n+2}}{\Sigma \vdash (:\mathbf{distinct}\ t_1\ \cdots\ t_{n+2}) : \mathsf{F}}$$

$$\frac{\Sigma \vdash p : S_1 \cdots S_n \ \mathsf{F} \quad \Sigma \vdash t_1 : S_1 \quad \cdots \quad \Sigma \vdash t_n : S_n}{\Sigma \vdash (p\ t_1\ \cdots\ t_n) : \mathsf{F}}$$

$$\frac{\Sigma \vdash \varphi : \mathsf{F}}{\Sigma \vdash (\mathbf{:not}\ \varphi) : \mathsf{F}} \quad \frac{\Sigma \vdash \varphi_1 : \mathsf{F} \quad \Sigma \vdash \varphi_2 : \mathsf{F}}{\Sigma \vdash (\mathbf{:impl}\ \varphi_1\ \varphi_2) : \mathsf{F}} \quad \frac{\Sigma \vdash \varphi_1 : \mathsf{F} \quad \Sigma \vdash \varphi_2 : \mathsf{F}}{\Sigma \vdash (\mathbf{:iff}\ \varphi_1\ \varphi_2) : \mathsf{F}}$$

$$\frac{\Sigma \vdash \varphi_1 : \mathsf{F} \quad \Sigma \vdash \varphi_2 : \mathsf{F} \quad \Sigma \vdash \varphi_3 : \mathsf{F}}{\Sigma \vdash (\mathbf{:ite}\ \varphi_1\ \varphi_2\ \varphi_3) : \mathsf{F}} \quad \frac{\Sigma \vdash \varphi_1 : \mathsf{F} \quad \cdots \quad \Sigma \vdash \varphi_n : \mathsf{F}}{\Sigma \vdash (k\ \varphi_1\ \cdots\ \varphi_n) : \mathsf{F}}$$

$$\frac{\Sigma,\ x : S \vdash \varphi : \mathsf{F}}{\Sigma \vdash (\mathbf{:exists}\ (x\ S)\ \varphi) : \mathsf{F}} \quad \frac{\Sigma,\ x : S \vdash \varphi : \mathsf{F}}{\Sigma \vdash (\mathbf{:forall}\ (x\ S)\ \varphi) : \mathsf{F}}$$

$$\frac{\Sigma \vdash t : S \quad \Sigma, x : S \vdash \psi : \mathsf{F}}{\Sigma \vdash (\mathbf{:let}\ x\ t\ \psi) : \mathsf{F}} \text{ if } t \text{ is ground}$$

$$\frac{\Sigma \vdash \varphi : \mathsf{F} \quad \Sigma, p : \epsilon \vdash \psi : \mathsf{F}}{\Sigma \vdash (\mathbf{:flet}\ p\ \varphi\ \psi) : \mathsf{F}} \text{ if } \varphi \text{ is closed and } p \notin \Sigma^{\mathrm{P}}$$

By *ground* and *closed* in the last two rules we mean containing no occurrences of elements of *var* and no free occurrences of elements of *var*, respectively, according to the standard meaning of free occurrence in $FOL^{=}$.

To reduce the level of nesting of formulas the *and*, *or*, and exclusive *or* connectives are defined, thanks to their associativity, as varyadic connectives taking zero or more arguments. The semantics of the connectives applied to zero arguments is the expected one: **(:and)** is equivalent to :**true**, while **(:or)** and **(:xor)** are both equivalent to :**false**.

We say that an element ψ of *let_formula* is *well-sorted* (with respect to Σ) if $\Sigma \vdash \psi : \mathsf{F}$ is derivable by the rules above.

Definition 4.2. The SMT-LIB language for formulas is the set of all closed well-formed formulas generated by *let_formula*.

Note that we consider closed formulas only. This is mostly a technicality, motivated by considerations of convenience. In fact, with a closed formula ψ of a signature Σ the particular mapping of variables to sorts defined by Σ is irrelevant. The reason is that the formula itself contains its own sort declaration for its variables, either explicitly, for the variables bound by a quantifier, or implicitly, for the variables bound by a *let*. Using only closed formulas then simplifies the task of specifying their signature, as it becomes unnecessary to specify how the signature maps the elements of *var* to the signature's sorts.

There is no loss of generality in this approach because, since we are interested in formula satisfiability, every formula $\varphi(\boldsymbol{x})$ with free variables $\boldsymbol{x}$ of sort $\boldsymbol{S}$ can be rewritten as $\exists \boldsymbol{x} : \boldsymbol{S}.\ \varphi(\boldsymbol{x})$. An alternative way to avoid free variables in benchmarks is proposed in Section 5.2

5 The SMT-LIB Format

In this section we propose a format for specifying theories and benchmark sets. As with formulas, this too is an attribute-value-based format. The main difference with formulas is that some of the attributes do not have a formally specified value—they just contain free text.

Ideally, a formal specification of these free-text attributes would be preferable to free text in order to avoid ambiguities and misinterpretation.

> *The choice of using free text for these attributes is motivated by practicality reasons: (i) these attributes are meant to be read by human readers, not programs and (ii) the amount of effort needed to devise a formal language for these attributes first and to specify their values in this language later does not seem justified by the current goals of SMT-LIB.*

As mentioned in the introduction, we propose that background theories and benchmark sets be specified separately in SMT-LIB. We described the proposed format for each in the following using the same kind of grammar used in Section 4.1 and also some of the non-terminal symbols defined there.

5.1 Specifying theories

In this version of the paper we consider the specification of basic theories only. We leave the problem of specifying combined theories to a later version.

> *A theory specification defines both an order-sorted signature for a theory and the theory itself.*

Formally, a theory specification is an element of the language generated by the non-terminal *theory* in the grammar below.

theory ::= (:**name** *string*
:**sorts** (*sort_symb*$^+$)
[:**subsorts** (*subsort_decl*$^+$)]
:**funs** (*fun_symb_decl*$^+$)
[:**preds** (*pred_symb_decl*$^+$)]
:**definition** *string*
[:**extensions** *string*]
)

subsort_decl ::= (*sort_symb sort_symb*)
fun_symb_decl ::= (*fun_symb sort_symb*$^+$)
pred_symb_decl ::= (*pred_symb sort_symb**)

Of all the elements of *theory* we consider only those that satisfy the following constraints:

1. sort symbols do not occur as function or predicate symbol as well, and similarly for function and predicate symbols—the sets of sort symbols names, function symbols, and predicate symbols are pairwise disjoint;

2. the sort symbols occurring in the :**subsorts**, :**funs** or :**preds** attribute are among the symbols listed in the :**sorts** attribute—all sort symbols used must be declared;
3. a function symbol does not occur more than once in :**funs**—no overloading of function symbols;
4. a predicate symbol does not occur more than once in :**preds**—no overloading of predicate symbols;
5. every sort symbol that occurs only on the left in the pairs of the :**subsorts** attribute occurs as the last symbol in a function symbol declaration in :**funs**—no empty sorts.

The signature of a theory is defined by the :**sorts**, :**subsort**, :**funs** and :**preds** attributes in the obvious way.

The :**subsort** and :**preds** attributes are optional because a theory might have a flat sort structure or lack predicate symbols. The :**sorts** attribute, however, is not optional. The way the language is designed, the signature must have at least one sort otherwise is is not possible to specify the interfaces of function and predicate symbols. This is no real limitation because for instance unsorted theories can be always seen as one-sorted.

The :**funs** attribute is also not optional. This is a consequence of constraint 5 above which forces the existence of at least one function symbol for at least one sort. The purpose of this constraint is to guarantee that for each sort S with no subsorts there is at least one well-sorted term of sort S. This is a technicality, well-known in the many-sorted logic literature. Although it is not strictly necessary, it somewhat simplifies the semantics and the proof theory of many-sorted logics without real loss of generality. In our case it is almost invariably satisfied. When it is not, it is enough to add in the :**funs** attribute the declaration of a new constant symbol of the appropriate sort.

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. Some theories (like the theory of real numbers) are well known, and so just a reference to their common name might be enough. For theories that have a small set of axioms (or axiom schemas), it might be convenient to list the actual axioms. For some other theories, a mix a formal notation and informal explanation might be more appropriate.

The optional :**extension** attribute is meant to document any notational conventions used in the listed benchmarks. This is useful because often the syntax of a theory is extended for convenience with syntactic sugar.[7]

5.2 Specifying Benchmarks

We propose to group benchmarks in benchmark sets.

[7] Think for instance of Presburger arithmetic, where a numeral n abbreviates the n-fold application of the successor function to zero, and the predicate $t_1 \leq t_2$ abbreviates the formula $t_1 < t_2 \vee t_1 = t_2$.

The specification of a benchmark set contains, in addition to the benchmarks themselves, a reference to their background theory, a description of the language fragment to which the benchmarks belong, and an optional specification of additional function and predicate symbols.

More formally, a benchmark set specification is an element of the language generated by the non-terminal *benchmark_set* in the grammar below.

benchmark_set ::= (:**name** *string*
:**theory** *string*
:**language** *string*
[:**extra_funs** (*fun_symb_decl*$^+$)]
[:**extra_preds** (*pred_symb_decl*$^+$)]

:**benchmarks** (*benchmark*$^+$)
)

benchmark ::= (:**formula** *let_formula* :**status** *status*)
status ::= :**sat** | :**unsat** | :**unknown**

Of all the elements of *benchmark_set* we consider only those that satisfy the following constraints:

1. the value of the :**theory** attribute coincides with the value of the :**name** attribute of some theory specification *tspec* in SMT-LIB;
2. the sort symbols occurring in (the value of) the :**extra_funs** or the :**extra_preds** attribute are among the symbols listed in the :**sorts** attribute of *tspec*;
3. a function symbol does not occur more than once in :**extra_funs**,
4. a predicate symbol does not occur more than once in :**extra_preds**;
5. a function symbol occurs neither in :**extra_preds** nor in *tspec*;
6. a predicate symbol occurs neither in :**extra_funs** nor in *tspec*;
7. all function (resp. predicate) symbols occurring in the :**benchmarks** attribute are declared either in *tspec* or in the :**extra_funs** (resp. :**extra_preds**) attribute.

The :**name** attribute provides a name for the set, for reference purposes.

The :**theory** attribute simply contains the name of a background theory specified in SMT-LIB.

The :**language** attribute specifies the specific subset of *let_formula* to which the listed benchmarks belong. The attribute is text valued because it has mostly documentation purposes. It is meant to help the benchmark user decide whether his solver can process the benchmark set. A natural language description of the sublanguage seems therefore adequate for this purpose.

The :**extra_funs** attribute complements the :**funs** attribute of the corresponding theory specification by declaring additional function symbols with their interface. The :**extra_preds** attribute has a similar purpose, but for predicate

symbols. In contrast to the symbols possibly defined in the :**extensions** attribute of a theory specification, which are interpreted in terms of the symbols in the theory, the symbols in :**extra_funs** and :**extra_funs** are uninterpreted in associated theory.

Uninterpreted function or predicate symbols are found often in applications of satisfiability modulo theories, typically as a consequence of Skolemization or abstraction transformations applied to more complex formulas. Hence SMT-solvers typically accept formulas containing uninterpreted symbols in addition to the symbols of their background theory. The :**extra_funs** and :**extra_preds** attributes serve to declare any uninterpreted symbols occurring in benchmarks listed in the :**benchmarks** attribute. The value of :**extra_funs** and :**extra_preds** attributes is specified formally because, in effect, it dynamically expand the signature of the associated background theory, hence it is convenient for it to be directly readable by satisfiability procedures for that theory.

The :**extra_funs** attribute is also useful for specifying benchmarks with free variables (such as quantifier-free ones). As mentioned in Section 4.1, our language does not allow benchmarks with free variables. As we saw, one way to circumvent this restriction is to close such formulas existentially. Another one is to replace the free variables by fresh constant symbols of the proper sort. In the second case, these constant symbols are declared in the :**extra_funs** attribute.

The :**benchmarks** attribute lists the actual benchmarks—formally defined as members of *let_formula*—specifying for each of them whether the benchmark is known to be (un)satisfiable in the background theory. Knowing about the satisfiability of a benchmark is useful for debugging new solvers.

At the moment we require that all function or predicates symbols in a benchmark set be declared either in the corresponding theory specification or in the :**extra_funs** and :**extra_preds** attribute. Depending on the feedback we will get from the SMT-LIB interest group, this restriction might be relaxed in a later version of this paper. Specifically, one might think of following the convention that any undeclared function or predicate symbol occurring in a benchmark is automatically considered as uninterpreted. Technically, this is feasible if we add to the proposed sort system a predefined top sort, i.e., a sort that is by definition a supersort of any other sort. In that case, an interface for a symbol (i.e. its input and output sorts) can be alway inferred automatically from the formula. We point out, however, that in essence this requires a solver to be able to do type inference, as opposed to just type checking, a considerable more complex task in case of signatures with subsorts.

6 Semantics

The semantics of the SMT-LIB language is provided by a translation of let_formulas and their signature into formulas of $FOL^{=}$.

We define a translation operator **Tr**(_) below, following a well-known relativization process for translating many-sorted logics into classical logic. Formally, the semantics of SMT-LIB benchmarks is defined as follows.

Definition 6.1. Let Σ be an order-sorted signature generated by a set *Sub* of *subsort_decl*'s, a set F of *fun_symb_decl*'s, and a set P of *fun_symb_decl*'s. Let T be a theory (of signature Σ) axiomatized by a set Ax of closed formulas well-sorted with respect to Σ. Then let ψ be a closed formula well-sorted with respect to Σ. We say that ψ is satisfiable in T iff the set

$$\{\mathbf{Tr}(sd) \mid sd \in Sub\} \cup \{\mathbf{Tr}(fd) \mid fd \in F\} \cup \{\mathbf{Tr}(ax) \mid ax \in Ax\} \cup \{\mathbf{Tr}(\psi)\}$$

of $FOL^{=}$ sentences is satisfiable in the classical sense.

The particular sets *Sub*, F, P, Ax in the definition above are meant to be elicited from the specification of a benchmark set containing the formula ψ and from the specification of the benchmark set's background theory.

A fine point to note about our translation semantics is that it effectively requires all background theories of SMT-LIB to be defined axiomatically (so that one can identify the set Ax needed in Definition 6.1). Defining the background theories by a set of axioms, as opposed to a set of models, say, looks like the only option if we want to avoid specifying a native algebraic semantics—one with order-sorted models—for our language, instead of the translation semantics below. Further discussion on this point is needed.

The translation operator $\mathbf{Tr}(\_)$ is defined in the following on sort declarations, function symbol declarations, and formulas. To describe the translation into $FOL^{=}$, we use a conventional syntax for $FOL^{=}$ sentences and abuse the notation a bit by sometimes ignoring the fact that the terms of the SMT–language have a LISP-like syntax instead of the usual mathematical notation.

Translation of sort declarations and function symbols declarations

$$\begin{aligned}
\mathbf{Tr}(\textit{sort_decl}) &= \mathbf{Tr}((S_1\ S_2)) = \forall x\,(S_1(x) \Rightarrow S_2(x)) \\
\mathbf{Tr}(\textit{fun_symb_decl}) &= \mathbf{Tr}((f\ S_1 \cdots S_{n+1})) \\
&= \forall x_1, \ldots, x_n\,(S_1(x_1) \wedge \cdots \wedge S_n(x_n) \Rightarrow S_{n+1}(f(x_1, \ldots, x_n)))
\end{aligned}$$

Translation of formulas For formulas, the translation operator is defined inductively on the structure of *let_formula*.

Recall that variable occurrences in a formula can be bound by a quantifier or by a :**let** binder, while propositional variable occurrences in a formula can be bound by an :**flet** binder. In the following, we denote by $\psi\{x \mapsto t\}$ the formula obtained from the formula ψ by simultaneously replacing every unbound occurrence of the variable x in ψ by the term t. Similarly, we denote by $\psi\{p \mapsto \varphi\}$ the formula obtained from ψ by simultaneously replacing every unbound occurrence of the propositional variable p in ψ by the formula φ.

$$
\begin{array}{ll}
\mathbf{Tr}((\textbf{:let } a\ t\ \psi)) & = \mathbf{Tr}(\psi\{x \mapsto t\}) \\
\mathbf{Tr}((\textbf{:flet } p\ \varphi\ \psi)) & = \mathbf{Tr}(\psi\{p \mapsto \varphi\}) \\
\mathbf{Tr}((\textbf{:not } \varphi)) & = \neg\mathbf{Tr}(\varphi) \\
\mathbf{Tr}((\textbf{:and } \varphi_1\ \cdots \varphi_n)) & = \mathbf{Tr}(\varphi_1) \wedge \cdots \wedge \mathbf{Tr}(\varphi_n) \\
\mathbf{Tr}((\textbf{:or } \varphi_1\ \cdots \varphi_n)) & = \mathbf{Tr}(\varphi_1) \vee \cdots \vee \mathbf{Tr}(\varphi_n) \\
\mathbf{Tr}((\textbf{:xor } \varphi_1\ \cdots \varphi_n)) & = \mathbf{Tr}(\varphi_1) \oplus \cdots \oplus \mathbf{Tr}(\varphi_n) \\
\mathbf{Tr}((\textbf{:ite } \varphi_1\ \varphi_2\ \varphi_3)) & = (\mathbf{Tr}(\varphi_1) \Rightarrow \mathbf{Tr}(\varphi_2)) \wedge (\neg\mathbf{Tr}(\varphi_1) \Rightarrow \mathbf{Tr}(\varphi_3)) \\
\mathbf{Tr}((\textbf{:exists } (x\ S)\ \varphi)) & = \exists x\ (S(x) \wedge \mathbf{Tr}(\varphi)) \\
\mathbf{Tr}((\textbf{:forall } (x\ S)\ \varphi)) & = \forall x\ (S(x) \Rightarrow \mathbf{Tr}(\varphi)) \\
\mathbf{Tr}(\textbf{:false}) & = \bot \\
\mathbf{Tr}(\textbf{:true}) & = \neg\bot \\
\mathbf{Tr}(p) & = p \\
\mathbf{Tr}((p\ t_1\ \cdots\ t_{n+1})) & = p(t_1, \ldots, t_{n+1}) \\
\mathbf{Tr}((=\ t_1\ \cdots\ t_{n+2})) & = t_1 = t_2 \wedge \cdots \wedge t_{n+1} = t_{n+2} \\
\mathbf{Tr}((\textbf{:distinct } t_1\ \cdots\ t_{n+2})) & = \bigwedge_{1 \le i < j \le n+2} \neg(t_i = t_j)
\end{array}
$$

7 Examples

In this section we present some examples of theory and benchmark set specifications written in the proposed format.

Most examples are about the theory of real numbers because this theory is particularly apt at illustrating the various aspects of the format, and in particular the point of distinguishing between logic, theories, and input language in SMT-LIB.

7.1 The theory of real numbers

There really is one theory T_R that captures the real numbers, the one that logicians usually call the theory of ordered real-closed fields. The literature, however, is full with theories and "logics" that one way or another have to do with the real numbers. For the purposes of SMT-LIB, most of these various theories and logics are best described in terms of the restriction that they impose on the class of formulas considered for satisfiability in T_R.

For instance, the satisfiability of formulas in so called "linear arithmetic" can be described as the satisfiability in T_R of formulas built as conjunctions of equations and inequations ($\le$) between *linear* terms, that is, terms reducible to linear polynomials.

Similarly, the satisfiability of formulas in so the called "separation logic" or "difference logic"can be described as the satisfiability in T_R of conjunctions of formulas of the form $m * (x - y) < n$ where x and y are variables and m, n are two numerals.

To avoid an undue proliferation of theories in STM-LIB we would specify only the full first-order theory of the real numbers, and consider its restrictions only at the level of benchmark sets. Similar considerations of course would apply

to other theories such as, for instance, the full-first order theory of the natural numbers and the restriction of it known as Presburger arithmetic.

We present two versions of the theory of the real numbers to highlight the different possibilities offered by the order-sorted framework.

Real numbers I

```
(:name "REALS-I"
 :sorts (Real)
 :funs ((0 Real) (1 Real)
        (~ Real Real)
        (+ Real Real Real) (* Real Real Real))
 :preds ((< Real Real))
 :definition "The standard, one-sorted theory of real numbers"
 :extensions "A numeral n > 1 abbreviates the sum (+ 1 ... (+ 1 1))
              of n ones;
              (- t_1 t_2) abbreviates (+ t_1 (~ t_2));
              (<= t_1 t_2) abbreviates (:or (< t_1 t_2) (= t_1 t_2))
              and similarly for >= and >"
)
```

A benchmark set for this theory might look like the following.

```
(:name "RB1"
 :theory "REALS-I"
 :language "Conjunctions of linear ground equations and inequations"
 :extra_funs ((x Real) (y Real) (f Real Real))
 :benchmarks (
  (:formula (:and (= (+ (* 5 x) y) 0)
                  (= (- x y) 4))
   :status :sat)
  (:formula (:and (= (+ x (+ y (f x))) 0)
                  (<= (- x y) 4)
                  (< (* 4 x) (f x)))
   :status :sat))
)
```

Note how this example uses the uninterpreted constants `x` and `y` as the unknowns of the first benchmark, and how the second benchmark also contains occurrences of the uninterpreted function symbol `f`.

The second example considers first-order benchmarks with no extra symbols.

```
(:name "RB2"
 :theory "REALS-I"
 :language "all first-order formulas"
 :benchmarks (
  (:formula (:not (:forall (?x Real) (:forall (?y Real)
                   (impl (< ?x ?y)
                         (:exists (?z Real) (:and (< ?x ?z) (< ?z ?y)))))))
   :status :unsat))
)
```

The third example considers difference logic-like benchmarks.

```
(:name "RB3"
 :theory "REALS-I"
 :language "Boolean combinations of atoms of the form
            (op (* m (- x y)) n)
            where op is =, <, <, >, or >=,
            m, n are numerals and x,y are variables."
 :extra_funs ((x Real) (y Real) (z Real))
 :benchmarks (
  (:formula (:or (:and (<= (- x y) 0)
                       (= (- x z) 7))
                 (< (* 3 (- y z)) 1))
   :status :unknown))
)
```

Real numbers II

Here is a second take on the theory of real numbers, motivated by the fact that in some applications one is often interested in restricting the range of problem variables to integer values only. For those applications an order-sorted version of the theory seems more convenient.

```
(:name "REALS-II"
 :sorts (Nat Int Real)
 :subsorts ((Nat Int) (Int Real))
 :funs ((0 Nat) (1 Nat)
        (~ Real Real)
        (+ Real Real Real) (* Real Real Real))
 :preds ((< Real Real))
 :definition "The standard theory of real numbers over the sort Real,
              together with the Nat and Int subsorts defined by the
              following axioms, which specify that the sort Nat is
              generated by 0,1 and +, while the sort Int is generated
              by 0,1, + and ~(unary minus).
              (:forall (?x Real)
               (:iff (:exists (?n Nat) (= ?x ?n))
                     (:or (= ?x 0)
                          (exists (?m Nat) (= ?x (+ ?m 1))))))
              (:forall (?x Real)
               (:iff (:exists (?z Int) (= ?x ?z))
                     (:exists (?n Nat) (:or (= ?x ?n) (= ?x (~ ?n)))))))"
 :extensions "A numeral n > 1 abbreviates the sum (+ 1 ... (+ 1 1))
              of n ones.
              (- t_1 t_2) abbreviates (+ t_1 (~ t_2))
              (<= t_1 t_2) abbreviates (:or (< t_1 t_2) (= t_1 t_2))"
)
```

The following benchmark set should make clear that subsorts so add to the power of the language, as changing sort constraints on a formula's variables changes its satisfiability in the theory.

```
(:name "RB4"
 :theory "REALS-II"
 :language "Conjunctions of linear equations"
 :extra_funs ((x Real) (y Real) (n Int) (m Int))
 :benchmarks (
  (:formula (:and (= (+ x y) 1)
                  (= (- (* 3 x) (* 3 y)) 1))
   :status :sat)
  (:formula (:and (= (+ m n) 1)
                  (= (- (* 3 n) (* 3 n)) 1))
   :status :unsat))
)
```

The benchmarks in this set represent the same system of equations. The only difference is that the system's unknowns are required to be real valued in the first case and integer valued in the second case. Since the system admits only a non-integer solution, the first benchmark is satisfiable in the theory while the second is not.

7.2 The theory of arrays

We conclude by providing a specification for another popular theory in the SMT literature, the theory of functional arrays (without the extensionality axiom). The specification of this theory is unproblematic because the theory is defined by just two axioms.

```
(:name "ARRAYS-I"
 :sorts (Index Element Array)
 :funs ((select Array Index Element) (store Array Index Element Array))
 :definition "The theory of arrays defined by the following axioms:
              (:forall (?a Array) (:forall (?i Index)
              (:forall (?e Element)
                (= (select (store ?a ?i ?e) ?i) ?e))))
              (:forall (?a Array)
              (:forall (?i Index) (:forall (?j Index)
              (:forall (?e Element) (:or (= ?i ?j)
                                         (= (select (store ?a ?i ?e) ?j)
                                            (select (?a ?j)))))))))"
)
```

8 Acknowledgments

We would like to thank the following members of the SMT-LIB interest group, in alphabetical order, for their initial suggestions and comments on the SMT-LIB format: Clark Barrett, Greg Nelson and the members of Sparta group at HP SRC, Harald Ruess, and Aaron Stump.

References

1. Holger Hoos and Thomas Stützle. SATLIB–The Satisfiability Library. Web site at: `http://www.satlib.org/`.
2. Silvio Ranise and Cesare Tinelli. SMT-LIB–The Satisfiability Modulo Theories Library. Web site at: `http://combination.cs.uiowa.edu/smtlib/`.
3. Geoff Sutcliffe and Christian Suttner. The TPTP Problem Library for Automated Theorem Proving. Web site at: `http://www.cs.miami.edu/ tptp/`.