A Flexible Proof Format for SMT: a Proposal

Frédéric Besson, Pascal Fontaine, Laurent Théry

A Flexible Proof Format for SMT: a Proposal*

Frédéric Besson
INRIA Rennes – Bretagne Atlantique, France
Frederic.Besson@inria.fr

Pascal Fontaine
University of Nancy and INRIA, Nancy, France
Pascal.Fontaine@loria.fr

Laurent Théry
INRIA Sophia-Antipolis – Méditerranée, France
Laurent.Thery@inria.fr

Abstract

The standard input format for Satisfiability Modulo Theories (SMT) solvers has now reached its second version and integrates many of the features useful for users to interact with their favourite SMT solver. However, although many SMT solvers do output proofs, no standardised proof format exists. We, here, propose for discussion at the PxTP Workshop a generic proof format in the SMT-LIB philosophy that is flexible enough to be easily recast for any SMT solver. The format is configurable so that the proof can be provided by the solver at the desired level of detail.

1 Introduction

Satisfiability Modulo Theory (SMT) consists in deciding the satisfiability of formulae belonging to a combination of theories. Over the past few years, the quality of SMT-provers has greatly improved. This is evaluated at the SMT-COMP, the annual competition for SMT. Current SMT-provers are highly optimised and engineered tools that are capable of deciding formulae of industrial size. For the moment, the output of most SMT-provers is just the simple answer: `sat` or `unsat`. This information is enough to evaluate their speed and relative soundness – especially when the status of formulae is known beforehand. But this may not be sufficient, and particularly when trusting the SMT solver is not an option, like in a skeptical cooperation of solvers.

The purpose of the current proposal is to tackle this problem and propose a generic proof format for SMT-provers. Compared to existing approaches our objective is to aim at a format that is sufficiently generic so that:

- any SMT-solver could generate a proof without too much effort;
- the proof could be checked by a trustworthy external verifier.

Our assessment is that previous attempts have produced formats that are either easy to generate but hard to check or hard to generate and easy to check. For instance, the SMT-solver `clsat` generates proofs that are already genuine proofs objects of the logical framework LF [9]. This approach is very challenging and intellectually attractive. Yet efficient generation and checking of LF proofs is still an open research area. Unlike [9], we advocate for a clear distinction between the proof generated by the SMT-solver and the proof-object that would be built by the checker. As a pay-off our proof format should be easier to produce and not require a substantial re-engineering of the SMT-solver. There are other proof-generating SMT-provers such as CVC3 [2], veriT [4] and Z3 [6, 5]. For those, the proof format is not totally formalised and proof reconstruction in a skeptical proof assistant is not a trivial task [8, 3]. One difficulty lies in the fact that certain proof steps of the SMT-provers are kept implicit and do not appear in the proof trace. We aim at providing a proof format that is *proof-assistant friendly*.

Pascal Fontaine, Aaron Stump (eds.); PxTP 2011, pp. 15-26

*This work was funded by the ANR Decert project.

1.1 Proof of satisfiability/unsatisfiability

When a formula is satisfiable, a checkable account of the satisfiability is a detailed model of the formula, such that every term, atom, literal, and sub-formula has a precise value. Giving this detailed model may be problematic when handling formulae with quantifiers. When the formula is unsatisfiable, SMT-solvers derive an inconsistency from the original formula. A proof of unsatisfiability is thus a checkable derivation from the original formula to an object which is trivially inconsistent. The *context* of such a proof is the set of all deduced facts. Initially, the context contains the original formula. Logical *rules* are applied to derive new formulae from formulae in the context. These new formulae are then added to the context. At the end of the proof, the empty clause (noted `()`) should belong to the final context.

Our proof format aims at providing a clear interface between SMT-solvers that generate proofs and checkers that verify the correctness of the generated proofs. We voluntarily restrict ourselves to a specific and limited fragment of the logics current SMT-solvers can deal with. This will let us experiment rapidly and get feedback on how the format should evolve and be improved. As a result, we are considering quantifier-free formulae with uninterpreted functions, equality and linear arithmetic only. In particular, we do not take quantifiers into account. This means that tasks such as skolemization and instantiation are not covered yet.

The work of the solver is usually composed of several phases that the proof format has to address:

- First, the formula may be rewritten: the formula is transformed to a somewhat simpler and equivalent one. This is done by identifying sub-formulae and sub-terms that clearly can be rewritten to simpler equivalent forms. Provers implement this by some rewriting rules, and those rewriting rules should have their derivation counterparts.

- SMT-solvers, being based on the SAT-solvers technology, also require conjunctive normal forms (CNF). The transformation phase, that translates the original formula to a set of clauses also requires a proof. Each clause of the CNF becomes a new fact added to the context. From those clauses, new clauses can be derived using resolution. Resolution is a complete method for the satisfiability of propositional formulae.

- Reasoning about theories can be understood as adding conflict clauses to the SAT-solver. These conflict clauses are tautologies according to the theories in action. To each theory will correspond a set of rules that will add clauses in the context. Equality exchange between different parts of the reasoner can simply be seen as resolution.

1.2 Rationale for the proof format

The SMT-LIB 2.0 format [1] is the *de facto* input standard of the SMT community. It provides a set of commands for interacting with solvers. While designing our proof format, we have been careful at being fully compatible with the SMT-LIB format. Other existing formats like TSTP [10] could have been valid candidates to represent SMT proofs but we wanted to have a format whose syntax was a direct extension of SMT-LIB syntax. For this reason, we inherit the syntax of terms and formulae but also certain well-formedness conditions from the SMT-LIB. This does not prevent the possibility of developing a translator to other formats like TSTP in the future.

The core of the proof format specifies the syntax and semantics for the existing SMT-LIB command `get-proof` – currently left unspecified. To our opinion, a *fixed* proof format is not viable because of the variety of existing SMT-provers. Our proposal is a proof format that is built upon a generic kernel enriched by prover-specific proof rules that can be obtained by a dedicated `get-proof-header` command (see Section 2.2).

Overall, the proof format is a trade-off between the requirements from proof consumers, who want a non-ambiguous syntax and semantics, and developers of SMT-solvers, who favour freedom in outputting proofs in a structure that their solvers can generate without sacrificing efficiency. In Section 2.5, we list a set of recommendations for getting a maximal benefit from the proof format.

The rest of the paper is organised as follows. Section 2 explains the syntax of the proof format. The format is illustrated by an example discussed in details in Section 3. Section 4 provides an operational semantics of the format by providing a reference implementation for a proof-checker.

2 Syntax

2.1 Data structures

This document inherits the syntax for terms (and thus formulae) from the SMT-LIB 2.0 standard [1]. Furthermore, clauses are used in several places. A clause is a list of formulae. For instance: `((= x y) (= (f x) (f y)))` represents a clause which is a disjunction of the formula $x = y$ and the formula $f(x) = f(y)$. The empty clause is the empty list `()`. For convenience, one can use the trivially valid clause `(true)`.

2.2 Proof header

The format is generic and therefore has to be instantiated with solver-specific proof rules. To ease proof-reconstruction by third-party tools, proof rules are declared belonging to a certain SMT-LIB logic, with an attached informal description. The header may be output on request by the SMT solver with a supplementary command `get-proof-header`, an addition to SMT-LIB 2.0. An example of such a header is given in Figure 2. Here is its syntax:

⟨rule_tag_def⟩ ::= `(define-rule-kw` ⟨keyword⟩ ⟨string⟩`)`
⟨rule_def⟩ ::= `(`⟨rule_id⟩ ⟨logic⟩ ⟨attribute⟩*`)`
⟨header⟩ ::= ⟨rule_tag_def⟩* ⟨rule_def⟩*

where ⟨attribute⟩ is defined as in SMT-LIB 2.0. Note that attribute values can be ⟨sexpr⟩ so they could possibly contain code (for instance for proof checkers). All keywords used as attributes in ⟨rule_def⟩ should be defined; there is currently no predefined keyword. Keywords are used to identify collection of rules. For example, they could be used to qualify rules that are handling quantifiers, skolemizations, or specific operations on connectors (e.g., conjunctions). A rule using ⟨logic⟩ should be such that SMT solvers implementing only the SMT-LIB 2.0 logic must be able to verify every clause deduced by this rule.

2.3 Proof script

An example proof script is presented in Figure 3. A proof script is a pair made of a proof context and a sequence of proof steps

⟨proof⟩ ::= ⟨context⟩ ⟨proof_step⟩*

where the context is an SMT-LIB 2.0 script using only commands `set-logic`, `declare-sort`, `define-sort`, `declare-fun`, `define-fun`, `assert`. The context is generally just a subset of the SMT-LIB 2.0 input

script. Furthermore, in asserts, in order to identify formulae, we require them to be explicitly named using the following SMT-LIB 2.0 notation:

`(assert (!` ⟨term⟩ `:named` ⟨clause_id⟩`))`

where ⟨term⟩ is the formula asserted, written in the SMT-LIB 2.0 syntax, and ⟨clause_id⟩ is an SMT-LIB 2.0 ⟨symbol⟩.

A proof step is defined by the following rule:

⟨proof_step⟩ ::= `(define` ⟨term_id⟩ ⟨term⟩`)`
| `(set` ⟨clause_id⟩ ⟨gen_clause⟩`)`
| `(seth` ⟨clause_id⟩ ⟨clause⟩`)`

where

- `(define` ⟨term_id⟩ ⟨term⟩`)` declares ⟨term_id⟩ as a short-name for the term ⟨term⟩.
- `(set` ⟨clause_id⟩ ⟨gen_clause⟩`)` constructs a new clause named ⟨clause_id⟩ using the *clause generation* rule ⟨gen_clause⟩ (see below). The identifier ⟨clause_id⟩ is an SMT-LIB 2.0 ⟨symbol⟩ and is used to refer to the clause in the current environment
- `(seth` ⟨clause_id⟩ ⟨clause⟩`)` is used to assert the hypotheses of a sub-proof. Again, ⟨clause_id⟩ is an SMT-LIB 2.0 ⟨symbol⟩.

Note that this version of the proof format does not cover quantifiers yet. So ⟨term⟩ is just a ground term.

2.4 Derived clauses

A clause can either be explicit or derived. Here we describe how it is derived:

⟨gen_clause⟩ ::= ⟨clause_id⟩
| `(`⟨rule_id⟩
`(:clauses (`⟨gen_clause⟩*`)` | `:all-clauses)`?
`(:terms (`⟨term⟩*`))`?
⟨attribute⟩*
`(:conclusion` ⟨clause⟩`)`?`)`
| `(subproof` ⟨proofstep⟩* `(:conclusion` ⟨clause⟩`)`? `)`

There exist three ways to derive a clause:

- ⟨clause_id⟩ is used to retrieve a clause formerly assigned a ⟨clause_id⟩ in the current environment (where ⟨clause_id⟩ is an SMT-LIB 2.0 ⟨symbol⟩).
- A clause can be derived using a (named) rule: as arguments an optional list of terms, and an optional result can be provided. When `:clauses` is used, the conclusion is a logical consequence of the given set of clauses. When `:all-clauses` is used, the conclusion is a logical consequence of an unspecified set of local hypotheses. When neither the tag `:clauses` nor `:all-clauses` is used, the rule introduces a tautology.

- A local proof can be declared. For simplicity, only local proofs from an empty context are supported; references to the current context in a local proof is not allowed. The clause attached to a local proof is composed of all the literals in all clauses introduced by the `seth` in the proof steps (these literals are negated) and the literals in the clause given after the `:conclusion` attribute. When this attribute is omitted, it is the clause attached to the last step that is considered as conclusion.

2.5 Recommendations

Figure 1 presents the whole grammar of the format. The format is very flexible and tries to avoid unnecessary constraints. Nevertheless, some obvious recommendations can already be made in order to allow effective validation by third-party tools:

- rules should be carefully described in the header;
- term sharing should be maximised in order to improve memory management;
- all the hypotheses in local proofs should be specified first.

These are recommendations and not requirements since external tools could always refactor proofs in order to meet these recommendations.

```
⟨rule_tag_def⟩  ::=  (define-rule-kw ⟨keyword⟩ ⟨string⟩)
⟨rule_def⟩      ::=  (⟨rule_id⟩ ⟨logic⟩ ⟨attribute⟩*)
⟨header⟩        ::=  ⟨rule_tag_def⟩* ⟨rule_def⟩*

⟨proof⟩         ::=  ⟨context⟩ ⟨proof_step⟩*

⟨proof_step⟩    ::=  (define ⟨term_id⟩ ⟨term⟩)
                 |   (set ⟨clause_id⟩ ⟨gen_clause⟩)
                 |   (seth ⟨clause_id⟩ ⟨clause⟩)

⟨gen_clause⟩    ::=  ⟨clause_id⟩
                 |   (⟨rule_id⟩
                         (:clauses (⟨gen_clause⟩*) | :all-clauses)?
                         (:terms (⟨term⟩*))?
                         ⟨attribute⟩*
                         (:conclusion ⟨clause⟩)?)
                 |   (subproof ⟨proofstep⟩* (:conclusion ⟨clause⟩)? )
```

Figure 1: The complete proof format

3 An example

In Figure 3, we present a proof in our format of the following formula

$$\neg((a = c) \wedge (b = c) \wedge ((f(a) \neq f(b)) \vee (p(a) \wedge \neg(p(b)))))$$

For readability, the proof does not use the facility of the format for sharing terms. It starts with the context which is just a subset of the SMT-LIB 2.0 input script. Every following command introduces a new clause (with the `set` command) until the final clause, the empty clause `()`. The proof relies on elementary rules. The description of these rules can be obtained using the command `(get-proof-header)` which could return (among other rule definitions) the definitions in Figure 2. In those definitions, the `:clauses`, `:terms` and `:conclusion` attributes respectively give the expected number of clauses (-1 if arbitrary), terms, and conclusion (0 if no conclusion is provided, that is, if the checker is expected to recompute the conclusion).

The `and` rule is used to deduce new clauses from conjunctive unit clauses: every conjunct of a conjunctive unit clause can indeed be itself introduced as a unit clause, as are clauses `c2`, `c3` and `c4` in the example in Figure 3. The `and_pos` rule generates a tautology of the form $\neg \wedge_i a_i \vee a_j$ for some j (see clauses `c5` and `c6`). The `or` rule introduces a clause from a disjunctive unit clause, by simply building the clause of the disjuncts (e.g. clause `c7`). Clauses `c2` to `c7` constitute the CNF of the input formula. Notice that the CNF transformation is not a naive transformation, but it does not explicitly introduce new (Tseitin) Boolean variables. Indeed, as a side effect of the fact that clauses are sets of formulae (and not just literals), it is easy to introduce definitional CNF transformations without introducing new Boolean variables: the formulae themselves stand for those Boolean variables. Using sharing, the CNF would be linear with respect to the size of the initial formula.

Rules `eq_congruent`, `eq_congruent_pred`, and `eq_transitive` introduce equational tautologies (for instance `c8`, `c9`, `c15` and `c16`). The resolution rule implements chain resolution. Note that this rule serves several purposes here. It is used for instance for resolution executed within the SAT-solver (e.g. to deduce `c18`), and it is used to build conflict clauses from generic tautologies (e.g. to build `c10`).

The `subproof` construct (see Figure 1) can always be inlined – maybe at the cost of adding more proof-rules. It does not change the expressive power of the proof-format but allows for more structured proofs. The `subproof` construct can also be used to minimise the number of proof rules. For instance, in the proof-script of Figure 3, the clause `c5` can alternatively be obtained by the following sub-proof.

```
(set c5 (subproof (seth h (and (p a) (not (p b))) )
          (set res (and :clauses (h) :conclusion (p a)))))
```

Note that the conclusion of a sub-proof (if omitted) can exactly be reconstructed as soon as the proof rule for the last derived clause in the sub-proof has an explicit conclusion. Here, the obtained clause is therefore `((not (and (p a) (not (p b)))) (p a))`.

4 Proof checking

In this section, we provide a reference implementation for a proof verifier for the generic proof format. The proof verifier is parametrised by prover-specific proof rules declared in the proof header. As a result, the validity of the proof verifier relies on the fact that proof rules are logically sound *i.e.,* only derive clauses that are logic consequences of the current known clauses.

The proof verifier takes as input a proof $\pi \in \langle \text{context} \rangle \times \langle \text{proof\_step} \rangle^*$ (see Section 2.3). and verifies that the proof steps indeed derive the empty clause *i.e.,* the conjunctions of the formulae asserted by the $\langle \text{context} \rangle$ part of the proof is unsatisfiable. The state of the verifier is made of a quadruple (Sig, N, S, cl) where

- *Sig* is a SMT-LIB signature built from the commands `declare-sort` and `declare-fun` of the proof context. Its purpose is to ensure that terms are well-sorted.

```
; get-proof-header returns

(and_pos QF_UF
  :comment "valid clause ((not (and a_1 ... a_n)) a_i)"
  :clauses 0 :terms 0 :conclusion 1)

(and QF_UF
  :comment "(and :clauses (c) :conclusion (a_i))
              where c = ((and a_1 ... a_n))"
  :clauses 1 :terms 0 :conclusion 1)

(or QF_UF
  :comment "(or :clauses (c) :conclusion (a_1 ... a_n) )
              where c = ((or a_1 ... a_n))"
  :clauses 1 :terms 0 :conclusion 1)

(eq_transitive QF_UF
  :comment "valid clause
             ((not (= x_1 x_2)) ... (not (= x_{n-1} x_n)) (= x_1 x_n))"
  :clauses 0 :terms 0 :conclusion 1)

(eq_congruent QF_UF
  :comment "valid clause ((not (= x_1 y_1)) ... (not (= x_n y_n))
                          (= (f x_1 ... x_n) (f y_1 ... y_n)))"
  :clauses 0 :terms 0 :conclusion 1)

(eq_congruent_pred QF_UF
  :comment "valid clause ((not (= x_1 y_1)) ... (not (= x_n y_n))
                          (not (p x_1 ... x_n)) (p y_1 ... y_n))"
  :clauses 0 :terms 0 :conclusion 1)

(resolution QF_UF
   :comment "Chain resolution of any number of clauses"
   :clauses -1 :terms 0 :conclusion 1)
```

Figure 2: The proof header

- N is a mapping from symbols to constants and function declarations. It is initialised by the `define-fun` context commands. Later on, for sharing purpose, it is updated by the proof command `define`. Note that unlike `define-fun`, the `define` command can only define constants[1].

- S is a stack of named assertions tagged with a boolean flag. The `assert` commands of the context construct an initial singleton stack where all the asserted formulae are tagged `true`. During the proof, derived clauses introduced by the `set` command are also tagged by `true`. The tag `false` is reserved to clauses introduced by the `seth` command. Those clauses are local hypotheses to be discharged by sub-proofs.

[1]For more flexibility, this restriction might be lifted in the future.

```
; get-proof returns

; Context

(set-logic QF_UF)
(declare-sort U 0)
(declare-fun p (Bool) U)
(declare-fun f (U) U)
(declare-fun a () U)
(declare-fun b () U)
(declare-fun c () U)
(assert (! (and (= a c) (= b c)
                (or (not (= (f a) (f b))) (and (p a) (not (p b))))) :named c1))

; Proof

(set c2 (and :clauses (c1) :conclusion ((= a c))))
(set c3 (and :clauses (c1) :conclusion ((= b c))))
(set c4 (and :clauses (c1)
             :conclusion ((or (not (= (f a) (f b))) (and (p a) (not (p b)))))))
(set c5 (and_pos :conclusion ((not (and (p a) (not (p b)))) (p a))))
(set c6 (and_pos :conclusion ((not (and (p a) (not (p b)))) (not (p b)))))
(set c7 (or :clauses (c4)
            :conclusion ((not (= (f a) (f b))) (and (p a) (not (p b))))))
(set c8 (eq_congruent :conclusion ((not (= b a)) (= (f a) (f b)))))
(set c9 (eq_transitive :conclusion ((not (= b c)) (not (= a c)) (= b a))))
(set c10 (resolution :clauses (c8 c9)
                     :conclusion ((= (f a) (f b)) (not (= b c)) (not (= a c))))))
(set c11 (resolution :clauses (c10 c2 c3) :conclusion ((= (f a) (f b)))))
(set c12 (resolution :clauses (c7 c11) :conclusion ((and (p a) (not (p b))))))
(set c13 (resolution :clauses (c5 c12) :conclusion ((p a))))
(set c14 (resolution :clauses (c6 c12) :conclusion ((not (p b)))))
(set c15 (eq_congruent_pred :conclusion ((not (= b a)) (p b) (not (p a)))))
(set c16 (eq_transitive :conclusion ((not (= b c)) (not (= a c)) (= b a))))
(set c17 (resolution :clauses (c15 c16)
                     :conclusion ((p b) (not (p a)) (not (= b c)) (not (= a c))))))
(set c18 (resolution :clauses (c17 c2 c3 c13 c14) :conclusion ())
```

Figure 3: A simple example with its proof.

- *cl* is the last derived clause. It is initialised to true (true) and for a valid proof is eventually set to the empty clause ().

The proof checking succeeds if the empty clause is eventually generated in a singleton stack for which all the formulae are tagged `true`.

The proof-verifier is executed in a state constructed by running the context commands. After running the context commands the proof state is such that:

- The signature *Sig* contains sort and function declarations (*Sig* is thereafter immutable)

- *N* contains constant and function declarations (during the proof checking, only new constants can

be defined)

- S is a singleton stack of the form $\{n_1 \mapsto cl_1^{tt}, \ldots, n_i \mapsto cl_i^{tt}\}$ where a binding (n_i, cl_i) is the result of the `assert` command

  `(assert (! f`$_i$ `:named n`$_i$`)).`

  cl_i being then the clause containing the sole formula `f`$_i$.

4.1 Conventions and auxiliary functions

The verifier makes use of auxiliary functions and predicates. Several of them are already part of the SMT-LIB 2.0 where they are used to give a semantics to SMT-LIB scripts. The pseudo-code does not introduce an explicit abstract syntax for the proof constructs. Instead we use directly the concrete syntax with the conventions that optional attributes are given a default value. More precisely, missing attributes of type list (for instance `:clauses`, `:terms`, `:attributes`) are given as default value the empty list and a missing attribute of type clause (for instance `:conclusion`) is given as default value the symbol null.

4.1.1 Well-formedness

As in the SMT-LIB we only consider well-formed terms.

Definition 1 (Well-formed terms)**.** *A term t is well-formed with respect to a signature Sig and an environment of names N (isWellFormed(Sig,N,t)) if*

- *all the symbols in the term are bound in N;*
- *the term is well-sorted according to the signature Sig.*

To ensure that a well-formed term will never be invalidated, the signature *Sig* and the environment *N* can only be augmented by additional declarations. It is therefore forbidden to overwrite a declaration. In case of violation, the verification is aborted.

4.1.2 Assertion stack

Proof scripts feature a notion of `subproof` to describe scoped proofs. As already stated, this construct does not increase the expressive power of the format. Its purpose is to structure proofs and thus facilitate their checking. For instance, each theory reasoner of the SMT solver can generate a sub-proof for each theory lemma or conflict clause. The advantage of `sub-proof` is that such theory lemma can be checked in isolation. To implement scoped proofs, the assertion stack is updated by `push` and `pop` operations[2]. Given a tagged clause cl^b, $clause(cl^b)$ returns the clause cl and $tag(cl^b)$ returns the boolean b. The verifier always accesses and updates the topmost assertion set. We write $top[s]$ for the clause bound to the symbol s in the topmost assertion set. Accessing a non-existent symbol aborts the verification. Overwriting a clause tagged with ff also aborts the verification *i.e.,* an assignment of the form $top[s] := cl^b$ where $tag(top[s]) = ff$ aborts the verification.

[2]They correspond to the functions `push 1` and `pop 1` of the SMT-LIB

4.1.3 Prover-specific proof rules

The verifier is parametrised by prover-specific proof rules that are trusted and could therefore be responsible for an invalid proof. Each proof rule is implemented by a function named according to the ⟨rule_id⟩ taking as arguments an optional list of clauses, terms and attributes and either returning a clause or reporting an error. An error immediately aborts the verification of the proof. The soundness requirement is that the generated clause must be a logic consequence of the clauses passed as arguments. A proof rule also takes as arguments the signature *Sig* and the environment of bindings *N*.

$$\langle \text{rule\_id} \rangle : Sig \times N \times \langle \text{clause} \rangle^* \times \langle \text{term} \rangle^* \times \langle \text{attributes} \rangle^* \to \langle \text{clause} \rangle \mid \langle \text{error} \rangle$$

4.2 Pseudo-code of the verifier

The verifier is written in an imperative style and therefore updates in-place the proof state (Sig, N, S, cl). The verifier is presented top-down and consists of 4 functions:

- checker is the top-level function and evaluates a whole proof. Upon success, it concludes that the input problem is not satisfiable.
- eval_proof evaluates all the proof steps in turn and updates the proof state accordingly;
- eval_proof_step performs a case analysis over the proof command;
- eval_gen_clause is responsible for deriving from the proof state new logic consequences by combining prover-specific proof rules.

A proof is valid if after executing the proof steps the clause *cl* is the empty clause () and there are no introduced local hypotheses (all clauses are tagged *tt*).

```
bool checker(Π) {
  eval_proof(Π); return (cl = ()) & (∀ i, tag(top[i]) = tt);
}
```

The function eval_proof executes in turn the proof steps.

```
void eval_proof (Π) { for−each (π ∈ Π) eval_proof_step(π); }
```

The function eval_proof_step performs a case analysis and updates the last derived clause.

```
void eval_proof_step(π){
  switch(π) {
    case (define tid trm) : requires(isWellFormed(trm));
        N[tid] := trm

    case (set cid gencl) :
        cl := eval_gen_clause(gencl);
        top[cid] := cl^tt;

    case (seth cid hyp) :
        requires(isWellFormed(hyp) & hasSortBool(hyp));
        top[cid] := hyp^ff;
}
```

The core of the verifier is the function eval_gen_clause which generates a novel clause logic consequence of the current proof state.

```
clause eval_gen_clause(gencl){
  switch(clgen){
    case clid : return clause(top[clid])

    case (rid :clauses (gc1, ..., gci)
              :terms (t1,..., tk)
              :attributes (a1, ..., an)
              :conclusion concl) :
      /* Recursively generate clauses */
      gcs := (eval_gen_clause(gc1),
              ...,eval_gen_clause(gci));
      /* Call the prover specific proof rule */
      cl  := rid(Sig,N,gcs,(t1,...,tk),(a1,...,an),concl);
      /* Check the conclusion if any */
      if concl != null & concl != cl then abort
      return  cl ;

    case (subproof prf :conclusion concl)
      push(); /* Local assertion set */
      eval_proof(prf);
      if concl != null & cl != concl then abort
      /* Generation of the conflict-clause */
      conflict := (not(h1),....,not(hn), cl);
      where {h1,...,hn} =
              {clause(top[i]) | tag(top[i]) = ff}
      pop();
      return  conflict;
}
```

5 Acknowledgements

The work presented here has been funded by the French ANR Decert initiative. This format is the result of many stimulating conversations with its members. We want to thank Aaron Stump for his remarks on a previous version of this document, and for numerous informal discussions. We also thank the anonymous reviewers for their insightful comments.

6 Conclusion and further work

The proof format presented in this paper is an extension of the SMT-LIB 2.0 format. We have tried as much as possible to find a good compromise between what can be output by an SMT-solver and what is needed in order for a simple checker to be able to verify a proof. We are aware that this format is not perfect and some subtle issues still need to be further investigated and discussed. Nevertheless we hope that the format can serve as a common basis that can be used by the SMT community.

The format is currently being implemented in the veriT solver distributed as open-source using the BSD licence. We are also currently investigating proofs for quantifier reasoning [7]; Skolemization, as a satisfiability preserving transformation which does not preserve logical equivalence, may raise difficult

issues. The CNF transformation presented here does not require Tseitin variables; if such variables are required by other solvers, then similar issues may also appear for CNF transformation.

In Section 4, we presented the code of a proof checker. A standalone checker for the format is one of our long term goals. However, in addition to the infrastructure presented in Section 4, this requires to implement the term data-structure, and the operators to manipulate terms. This comes to reimplementing (parts of) a kernel like those found in mainstream proof assistants. We are currently implementing a module to replay proofs in the present format within Coq, and we plan to study replaying this proof format within LFSC.

References

[1] C. Barret, A. Stump, and C. Tinelli. The SMT-LIB standard: Version 2.0, 2010. Latest official release of Version 2.0 of the SMT-LIB standard.

[2] C. Barrett and C. Tinelli. CVC3. In *CAV '07*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[3] S. Böhme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *ITP'2010*, volume 6172 of *LNCS*, pages 179–194. Springer, 2010.

[4] T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In *CADE'09*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.

[5] L. M. de Moura and N. Bjørner. Proofs and Refutations, and Z3. In *Proc. of the LPAR 2008 Workshops*, volume 418 of *CEUR Workshop Proceedings*, 2008.

[6] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[7] David Déharbe, Pascal Fontaine, and Bruno. Quantifier inference rules for SMT proofs, 2011. Workshop on Proof eXchange for Theorem Proving (PxTP).

[8] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *PDPAR '05*, volume 144(2) of *ETCS*, pages 43–51. Elsevier, 2006.

[9] A. Stump and D. L. Dill. Faster Proof Checking in the Edinburgh Logical Framework. In *CADE-18*, volume 2392 of *LNCS*, pages 392–407. Springer, 2002.

[10] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, pages 201–215, 2004.