Proof Translation and SMT-LIB Benchmark Certification: A Preliminary Report *

Yeting Ge[1], Clark Barrett[1]
[1]New York University, `yeting|barrett@cs.nyu.edu`

Abstract

Satisfiability Modulo Theories (SMT) solvers are large and complicated pieces of code. As a result, ensuring their correctness is challenging. In this paper, we discuss a technique for ensuring soundness by producing and checking proofs. We give details of our implementation using CVC3 and HOL Light and provide initial results from our effort to certify the SMT-LIB benchmarks.

1 Introduction

Satisfiability Modulo Theories (SMT) solvers have been successfully applied in many verification applications. As modern SMT solvers add optimizations and features, they are becoming more complicated than ever before. At the same time, SMT solvers are increasingly being used in applications where the correctness of the solver is essential. With currently available verification techniques, it would be extremely difficult to verify that a modern SMT solver is correct. Even if such a proof were done, it would be difficult to maintain in the face of constant changes to the solver. One alternative is to have the SMT solver produce a record of its proof search and then use a small, trusted proof-checker to check the proof.

However, the approach of generating and checking proofs from SMT solvers faces several challenges. The first challenge is to design a suitable set of proof rules. Unlike SAT solvers, for which only one proof rule (Boolean resolution) is sufficient for proof-checking, SMT solvers require a much richer set of proof rules, which depend on the background theories supported and the decision procedures employed. There are also trade-offs to be considered in the selection of proof rules. On the one hand, a small set of simple rules is better for proof-checking. On the other hand, a larger set of more complex rules makes things easier for the implementer of the SMT solver. An additional issue is the maintenance of the proof rules. As functionality is

*This work was partially supported by National Science Foundation grant number 0551645.

added and modified over time, proof rules may change and new proof rules may be needed. Adding support for these changes to the proof-checking strategy thus incurs additional maintenance effort.

The next challenge is to implement a proof-checker. Proofs of nontrivial SMT benchmarks are far too big to be readable by a human. Thus, proofs must be checked by a trusted proof-checking algorithm. Depending on the number and complexity of proof rules, the task of a proof-checker may range from fairly simple to very complex. One representation of a proof is as a tree in which each node is labeled with a formula. The root of the tree represents the theorem being proved. Each internal node represents the application of a proof rule used to derive the formula labeling that node from the formulas labeling its children. For such a proof tree, the task of a proof-checker is to check that the deduction represented at each node in the tree is valid. For simple rules, such as deriving $\neg true$ from *false*, a simple syntactic check is sufficient. However, for more complicated rules, (for example, a single proof rule could be used to encapsulate normalization of linear arithmetic terms), a sophisticated algorithm requiring many steps may be needed.

There seems to be an unavoidable trade-off between performance and ease of coding the SMT solver (which leads to many complex proof rules) and the simplicity of the proof-checker which is desirable in order to minimize the amount of code that must be trusted (and also to minimize the effort required in building and maintaining the checker).

There is, however, a solution that has most of the advantages of both. The idea is to use another existing theorem prover to check proofs from the SMT solver. This approach enables the use of fairly complicated rules in the SMT solver as long as the reasoning behind the rules can be reproduced in the other prover. The additional work that must be done is then to *translate* each rule into the language and methodology of the other theorem prover. A successful check of the proof results in a theorem in the other prover. Notice that we have reduced the problem of trusting the SMT solver to the problem of trusting the other prover. However, if the other prover is chosen carefully, specifically if the choice is made to use a prover that has a small set of simple core proof rules, then the result is a system in which the SMT solver can use complex proof rules, while at the same time the set of rules that must be trusted is small and simple.

In this paper, we describe our experience with this paradigm. The SMT solver is CVC3 [4], and the proof-checker is HOL Light [5]. To motivate and test the system, we applied it to benchmarks from the SMT-LIB library [3]. These benchmarks are used as points of comparison in many papers as well as in the annual SMT-COMP competition. Every benchmark in SMT-LIB contains a status field indicating whether it is satisfiable, unsatisfiable, or unknown. While benchmark providers and SMT-LIB maintainers do their best to ensure that the status fields are correct, occasionally benchmarks are incorrectly labeled resulting in confusion or controversy.

Our eventual goal is to *certify* as many unsatisfiable benchmarks as possible by producing and checking their proofs. Here, we report on our initial progress towards this goal. Ultimately, satisfiable benchmarks could (and should) also be certified by producing and checking models, but that is beyond the scope of this effort.

The paper is organized as follows. Section 2 gives a brief introduction to CVC3 and its proof system. Section 3 describes the theorem prover HOL Light. Section 4 discusses the translation procedure and several obstacles that had to be overcome in order to make it work. Section 5 discusses our experience running the system on the SMT-LIB benchmarks. Section 6 discusses related work, and Section 7 concludes.

2 CVC3

CVC3 is the latest in a series of SMT solvers (CVC, CVC Lite, CVC3). It aims to be both a platform for SMT research as well as a robust tool for use in verification applications. CVC3 is open-source, is maintained by a number of contributing developers, and enjoys a large and active user community. In order to achieve competitive performance on large benchmarks, CVC3 employs a number of optimization strategies which complicate the code. For instance, CVC3 implements its own memory manager, has reference counting schemes for expressions and theorems, and uses sophisticated data structures for backtracking. At the time of this paper, the code base consists of nearly 100,000 lines of intricate C++ code.

Because applications of theorem provers like CVC3 need to be able to rely on correct results, it is of the utmost importance that the complexity of CVC3 not compromise its correctness. In particular, a theorem prover that is unsound (i.e. reports that a theorem is unsatisfiable when it is actually satisfiable) could lead to missed bugs in critical applications.

One of the primary goals with the CVC family of systems has been to have high confidence in their soundness. The first system, CVC, pioneered the use of proofs within a state-of-the-art SMT solver [9]. The current system, CVC3, builds upon a proof infrastructure developed for CVC Lite [2]. Here, we give a brief overview of CVC3's proof system.

2.1 Proofs

A *proof* is a tree in which each node is labeled with a formula. The formulas at the leaves of the tree are called *assumptions* and the formula at the root is called the *conclusion*. Assumptions may be designated as *open* or *closed*.

A *sequent* is a pair $\Gamma \vdash \phi$, where Γ is a set of formulas and ϕ is a formula. Since we are often interested only in the assumptions and the conclusion, the sequent $\Gamma \vdash \phi$ is used to represent any proof whose open assumptions are Γ and whose conclusion is ϕ.

A *proof rule* or *inference rule* is a function which takes one or more proofs (called *premises*) and returns a new proof (the *consequent*) whose root node has each of the input proofs as its children. A proof rule specifies what formula should label the new root node and may also change the designation of one or more assumptions from open to closed.

Proof rules depend only on the assumptions and conclusions of their premises and can thus be described using sequents. We denote a proof rule as follows:

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

where the P_i's are sequents representing the premises and C is a sequent representing the new proof tree. The proof rule takes any set of proofs which match the P_i's and returns a new proof whose root is labeled by the right-hand side of C. If an assumption appears in some P_i but not in C, then that assumption is closed in the proof tree constructed by the proof rule. If there are no premises, the rule is called an *axiom*.

A sequent $\Gamma \vdash \phi$ is *valid* if the conjunction of the assumptions in Γ implies ϕ. A proof rule is *sound* if the validity of all its premises implies the validity of the conclusion. It is not hard to see that if all the proof rules are sound, then any sequent representing a proof constructed using those proof rules is valid.

2.2 Proof Rules

The most basic rule is the assumption axiom. This rule, together with a few other simple rules, are shown below.

$$\frac{}{\phi \vdash \phi}\text{assume}$$

$$\frac{\Gamma_1 \vdash \phi \leftrightarrow \psi \quad \Gamma_2 \vdash \psi \leftrightarrow \theta}{\Gamma_1 \cup \Gamma_2 \vdash \phi \leftrightarrow \theta}\text{iffTrans}$$

$$\frac{\Gamma_0 \vdash \alpha_0 \quad \Gamma_1 \vdash \alpha_1 \quad \ldots \quad \Gamma_n \vdash \alpha_n}{\Gamma_0 \cup \Gamma_1, \ldots, \Gamma_n \vdash \phi \leftrightarrow \phi'}\text{simplify}$$

Some proof rules (like the middle one above), have results that are completely determined by the premises. Others (like the other two) require additional parameters. For instance, `assume` has no premises and takes ϕ as a parameter, producing the sequent $\phi \vdash \phi$. Similarly, `simplify` takes a set of premises $\Delta = \{\Gamma_i \vdash \alpha_i \mid i \in \{0 \ldots n\}\}$ and the formula ϕ to be simplified as a parameter. It returns a sequent for $\phi \leftrightarrow \phi'$ where ϕ' is obtained by replacing all instances of the literals in Δ by *true* (and their negations by *false*) and applying simple Boolean rewrites to the result.

At the time this paper was written, there were 298 proof rules in CVC3. They include basic first-order rules, rules for propositional logic, and a variety of rules for theory-specific reasoning. They range from extremely simple to very complex.

2.3 Implementation

In CVC3, one of the basic classes is the `Theorem` class. Each instance of this class represents a proof and contains the sequent, i.e. the assumptions and conclusion for this proof. These `Theorem` objects exist even in the high-performance non-proof-producing version of the code. In fact, the assumption lists are critical for producing conflict clauses (see [2]). If proof-production is enabled, then each `Theorem` object in addition contains the actual proof tree represented as a directed acyclic graph (i.e. identical subtrees are shared). Each proof rule is implemented as a function which takes 0 or more `Theorem` objects (the premises) as well as any necessary parameters as input and produces a new `Theorem` (the consequent) as output. These functions exist in specially designated *trusted* code modules. A compile-time check ensures that only trusted modules can create new `Theorem` objects. In addition, each proof rule function checks that its premises are of the right form. These features help ensure that soundness bugs can only be the result of problems in the trusted code modules.

Implementing proof production has been valuable in helping shape and understand the design of the system. More importantly, it has caught and prevented bugs in CVC3. Recently, some changes to the arithmetic module uncovered a soundness bug (a previously known satisfiable benchmark was reported unsatisfiable). We ran our translator and found one step of the proof that could not be validated. A careful examination of the proof rule in question showed that the proof rule itself was not sound. This bug had persisted in CVC3 for years and would have been extremely difficult to find without the proof system.

3 HOL Light

HOL Light is a general purpose interactive theorem prover based on higher order logic. Like other HOL-like systems, HOL Light is capable of formalizing most of useful mathematics. In particular, it is capable of formalizing the theories and reasoning used in SMT solvers.

HOL Light is built on top of a very small trusted logical core. The logical core implements a proof system consisting of ten inference rules, mostly about equality, three axioms, and two principles of conservative definitional extension. It is implemented using only 430 lines of Ocaml code. Except for equality, all other logical symbols are defined, even the propositional connectors like "$\wedge$" and "$\vee$". HOL Light's definitional extension mechanism guarantees that each definition is sound and that any theorems proved are valid as long as the logical core is valid. In addition to being small, the majority of the trusted core of HOL Light has itself been verified formally [6]. As John Harrison, the author of HOL Light, has stated, "it sets a very exacting standard of correctness" [1].

HOL Light is programmable and can easily be extended. Derived proof rules and decision procedures can be implemented as Ocaml functions. Many such derived functions exist as part of HOL Light already. For example, the function *REAL_ARITH* is a decision procedure for proving basic facts about arithmetic. HOL Light also includes decision procedures for propositional and first-order reasoning. These tools can be leveraged for our proof-checking purposes.

4 Proof Translation

When CVC3 is presented with a verification task in SMT-LIB format, it may respond with "satisfiable", "unsatisfiable", or "unknown" (or it may timeout or run out of memory). When the result is "unsatisfiable", CVC3 can produce a proof using its proof system. This proof is used as input to a translation program written on top of HOL Light. The goal of the translator is to read the CVC3 proof and reproduce the same reasoning steps in HOL Light. In order to do this, the translator must be able to translate both the formulas and the proof rules. In this section we discuss how this is done with emphasis on specific challenges that had to be overcome.

4.1 Translation of formulas

CVC3 uses the language of many-sorted first-order logic, while HOL Light is based on higher order logic. Because the theories used in CVC3 can be defined (or are already defined) in HOL Light, it is fairly straightforward to translate formulas of CVC3 into formulas of HOL Light. There are, however, a few idiosyncrasies of the SMT-LIB format that are a bit challenging for HOL Light.

For example, SMT-LIB supports a built-in predicate of variable arity called `distinct`. $\texttt{distinct}(x_1, x_2, ..., x_n)$ means $\forall\, i\, j : [1 \ldots n].\ (i \neq j \rightarrow x_i \neq x_j)$. Because predicates in HOL Light must have a fixed arity, we model this predicate by defining a set of parametrized predicates $\texttt{distinct}_n$, where n is the arity. These predicates are defined only when needed by the translator.

The translation of variables and constants of real and integer types is a bit tricky. CVC3 allows integers to be used as arguments to real operators. This is not allowed in HOL Light. Thus, during translation, if integers and reals appear in the same formula, the integers are lifted into reals.

4.2 Translation of proof rules

For each proof rule in CVC3, we write an Ocaml function whose purpose is to get HOL Light to prove the conclusion given HOL Light theorems for the premises. A naive approach is just to call built-in HOL Light functions and hope they will succeed. For instance, we could call the built-in HOL Light function `REAL_ARITH` to do reasoning about arithmetic. This approach sometimes works for simple rules and formulas, but is too slow to use in general.

A much better method is to prove generalized versions of each proof rule in HOL Light ahead of time and then just instantiate these theorems. For example, consider the following CVC3 proof rule (where x and y are parameters that must be integers):

$$\frac{}{\vdash x < y \leftrightarrow x \leq y - 1}\ \text{lessThanEqRhs}$$

The corresponding theorem in HOL Light is `!x y.  x:int < y <=> x <= y + (-- &1)`. When translating the proof rule `lessThanEqRhs`, we first translate the terms that are used as parameters and then use them to instantiate this theorem. Instantiation of existing theorems is highly efficient in HOL Light and can be used whenever a proof rule corresponds directly with a HOL Light theorem.

Sometimes a proof rule cannot be represented by a single theorem. For example, CVC3's `or_distributivity` rule generates a theorem $(A \wedge B_1) \vee (A \wedge B_2) \vee ... \vee (A \wedge B_n) \leftrightarrow A \wedge (B_1 \vee B_1 \vee ... \vee B_n)$. For such proof rules, We can create a customized theorem on the fly and then instantiate it. In this case, we prove a HOL Light theorem in which A and each B_i are replaced by universal propositional variables. Even for such cases, instantiation of a general theorem is typically faster than proving the particular instance of the proof rule directly because the formulas appearing in the instance (i.e. those used to instantiate A and the B's) could be arbitrarily complex.

One special rule in CVC3 is the skolemization rule:

$$\frac{}{\vdash \exists x.P(x) \leftrightarrow P(c)}\ \text{skolem}$$

In this rule, c is a particular constant that is always used as the witness for $P(x)$. In classical first-order logic, this rule is actually unsound. However, we can use HOL Light's *choice operator* to translate this rule soundly. We simply translate the special constant c as $\varepsilon x.P$. Here, $\varepsilon x.P$ means some x that makes P true if there is one.

Sometimes CVC3 and HOL Light have similar proof rules. For example, `subst_op` in CVC3 and `SUB_CONV` in HOL Light are both rules for substitutions. However, the `subst_op` has a variant in which only some of the children are substituted. For this rule, a modification of the existing code for `SUB_CONV` can be used to do the translation.

Finally, some proof rules are too complicated for any of these approaches and custom translation functions must be written for them. An example is the `rewrite_and` rule. This rule flattens nested conjunctions, removes conjunctions containing *true*, and performs several other rewrites on conjunctions. We wrote a custom translation function that makes use of a proof rule in HOL Light for rewriting conjunctions.

4.3 Translation of propositional reasoning

Most modern SMT solvers use a separate SAT solver to do propositional reasoning and CVC3 is no exception. In previous versions of CVC, proofs could only be obtained by using a slower, custom-built, proof-producing SAT solver [2]. However, modern SAT solvers like zChaff and Minisat can dump a resolution proof for unsatisfiable formulas. We followed a similar approach to that taken by others (e.g. [11]) to produce a complete proof, given the resolution proof.

The rule for propositional resolution can be described as follows.

$$\frac{\Gamma_1 \vdash A \vee B \quad \Gamma_2 \vdash \neg A \vee C}{\Gamma_1 \cup \Gamma_2 \vdash B \vee C}\,\text{bool\_resolution}$$

We experimented with several methods of implementing the resolution rule in HOL Light. One possibility, if A is to be resolved, is to first reorder the disjunctions and move A and $\neg A$ to the front of their respective clauses, removing any duplicate occurrences at the same time. We can then instantiate the following theorem: $(A \vee B) \wedge (\neg A \vee C) \leftrightarrow (B \vee C)$. Unfortunately, translating a large resolution proof using this method turns out to be quite slow. Fortunately, there is a better way. As described in [10], the representation of CNF clauses can be changed into so-called *Sequent Representation*. The key idea is to represent the literals of a clause as assumptions. For instance, $\Gamma \vdash A \vee B$ is represented as $\Gamma, \neg A, \neg B \vdash \mathit{False}$. HOL Light uses a set to store the assumptions, so no reordering is needed, and when two assumption lists are merged, the duplicated literals are removed automatically. In the latest version of HOL Light, the assumption lists are ordered, which further speeds up propositional resolution.

A final point about translating the propositional reasoning has to do with the initial conversion to CNF before running the SAT solver. CVC3 uses a standard Tseitin-style conversion algorithm which introduces additional variables. Each of these new variables is a placeholder for some other formula. However, proving the equisatisfiability of the CNF formula with all of these new variables is an unnecessary complication. Instead, as we replay the resolution proof, we substitute the original formulas for the placeholder variables at each step. Care must be taken to distinguish for instance, the clause $A \vee B \vee C$ from the clause $p \vee C$ where p is a placeholder for $A \vee B$, but with this caveat, there is no difficulty checking the resolution proof. Notice that, in particular, the clauses introduced by converting some internal node in the formula to CNF are all tautologies, and can thus be proved easily by HOL Light.

4.4 Final check

If the proof translation in HOL Light succeeds, the result is a theorem of HOL Light. However, it may be difficult to determine, especially for large problems, whether the theorem proved by HOL Light is in fact the same as the original problem posed to CVC3. A bug in the proof or the proof translation could result in HOL Light successfully proving the wrong theorem. To eliminate the need to trust the proof or the proof translation, we added one final check to make sure that theorem proved in HOL Light is indeed the theorem we want to prove. This is done by translating the original problem into HOL Light's language directly and comparing the result to the theorem that was proved. This reduces the code that must be trusted (besides HOL Light) to CVC3's parser and the formula translator, which are trivial compared to the rest of the system.

5 Experimental results

We tested the system on a subset of the AUFLIA benchmarks from SMT-LIB. All tests were run on AMD Opteron-based (64 bit) systems with 2GB of RAM running Linux. CVC3 was given a time limit of one minute. The translator was given a time limit of 10 minutes.

Table 1 summarizes the results. Each row shows, from left to right, the total number of unsatisfiable cases in this family, the number of cases that CVC3 successfully proved to be unsatisfiable, the total and average time spent by CVC3 for the family, the number of cases for which the proof translation succeeded, and the total and average time for the translation. As seen from the table, the amount of time spent for the proof translation in HOL Light varies significantly, from less than 10 times the amount of time spent in CVC3 to nearly 100 times the amount of time spent in CVC3. There are several cases in the `simplify2` family for which the proof translation fails. Some of these time out, and others contain proof rules not yet fully supported by the translator.

The table does not list results for the "boogie" family of benchmarks because complete results are not yet available. However, we did find something very interesting. CVC3 reported unsatisfiable for two cases in labeled as satisfiable. We were able to translate these proofs successfully, meaning that these two benchmarks were incorrectly labeled.

category		CVC3			HOL Translation		
	Cases	Proved	Total	Ave	Proved	Total	Ave
simplify	833	833	814.30	0.98	833	16249.33	19.51
simplify2	2329	2306	2408.95	1.11	2164	19153.34	8.85
Burns	14	14	0.30	0.02	14	19.37	1.38
Ricart	14	13	0.89	0.07	13	228.80	17.60
piVC	41	41	4.92	0.12	41	59.40	1.45

Table 1: Results on a selection of AUFLIA benchmarks

6 Related work

Moskal [8] proposed a rewriting system for proof checking of SMT solvers. His implementation emphasizes speed and compactness. Our system, while slower, emphasize trustworthiness. Our system ultimately provides a very strong guarantee of correctness, and essentially none of the code of the SMT solver or the proof translator need be trusted.

Our own previous work in this direction [7] described our initial efforts to combine CVC Lite and HOL Light. That work emphasized using CVC Lite as an external decision procedure for HOL Light. Here, we emphasize HOL Light's value as a proof-checker for CVC3. We also give a more detailed description of the system and give results on the SMT-LIB benchmarks.

7 Conclusion

CVC3 is an SMT solver with many diverse proof rules. The proof rules were designed to enable high-performance and to be convenient for CVC3 developers. This is at odds with the goal of having a small and simple trusted core. The problem can be alleviated by translating proofs into another prover that does have a small trusted core, in this case HOL Light. We plan to continue our efforts to certify as much of the SMT-LIB library as possible. We also plan to continue improving and tuning the translator.

8 Thanks

We would like to thank John Harrison for his help with HOL Light, and we would also like to thank Sean McLaughlin for writing the first version of the proof translator.

References

[1] `http://www.cl.cam.ac.uk/~jrh13/hol-light/`.

[2] C. Barrett and S. Berezin. A proof-producing boolean search engine. In *Proceedings of the 1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '03)*, July 2003. Miami, Florida.

[3] C. Barrett, S. Ranise, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2008.

[4] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[5] J. Harrison. Hol light: A tutorial introduction. In M. K. Srivas and A. J. Camilleri, editors, *FMCAD*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer, 1996.

[6] J. Harrison. Towards self-verification of hol light. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.

[7] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In A. Armando and A. Cimatti, editors, *Proceedings of the 3rd Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, Jan. 2006. Edinburgh, Scotland.

[8] M. Moskal. Rocket-fast proof checking for smt solvers. In K. Jesen and A. Podelski, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 2008.

[9] A. Stump and D. Dill. Faster proof checking in the edinburgh logical framework. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction (CADE '02)*, volume 2392 of *Lecture Notes in Computer Science*, pages 185–222. Springer, 2002.

[10] T. Weber. Efficiently checking propositional resolution proofs in isabelle/hol. volume 212 of *CEUR Workshop Proceedings*, 2006.

[11] T. Weber. Integrating a SAT solver with an LCF-style theorem prover. In A. Armando and A. Cimatti, editors, *Proceedings of the 3rd Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 67–78. Elsevier, Jan. 2006. Edinburgh, Scotland.