Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 144 (2006) 15–26
www.elsevier.com/locate/entcs

# Tool Building Requirements for an API to First-Order Solvers

Jim Grundy[1]

*Strategic CAD Labs, Intel Corporation, Hillsboro, OR, USA*

Tom Melham[2]

*Computing Laboratory, Oxford University, Oxford, England*

Sava Krstić[1]

*Strategic CAD Labs, Intel Corporation, Hillsboro, OR, USA*

Sean McLaughlin[3]

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA*

**Abstract**

Effective formal verification tools require that robust implementations of automatic procedures for first-order logic and satisfiability modulo theories be integrated into expressive interactive frameworks for logical deduction, such as higher-order logic theorem provers. This paper states some pragmatic requirements for implementations of decision procedures that make them well-suited to integration into such frameworks. The aim is to open a dialogue with the designers of decision procedure software that will lead to greater and easier uptake of their implementations by verification users.

*Keywords:* abstract syntax, CVC Lite, decision procedure, first-order logic, forte, higher-order logic, *reFLect*, tool integration

[1] Email: {jgrundy,skrstic}@ichips.intel.com
[2] Email: tom.melham@comlab.ox.ac.uk
[3] Email: seanmcl@cmu.edu

1571-0661/$ – see front matter 
doi:10.1016/j.entcs.2005.12.003

# 1 Introduction

A range of proof technologies for formal verification of high-performance processor designs are employed at Intel Corporation [13]. The verification framework, currently known as *forte3*, [4] is built around the *reFLect* functional programming language [7]. Several verification technologies are integrated into the *forte3* framework—including BDDs, SAT procedures, model checkers, and a higher-order logic theorem prover similar to HOL [6]. The *reFLect* language is used to coordinate deployment of the integrated tools to solve large and challenging verification problems, as well as to explore specifications and hardware designs by symbolic execution. It is also the underlying term language of the higher-order logic theorem prover.

There are many applications for first-order reasoning in this framework, and a few experiments with integrating decision procedures into *reFLect* have already been done. Our aim is to provide support for first-order proof automation (modulo theories) within the system's higher-order logic theorem prover. For example, we have integrated an implementation of Harrison's model elimination procedure [8] and have found it to be an invaluable workhorse for first-order proof support. We have also integrated CVC Lite [2] and used it in an experimental project.

Based on this experience, and without claiming originality, we give an account of some pragmatic requirements for smooth integration of decision procedure implementations into a system like *reFLect*. We begin by examining our application context: the nature of the *forte3* tool and how we use first-order solvers within it. We then comment on the design of first-order languages for communication with decision procedure components. We then present our requirements for supporting an in-memory integrations through an *application program interface* (API).

We are aware that some of our requirements are competing, and perhaps seemingly contradictory. Some of them are therefore stated more tentatively as 'desires'. But we hope that by raising these issues we might open a dialogue with the designers of decision procedure software that will lead to greater and easier uptake of their implementations by verification users. We also offer these observations as a contribution to discussion of SMT-LIB [12], an initiative to produce a standard language for benchmark problems in first-order satisfiability modulo theories.

[4] *forte3* may be downloaded for academic use.
http://www.intel.com/software/products/opensource/tools1/verification

## 2 Application Context

We are interested in applying solvers and simplifiers for first-order logic in the context of an interactive reasoning tool for higher-order logic. In our case, the tool is the theorem prover developed at Intel, but the general scenario is a common one. Commercially valuable verifications exceed the reach of automatic tools, so an interactive proof construction framework and an expressive, general logic are required. Just as importantly, however, these verification efforts are too large to be done entirely manually, so automated reasoning applied to selected subproblems is also essential.

In such a setting, some variant of typed higher-order logic is commonly adopted for the interactive reasoning tool because it helps to capture requirements and designs abstractly. The logic's type system facilitates data modeling, and type checking prevents trivial errors. On the other hand, the most successful automated reasoning tools are for first-order logic. Fortunately, verification problems are typically expressed in the first-order fragment of higher-order logic or reduce to first-order subproblems. And even higher-order properties can sometimes be proved by first-order reasoning when appropriately encoded [9]. This clearly leads to the desire to integrate first-order solvers into interactive higher-order reasoning tools.

### *2.1 In-Memory Integrations*

In our interactive proof system, a user may invoke a solver to attempt a proof of a formula manually reduced to what appears to be a solvable form. For such applications, a command to check the validity of a formula could be written to a file in the concrete syntax of an external tool, which could then be invoked on that input. This method of communication, however, is less tenable when we consider more automated and speed sensitive applications.

For example, the rewriting subsystem of our theorem prover traverses expressions, searching for sites where it may apply a rewrite from a collection of conditional equations supplied by the user. Having found a match with the left-hand side of an equation, it attempts to check the condition under the assumptions it has accumulated from the context of the matching site. If the condition is solved the rewrite is performed. Either way, the process continues until some user specified criteria are met, usually when no more rewrites apply. This kind of application can produce many calls to a solver, with a high degree of commonality in sub-terms and formulas from call to call. File based communication does not seem suitable here.

**Requirement 1** *We need an in-memory API to terms and formulas in the native data-structures of a tool for first-order logic. The API must also allow*

*us to invoke solving and simplification routines on the terms and formulas so constructed.*

### *2.2 State in the Interface*

Interactive reasoning tools are usually implemented in functional programming languages, and those for higher-order logic invariably in some dialect of ML. Our higher-order proof tool, for example, is implemented in *reFLect*, a dialect of ML with reflection features similar to those of LISP. Functional languages commonly employ an eager evaluation strategy and support both purely functional and stateful programming. But in practice, the simplicity afforded by the functional style gives the tool designers a bias towards interfaces that minimize the number of operations modifying visible state. Our preference for functional style is even stronger in *reFLect* because it uses lazy evaluation, making the order in which state changes occur hard to predict.

**Desire 2** *The API to a first-order tool should minimize the number of operations modifying the visible state. Where applicable, it should be possible to undo a state modifying operation.*

If presented with a state-modifying API we can always restore transparency to our code by providing a monadic interface to it [14], as we do in our integration with CVC Lite. But the overhead of the monadic style only emphasizes our preference for an API that minimizes state modifications.

If an API provides a facility to undo state changing operations then we can create pure functions that perform state updates internally, only to revoke them before returning their result. Consider the contextual rewriting scenario described in the previous section. As the rewriter traverses a term, it passes through contexts where local assumptions may be used to solve the conditions of a rewrite. The CVC Lite API includes state-modifying operations to add an assumption to the proof state and to roll-back to an earlier state. As the rewriter enters a context, it can add local assumptions to the CVC Lite proof state, and drop them again as it exits. There are contexts within contexts, of course, but the assumptions exhibit a last-in-first-out behavior that fits with the CVC Lite interface and minimizes the number of operations that need to be done during rewriting. When rewriting terminates, CVC Lite is returned to its original state, and so the rewriter may be treated as a pure function.

## 3 Logic Design Issues

Most well-known decision procedures and the available SMT solvers implementing them are based on first-order logic. Consequently, the SMT-LIB

standard language is many-sorted first-order logic with equality, enhanced with a handful of non-standard constructs such as conditionals and local lets. On the other hand, the logic in which hardware and its properties are captured for formal verification at Intel (and elsewhere) is a variant of higher-order logic with a parametrically polymorphic type system. The need for a smooth and sound transition between the two logical systems raises some issues, of which we mention just two.

### *3.1 Typing and Polymorphism*

Hurd has proposed a translation scheme from higher-order logic to first-order logic that can represent any higher-order expression [9]. The scheme compiles $\lambda$-terms into combinatory form and replaces all function applications by applications of a binary uninterpreted function $\bullet$. Take, for example, the term '$n + m$'. In higher-order logic, the two-place function $+$ is curried and has higher-order type $int \to (int \to int)$, so our term is really '$(+\ n)\ m$', and so gets translated into the first-order term '$\bullet\ (\bullet\ (+, n), m)$'.

In general, Hurd's scheme treats all functions, except equality, as uninterpreted. In reality, we need a mix of interpreted and uninterpreted translation: when the first-order prover has a suitable interpretation for a function symbol (like $+$ in the example above), we should use a direct translation; otherwise, we translate into $\bullet$-terms.

Type information is simply dropped in this translation scheme, making the approach generally unsound. The danger is that the first-order tool may find a proof for the translated formula using an expression untypable in higher-order logic. Hurd guards against this by attempting to replay any proof discovered for the translated formula in the higher-order logic. He estimates that proofs fail to replay in fewer than one in a hundred attempts. He has also explored a more elaborate translation, in which type information is encoded in first-order logic. This takes care of any logical problems, but at the cost of larger first-order terms.

Hurd's encoding assumes an untyped first-order logic. This is convenient, because the operator '$\bullet$' can then be used for translation of function application terms of arbitrary types. On the other hand, if the target first-order logic is monomorphically typed (as is the many-sorted SMT-LIB), then there will be problems representing $\bullet$ as a single operator. We would need to introduce a family of operators, one for each distinct function type that occurs in the expression to be translated. Note, however, that if our first-order logic came with parametric polymorphism, it would be possible to encode the '$\bullet$' operator and there would be no need to replay proofs or encode types to ensure soundness.

```
let RND pc rc s sgf =
  let L = Lsb pc sgf in
  let G = Guard pc sgf in
  let RS = RoundS pc sgf in
  let rbit = (rc '=' TO_ZERO)     => F
           | (rc '=' TO_POS_INF) => ((NOT s) AND (G OR RS))
           | (rc '=' TO_NEG_INF) => (s AND (G OR RS))
           | (rc '=' TO_NEAREST) => (RS => G | (L AND G))
           | F in
  Result rbit pc sgf;
```

Fig. 1. Local lets give increased clarity in specifications.

**Requirement 3** *The input logic of first-order prover components should either be untyped or, preferably, support parametric polymorphism.*

There is also a simpler motivation for this requirement. An industrial verification problem will typically have several structurally similar data types in play at once—multiple kinds of lists (integer lists, Boolean lists) or different flavors of arrays. Encoding all these as special cases into a monomorphically typed logic is inconvenient and can be avoided with polymorphism. CVC Lite allows some interpreted functions, e.g. array operations, to behave as if parametrically typed, presumably to accommodate such practical concerns. But we would prefer general support for such types.

### *3.2 Local Lets*

In higher-order logic, formulas are just terms of boolean type, not a distinct syntactic category, as in the first-order logic. A single let construct like '**let** $v = t$ **in** $u$', where $t$ and $u$ range over terms suffices for all local variable definitions.

We wish to have such a construct in the first-order logic we are translating into, for at least two reasons. First, it provides a concise form of expression, which is important in any tool that allows user interaction in the reasoning process. With *reFLect*, we make liberal use of local lets for readability, as in the specification given in Figure 1 of the float-point rounding operation used in an actual processor verification [1]. Second, having local lets facilitates space-saving sharing in off-line (textual) representations of otherwise large formulas.

**Desire 4** *The logic should include let constructs for introducing local terms and local formulas in both terms and formulas.*

With a similar motivation, SMT-LIB has (separate) let constructs for introducing local terms and formulas in formulas. However, there is no construct to introduce locally-defined terms in terms, so translation from the freer syntax of higher-order logic into this language might be awkward. On the other hand, adding two further let constructs to SMT-LIB, for terms in terms and

terms in formulas, seems rather messy. We briefly consider an alternative in section 4.3.

# 4 Logical API Requirements

In this section we give some guidance on the design of an API for constructing and manipulating terms and formulas, and consider consequences for the description of the logic itself.

## *4.1 Constructing Logical Structures*

Because of their different strengths, multiple first-order solvers—additional to CVC Lite—will be considered for integration with *forte3*. We expect SMT-COMP may make the field resemble that of SAT solvers, with rapid arrival of tools and shifts in relative performance of existing ones. This is exciting but worrying; we relish the prospect of increasingly powerful tools to integrate, but what if they all have different APIs?

The SMT-LIB standard addresses this by providing a common textual format for communicating with first-order tools. We would like a standard to address the need for a common API for in-memory communication. The internal data-structures of a solver will vary from tool to tool, and will change as people explore optimizations, but the abstract syntax of the logic implemented should remain the same. It is therefore the best guide for a standard API.

**Requirement 5** *We need a standard API for in-memory communication with first-order solvers. The API for constructing terms and formulas should follow the abstract syntax of the logic.*

Consider the experience of developers from the HOL family of theorem provers [6]. The abstract syntax of the logic implemented by the various versions of HOL—including HOL88, HOL90, HOL98, HOL4, HOL-Light and Proof Power—is identical, and essentially the same as in Church's original formalization of the simple theory of types. Those systems have all retained the same API for manipulating terms, based on the common abstract syntax of those terms. They have done this despite being implemented in four different dialects of ML (and some LISP in the case of HOL88) and exploring various underlying optimizations to the representation of terms, including de Bruijn vs. name carrying representations and a delayed substitution mechanism.

In contrast to this, the API for manipulating terms in CVC Lite is influenced not only by the abstract syntax of the logic, but also by the theories and decision procedures used for reasoning in that logic. The way a function

application is constructed, for example, depends on whether the function is a tuple construction, update or selection; a record construction, update or selection, an uninterpreted function, or one of the primitive interpreted functions. As a consequence, the CVC Lite term manipulation API undergoes significant change between versions of the tool, and is unlikely to be duplicated in another first-order logic tool.

### *4.2 Inspecting Logical Structures*

We are particularly interested in tools that can act as simplification engines as well as solvers. Our integration with CVC Lite packages the CVC Lite simplifier as a *conversion*, a general form of trusted expression transformation that users of our theorem prover can pass into our rewrite engine [11]. The user can simplify a goal by replacing every subexpression that CVC Lite can simplify with the result of the simplification.

To implement simplifications, we traverse an expression in our logic and construct a translation in the logic of the first-order tool, invoke the first-order simplifier, and traverse the simplified expression to translate the result back. To translate back, we must be able to distinguish each syntactic form of the terms and formulas of the first order tool, and access any sub-terms and formulas of those forms.

**Requirement 6** *The API should also support the discrimination and destruction of terms and formulas.*

The interface should follow the style described by McCarthy for the construction and manipulation of abstract syntax trees [10]. McCarthy's approach uses functions to construct each syntactic form of expression, discriminate expressions based on their syntactic form, and destruct an expression of a particular syntactic form into its components. Developed with the advent of the functional programming in LISP, McCarthy's approach to abstract syntax fits cleanly with the functional meta-languages used to implement interactive theorem provers. It can fit just as well into the object-oriented style often used to implement first-order solvers, with subclasses of an expression class used for each syntactic form, and dynamic casting used to discriminate syntactic forms and then access their components.

### *4.3 A Simple Abstract Syntax*

Syntactic complications can be left to a concrete syntax and its translation into the abstract syntax. This guiding principle for designing an abstract syntax is especially important for us as tool integrators. To construct and manipulate integrated data structure values with ease, we want the simplest

possible interface to them, wanting at the same time the interfaces to follow the abstract syntax of the logic.

**Desire 7** *The abstract syntax of the logic should be as simple as possible.*

The abstract syntax proposed for SMT-LIB strives for simplicity, but perhaps could be simpler still. In particular, the '**if _ then _ else _**' formula construction does not seem to offer value over adding a three-place operation with this meaning to the family of connectives. In some regards the CVC Lite abstract syntax is simpler than that proposed for SMT-LIB, and more closely resembles the higher-order logic of our host tool by not making a distinction between formulas and terms. This neatly removes the need for a multiplicity of **let** constructions as considered in section 3.2.

## 5 Data API Requirements

In this section we discuss some low-level operations on terms and formulas required for in-memory integrations to support the more obvious high-level operations discussed until now.

### *5.1 Memory Management API Requirements*

Consider again using an integrated first-order tool to discharge conditions in our rewriter. Multiple calls to the solver exhibit a high degree of commonality of sub-terms and formulas. To avoid rebuilding common structures, we must retain references to them between calls. This demands some exposure of the memory management strategy of the first-order tool.

**Requirement 8** *The API to the first-order tool must allow us to communicate that we are retaining a reference to a term or formula, and also when we cease to retain it.*

This requirement has a number of implications depending upon the garbage collection strategy employed by the first-order tool. The simplest case occurs when the tool uses a reference counting scheme. We need only have the ability to increment and decrement reference counts. If references are communicated via smart pointers encapsulating reference counting then the API is particularly easy to use. The CVC Lite API works in this way.

Several propositional tools integrated with *reFLect* use mark-and-sweep garbage collection. Such tools require APIs that expose, separately, calls to initiate the mark and sweep operations, allowing us to mark the structures *reFLect* references as well as the structures the tool references before sweeping.

We assume that mark-and-sweep will be invoked by the *reFLect* based integration framework. Integrated tools must provide an API to register a method to inform *reFLect* that it *desires* a garbage collection. If an integrated tool may *require* immediate garbage collection, then it must provide routines through which we can register methods for it initiate the mark-and-sweep operations of *reFLect*.

### *5.2 Data Indexing API Requirements*

When integrating first-order tools with support for simplification, we translate higher-order terms into first-order terms and formulas, and translate back after simplification. On the way back, we wish to avoid re-translating unchanged or common sub-terms and formulas. We also wish to generate sharing. We can do this via tables relating higher-order terms and their first-order translations. To translate in either direction, we lookup the result of any previous translation, only if there was none do we perform the translation and store the result for reuse. We do this recursively for sub-terms and formulas.

**Requirement 9** *We require syntactic equality test and hashing functions for terms and formulas. Hashes should not be recomputed for shared structures.*

A lookup-based translation can minimize both translation time and the size of the translated value. The CVC Lite system provides functions to hash terms and formulas, which it uses to ensure maximal sharing. But, relying on a feature like this alone will not ensure minimal re-translation effort. Even minimal re-translation will not ensure minimal translation time. The time to compute a hash and the time to perform a translation are both proportional to the size of the structure to be translated. No saving is realized unless hashing is faster than translation. To achieve a saving, the first-order tool must store hash codes with terms and formulas at their time of construction, or cache hash codes so they are not recomputed for shared structures.

It may be useful to consider including arbitrary total ordering functions for terms and formulas in the API, allowing them to be used as keys in tree based structures. Of course, a hash trivially generalizes this since we can compare the hash codes, but if hashes are not precomputed then it may be faster to just test two terms for their relative ordering.

**Desire 10** *We desire total ordering functions on the syntax of terms and formulas.*

Hashing and comparison functions may identify some semantically equal terms and formulas, provided the functions behave consistently. For example, if alpha-equivalent formulas are considered identical, then they should share

the same hash. While a tool may identify some semantically equal terms and formulas, it should not expand the collections of such structures it identifies as it makes deductions. Doing so would invalidate the translation information accumulated.

**Requirement 11** *The results of syntax-based hashing and comparison functions should not change.*

## 6 Conclusion

We have stated some pragmatic requirements for integrating first-order solvers into verification tools. We write from the perspective of developing verification tools at Intel, concentrating on integrating first-order solvers into a higher-order logic theorem prover written in a functional language. But the issues raised apply to more generally; many were encountered and addressed in the design of the PROSPER toolkit [5], an integration framework for both in-memory and distributed communication of higher-order logic data. Others have had similar experiences, starting with Boyer and Moore [3] and more recently with PVS [4]. This paper serves to emphasize and offer precise guidance in line with this other work.

Our requirements have focused on the need for a suitable API to the data structures representing a first-order language for solvers. Ideally, the API to be part of a standard such as SMT-LIB. One could imagine the standard including a parser that uses the API and can be linked against a proof tool supporting it. Entries to a competition like SMT-COMP would be submitted as libraries to be linked against a stock benchmark parser and test bench.

## Acknowledgement

We thank Jeremy Casas and John O'Leary for their thoughts on this material.

## References

[1] Mark D. Aagaard, Robert B. Jones, Thomas F. Melham, John W. O'Leary, and Carl-Johan H. Seger. A methodology for large-scale hardware verification. In Warren A. Hunt, Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design: 3rd International Conference, FMCAD*, volume 1954 of *LNCS*, pages 263–282. Springer, 2000.

[2] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification: 16th International Conference, CAV*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.

[3] Robert S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In J. E. Hayes, Donald Michie, and J. Richards, editors, *Machine Intelligence: 11th Conference*, pages 83–124. Oxford, 1985.

[4] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, and Natarajan Shankar. Integrating verification components: The interface is the message. In *Monterey Workshop on Software Engineering Tools: Compatibility and Integration*, 2004.

[5] Louise A. Dennis, Graham Collins, Michael Norrish, Richard J. Boulton, Konrad Slind, and Thomas F. Melham. The PROSPER toolkit. *International Journal on Software Tools for Technology Transfer*, 4(2):189–210, 2003.

[6] Michael J. C. Gordon and Thomas F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge, 1993.

[7] Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2), 2006.

[8] John Harrison. Optimizing proof search in model elimination. In Michael A. McRobbie and John K. Slaney, editors, *Automated Deduction: 13th International Conference, CADE*, volume 1104 of *LNCS*, pages 313–327. Springer, 1996.

[9] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number CP212448 in NASA Technical Report, pages 56–68, 2003.

[10] John McCarthy. Towards a mathematical science of computation. In Cicely M. Popplewell, editor, *Information Processing: Congress*, pages 21–29. Elsevier, 1963.

[11] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[12] Silvio Ranise and Cesare Tinelli. *The SMT-LIB Standard*, 1.1 edition, 2005.

[13] Tom Schubert. High level formal verification of next-generation microprocessors. In *Design Automation: 40th Conference, DAC*, pages 1–6. ACM, 2003.

[14] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52, 1995.