Integrating SVC and HOL with the Prosper Toolkit*

Alan Stevenson[1] and Louise A. Dennis[2] **

[1] Department of Computing Science, University of Glasgow, G12 8QQ, UK
[2] Division of Informatics, University of Edinburgh, EH1 1HN, UK louised@dai.ed.ac.uk

Abstract. We describe an integration of the SVC decision procedure with the HOL theorem prover. This integration was achieved using the PROSPER toolkit. The SVC decision procedure operates on rational numbers, an axiomatic theory for which was provided in HOL. The decision procedure also returns counterexamples and a framework has been devised for handling counterexamples in a HOL setting.

1 Introduction

The PROSPER project[1] is researching and developing a toolkit [4] that allows an expert to easily and flexibly assemble *proof engines* from existing tools to provide embedded formal reasoning support inside applications. The ultimate goal is to make the reasoning and proof support invisible to the end-user—or at least, more realistically, to incorporate it securely within the interface and style of interaction to which they are already accustomed. One of the main facets of the project is the use of plugin technology in order to prevent the continual re-implementation of existing techniques.

This paper describes the construction of a PROSPER plugin based upon the Stanford Validity Checker (SVC)[14]. This work has included providing an axiomatic theory for rationals in HOL and a framework for handling counter-examples.

In §2 and §3 we give overviews of the PROSPER toolkit and SVC respectively. In §4 we discuss our integration of the two systems including an overview of the theory of rational numbers and the framework for counterexamples. In §5 we describe the results of experimental evaluation of the system. In §6 we look at related work and in §7 we discuss further work needed on the plugin.

2 The Prosper Toolkit

A central part of PROSPER's vision is the idea of a proof engine—a custom built verification engine which can be operated by another program through an Application Programming Interface (API). A proof engine can be built by a system developer using the toolkit provided by the project. A proof engine is based upon the functionality of a theorem prover with additional capabilities provided by 'plugins' formed from existing, off-the-shelf, tools. The toolkit includes a set of libraries based on a language-independent specification for communication between components of a final system. The theorem prover's command language is treated as a kind of scripting or glue language for managing plugin components and orchestrating the proofs.

The central component is based on a theorem prover because this comes with ready made concepts of term, theorem, and goal, which are important for managing verifications. A side benefit is that all the functionality in the theorem prover (libraries of procedures, tactics, logical theories, etc.) becomes available to a developer for inclusion in their custom proof engine. This does not prevent theorem proving being very lightweight if desired.

* The authors would like to thank Tom Melham for his help with the research reported here.
** Corresponding Author
[1] http://www.dcs.gla.ac.uk/prosper

A toolkit has been implemented based around HOL98, a modern descendent of the HOL theorem prover [5]. The toolkit provides several plugin components based on external tools which offer APIs to a proof engine. It also provides support to enable developers of other verification tools to offer them as PROSPER plugins.

The application, proof engine and plugins act as separate components in the final system (Figure 1). In the first prototype they are also separate processes. Communication between them is treated in a

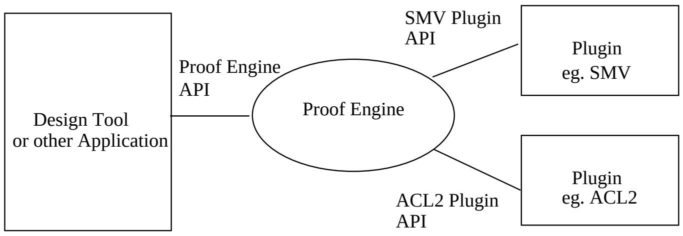

Fig. 1. A system built with the PROSPER toolkit

uniform manner specified by the PROSPER *Integration Interface.*

2.1 The Prosper Integration Interface

A major part of the PROSPER methodology is the PROSPER Integration Interface (PII), a language-independent specification of communication for verification. This specification is currently implemented in several languages (C, C++, ML, Java, Ada, λProlog and Python) allowing components written in these languages to be used together.

The PII consists of several parts. The first is a datatype, called *interface data*, for all data transferred between an application and a proof engine and between a proof engine and its plugins. A major part of the datatype is the language of higher order logic used by HOL and so any formula expressible in higher order logic can be passed between components. Many plugins operate with logical data that is either already a subset of higher order logic (e.g. predicate calculus and propositional logic) or embeddable in it (e.g. CTL). The second part consists of a datatype for the results of remote function calls and support for installing and calling procedures in an API. There are also parts for managing low level communication, which are largely invisible to an application developer.

The developer of a verification tool can adapt it so that it can be used as a PROSPER plugin. A plugin developer programs both in ML and in the plugin's own implementation language. The developer will place chosen entrypoints to the plugin into an API database. In the plugin's implementation language they will translate any arguments needed by these functions into interface data. In the theorem prover's command language they will need to unpackage these entrypoints again so they present themselves as language-specific bindings in that language (ML). In particular any additional theories required (i.e. an embedding of the logic used by the plugin into higher order logic) should be provided by the plugin developer.

3 SVC

SVC stands for the Stanford Validity Checker. This tool has been in development at Stanford for several years, and is used both in research, and as a formal hardware verification tool, as seen in such papers as [8, 1]. SVC allows users to check formulae based on a subset of first order logic and boasts of its efficient and automatic decision procedures.

The logic for these decision procedures includes booleans, uninterpreted functions and linear arithmetic. There are also interpreted functions such as array operations, bit vectors and parts of linear arithmetic.

SVC provides a variety of commands to the user, from a traditional command-line user interface. At its simplest level, proof is carried out through use of the `check_valid` command, which is given an expression in SVC logic, and will return an answer based on the validity of that expression, and possibly a counter-example, if the expression is found to be invalid.

SVC also provides numerous other commands which aid in proof. These can relate to various command flags, which affect the way SVC works, or manipulate SVC's *context* which stores proven results.

4 Integrating SVC and HOL

We attempted to integrate SVC and HOL following, as closely as possible, the methodology set down by [4]. This methodology outlined the following set of tasks:

1. Provide an implementation of the PII in the implementation language of SVC.
2. Identify entry points into SVC to make accessible to HOL.
3. Provide a translation between interface data and the formula syntax of SVC.
4. Provide ML language bindings in HOL to make the operation of SVC appear as natural as possible to a HOL user.

We have provided a link up based on this methodology [2].

4.1 Implementing the PII

SVC is written in C++. An implementation of the PII already existed in C and it was a relatively trivial exercise to adapt this to C++, in fact much of the relevant code was simply imported directly into C++ without any change or alteration at all. All the work required was standard reworking of C code to allow it to be called from C++.

4.2 Entry Points into SVC

SVC has a number of access points through its command line interface. Ideally we would have liked to have provided most of these to HOL users. However, given time constraints, we chose to provide only `check_valid` and `reset`. `check_valid` is SVC's primary command for proof, to which the user supplies an expression, and the system will return a result based on the validity of that expression. This can be either VALID, INVALID or INVALID with a counter-example. `check_valid` will do this by setting up a context for the current proof, in which the actual validation takes place, and the results will be stored, which can then be used in further proofs.

`reset` is the command used to clear the current context, when it is felt that previous results will no longer be needed, allowing the user to start with a "clean" SVC session.

These entry points were placed in a Prosper API for SVC.

4.3 Translation between interface data and SVC's formula syntax

SVC has a large formula syntax for formulas expressed in arithmetic, records and bit-vectors. We chose for a first implementation only to cater for a subset of this language which corresponded to the arithmetical expressions.

[2] `http://www.dcs.gla.ac.uk/~stevenat/svc_plugin.tar.gz`

The translation from interface data into formula syntax thus depended upon the representation of rational numbers in interface data. To determine this we needed to know how the rationals were represented in HOL since that would determine their presentation in interface data (the translation between HOL terms and interface data is already decided and implemented and acts as the gold standard for other translations). Our theory of rational numbers is described below.

The translation on the C++ side of the interface is performed simply, by taking a PII term, and identifying the kind of term we have (equality, variable, application etc). The PII provides similar methods to HOL for the decomposition of terms, and using these functions we extract the necessary information from the term (which can be identified further by checking constituent parts for known function names, such as those defined in the rational number theory) such as function arguments, variable names and numerical constants. With this information, the appropriate SVC expression is then built up, using constructors provided by SVC. For example, specific objects are provided for numerical expressions, function expressions and addition expressions. These all stem from a generic expression object within SVC.

The Theory of Rationals SVC assumes that all numbers appearing in arithmetical expressions are rational numbers. There is no full theory of rationals in HOL although John Harrison provided a theory of *half-rationals* to support his construction of a theory of real numbers [15]. It is preferable to provide *definitional* theories for HOL (i.e. constructing a new theory from an old one by definitions and then proving the axioms of the new theory). However it is possible and in many cases simpler to provide an *axiomatic* theory which we chose to do in this case. Our theory of rationals is based upon the description in *Axiomatic Set Theory* [16].

The theory provides a new HOL type, *rat*, with which to represent rational numbers. It provides the usual constants representing addition, multiplication and inequalities, and also defines the axioms for the rational numbers. The main problems came with the representation of numbers with this rational theory. This is solved by providing three functions:

1. `rat` - takes in something of type `num` (a HOL type for natural numbers) and returns a `rat`. This allows us to parse numerical constants as rational numbers.
2. `rneg` - takes in something of type `rat` and returns its negation. This function allowed us to define negative rational numbers from within HOL.
3. `rdiv` - Taking in two numbers of type `rat` this constant is defined to allow us to represent division of rational numbers. This is especially important as it allows us to define fractions.

By defining the axioms and functions in this way, it is hoped that this will allow for possible expansion of the theory, by implementing the functions, and by properly proving the axioms.

With this in place we were able to handle a reasonable subset of SVC's input language. We have a representation within HOL with which users can specify boolean and rational number predicates, including most of the usual logical operators (AND, OR, NOT) as well as rational functions (negation, addition, multiplication and inequalities) and fractional numbers. These statements allow us to represent a large part of SVC syntax, although at current standing, bit-vectors and records cannot be represented within HOL, due to the lack of a suitable HOL theory, although there is nothing to suggest that this could not change in future years, allowing the expansion of our theory to encompass more, if not all, of the SVC syntax.

4.4 ML language bindings for the SVC API

Recall that we had placed two SVC functions in its PROSPER API. We now needed to present these in some way to HOL users. We implemented two HOL entry points based upon `check_valid`. The first (also called `check_valid`) is of type `Term -> Term`. The user supplies a term which represents a boolean statement. This statement can be composed of boolean and rational expressions as shown above. This term will then be sent to the plugin using the PII where an SVC context is created, and validity checks performed. The user will then get another term as a result (assuming an error has not ocurred) which will be one of `True`, `False` or a term representing the counter-example calculated by SVC (This counter-example is naturally presented in the same syntax as the initial term provided by the user).

Check valid provides an SVC centric entrypoint which is not of a great deal of use for proof in HOL unless the formula sent to SVC is a subgoal of a proof in its own right. A more HOL centric entrypoint is `svcprove` which will return a HOL theorem which can then be saved and recalled when the result is needed at a later date. This function is essentially a wrapper for `check_valid` which parses the returned result, and constructs the HOL theorem based on whether the result was `True` or `False`. If the result is a counter-example, then the original Term is equated to False, and the counter-example is provided as an assumption to the theorem.

A major part of HOL is the exploitation of conversions. "A conversion is a rule that maps a term to a theorem expressing the equality of that term to some other term" [5]. They are particularly useful for providing rewrite rules to rewriting tactics. `SVCprove` is a conversion which means it can be naturally and easily used in the course of HOL proof, in particular it can be exploited by rewriting techniques.

Counter Examples SVC can return counter examples for non-theorems. Counter examples are not of direct use in HOL proofs however they can be very useful in "debugging" non theorems and in guiding proof. Hence counterexamples can provide useful information to a HOL user or potentially to the user of a design tool which incorporates PROSPER style verification capabilities. We needed to provide a framework in which counterexamples could be returned to a HOL user and possibly to a design tool beyond.

SVC provides counter-examples in the form of formulae representing a set of circumstanes under which the original expression will fail. These counter-examples are stored within the context and are accessed through the context's `Splitter` functions. These functions allow you to access the most recent splitters (counter-examples) and will provide them as SVC expressions. These expressions are then translated into PII terms in a similar way to which PII terms become SVC expressions (in that the specific type of expression is identified, and then the methods SVC provides for decomposing these expressions are employed to retrieve the appropriate information, which is then supplied as arguments to PII methods for constructing terms).

Counter-examples are returned as assumptions to a HOL theorem equating the goal expression to false. This was felt to be a natural way of viewing the available information, as well as supplying it to the user in a way which allows it to be stored for later use (as opposed to simply storing the truth of the original term, and having the counter-example separate).

This is a generic approach. As well as the advantages stated above (the ability to store counter-example and theorem as one statement for future reference, as well as human readability – `if` *counter-example* `then` *Term* `equals False`) HOL provides simple functions for the decomposition of these expressions, making it easy to extract the counter-example or leave it in as necessary. If the counter-example and theorem were separate there could be difficulties combining them later on (e.g. if the counter-example is kept in a type that cannot be easily stored for future use).

5 Evaluation

Ideally we would have liked to have tested the efficiency of the SVC plugin against an internal HOL decision procedure. However, no appropriate decision procedures exist for rational numbers. As a result we can only note that it is now possible to work with rationals in HOL which it wasn't previously, and that with relatively little work we were able to give access to a powerful decision procedure in that domain without having to implement it from scratch. In this way we have provided a new theory in HOL with sophisticated theorem proving support with relatively little effort.

Below are a variety of test examples, which show the usage of the rationals theory, as well as the SVC commands we have provided.

5.1 A Simple Example

This is a simple example showing a trivial use of `check_valid`.

```
- check_valid (Term ‘(x = rat 6)‘);
> val it = ‘(x = rat 6) = T‘ : Term.term
```

5.2 Addition and Multiplication

This next example uses `svcprove` to decide an equality expression involing addition and multiplication.

```
- svcprove (Term ‘((rat 3) rtimes m) rplus ((rat 4) rtimes n) = n rplus n rplus n
rplus n rplus (m rtimes (rat 3))‘);
> val it =
[]
|- (rat 3 rtimes m rplus rat 4 rtimes n =  n rplus n rplus n rplus n rplus m rtimes
rat 3) =  T
: Thm.thm
```

5.3 Inequalities and negation

This example shows a proof involving inequalities, as well as the `rneg` function:

```
- svcprove(Term ‘rneg (rat 4) rless rat 3‘);
> val it = [] |- rneg (rat 4) rless rat 3 = T : Thm.thm
```

5.4 Division by Zero

This final example shows the result of trying to divide by zero:

```
- check_valid(Term ‘rdiv (rat 1) (rat 0) = x‘);
! Uncaught exception:
! Fail "Term is not an SVC term"
```

6 Related Work

Several decision procedures have been converted into Prosper plugins, most notably the SMV model checker [9], the first order logic reasoner Gandalf [13, 7], the ACL2[2] theorem prover and Prover technology's proof tool based on Stälmarck's decision procedure [11, 10]. The Prover plugin also deals with the return of counter examples. It presents a countermodel, a list of pairs of expressions and their truth values, as a third result of the decision procedure (along with true and false). This presentation is rather ad hoc and doesn't present the counter model together with the original goal as we do, nor does it allow the results of the decision procedure to be used in conversions. It wouldn't be difficult to wrap up the Prover plugin's output to match our representation. The approach we have adopted here integrates the plugin more seamlessly within HOL and is general enough to cover the Prover plugin's output.

SVC has been integrated into Isabelle as part of Isabelle/DC, a proof assistant for the real-time logic Duration Calculus[6]. However, as a result of using Prosper methodologies to construct the SVC plugin, it now shares a consistent programming interface with other Prosper plugins, which will lead to easier future development.

7 Future Work

To properly assess and evaluate the SVC plugin our axiomatic theory of rationals needs to be converted into a definitional one. One route to pursue here might be to treat the rational numbers as a restriction of the reals.

We would also like to expose more of SVC's entrypoints to HOL. We do not at present, have any plans to extend the formula syntax handled since HOL has no appropriate theories for records or bit vectors. However should these become available then it would be interesting to evaluate the use of the decision procedure in conjunction with them.

8 Conclusion

We have integrated SVC into HOL using the technology of the PROSPER toolkit. This integration has succeeded with encouraging results and serves as an example of how decision procedures can be integrated into HOL using the PROSPER approach without the necessity of re-implementation.

One of the problems we encountered was a mismatch between the objects handled by SVC (rationals, records and bit vectors) and the theories in HOL. While we have provided an axiomatic theory for rational numbers in HOL it seems likely that if the PROSPER project wishes to encourage more integration of this kind then a wider set of theories will need to be implemented since it would be unrealistic to expect tool vendors to implement their own theories for standard types such as rational numbers.

We have developed a framework within which counterexamples should be presented to HOL. We intend that this framework should be general and provide a standard format for accessing counterexamples. In this way design tools wishing to exploit counter-examples (to provide debugging information to a user, for instance) need only know that a plugin can return counter-examples and need not worry about the manner in which the counter-example is presented.

References

1. C. Barrett, D. Dill and J. Levitt, Validity Checking for Combinations of Theories with Equality, M. Srivas and A. Camilleri (eds), *Formal Methods In Computer-Aided Design (FMCAD'96)*, Lecture Notes in Computer Science 1166, Springer-Verlag, pp 187–201, 1996.
2. B. Brock, M. Kaufmann, and J Moore, ACL2 Theorems about Commercial Microprocessors. M. Srivas and A. Camilleri (eds), *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96)*, Lecture Notes in Computer Science 1166, Springer-Verlag, pp. 275–293, 1996.
3. A. Church, A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.
4. L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon and T. Melham, The PROSPER Toolkit, S. Graf and M. Schwartzbach (eds). *TACAS 2000*, Lecture Notes in Computer Science 1785, pp. 78–92, 2000.
5. M. J. C. Gordon and T. F. Melham (eds), *Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993.
6. S. T. Heilmann, *Proof Support for Duration Calculus*, Department of Information Technology, Technical University of Denmark, PhD-thesis, January 1999
7. J. Hurd, Integrating Gandalf and HOL. Y. Bertot, G. Dowek, A. Hirshowitz, C. Paulin and L. Théry (eds). *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 1690, Springer-Verlag, pp. 311–321, 1999.
8. J. R. Levitt, Formal Verification Techniques for Digital Systems. *PHD Thesis*, Stanford University, December, 1998.
9. K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers. 1993.
10. M. Sheeran and G. Stålmarck, A tutorial on Stålmarck's proof procedure for propositional logic. *The Second International Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science 1522, Springer-Verlag, pp. 82–99, 1998.

11. G. Stålmarck and M. Säflund, Modelling and Verifying Systems and Software in Propositional Logic. *Proceedings of SAFECOMP '90*, Pergamon Press, pp. 31–36, 1990.
12. M. Staples, Linking ACL2 and HOL, Technical Report No. 476, University of Cambridge Computer Laboratory. 1999.
13. T. Tammet, A resolution theorem prover for intuitionistic logic. *13th International Conference on Automated Deduction*, Lecture Notes in Computer Science 1104, Springer-Verlag, pp. 2–16, 1996.
14. SVC Home Page, *The Stanford Validity Checker*, http://verify.stanford.edu/SVC.
15. John Harrison, *Constructing the real numbers in HOL*, University of Cambridge.
16. Patrick Suppes, *Axiomatic Set Theory*, D. Van Nostrand Company, Inc, 1960.