

Experimentation with a Minimalist Prover

Hampton Smith

Abstract—Central to the design philosophy of RESOLVE is the hypothesis that a well designed verification system will—for day-to-day programs—generate VCs that are straightforward to prove. In order to investigate this hypothesis, we have been developing a term-rewrite prover at Clemson with the goal of exploring the minimal functionality required to verify reasonable programs written using the RESOLVE language. This paper describes our progress and the functioning of this prover.

Index Terms—automated proving, verification, software engineering, formal methods

I. INTRODUCTION

THREE different strategies are currently used to dispatch verification conditions (VCs) generated by systems in the RESOLVE family of languages. The first is to use an existing third-party prover—current RESOLVE systems use the Isabelle proof assistant [1]. The second is to use a special-purpose decision procedure such as the Split Decision engine being developed at Ohio State. The third is to use a home-brewed prover such as the minimalist rewrite prover we are developing at Clemson. Each of these options provides its own advantages and drawbacks, but it is the solution we are developing at Clemson that is the focus of this paper.

Our prover differs from third party provers such as Isabelle primarily in that it is focused on minimal, general functionality. While Isabelle allows for multiple decision procedures based on context, along with a general rewrite engine and a satisfiability engine, all of which provide special-purpose procedures for such wide ranging topics as set theory, number theory, and boolean theory, the prover we are developing at Clemson focuses on a rewrite engine alone, which is applied in any situation.

Our prover differs from a decision procedure in its generality: it can be applied to any domain so long as a theory of that domain has been certified and added to the library. A decision procedure has the advantage of being able to make use of domain specific information to organize proofs and make decisions, while a general prover has the advantage of being able to be applied to diverse situations without modification.

The remainder of the paper will be organized as follows: in section 2 we will motivate the development of a new prover, in section 3 we will describe the technical workings of the

prover we are developing, in section 4 we will discuss optional optimizations currently available in our prover, in section 5 we will discuss planned future development on the prover, and in section 6 we will discuss the capabilities of other provers, and finally in section 7 we will provide some concluding remarks.

II. RATIONALE

An integral part of the hypothesis driving RESOLVE development is that—while verifying *any* program is an unrealistic goal, disallowed as it is by Goedel's Incompleteness Theorem, *most* properly engineered programs ought to be straightforward to prove, since the human programmers feel convinced they work. It follows from this hypothesis that an “industrial strength” prover should not be required to dispatch the VCs arising from a well-engineered system: the prover need only keep up with the small leaps of logic that a programmer feels are relatively obvious from step to step in his program. This opens up the option for a minimalist prover, in which only the most basic functionality is hard-coded and the remainder of the behavior is effectively a parameter drawn from the mathematical theory files.

In order to test this hypothesis, we at Clemson University have been building such a prover from the ground up. The prover is exceedingly simple: consisting of a theory-propagation preprocessing step, during which each VC's consequents are expanded with those conclusions that follow from the given assumptions, and a proof-searching step that forms the bulk of the proving process, in which the proof space is searched by making repeated simple substitutions—that is, replacing left hand sides of equalities with right hand sides, or *visa versa*. All other data about the proving process is defined in mathematical theories, which provide both implications to be used in theory propagation and also equalities to be used during the substitution step.

In addition, since we are developing this prover from the ground up, we have the added advantage of assuring that its mathematical underpinnings match those of the rest of the RESOLVE system and that the prover can be easily interfaced with by the verification condition generator.

III. PROCESS

The input to our rewrite prover is the output from the VC generator—VCs, in symbolic format—along with the corpus of theorems defined in visible theories. The symbolic format of the VCs is a modified form of a RESOLVE abstract syntax tree. In order to ease reasoning about VCs and theorems alike, both are broken into conjuncts to divorce sequences of

Manuscript received August 11, 2009. This research is funded in part by NSF grants CCF-0811748 and DMS-0701187.

Hampton Smith is with the School of Computing at Clemson University, e-mail address: hamptons@cs.clemson.edu.

conditions from cumbersome series of *ands*. As a preprocessing step, simple variable expansions where the left or right hand side of the conjunct is a variable name are replaced in place.

Before proving begins, available theorems are split into to groups: those that take the form on an equality and those that take the form of an implication. At present, no other kinds of theorems are allowed. Implication theorems are applied to the antecedents of each VC as a preprocessing step, by way of performing theory propagation. So, for example, we may have a VC such as:

VC Version 1

$$|S| > 0 \text{ and } |S \circ T| \leq |R| \rightarrow |T| < |R|$$

And an implication theorem of the form:

Theorem 1

$$\text{For all } i: Z, i > 0 \rightarrow i \neq 0$$

So that, during preprocessing, the VC becomes:

VC Version 2

$$|S| > 0 \text{ and } |S \circ T| \leq |R| \text{ and } |S| \neq 0 \rightarrow |T| < |R|$$

It should be noted that currently theorems are limited to universal quantifiers without *where* clauses, as these can raise additional proof obligations.

This theory propagation step is performed in rounds, with each theorem offered the chance to bind multiple times to the antecedents of the VC. In this way, multiple-step jumps of logic can be simulated. Supposing we also had this theorem:

Theorem 2

$$\text{For all } u, v, w : \text{Str}(\text{Gamma}), \\ |u| \neq 0 \text{ and } |u \circ v| \leq |w| \rightarrow |v| < |w|$$

After round one, VC Version 1 would become VC Version 2 (since the antecedent for Theorem 2 cannot be directly established). However, with the application of Theorem 1, the antecedent of Theorem 2 can afterward be directly established and so, after round two, the VC becomes:

VC Version 3

$$|S| > 0 \text{ and } |S \circ T| \leq |R| \text{ and } |S| \neq 0 \text{ and } |T| < |R| \rightarrow |T| < |R|$$

Which can of course be immediately dispatched. After each round, duplicates and useless tautologies (such as symmetric equalities) are removed. After some predetermined number of rounds (which is a parameter to the prover), preprocessing ends and the prover begins to explore the proof space directly.

In each step of the proof search, any conjuncts of the consequents of the VC that follow directly (i.e., appear in in the antecedents, are symmetric equalities, or are tautologically true by some theorem) are removed from consideration. If all conjuncts have been removed, the VC is proved and we're done. If not, we attempt to apply a proof step. A proof step can take the form either of a theorem application or an antecedent substitution. Each possibility is tried in turn. At this stage, only theorems in the form of equalities are considered and modifications are only made the consequents of the VC. Each theorem and each antecedent is cycled through and all possible applications are considered with a recursive step. If no application yields a proof, the prover backtracks.

In order to assure termination, proof lengths are tethered such that proofs longer than a certain length are not considered. This length is a parameter to the prover, and in this way the proof-space, while often very large, is never infinite.

IV. CURRENT OPTIMIZATIONS

Our rewrite prover was quite specifically implemented as straightforwardly as possible, taking to heart Knuth's advice that premature optimization is the root of all evil. A few optimizations have, however, already been added where they have proved useful.

Firstly, the proof space is searched in "layers", with all proofs of length two being searched first, then proofs of length three, and then proofs of length four or five. A compiler flag allows proofs of length six to optionally be searched as a last resort, but this often leads to searches that run for twenty minutes or more. This dramatically increases the speed at which proofs are found in most cases, since most VCs are very straightforward and require only a few steps. The cost, of course, is that longer proofs take longer to than necessary to discover, since all the proof paths at lengths one to three have to be re-generated to explore those proofs of length four and five.

Secondly, as the proof space is searched, the prover keeps track of each transformation on the VC along the current path. The moment a VC state is repeated, the prover backtracks rather than search from that point, since clearly if we can't find a proof with length no more than four starting from $x \rightarrow y$,

we aren't going to find one with length no more than two. Currently this is implemented in a brute force fashion—each VC state is pushed onto a stack as we recurse, and each new VC is compared to every VC on the stack.

A third optimization is to prioritize the order in which theorems are applied. Originally theorems were applied in alphabetical order by theorem name (essentially randomly). Thus the order of theorem application was the same for each VC. With prioritization, however, the theorems are ranked by some fitness function (a parameter to the prover) once for each VC, and then applied in order from most to least fit. The default fitness function gives higher fitness to theorem applications that simplify (which here means “result in fewer variables and function applications”) and to theorem applications that have high overlap between those types and named functions that appear in the theorem and those that appear in the consequents of the VC.

The effect of applying these optimization is summarized in Table 1. The *Backtracking Alone* heading is omitted in the table because currently backtracking adds so much overhead to each proof steps that long proof searches take far too long to terminate. The metrics used are *proofs considered*, where a considered proof is any successful application of a rule (either a theorem or an antecedent substitution), regardless of whether applying that rule leads to a successful proof or a backtrack; and *average time*, which is expressed in milliseconds. Each of these metrics is important, since the former indicates something of the complexity of the proof, while the latter indicates the efficiency of the proof. The VCs considered are taken from a selection sort realization of sorting on queues. They are not representative of the VCs, but rather were chosen as “interesting” examples that took more effort than most to prove. The VCs are as follows (with some irrelevant expressions removed for brevity):

VC 0_7

$$(Q = (<??Min> \circ ???Q)) \text{ and}$$

$$(Is_Permutation(((?New_Queue \circ ??Q) \circ <?Min>), Q)) \text{ and}$$

$$(??Q = (<?Considered_Entry> \circ ?Q))$$

=====>

$$Is_Permutation($$

$$?New_Queue \circ <?Min> \circ ?Q \circ <?Considered_Entry>,$$

$$Q)$$

VC 2_3

$$(|Q| \neq 0) \text{ and}$$

$$(Is_Permutation(?New_Queue \circ ?Q \circ <?Min>, Q) \text{ and}$$

$$(\text{not}(!?Q| > 0)))$$

=====>

$$! ?New_Queue| = (|Q| - 1)$$

VC 3_8

$$(Is_Universally_Related(LEQV, ?Sorted_Queue, ??Q)) \text{ and}$$

$$(!?Q| > 0) \text{ and}$$

$$(Is_Permutation(?Q \circ <?Lowest_Remaining>, ??Q) \text{ and}$$

$$(Is_Universally_Related($$

$$LEQV, <?Lowest_Remaining>, ?Q)) \text{ and}$$

=====>

$$Is_Universally_Related(LEQV,$$

$$(?Sorted_Queue \circ <?Lowest_Remaining>), ?Q)$$

Now, the table:

TABLE 1
OPTIMIZATION EFFECTS

Optimizations	Proofs Considered	Average Time in milliseconds (over 5 trials)
No Optimizations		
0_7	1258410	145596
2_3	33	69
3_8	61	53.2
Layered Search Alone		
0_7	31456	4963.8
2_3	10	44.6
3_8	7	35.2
Prioritization Alone		
0_7	88162	11204.8
2_3	30	50.4
3_8	2	29
All three together		
0_7	2226	1392.6
2_3	229	93
3_8	2	28.8

V. FUTURE WORK

There are a number of optimizations we ave yet to explore which may improve the efficiency of our rewrite prover while keeping it very simple. The one we are actively working on is to allow theories to provide “hints” (backed up with suitable proofs, of course) that certain operations are associative or commutative under certain circumstances. If, for example, we know that an operation is associative and commutative under all circumstances, we can store chained operands internally as an unordered multiset rather than a hierarchical tree that encodes such useless (in this situation) information as operation precedence and operand order. In this way we collapse many subspaces from the proof space that represent basically equivalent searches (such as re-ordering the operands). In this way, we can create a theorem in *String_Theory* like:

Associativity Theorem *Permutation_Associativity*:

For $t, u, v, w : \text{Str}(\text{Gamma})$,
 $\text{Is_Permutation}((t \circ u) \circ v, w) =$
 $\text{Is_Permutation}(t \circ (u \circ v), w)$;

When this feature is fully implemented, the prover, upon seeing such a theorem, will transform chains of concatenations that appear in the specified context into an internal representation that reflects their associativity. A similar syntax will exist to deal with commutativity.

Some other possible future optimization options include:

- Searching the proof space starting only with those theorems deemed most relevant by the fitness function, then expanding the set of considered theorems until a proof is found or all theorems are considered.
- Prioritizing theorems once at each level of the proof for a VC, rather than once for each VC, such that the first theorem attempted at any level is the one that brings the VC's consequents closest to being proved, as determined by some fitness function.

In addition, there are a number of places where the proving algorithm itself will eventually need to be changed. As one example, all theory propagation is currently done naively in a pre-processing step. However, in the presence of thousands of theorems, this will likely result in an exponential explosion in the size of VC antecedents, with an associated jump in the time requires at each proof step. Eventually, some tactic will have to allow for more selective application of theory propagation, either reducing the set of considered theorems, or allowing theory propagation to be done on-the-fly during proof searches, rather than as a preprocessing step.

VI. RELATED WORK

A. Isabelle [1]

Programmed in SML, Isabelle is similar to RESOLVE in that it defines a narrow logical kernel upon which all other development is based. However, unlike RESOLVE, Isabelle is styled as a “framework,” with additional logics and tactics hard-coded to extend its SML code, resulting in different flavors of Isabelle such as “Isabelle/HOL” for higher order logic and “Isabelle/ZF” for Zermelo-Fraenkel set theory, Isabelle contains a rewrite prover similar to the one we have developed but it is often used as a simplifying step before passing its result onto one of the decision procedures, rather than as the prover proper.

B. Z3 [2]

Developed by Microsoft Research, Z3 is the backend prover for the Boogie intermediate verification language, which is used primarily as the intermediate language for the

Spec# effort to extend C# with verification capabilities. It is an SMT solver (Satisfiability Modulo Theories) with support for quantifiers that leverages hard-coded logics for real and integer arithmetic, bit-vectors, and uninterpreted functions and symbols, among others. It differs from our prover both in that its theories are hard-coded and that it uses different procedures for different domains.

C. Simplify [3]

The basis for the Extended Static Checking family of verification systems (including, most notably, ESC/Java), Simplify attempts to establish theorems by testing the satisfiability of their negation. Simplify includes hard-coded theories for boolean logic, arithmetic, and maps, among others. As with the Z3 prover, Simplify differs from our prover both in that it uses hard-coded theories, and it does not apply a general proof procedure to all domains.

VII. CONCLUSION

Despite its modest capabilities, the prover we have developed is capable of dispatching VCs generated by a number of example programs, including both simple programs such as reversing an unbounded stack or transferring the contents of one to another, and more advanced programs such as a selection sort realization of bounded queue sorting. The former examples can be completely verified, while the latter generates 39 VCs, of which all but two can be proved. Those that cannot be proved are the result of an assertion in the RESOLVE source code that erroneously does not make its way into the final VCs—that is, the VCs themselves are unprovable.

We feel this result lends weight to the hypothesis that a straight-forward substitution prover may well be far more effective when applied to the VCs arising from a properly engineered verification system than most have guessed. We intend to apply the prover to larger and more diverse examples to further experiment with what capabilities are absolutely necessary in such a situation, and begin to use the prover to test the scalability of a minimalist prover.

VIII. ACKNOWLEDGEMENTS

This research is funded in part by grants from the National Science Foundation: CCF-0811748 and DMS-0701187. The author would like to thank Murali Sitaraman and Bill Ogden for their many insights that motivated and shaped this prover.

REFERENCES

- [1] M. Makarius. (2009, April 19th). The Isabelle/Isar Reference Manual [Online]. Available: <http://www.cl.cam.ac.uk/research/hvg/Isabelle/>
- [2] L. de Moura and N. Bjorner. “Z3: An Efficient SMT Solver,” in Proc. Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, 2008.
- [3] D. Detlefs, G. Nelson, and J. B. Saxe, “Simplify: A Theorem Prover for Program Checking,” Hewlett Packard, Technical Report HPL-2003-148, 2003.