Experimentation with a Minimalist Prover for Discharging Verification Conditions of Correctness

Hampton W. Smith, Blair Durkee, and Murali Sitaraman

Technical Report RSRG-14-01
School of Computing
100 McAdams Hall
Clemson University
Clemson, SC 29634-0974

July 2014

Experimentation with a Minimalist Prover for Discharging Verification Conditions

Hampton Smith[1], Blair Durkee[2], and Murali Sitaraman[2]

[1] Mountain View, CA, USA
hampton.w.smith@gmail.com
[2] School of Computing, Clemson University, Clemson, SC 29634, USA
{bdurkee | msitara}@clemson.edu

Abstract. Specialized decision procedures have shown much promise for discharging verification conditions (VCs) that arise from establishing correctness of programs. This paper explores the merits of an alternative approach for proving VCs using a general, yet minimalist rewriting prover. The advantages of the prover are that it is independent of the combination of theories involved in the VCs. So if a new theory is employed in specification of a component, only the results from the theory in standard mathematical form are necessary for the prover to verify VCs resulting from using that component. No special translations to the logic of the provers are necessary. The thesis behind why a minimalist prover might have promise is that the VCs arising from properly justified programs are relatively simple to establish and do not require deep thinking, given suitable mathematical results. Experimentation with a suite of benchmark examples is promising. The paper also includes lessons learned and directions for further research.

Keywords: automated proving, benchmarks, evaluation, modular reasoning

1 Introduction

The development of a verifying compiler–one that proves programs correct with respect to their specifications–is a grand challenge for the computing community [9] . Automated and modular verification of full functional behavior of (generic) software components remains an important research area. A variety of systems, such as Dafny, RESOLVE, VeriSoft, and others summarized in [13] from a VSTTE 2010 Competition are available today for proving verification conditions (VCs) that arise from establishing software correctness. Underlying these systems is a range of proof assistants, such as Coq [15], and Isabelle/HOL [17], and sophisticated provers, such as Z3 [6].

This paper presents results from an experimentation with a suite of benchmarks to check the thesis that if software is engineered well and verification is modular, then the correctness conditions would not be difficult to prove. Specifically, given suitable mathematical results, it experiments with a minimalist rewriting prover to discharge the VCs. Such a prover is of interest for a variety

of reasons. Since the prover is independent of the mathematical domains involved and simply employs pre-established results from those domains, it is suitable for proving VCs that span multiple theories. Moreover, the prover works independently of whether the assertions to be proved are of first order or a higher order. If the specification of a new software component employs a new theory, then there is no additional effort required to prove VCs involving that specification, as long as the prover has access to suitable results in that theory. So the prover could readily push the boundary of automatically provable VCs that cut across multiple theories. The proof process employed by the prover is also closer in spirit to human reasoning. Finally, it might be easier to check the logic of the prover itself, because there is only one procedure to check as opposed to several specialized procedures. These are among the motivations for the research experimentation.

It is important to underscore what the minimalist prover is not meant to be. It simply employs results (e.g., theorems) from mathematical theories to prove the VCs, and will not be able to prove the validity of the results themselves, either automatically or with human assistance. These theorems need to be validated using other automated provers or proof assistants, such as Coq or Isabelle. Also, since a key objective of the minimalist prover is to enlarge the types of VCs that are proved automatically (but not necessarily how quickly they are proved), a practical verification system could use it in conjunction with other existing provers and terminate as soon as any of them terminate with a result. The system in [18], for example, runs both SplitDecision [1], and Z3 [6], on the same VCs.

The key thesis underlying why a minimalist prover such as the one discussed here is viable is that the VCs arising from properly-designed and justified software do not require deep thinking [12] to prove and require only a few steps. The results in this paper provide additional evidence for this thesis in that little proof searching is required for several of the VCs arising from benchmarks used in our experimentation. One key idea in minimizing proof steps is the development and use of assertions in specifications using mathematical developments that avoid the need for explicit quantifiers. For example, to assert that an output queue `Q` is a permutation of an input queue (`#Q`), we would state `Is_Permutation (Q, #Q)`, instead of using explicit quantifiers. This approach of eliminating quantifiers, while easing automation, also enhances human comprehension. As long as there are sufficient results involving permutation, proofs VCs involving the notion become straightforward.

The rest of the paper is organized into the following sections. Section 2 contains background and related work. Section 3 summarizes the heuristics employed by the minimalist prover. Section 4 summarizes the suite of benchmarks and solutions used for evaluation, highlighting key points pertaining to the prover. Section 5 contains results of our experimental evaluation, observations, and lessons learnt. Section 6 has our conclusions.

2 Background and Related Work

The distinction between different types of provers is often not clear-cut. In general, all provers include some form of term rewriting, decision procedures, and SAT solvers.

SAT solvers attempt to find valuations of boolean variables to satisfy a given formula. While in general this is NP-hard, a number of branch-and-bound techniques can be used to drastically reduce the problem of space for practical formulae. Almost universally such provers use the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, which is a constellation of specific heuristics and refinements that reduce the search space considerably in many cases. A good overview of this algorithm along with some recent refinements may be found in [16].

Many techniques exist for translating complex programmatic proof obligations into equisatisfiable boolean representations. An excellent summary of decision procedures may be found in [2]. A number of DPLL-based, SMT-solvers exist. Yices [7] and Z3 are popular provers based at their core on SAT-solvers. The built-in theories included in Z3 (the prover behind Dafny [19], among others), for example, are empty theory, linear arithmetic, nonlinear arithmetic, bitvectors, arrays, datatypes, quantifiers. The topic of engineering "plug ins" for new theories layered over the base theories is the topic of [3].

Term-rewrite Provers provers are the most flexible, but also the most difficult to optimize. An excellent example of this kind of prover being used in a practical style system is that used in the ACL2 system [10]. They are, by far, the most similar to how a human would proceed with a proof–matching theorems against a set of known facts and goals, transforming them until the desired result is derived. This general matter matching approach means that they can be applied to any domain about which a set of theorems has been established. ACL (unlike the minimalist prover discussed here) utilizes a complex series of hints, provided by the person developing the theory, about which theorems should be applied in which way, before applying them sequentially. Additionally, while our prover strives to operate over our distinct mathematical subsystem, ACL2's mathematics are much more tightly bound with its underlying language, Lisp. Unlike our minimalist prover, a notable feature of the ACL2 prover is its ability to automatically discover inductive proofs over certain restricted domains.

3 Minimalist Prover Summary

The prover discussed in this paper is one of the elements of the RESOLVE push-button verifying compiler [20] and it has been used in CS education for multiple years underneath a web IDE for the verifying compiler [5].

A complete discussion of the prover and its evaluation may be found in [21]. To validate the claim that well-specified and well-engineered software supported by a flexible, extensible mathematical system often yields VCs that are sufficiently straightforward to be dispatched by a minimalist rewrite prover, we have built a prover with absolutely minimal capabilities and slowly enhanced it through experimentation with general heuristics to a point where it can prove

almost all the VCs arising from a series of benchmarks. The heuristics are motivated in part by following human reasoning.

The initial version relied only on equality theorems (i.e., those of the form `A = B`) were considered, and they were permitted to be applied in either direction (i.e., replacing `A` with `B` or replacing `B` with `A`). This alone turned out to be sufficient to prove a number of VCs, because in many VCs goals are often close to the givens–an observation also made in [12].

The initial version was next extended to generate new results from existing antecedents (givens) to be able to prove the consequent (goal) of VCs such as the one below, using available set of theorems. This VC arises from verification of a Stack flipping example where Stacks are conceptualized as mathematical strings (similar to, but simpler than finite sequences because there is no indexing). `String_Theory` includes notations `o` for string concatenation, `||` for length, and `< >` for a string containing a single entry. In the VC, `Next_Entry'` is a single entry, whereas `S`, `S'`, `S''`, and `S_Flipped'` are strings that denote Stack values.

```
goal:
(|S_Flipped'| < Max_Depth)

given:
(((((Max_Depth > 0) and
(|S| <= Max_Depth)) and
S = (Reverse(S_Flipped') o S")) and
|S"| /= 0) and
S" = (<Next_Entry'> o S'))
```

Fig. 1: VC for the `requires` clause of `Push`

The proof for this VC is relatively straightforward: note that `|S| <= Max_Depth` and `S` itself is made up of:

`Reverse(S_Flipped')` and `S"`.

Since `|S"| /= 0`, the length of `Reverse(S_Flipped')` must be strictly less than `Max_Depth`. Since reversing the string does not affect its length, the length of `S_Flipped'` must also be strictly less than `Max_Depth`.

For the above example, a theorem that produces a new result `<` from givens involving `<=` and `=` needs to be applied.

In subsequent improvement, we introduced a pre-processing step, which we termed "minimization"[3] We qualified such steps resulting from minimization as those that maintain the tautological property (i.e., could not make a tautologically true VC into one that is not tautologically true) and that *strictly reduces* the number of function applications.

The rest of the section discusses a number of heuristics and optimizations to aid us in dispatching not general mathematical statements, but rather, the sorts of VCs that arise from code. Experimental data on the effectiveness of the

[3] We opted for this terminology over others loaded with existing mathematical meaning such as "simplification", because minimization merely provides a best-effort heuristic for reducing function application count. Changing the available theorems or even the order of these theorems may affect the outcome of minimization.

heuristics along with lessons learnt is discussed in a subsequent section. Clearly, some are more fruitful and require further exploration than others.

Intelligent Antecedent Development Antecedent development refers to the process of establishing new known facts based on existing ones. For example, given `A ---> B` as a VC and a theorem stating `A ---> C`, we may develop the VC's antecedent into `A and C ---> B`. Because such developments always hold true and may often be efficiently searched, it is generally useful to spend some time expanding our list of known facts before embarking on our proof search. In some sense, this increases the size of the "target" we are trying to hit–since transforming the consequent into a match for *any single* antecedent allows us to establish the VC. However, antecedent development that does not directly pertain to consequent proving could also lead to wasted effort. We therefore use a number of heuristics to attempt to avoid useless antecedent development.

Qualify and Avoid Useless Transformations. Ultimately, for automated verification to scale up, we must insulate the end users—whether the programmer or the mathematician—from the details of the prover. To this end we have routinely rejected designs and methodologies that require the theory developer to "tag" theorems or otherwise provide hints to the prover inside a theory or a program, a technique that is frequently employed by other mathematical reasoning systems (see, e.g., [11]). Even without explicit tagging, it is often possible to qualify different kinds of resulting transformations and thus treat them differently based on their unique usefulness in a given situation. For example, consider this theorem from `String_Theory`:

```
Theorem Concatenate_Empty_String_Identity_Right:
        For all S : SStr, S o Empty_String = S;
```

Certainly, to apply this theorem from left to right would often be useful, but to apply it in the other direction adds little to the coverage of our antecedent. The antecedent development stage therefore does not prioritize such identity-maintaining transformations when they increase the number of function applications.

Develop Antecedents Only About Relevant Terms. It is often the case that a VC contains many irrelevant antecedents providing information that is not useful to the final proof. In general it is not possible to sort relevant from irrelevant without first arriving at a proof. As an extreme example of this, along dead code paths, contradictory antecedents with no other bearing on the consequent may render otherwise unprovable VCs true. However, we recall our hypothesis that all VCs ought to be straightforward and thus apply a simple heuristic: a new antecedent should only be considered useful if it tells us something about a term that appears in the consequent[4].

While in a general proof system, this would limit our useful developments, since proofs might be quite deep, based on our hypothesis that reasoning should

[4] We take note of any equalities such as `A = B`, where `A` and `B` are terms, to permit developments about terms that are equivalent to terms that appear in the consequent.

be straightforward we expect that any required information should already be available roughly in the vocabulary of the consequent.

Maximize Diversity. Any transformation that may be applied to develop the antecedent has a dual that could be instead applied to the consequent. We've therefore tried to explore which kinds of steps are most advantageous during the antecedent development phase and which are better left for the consequent exploration phase. Intuitively, we note that the antecedent development phase, which occurs once as a preprocessing step and requires no back-tracking, is an ideal place to apply more computationally expensive steps; by contrast, the consequent exploration phase, which does not accumulate related facts and is therefore not subject to a combinatorial explosion of space, is an ideal place to explore many small variations[5]. We therefore hypothesized that antecedent-development time would be best spent establishing varied facts that put antecedents in new terms. To qualify this, we added the requirement that each antecedent introduced must both eliminate some term and introduce one.

Take, for example, this partial VC:

```
|S| > 0
  ---->
S /= Empty_String
```

We would permit the development `|S| /= 0`, since it eliminates the greater-than symbol and introduces the not-equal symbol, but not `|S| > 0 + 1`, since, while it introduces two symbols, it does not eliminate any.

Maintain Simplicity. After each round of antecedent development, we minimize the resultant antecedents. This provides two advantages: first, it supports the diversity maximization from above by exposing antecedents that essentially establish the same fact in the same terms (which are then removed as redundant); second, it increases the liklihood of transforming the consequent into an antecedent during the proof exploration phase, since both antecedents and consequent orbit the same minimized form.

Intelligent Consequent Exploration Consequent exploration refers to the final phase of the proof search during which the consequent is repeatedly transformed using available theorems until it reduces to `true` or the search times out.

However, given the large number of available theorems and their frequently closed nature, the consequent space to be explored is extremely large, suffering from a combinatorial explosion of time. It therefore becomes important to search this space in a reasonable manner, since in general an exhaustive search will be computationally infeasible.

[5] This informal analysis is confirmed empirically later, but we note that it relies on a number of assumptions about the nature of the proving task and the specific proving algorithm.

Minimize Complications. Before beginning consequent exploration in earnest, we apply minimization. This eliminates many complications that a human user would naturally eliminate.

Qualify and Avoid Useless Transformations. As during antecedent development, we identify transformations that simply introduce identity functions and ignore them during our proof search.

Detect Cycles. While the VC state itself is mutable, all components of the state (the expressions and their types) are immutable, which lends itself to efficient calculation of expression hashes. Using a polynomial hash, the overall hash of the VC state can be updated in constant time each time an expression is added, removed, or modified. In this way, we can efficiently detect cycles and thus not bother to inductively explore beyond them.

This ability also supports our desire to divorce theory creation from reasoning about the prover in the following way: mathematical systems like ACL2 [11] and Issabelle [23] require theorems to be annotated with the direction in which they should be applied in order to prevent cycles that arise from the same theorem being applied repeatedly backward and forward. Our prover is able to detect such a situation and avoid it.

Tether Search. The main consequent exploration algorithm is a depth-first search. While cycle-detection eliminates one class of infinite, unproductive paths, others exist. For example: the prover might repeatedly add one to both sides of an equation. Consistent with our hypothesis that VCs should be straightforward to prove, we tether the search to a short length, after which no further exploration will be applied. While this depth is parameterizable, we've found that the default of four is generally sufficient.

Step Intelligently. Rather than step blindly, we apply a greedy best-first algorithm in our search. In order to quantify what we mean by "best", we must establish a useful fitness function for each transformation. Empirically, we have found that re-calculating fitness at each step is far too costly in terms of time, but we are able to find a sensible ordering of available theorems on a per-VC basis.

We identified three criteria on which to determine the fitness of a transformation: 1) what effect does applying the transformation have on the unique symbols present? Does it introduce new unique symbols? Does it eliminate existing ones? 2) what effect does applying the transformation have on the number of function applications? 3) how many symbols does the transformation share with the VC?

After experimentation, we have found that the third criteria is not useful, since a transformation attempting to match symbols not contained in the VC can never be applied, and thus any such symbols that might be *produced* by the transformation, must be newly unique, which is subsumed by the second criteria.

By deprioritizing transformations that will introduce new symbols, we prevent the prover from "entering new territory" before it's finished exploring all the ways it can transform the current symbols. Similarly, by deprioritizing transformations that increase the number of function applications, we encourage proof

states toward simplicity and parsimony that are easier to explore. In a *general* proof system, one might assert that these tactics are just as likely to lead us *away* from a correct proof as to bring us closer, but in the specific domain of verifying well-engineered software, we hypothesize that the programmer is not taking leaps of logic and thus the reasoning should be simple.

4 Summary of Benchmarks and Solutions

4.1 Benchmarks

For experimental evaluation, we have used a set of incremental verification benchmarks, discussed in a previous VSTTE publication [22]. These benchmarks are intended to provide a basis for experimentation and discussion among verification efforts. In [8], the author presents a series of benchmarks implemented in RESOLVE in order to demonstrate the VC-generation capabilities of the RESOLVE compiler. The VCs arise not only from establishing the end results, but also from other sources such as checking preconditions of called operations, checking validity of programmer-supplied loop invariants and progress metrics for termination, and in one case, checking the validity of representation invariants. The VCs span multiple theories, including integer theory, string theory used in the specification of queues, and lambda functions that are used to model arrays.

The benchmarks and results from attempting to prove them with the minimalist prover are the topics of the next two sections.

Benchmark 1: Recursive and Iterative Integer Code Verify an operation that adds two numbers by repeated incrementing. Verify an operation that multiplies two numbers by repeated addition using the first operation to do the addition. Make one algorithm iterative, the other recursive. [6]

The verified solution accounts for Integer bounds and termination. The VCs involve integer theory.

Benchmark 2: Binary Search of an Array Verify an operation that uses binary search to find a given entry in an array of entries that are in sorted order.

Our solution is for a generic array parameterized by a type. It includes an ordering relation as a parameter to the specification and a comparator operation as a parameter to the code. Of the 45 VCs generated by this example, we are able to mechanically verify 39. The six VCs–the only ones that are not proved–do not represent a failure not of the prover, but rather a temporary, syntactic limitation at the time of submission. Specifically, the compiler does not properly process the required theorems that require the application of an arbitrary lambda expression (as opposed to a named function). Once these theorems are in place, proof of the VCs should be straightforward for the minimalist prover, as explained in the next subsection.

Benchmark 3: Sorting a Queue Specify a user-defined FIFO queue ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client defined order.

[6] Refer to [22] for this and all other problem statements.

The solution approach illustrates several interesting aspects and it is detailed in the next subsection. All 31 VCs for this example are proved automatically.

Benchmark 4: Reversing a Queue Specify a user-defined FIFO queue ADT that is generic (i.e. parameterized but the type of entries in a queue). Verify an operation that reverses the entries in a given queue.

As in the case of sorting solution above, our solution here is also modular and is based on the specification of a bounded queue. All VCs are proved.

Benchmark 5: Data abstraction verification Specify a user-defined LIFO Stack data abstraction that is generic (i.e., parameterized by the type of entries in the Stack). Verify an array implementation of that data abstraction.

The solution involves an implementation that is documented with an implementer-supplied abstraction function and representation invariant assertions. Generated VCs involve assertions containing strings, lambda functions, and numbers, and all of them are proved.

Benchmark 6: Specification Using a Different Theory Specify a tree abstraction, then specify, implement, and verify an operation that modies the tree before restoring it to its original form.

This benchmark and solution used only to illustrate that the prover can work with new theories and results. Resulting VCs from a simple example are proved.

4.2 Solution for Sorting a Queue with Specification Engineering

Solution Discussion The following is a specification of a queue sorting enhancement in RESOLVE [20]. As with our solution to the searching benchmark, it takes as a parameter a client-provided definition to define the sorted order, `LEQV`. The specification requires that this is a total preorder, i.e., it is total and transitive. The specification also makes use of two higher-order definitions:

`Is_Coformal_With(...)` and `Is_Permutation(...)`

to ensure that the final queue is ordered according to `LEQV` and contains the same elements as those originally provided, respectively. `String_Theory` contains these definitions as well as various results involving these definitions.

```
Enhancement Sorting_Capability(definition LEQV(x, y: Entry): B)
    for Queue_Template;
    uses String_Theory, Ordering_Relations_Theory;
requires Is_Total_Preorder(LEQV));

    Operation Sort(updates Q: Queue);
        ensures Is_Conformal_With(LEQV, Q) and
                Is_Permutation(\#Q, Q);
end Sorting_Capability;
```

A straightforward selection sort realization is provided in Figure 2. The realization is parameterized by an ordering operation (named `Are_Ordered)` that must be client supplied, and it is used in the local `Remove_Min` procedure that is called by `Sort`. The procedures are annotated with loop invariants (avoiding quantifiers) and termination progress metrics. In the realization, the notation :=: denotes a swap operator.

This implementation is fully verifiable. While others have verified similar generic implementations automatically, those have relied on decision procedures. For example, see [14] and the associated commentary [4].

```
Realization Selection_Sort_Realiz
  (Operation Are_Ordered(restores E1, E2: Entry): Boolean;
ensures Are_Ordered = LEQV(E1, E2);)
  for Sorting_Capablity of Queue_Template;
  uses String_Theory;

  Operation Remove_Min
    (updates Q: Queue; replaces min: Entry);
      requires |Q| /= 0;
      ensures Is_Permutation(Q o <min>, #Q) and
              Is_Universally_Related
                (<min>, Q, LEQV) and
              |Q| = |#Q| - 1;
  Procedure
      Var considered: Entry;
      Var new: Queue;

      Dequeue(min, Q);
      While (Length(Q) > 0)
          changing Q, new, min, considered;
          maintaining Is_Permutation
                        (new o Q o <min>, #Q) and
                      Is_Universally_Related
                        (<min>, new, LEQV);
          decreasing |Q|;
      do
          Dequeue(considered, Q);
          If Are_Ordered(considered, min) then
              min :=: considered;
          end;
          Enqueue(considered, new);
      end;
      new :=: Q;
  end Remove_Min;

  Procedure Sort(updates Q: Queue);
      Var sorted: Queue;
      Var lowest_remaining: Entry;

      While (Length(Q) > 0)
          changing Q, sorted, lowest_remaining;
          maintaining Is_Permutation
                        (Q o sorted, #Q) and
                      Is_Conformal_With
```

```
                        (LEQV, sorted) and
                      Is_Universally_Related
                        (sorted, Q, LEQV)
          decreasing |Q|;
       do
          Remove_Min(Q, lowest_remaining);
          Enqueue(lowest_remaining, sorted);
      end;
      Q :=: sorted;
  end Sort;
end;
```

Fig. 2: A selection sort realization for the `Sorting` enhancement

4.3 Discussion of Example Provable and Unprovable VC

This subsection presents a VC that is not proved, and it corresponds to the inductive case of the while loop's maintaining clause in binary search (irrelevant givens omitted for brevity).

```
goal:
  lambda(j: Z).
  ({{key     if j = (low' + ((high' - low') / 2))
     A'(j)   otherwise}}) = A

given:
  A " = A and
  A ' = lambda(j: Z).
  ({{midVal"  if j = (low' + ((high' - low') / 2))
     A"(j)    otherwise}}) and
  A"((low' + ((high' - low') / 2))) = key)
```

The mechanical prover would take the following steps. It would first replace instances of `A"` with `A`. Then it would replace `A'` with its full expansion in the goal, resulting in a new goal. Finally, the instances of `key` will be replaced with its expansion. At this point, it would apply a relevant theorem about lambda expressions from function theory (and expression of this theorem is what is not currently handled by the compiler).

```
theorem Shadowed_Function_Piece:
    for all T, R: MType,
    for all t: T,
    for all r: R,
    for all f: T -> R,
        f = lambda(x : T).
          ({{f(x)  if x = t
            lambda(y: Z).
              ({{r    if y = t
                 f(y) otherwise}})
             (x) otherwise}})
```

Application of the theorem would lead to `A = A`, at which point the proof is trivial. Each of these six currently unprovable VCs have a similar straightforward proof.

5 Prover Evaluation

The evaluation of the minimalist prover orbits three fundamental questions: first, does it succeed in proving verification conditions arising from programs; second, are our heuristics effective at increasing the usefulness of the prover; and third, does the data gathered by the prover expose any interesting properties of VCs that arise from well-engineered programs.

The first and second questions are explored by considering a series of verification benchmarks. In this section, we use the same benchmarks to explore how our various heuristics impact the effectiveness of the prover. To answer the third question, we provide some observations and conclusions based on the collected data.

5.1 Evaluation Metrics

We focus on three primary metrics for exploring the effectiveness of our prover and analyzing the difficulty of VCs. These are:

- **VCs proved.** The number of VCs that are successfully dispatched by the prover. Since the proof space is quite large, the prover often runs with a set timeout, so in reality this metric refers to the number of VCs that were proved within a given timeout.
- **Proof steps.** The number of relevant steps taken by the prover. One of the features of the prover is to prune steps that do not impact the final result.
- **Search steps.** A subset of the overall proof steps including only those steps taken during the consequent search process.

The specific numbers for verification of the sorting example discussed in the last section is given in Figure 3, and the cumulative statistics for all benchmarks is given in Figure 4.

5.2 Summative Heuristic Performance

In order to evaluate the effectiveness of our heuristics, the prover has been instrumented so that each of six different heuristics could be disabled individually. The heuristics we targeted are: ignoring useless transformations, developing the antecedent only about relevant terms, focusing on diversity in antecedent development, minimizing both consequent and antecedent, cycle detection, and transformation prioritization.

We then re-ran the verification process on each of the benchmarks and collected data about the changes to our various metrics. This data is summarized in Figure 4.

	σ	Steps	Search
VC 0_1	242	6	0
VC 0_2	73	5	0
VC 0_3	56	5	0
VC 0_4	198	6	1
VC 0_5	300	8	0
VC 0_6	372	9	3
VC 0_7	271	8	0
VC 0_8	136	10	1
VC 0_9	166	9	3
VC 1_1	100	5	0
VC 1_2	165	10	1
VC 2_1	136	5	0
VC 2_2	237	8	0
VC 2_3	197	5	0
VC 2_4	284	6	1
VC 2_5	387	14	0

	σ	Steps	Search
VC 2_6	278	12	3
VC 2_7	108	10	2
VC 2_8	139	7	0
VC 3_1	141	5	0
VC 3_2	213	8	0
VC 3_3	124	5	0
VC 3_4	197	6	1
VC 3_5	239	14	0
VC 3_6	200	10	1
VC 3_7	194	10	2
VC 3_8	165	7	0
VC 4_1	139	5	0
VC 4_2	150	12	0
VC 4_3	189	5	0
VC 4_4	262	18	3

Fig. 3: Selection `Sort` enhancement realization results

As expected, we see that the heuristics are at least partially responsible for a number of VCs being proved. Ignoring useless transformations, limiting development to relevant terms, diversifying antecedents, minimizing, and prioritizing transformations all make the difference between some VCs proving or not proving. Of those, it is interesting to note that diversifying antecedents is time consuming, but it enabled six more VCs to prove. This may indicate that it would benefit from being targeted for further optimization to reduce the cost of applying it. On the other hand, ignoring useless transformations, limiting development to relevant terms, minimization, and prioritization are nothing but a net gain. They are collectively responsible for permitting 43 additional VCs to prove.

An important metric is the change in search steps, since from an algorithmic perspective these steps are the most expensive to take: they contribute to the overall combinatorial explosion of time as we must inductively search deeper and deeper for a solution. We note that three of our heuristics improve this metric, while only one has adverse effects (and a small one at that). All told, we may summarize the net effect of all our heuristics as saving 51 search steps.

It may at first be unclear how minimization can eliminate thirty-five search steps but only three total proof steps—after all, search steps are a subset of total proof steps. Consider, however, that minimization is a heuristic for taking *extra steps* that are likely to be useful. The discrepancy of thirty-two steps represent steps that were *not* taken in the preprocessing phase, but instead delayed to the consequent exploration phase. This increases the complexity of the search as steps taken in the exploration phase contribute to the combinatorial explosion of the search space, while steps taken in the preprocessing phase do not.

	$\sum \Delta$Proved	$\overline{\Delta\text{steps}}$	$\sum \Delta$steps	$\overline{\Delta\text{search}}$	$\sum \Delta$search
With useless transformations	-12	0	0	0	0
Developing about irrelevant terms	-1	0	0	0	0
Not checking for diversity of givens	-6	-0.02	-4	-0.01	-1
No minimization	-10	0.02	3	0.28	35
No cycle detection	0	0.08	11	0.08	11
No prioritization of transformations	-19	0.05	6	0.04	5

Fig. 4: Summary of heuristic evaluation results. From left to right the columns are: total change in the number of proved VCs (negative means fewer were proved), average change to the number of steps required, total change in number of steps required, average change in number of search steps required, and total change in number of search steps required. Average and total changes only take into account VCs that were proved.

5.3 Observations and Lessons Learnt

These six examples were chosen to be representative of the sorts of problems to which our prover can be applied, but do not constitute the total body of components we can automatically verify. We have a library of many other verified operations operating on stacks, queues, lists, and integers. Also, through the web IDE, our students have developed and verified numerous components as a part of their classroom exercises [5].

These benchmarks together involve 133 VCs ranging over the mathematical domains of functions, trees, strings, integers, and booleans. Of those, we are able to mechanically verify 127, with the remaining 6 provable module improvements to RESOLVE's language representation. The median number of proof steps was seven and the median number of search steps was zero.

As we have previously mentioned, we are particularly interested in the search step metric since it represents the only non-deterministic portion of the proof search. That the median number of such steps is zero is extremely heartening and we are further encouraged that no VC required more than four. Figure **??** shows a histogram of the number of VCs requiring different numbers of search steps.

The majority (86, or 65%) of VCs required no search steps once all heuristics were applied. 40 (30%) required three or fewer steps, while only a single VC required four search steps. This data supports our hypothesis that VCs arising from well-engineered software should require only simple analysis with a minimalist prover to dispatch.

6 Conclusions

Verification conditions (VCs) of correctness that arise from well-engineered and properly-justified programs are relatively straightforward to prove, provided there are suitable supporting mathematical developments and sound results. When this is the case, a minimalist prover can discharge the VCs automatically. A key advantage of the prover is that it is independent of the mathematical domains that are involved in the VCs, and is thus attractive for proving VCs that span multiple theories. Using experimentation with a suite of benchmarks, we have evaluated various heuristics that such a prover might use and the relative benefits of those heuristics. The results of verification on the benchmarks are promising. However, further experimentation and research is necessary to confirm the benefits.

7 Acknowledgments

We thank members of the RESOLVE/Reusable Software Research Groups at Clemson and Ohio State for their comments concerning the contents of this paper. This research is funded in part by US National Science Foundation grants DMS-0701187, and CCF-1161916.

References

1. B. Adcock. *Working Towards the Verified Software Process*. Phd dissertation, Ohio State University, 2010.
2. C. Barrett. decision procedures: An algorithmic point of view, by daniel kroening and ofer strichman, springer-verlag, 2008. *Journal of Automated Reasoning*, 51(4):453–456, 2013.
3. N. Bjørner. Engineering theories with z3. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, APLAS'11, pages 4–16, Berlin, Heidelberg, 2011. Springer-Verlag.
4. D. Bronish and H. Smith. Robust, generic, modularly-verified map: a software verification challenge problem. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 27–30, New York, NY, USA, 2011. ACM.
5. C. Cook, H. Harton, H. Smith, and M. Sitaraman. Specification engineering and modular verification using a web-integrated verifying compiler. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1379–1382, June 2012.
6. L. de Moura and N. Bjrner. Z3: An efficient smt solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78800-3_24.
7. B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, SRI International, 2006.
8. H. Harton. *Mechanical and Modular Verification Condition Generation for Object-Based Software*. Phd dissertation, Clemson University, School of Computing, Dec. 2011.

9. T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50:63–69, January 2003.
10. W. Hunt, M. Kaufmann, R. Krug, J. Moore, and E. Smith. Meta reasoning in acl2. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin / Heidelberg, 2005. 10.1007/11541868_11.
11. M. Kaufmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Trans. Softw. Eng.*, 23:203–213, April 1997.
12. J. Kirschenbaum, B. Adcock, D. Bronish, H. Smith, H. Harton, M. Sitaraman, and B. Weide. Verifying component-based software: Deep mathematics or simple bookkeeping? In S. Edwards and G. Kulczycki, editors, *Formal Foundations of Reuse and Domain Engineering*, volume 5791 of *Lecture Notes in Computer Science*, pages 31–40. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-04211-9_4.
13. V. Klebanov, P. Müller, N. Shankar, G. T. Leavens, V. Wüstholz, E. Alkassar, R. Arthan, D. Bronish, R. Chapman, E. Cohen, M. A. Hillebrand, B. Jacobs, K. R. M. Leino, R. Monahan, F. Piessens, N. Polikarpova, T. Ridge, J. Smans, S. Tobies, T. Tuerk, M. Ulbrich, and B. Weiß. The 1st verified software competition: Experience report. In M. Butler and W. Schulte, editors, *FM*, volume 6664 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2011.
14. K. Leino and R. Monahan. Dafny meets the verification benchmarks challenge. In G. Leavens, P. O'Hearn, and S. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 112–126. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15057-9_8.
15. LogiCal Project. *The Coq proof assistant reference manual*, 2004. Version 8.0.
16. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53:937–977, November 2006.
17. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.
18. Resolvevc generator, Sept. 2010.
19. K. Rustan and M. Leino. Dafny: An automatic program verifier for functional correctness. LPAR 16 (to appear), 2010.
20. M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith, and B. W. Weide. Building a pushbutton RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing (to appear)*, 2010.
21. H. Smith. *Engineering Specifications and Mathematics for Verified Software*. Phd dissertation, Clemson University, School of Computing, 2013.
22. B. W. Weide, M. Sitaraman, H. K. Harton, B. M. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Frazier. Incremental benchmarks for software verification tools and techniques. In *Verified Software: Theories, Tools, and Experiments (VSTTE)*, pages 84–98, 2008.
23. M. M. Wenzel and T. U. Mnchen. Isabelle/isar - a versatile environment for human-readable formal proof documents. In *TPHOLS*, pages 167–184, 1999.