Automatically Proving Thousands of Verification Conditions Using an SMT Solver: An Empirical Study

Aditi Tagore, Diego Zaccai, and Bruce W. Weide

Dept. of Computer Science and Engineering
The Ohio State University
Columbus, Ohio 43210, USA
{tagore.2,zaccai.1,weide.1}@osu.edu

Abstract. Recently it has become possible to verify full functional correctness of certain kinds of software using automated theorem-proving technology. Empirical studies of the difficulty of automatically proving diverse verification conditions (VCs) would be helpful. For example, they could help direct those developing formal specifications toward techniques that tend to simplify VCs. They could also help focus the efforts of those improving automated theorem-proving tools that are targeted to handle VCs. This study explores two specific empirical questions of this sort: How does an SMT solver perform on VCs that involve user-defined mathematical functions and predicates? When it does not perform well, what can be done to improve the prospects for automated proof? Experience using Z3 to prove VCs for a solution to a fully generic sorting benchmark, along with thousands of other VCs generated for both clients and implementations of dozens of RESOLVE software components, suggests that providing the prover with universal algebraic lemmas about user-defined mathematical functions and predicates results in better outcomes than expanding (unfolding) definitions. The importance of such lemmas might not be surprising to those who have tried to carry out such proofs manually or with the help of an interactive prover, but the damage sometimes caused by expanding definitions might be unexpected. A large empirical study of these phenomena in the context of automated software verification has not been previously reported.

1 Introduction

We took the following steps for this study. First, we selected a variety of about 50 RESOLVE [1] software components comprising about 2000 lines of code, including the components involved in an earlier empirical study of different issues [2]. This code includes arithmetic algorithms over integers and natural numbers; sorting of arbitrary items with arbitrary orders; and a variety of client-view manipulations of, and internal data representations for, stacks, queues, lists, sets, etc. These components have specifications based on standard design-by-contract principles with pre- and post-conditions, and the code to be verified includes the

A. Goodloe and S. Person (Eds.): NFM 2012, LNCS 7226, pp. 195–209, 2012.

necessary annotations: loop invariants, progress metrics, representation invariants, and abstraction functions, as appropriate, but very few other assertions. Second, we generated the 4028 VCs needed to prove the full functional correctness of all this code, including termination, by using the RESOLVE VC generator [3]. We believe all these VCs are valid; while we have plenty of code that contains (mostly intentional) bugs, all such code was removed from our library for this study. Third, we machine-translated each VC into Dafny [4] from which it was fed to the SMT solver Z3 [5] for an automated proof attempt, and we recorded what happened. Fourth, we studied carefully the 503 VCs that were *not* proved automatically by Z3 (about 12%; see the red or dark gray region in the middle of the top bar in Figure 1) and tried to determine which changes to specifications, code, or anything else *under the control of the software developer* would improve automated proof success. We emphasize that the intent of this tool-chain is to prove full functional correctness of the code by automated theorem-proving technology, as opposed to abstracting that code to possibly simpler but incomplete finite-state models to be analyzed by a model-checker.

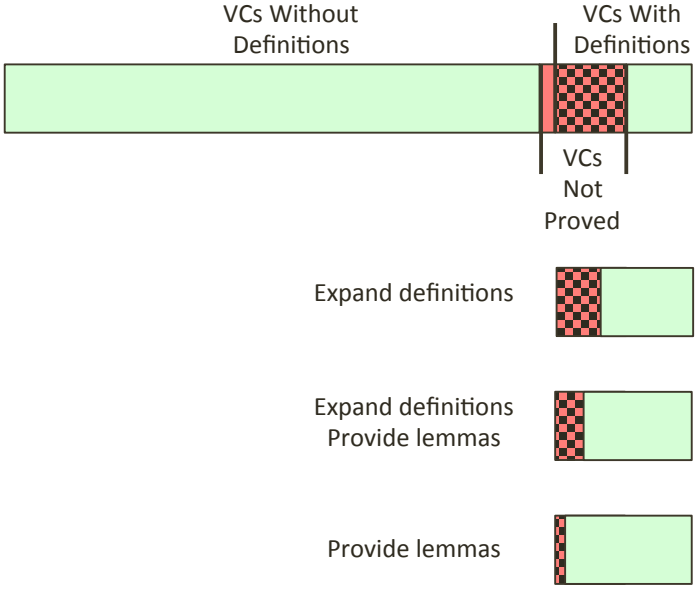

Fig. 1. Results of Proof Attempts on 4028 VCs: For the top bar, user-defined symbols appearing in VCs are treated as uninterpreted function or predicate symbols; for the lower three bars, their definitions are expanded to the prover and/or universal lemmas about those functions and predicates are provided to the prover, as indicated

It turned out that 425 of the 503 VCs *not* proved by Z3 in the first step were among the 817 VCs that involved used-defined mathematical function or predicate symbols. The automated proof success rate for these VCs was so low—about 48%, compared to about 98% for VCs not involving such symbols—that

we chose to focus our attention on what one might be able to do about them to improve the automated proof success rate. The rest of this paper is about these 425 VCs (represented in the checkerboard pattern in the top bar of Figure 1) and others like them that might arise in future software verification efforts. It explains how we improved the automated proof success rate for VCs involving user-defined mathematical functions and predicates from 48% to 93%. It is likely that formal specifications of software will often involve some *ad hoc* user-defined mathematical functions and predicates of the sort seen in these VCs, especially when domain-specific software components and application software are involved. So these results are potentially significant.

In considering these 425 VCs, we took the viewpoint of software engineers or mathematical specialists working on the software to be verified. We did not seek to reverse-engineer or second-guess Z3. Any advice provided here is, therefore, intended to be used by software verification researchers (and soon, we believe, by professionals working under a verified software paradigm). That advice is related to issues these "users" of software verification tools should both understand and be able to control, not to internal details of a black-box automated theorem-prover like Z3 that happens to be part of a tool chain. If any results also end up being of interest to those developing better automated provers for VCs, so much the better. We therefore wish to emphasize that none of our results should be interpreted as being critical of Z3. Indeed, both Z3 and the other automated provers we have used in the RESOLVE software verification tool-chain [3] perform far better than their predecessors of the 1990s, when we attempted a similar study on a smaller scale and found that few VCs could be proved automatically. In fact, maybe Dafny/Z3 can do better today on some VCs than the version available at the time of the study: Dafny version 2, which uses Z3 version 2.15. However, we are confident our basic conclusions would remain valid if Z3 were incrementally improved, or even if a different automated prover were substituted for it.

The paper makes two primary contributions. The first contribution is the empirical study itself, including its scale and the conclusions drawn from it: the importance of providing universal algebraic lemmas about user-defined mathematical function and predicates appearing in formal specifications, combined with the importance of not expanding definitions to reveal hidden quantifiers and even more definitions. Overviews of the study process and results are presented here, and full details are available via a companion website that is the first effort to provide such detailed information about so many VCs and the results of attempting to prove them automatically. The second contribution is the first entirely automatic proof of sorting code in which both the type of items being sorted and the total pre-ordering by which they are sorted are client-supplied parameters; this is one of the software verification benchmarks proposed in [6].

2 User-Defined Mathematical Functions and Predicates

Any practical language for writing mathematics must allow new mathematical functions and predicates to be introduced via **user-defined symbols**. This is

particularly important for a software specification language because specifications are mathematical statements meant to be read by software developers. Indeed, carefully considered and well-named mathematical functions and predicates can dramatically simplify mathematical statements while providing intuition and understanding for humans.

As a simple example, consider a specification involving odd integers. If the mathematical language used to write this specification has no built-in functions or predicates with which to make direct statements about odd integers, then the specifier might choose to write out, everywhere she needs to say "n is odd", a relatively cumbersome expression such as $\exists k : integer\ (n = 2k + 1)$. The capability to introduce new mathematical symbols and to define them gives her the alternative of saying "$ODD(n)$" in all these places; of course, if she introduces the symbol ODD she must say once and for all that $ODD(n)$ is defined as $\exists k : integer\ (n = 2k + 1)$. We note as an important empirical matter that it is typical for a user-defined mathematical function or predicate symbol such as ODD to hide one or more quantifiers in its definition.

RESOLVE does, therefore, support user-defined mathematical function and predicates. There are a number of mathematical theories and associated operators built-in to the language: booleans, integers, tuples, strings, finite sets, functions, relations, etc. In this study, we deal only with new user-defined mathematical functions and predicates within these theories (i.e., using their mathematical types, or sorts), not with entirely new user-defined mathematical theories.

There are two distinct ways in RESOLVE to introduce a new mathematical function or predicate symbol via a **signature** and provide its definition in a **body**:

- **Explicit definition**: the body is an expression of the result type of the function or predicate.
- **Implicit definition**: the body is an assertion (involving the function or predicate symbol being defined) that uniquely characterizes the function or predicate.

The definition of ODD above uses an explicit definition. An inductive definition is one style of implicit definition. Detailed examples of both explicit and implicit definitions are discussed in Section 5.

It is also possible in RESOLVE for a mathematical function or predicate to be a parameter to a software component. A body does not appear in the software component where this parameter is introduced, but rather in some other component (typically a client program) that completes the definition by binding that parameter to a specific mathematical function or predicate symbol with the same signature. RESOLVE permits the specifier introducing such a parameter to place a **restriction** on the definition ultimately to be bound to it by stating a property that the definition must satisfy—without uniquely characterizing the mathematical function or predicate. A specific example, a client-supplied ordering relation for sorting, is central to the discussion in Section 5.

In a verified software paradigm as envisioned by Hoare [7], mathematical statements in software specifications are seen not only by software developers but also

by automated provers as they attempt to prove verification conditions (VCs). If user-defined symbols are simply expanded ("unfolded") into their bodies in the VC proof process, thereby reintroducing the quantifiers and other complexities they were designed to bury for the software developer, then the benefits of user-defined mathematical function and predicate symbols are limited to the human writers and readers of specifications.

Sometimes, no knowledge at all about the definition is needed to prove a VC in which a user-defined mathematical function or predicate symbol appears [2]; it can be treated as an uninterpreted symbol. One assumption in a VC might have a form such as $ODD(n) \implies P$, while another assumption is simply $ODD(n)$. The prover concludes P from these two assumptions and proceeds to use that fact in the proof of the conclusion of the VC—without knowing anything more about ODD. Expanding definitions is not needed here.

Other times, when certain properties associated with a user-defined function or predicate such as ODD are needed in the proof of a VC, they can be stated as universal algebraic lemmas, e.g., $ODD(n) \iff \neg\ ODD(n+1)$; or, restated without free variables, $\forall n : integer(ODD(n) \iff \neg\ ODD(n+1))$. Again, expanding definitions is not needed.

In summary, then, when a user-defined mathematical function or predicate symbol appears in a VC, automated proof of that VC may seem to require the prover to have neither, one, or both of (a) the expanded definition, and (b) appropriate lemmas about it. The question studied in this paper is to what extent it is helpful to keep the complexity of expanded definitions hidden from an automated prover, since automated provers (like humans) often have considerable difficulty dealing with quantifiers.[1] The answer is that simply providing the prover with universal algebraic lemmas about user-defined mathematical functions and predicates is generally far better than expanding their definitions.

3 The Tool Chain

Several tools were integrated for this study to process the pipeline from specifications and code to proof of correctness. Our specifications are written in RESOLVE [1] using some of its built-in mathematical theories and its capabilities for making both explicit and implicit definitions and its support for generic parameters. Code to be verified is also written in RESOLVE. VCs are generated by the OSU RESOLVE verification tools [3] using proof rules described in [8,9]. These VCs along with the theories and mathematical definitions used in them are then (by further automatic translation) expressed as assumptions and assertions in Dafny [4,10]. Dafny translates these "VC programs" into the Boogie intermediate language [11], which in turn generates its own VCs to be proved by the SMT solver Z3 [5].

Dafny is a tool for writing programs annotated with various mathematical and specification statements. Dafny's features for making definitions and for writing

[1] Not in the case of ODD, which is easy to reason about automatically because it is in Presburger arithmetic, but in general.

assumptions and assertions make it an attractive translation target. We encode each definition as a static function in Dafny, with the definition body encoded as an assumption about it. Each VC generated from the RESOLVE code is encoded in an ordinary method in Dafny that consists of a series of assumptions followed by an assertion: each assumption A_i in the VC generates an `assume` statement, and the conclusion C generates an `assert` statement.

It would have been possible to translate RESOLVE specifications and code for realizations into Dafny or Boogie, but this would have introduced the reference semantics that is inherent in these languages and that RESOLVE avoids. So, we encode RESOLVE VCs themselves into Dafny programs rather than translating them into Z3's input format. The reason for this design choice (rather than providing Z3 with an axiomatic description of, say, RESOLVE's string theory, and relying on its core logic capabilities alone) is that Dafny offers similar mathematical theories into which we can directly translate RESOLVE-generated VCs. So, for instance, Dafny's sequences can be used as a translation target for RESOLVE's strings. Having no direct control over any details in this entire black-box back-end of the tool chain, we trust, of course, that Dafny's interface to Boogie and Boogie's to Z3 appropriately encode these features and that Z3 proves only valid VCs; we have no reason to suspect otherwise.

4 The Study

For the 817 VCs involving user-defined mathematical functions and predicates, we followed a sequence of four steps in an attempt to prove each of them.

In step 1, we attempted to prove each VC without providing Dafny with any information (except the signature) for any user-defined mathematical function or predicate symbol appearing in it. This is also the only step in which we tried to prove VCs that do not have any user-defined symbols in them at all, since providing more information about things not even indirectly appearing in a VC presumably cannot help the prover. In principle, this still allows us to prove any VC whose proof does not require the body or any other knowledge about a user-defined symbol. In Table 1 we see (also shown graphically in Figure 1) that Z3 performed extremely well on VCs without any user-defined mathematical functions or predicate symbols, proving almost 98% of them. It even proved 48% of the VCs that contain such symbols, without any knowledge at all about the underlying mathematical functions and predicates they denote, as illustrated with a similar hypothetical situation involving *ODD* in Section 2.

In step 2, we provided as an additional assumption of each VC an expanded definition of each user-defined symbol appearing in it. We provided its body for an explicit or implicit definition and its restriction for a parameter. An important point here is that the typical body or restriction involves at least one quantifier and sometimes alternation of quantifiers. This raises the level of complexity of the VCs for the automated prover. Indeed, hiding this complexity from the reader of specifications is one of the primary reasons for introducing a new user-defined mathematical function or predicate in the first place. Yet with expanded definitions available, Z3 nonetheless proved many more of the VCs that

involve user-defined mathematical function or predicate symbols: 68% of them, compared to 48% without this additional information.

In step 3, we added to the assumptions of each VC some universal algebraic lemmas about the user-defined mathematical functions and predicates appearing in it, or in the expanded definitions already added as assumptions in step 2. Most of the provided lemmas were from reusable mathematical developments related to the definitions and independent of any VC, as illustrated with *ODD* in Section 2. A few others were not independently identified as mathematically interesting and highly reusable, but rather were directly suggested by human proof attempts on particular VCs that were not proved automatically by Z3. This improved the success rate for Z3 to about 79% of the VCs when *both* expanded definitions and universal algebraic lemmas were available.

In step 4, we removed the bodies of definitions, but left the restrictions on user-defined symbols that are parameters; these restrictions are generally so central to the code in which they appear that they must be kept to have any hope of proving the resulting VCs. Moreover, they tend to be syntactically indistinguishable from universal algebraic lemmas, as seen in Section 5. The *removal* of expanded definitions significantly improved the success rate of Z3 to about 93% of the VCs involving user-defined mathematical functions and predicates.

Table 1. Summary of Empirical Results

Component family	VCs without a user-defined symbol	VCs with a user-defined symbol			
		No expansion; no lemmas	Expansion; no lemmas	Expansion; lemmas	No expansion; lemmas
List	(58/58)	(52/84)	(78/84)	(80/84)	(83/84)
Queue	(48/48)	(108/248)	(155/248)	(222/248)	(240/248)
Sequence	(53/53)	(50/98)	(75/98)	(79/98)	(87/98)
Stack	(6/7)	(7/25)	(17/25)	(25/25)	(21/25)
Integer	(755/788)	(-/-)	(-/-)	(-/-)	(-/-)
Natural	(2147/2190)	(32/48)	(43/48)	(46/48)	(46/48)
Set	(22/23)	(100/242)	(130/242)	(129/242)	(215/242)
Array	(3/3)	(21/31)	(30/31)	(31/31)	(31/31)
BooleanFacility	(23/23)	(-/-)	(-/-)	(-/-)	(-/-)
Others	(18/18)	(22/41)	(30/41)	(35/41)	(38/41)
VCs proved / VCs total	(3133/3211)	(392/817)	(558/817)	(647/817)	(761/817)
% VCs proved	97.6%	48.0%	68.3%	79.2%	93.1%

Table 1 summarizes the empirical study results. Detailed information showing specifications and code for all components, the VCs generated, the user-defined symbols with their bodies and restrictions, and the lemmas are available at http://resolve.cse.ohio-state.edu:8080/archive/nfm2012/ .

5 Example: Fully Generic Sorting

To illuminate issues and results regarding what happens when trying to prove VCs—by expanding the definitions of user-defined mathematical function and predicates and/or by providing universal algebraic lemmas about them—we offer one of the verification benchmark problems proposed in [6]: a generic sorting program in which both the type of the entries to be sorted and the ordering are client-supplied parameters. We specify and then verify an implementation (using merge sort) of an operation that sorts an ordered collection of entries of a user-supplied type according to a user-supplied total pre-order. This benchmark has been addressed by others, e.g., [12], for the case of sorting a fixed type (integers) according to a fixed total pre-order ($\leq$). The fully generic version offered as a benchmark is challenging precisely because it involves an obvious need to introduce some user-defined symbols to simplify the specification. Moreover, the code for the merge sort algorithm is not trivial and involves the standard sequential programming constructs including recursion.

We store the elements to be sorted in a Queue, whose mathematical model is a string of (the mathematical model of) the type of the elements to be sorted. The contract QueueTemplate provides four typical operations: Enqueue, Dequeue, Length, and Is_Empty. The elements inside a Queue can only be obtained by the Dequeue operation in a FIFO order, and there is no way to access the elements inside a Queue other than removing them from it. All code for this example can be found, accompanied with explanations of the relevant RESOLVE language features, on the web site mentioned at the end of Section 4.

Figure 2 shows the contract of an enhancement, or extension, of the QueueTemplate contract, called SortExtension. It specifies an operation to Sort a Queue. Note that the contract is parametrized by a binary relation ARE_IN_ORDER that is restricted to be a total pre-order. The ellipsis in this code is where the user-defined mathematical function and predicates in Figure 3 appear. These four definitions (signatures and their bodies) are structured in such a way that the two appearing in the contract of Sort (ARE_PERMUTATIONS and IS_NONDECREASING) are defined in terms of the other two (OCCURS_COUNT and PRECEDES). This style of making definitions is typical. The intent of each definition is as follows:

- OCCURS_COUNT is the number of occurrences of its second argument (an Item) in its first argument (a string of Items). This is an implicit definition, introduced by the keyword **satisfies** followed by an assertion in which OCCURS_COUNT appears.
- ARE_PERMUTATIONS is true whenever its two arguments (strings of Items) are permutations of one another. This is an explicit definition, introduced by the keyword **is** followed by an expression of the result type.
- PRECEDES is true whenever every entry in its first argument (a string of Items) is "in order with" every entry in its second argument (a string of Items), where the order is based on the relation ARE_IN_ORDER.
- IS_NONDECREASING is true iff its argument (a string of Items) is sorted.

```
contract SortExtension (
  definition ARE_IN_ORDER (x: Item, y: Item): boolean satisfies restriction
   for all z: Item ((ARE_IN_ORDER (x, y) or ARE_IN_ORDER (y, x)) and
     (if (ARE_IN_ORDER (x, y) and ARE_IN_ORDER (y, z)) then ARE_IN_ORDER (x, z)))
  ) enhances QueueTemplate
  ...
  procedure Sort (updates q: Queue)
    ensures
      ARE_PERMUTATIONS (q, #q) and IS_NONDECREASING (q)
end SortExtension
```

Fig. 2. Sort Specification

```
definition OCCURS_COUNT (
  s: string of Item,
  i: Item
 ) : integer satisfies
 if s = <> then
  OCCURS_COUNT (s, i) = 0
 else there exists x: Item,
                  r: string of Item
  (s = <x> * r and
   (if x = i then OCCURS_COUNT (s, i)
      = OCCURS_COUNT (r, i) + 1
    else OCCURS_COUNT (s, i)
      = OCCURS_COUNT (r, i)))

definition ARE_PERMUTATIONS (
  s1: string of Item,
  s2: string of Item
 ) : boolean is
 for all i: Item
  (OCCURS_COUNT (s1, i)
     = OCCURS_COUNT (s2, i))
```

```
definition PRECEDES (
  s1: string of Item,
  s2: string of Item
 ) : boolean is
 for all i, j: Item
  where (OCCURS_COUNT (s1, i) > 0 and
         OCCURS_COUNT (s2, j) > 0)
    (ARE_IN_ORDER (i, j))

definition IS_NONDECREASING (
  s: string of Item
  ) : boolean is
 for all a, b: string of Item
  where (s = a * b)
    (PRECEDES (a, b))
```

Fig. 3. Mathematical Definitions Used in SortExtension

In Section 6 we use sample VCs from the MergeSort realization of the contract in Figure 2 to illustrate the kinds of VCs proved in each step of the study. It is important to notice that this realization is parametrized by a **control**-valued *programming* function AreInOrder which returns true whenever its arguments satisfy the *mathematical* relation ARE_IN_ORDER (described previously and arising as a separate parameter to the SortExtension contract). The full separation of mathematical and programming functions as illustrated here is an important distinctive feature of RESOLVE. As it is often difficult to select names that convey this distinction, we adopt a typographical convention: all-upper-case identifiers are reserved for mathematical functions and predicates. It is worth noting that since we view the VCs to be putative mathematical theorems, only mathematical symbols (not the names of programming entities) appear in them.

6 VCs from the Fully Generic Sorting Example

In this section, we show some VCs from the MergeSort verification that are proved in each step of the study—and some that are not. The intent is to give

an indication of some of the difficulties that arise in this example—and some that do not. A total of 62 RESOLVE VCs are generated from the MergeSort code, of which 58 mention user-defined mathematical functions or predicates.

Figure 4 shows a VC that is proved in step 1 of the study, i.e., with all user-defined symbols treated as uninterpreted function and predicate symbols. Although this VC involves the predicate ARE_PERMUTATIONS in its assumptions and conclusions, it is proved without knowledge of anything more than the signature of ARE_PERMUTATIONS. This VC is proved easily due to a contradiction involved in one of the assumptions. Less than half (28 of 58) of the interesting VCs were proved in the first step.

```
var q1_4, q2_4, q1_0 : seq<T> ;
var x_6: T ;
...
assume ARE_PERMUTATIONS((((([x_6] + [])
 + q1_4) + q2_4), ((q1_0 + []) + []));
assume (([x_6] + []) == []);
...
assert ARE_PERMUTATIONS((((] + (q1_4
+ [x_6])) + q2_4), ((q1_0 + []) + []));
```

Fig. 4. VC Proved in Step 1

```
var q2_3, q1_0 : seq<T> ;
var q2Item_3: T ;
...
assume (|q1_0| > 0);
assume (|([q2Item_3] + q2_3)| > 0);
...
assert ARE_PERMUTATIONS(
((([] + q1_0) + q2_3) + [q2Item_3]),
((([] + q1_0) + q2_3) + [q2Item_3]));
```

Fig. 5. VC Proved in Step 2

Figure 5 shows a VC proved in step 2 but not in step 1. Its assumptions include the expanded definitions shown in Figure 3 (but not shown here). The conclusion of this VC states that a particular string is a permutation of itself, which unsurprisingly cannot be proved if ARE_PERMUTATIONS is treated as an uninterpreted predicate symbol. With expanded definitions provided, Z3 is able to prove 36 of the 58 VCs containing user-defined function and predicate symbols.

Notice that expanding definitions opens up everything to the prover, including the definitions of OCCURS_COUNT and PRECEDES that do not directly appear in the specification of Sort. In software engineering terms, we might say that expanding definitions breaks encapsulation and flattens out all the underlying mathematical machinery devised to write the specification. There is no information hiding—either from a human reader or a prover—when definitions are expanded.

Figures 6 and 7 show examples of two VCs that are proved in step 3 but not in step 2. In addition to expanding definitions, we now provide the prover with some simple universal algebraic lemmas about the user-defined mathematical functions and predicates. We encode some lemmas involving ARE_PERMUTATIONS and IS_NONDECREASING as shown in Figures 8 and 9. Z3 proves 57 of the 58 interesting VCs with this additional information.

The VC in Figure 6 contains among its assumptions (not shown here) the expanded definitions of ARE_PERMUTATIONS and OCCURS_COUNT. It also has several more assumptions, all but one elided as irrelevant to the proof. To a human, the conclusion seems easily provable from the given assumption along with an understanding of concatenation and the meaning of ARE_PERMUTATIONS. Some of the lemmas provided as additional assumptions capture this feature.

```
var q1_0, q2_3, q1_6, tmp_4, q2_4 : seq<T> ;
var q2Item_3, q1Item_6, q2Item_4 : T ;
...
assume ARE_PERMUTATIONS((((tmp_4 +
([q1Item_6] + q1_6)) + q2_4) + [q2Item_4]),
 ((([] + q1_0) + q2_3) + [q2Item_3]));
...
assert ARE_PERMUTATIONS(((((tmp_4 +
[q2Item_4]) + q2_4) + q1_6) + [q1Item_6]),
 ((([] + q1_0) + q2_3) + [q2Item_3]));
```

Fig. 6. VC Proved in Step 3

```
var tmp_4, q2_4: seq<T>;
var q2Item_4: T;
...
assume IS_NONDECREASING(
 ((tmp_4 + [q2Item_4]) + q2_4));
...
assert IS_NONDECREASING(
 ([q2Item_4] + q2_4));
```

Fig. 7. VC Proved in Step 3

The VC in Figure 7 also includes two expanded definitions (not shown here), and has several other assumptions, all but one elided as irrelevant. To a human, it is obvious that if IS_NONDECREASING holds for a string formed by the concatenation of three strings, then it holds for the concatenation of two of them in the same order. But here it is less clear how long a reasoning path is required for an automated prover to notice this without some simple lemmas to help.

```
1. forall a:seq<T>:: ARE_PERMUTATIONS(a,a)

2. forall a:seq<T>, b:seq<T>, c:seq<T>:: ARE_PERMUTATIONS(a,b) &&
    ARE_PERMUTATIONS(b,c) ==> ARE_PERMUTATIONS(a,c)

3. forall a:seq<T>, b:seq<T>:: ARE_PERMUTATIONS(a,b) ==> ARE_PERMUTATIONS(b,a)

4. forall a:seq<T>, b:seq<T>, c:seq<T>:: ARE_PERMUTATIONS((a + b) + c, a + (b + c))

5. forall a:seq<T> , b: seq<T> :: a == b ==> ARE_PERMUTATIONS(a, b)

6. forall a:seq<T>, b:seq<T>:: ARE_PERMUTATIONS(a,b) ==> |a| == |b|
```

Fig. 8. ARE_PERMUTATIONS Lemmas

An obvious and important question is, "Which lemmas should be provided to the prover?" The lemmas in Figure 8 for ARE_PERMUTATIONS are the usual equivalence relation properties along with a few others that might be given as problems in an undergraduate math/logic textbook. Given lemma 1, lemma 4 merely restates that concatenation is associative. It turns out to be important for Z3 to have this property separately in order to prove some of the MergeSort VCs involving ARE_PERMUTATIONS. Lemma 5 says the same thing as lemma 1; yet it helps Z3 prove some VCs where lemma 1 does not. In short, none of these lemmas is surprising except possibly for the fact that it helps Z3 when properties are stated in a particular way. This is not a shortcoming of the concept of using universal algebraic lemmas in proofs of VCs, but rather appears to be the expression of a current limitation on how they are processed by the prover.

The lemmas added for IS_NONDECREASING are more extensive. The most basic set are lemmas 1 through 3 in Figure 9. The second set (4 through 6) are of a different nature, relating various concatenations of strings satisfying

```
1. IS_NONDECREASING([])

2. forall x:T :: IS_NONDECREASING([x])

3. forall q:seq<T> :: |q| <= 1 ==> IS_NONDECREASING(q)

4. forall x:seq<T>, y:seq<T>:: IS_NONDECREASING(x + y) ==>
      IS_NONDECREASING(x) &&  IS_NONDECREASING(y)

5. forall a:seq<T>, b:seq<T>, c:seq<T>:: IS_NONDECREASING(a + b + c) ==>
      IS_NONDECREASING(a + b) && IS_NONDECREASING(b + c) &&
      IS_NONDECREASING(a + c)

6. forall a:seq<T>, b:seq<T>, c:seq<T>:: IS_NONDECREASING(a + c) &&
      IS_NONDECREASING(c + b) && c!= [] ==> IS_NONDECREASING(a + c + b)

7. forall a:T, b:T :: ARE_IN_ORDER(a,b) ==> IS_NONDECREASING([a] + [b])

8. forall a:seq<T>, b:seq<T>, x:T, y:T :: IS_NONDECREASING(a + [x]) &&
      IS_NONDECREASING([y] + b) && ARE_IN_ORDER(x,y) ==>
      IS_NONDECREASING(a + [x] + [y] + b)

9. forall a:seq<T>, x:T, y:T ::IS_NONDECREASING([x] + a + [y]) ==>
      ARE_IN_ORDER(x,y)
```

Fig. 9. IS_NONDECREASING Lemmas

IS_NONDECREASING. The third set (7 through 9) are different still, relating ARE_IN_ORDER and IS_NONDECREASING. All these lemmas were proved interactively with the help of Isabelle used as a proof assistant.

```
var q1_0,q2_3,tmp_17,q2_17 : seq<T> ;
var q2Item_3,q2Item_17 : T ;

assume ARE_PERMUTATIONS((((tmp_17+[])+q2_17)+[q2Item_17]),
            ((([]+q1_0)+q2_3)+[q2Item_3]));
assert ARE_PERMUTATIONS(((tmp_17+[q2Item_17])+q2_17),(q1_0+([q2Item_3]+q2_3)));
```

Fig. 10. VC Proved in Step 4

Figure 10 shows the lone VC not proved in a previous step but successfully proved in step 4, where we remove expanded definitions and leave only universal algebraic lemmas about them. Z3 now proves all 58 interesting lemmas.

It is important to note that in the absence of the expanded definitions, providing universal algebraic lemmas does *not* open up everything to the prover. In particular, here the very existence of OCCURS_COUNT and PRECEDES remains hidden because they do not directly appear in the specification of Sort. In software engineering terms, providing universal algebraic lemmas about defined functions and predicates respects encapsulation and leverages all the underlying mathematical machinery devised by the software developer to write the specification. Information hiding survives when definitions are not expanded.

7 Discussion

Intuition for believing that otherwise troublesome quantifiers in VCs might be finessed by introducing universally quantified lemmas comes from observing the practice in calculus where, for example, most results are established (by humans) not by appealing to a complex nested quantification like that required to define the concept of a limit, but rather by reusing universal algebraic results proved separately, e.g., $lim\ (f + g) = lim\ f\ +\ lim\ g$. A similar situation characterizes reasoning using big-O notation. The value of universal algebraic lemmas seems clear from such experience with fully manual proofs, as well as from anecdotal evidence involving interactive proofs (e.g., [13,14]). However, it is less clear that such lemmas should be *so helpful* to an automated prover that they should help it produce—fully automatically—proofs of VCs involving complex definitions.

To see why this is plausible, consider a VC in which user-defined symbols appear, but in which no definitions are expanded and no properties about those definitions appear. The form of such a VC as generated by the OSU RESOLVE verification tools [3] is always $\bigwedge A_i \implies C$. When definitions are used to hide *all* quantifiers in software specifications (a recommendation we have followed in the specifications used in this study), all the variables (call them $x_1, ..., x_m$) in the assumptions $A_1, ..., A_n$ and the conclusion C are free variables; equivalently, there is an implicit universal quantifier in front: $\forall x_1, ..., x_m(\bigwedge A_i \implies C)$. Depending on the combination of functions and predicates appearing in the VC, automated provers may do well or not so well on it. Yet if there is trouble then at least it is not the fault of the quantifier structure, because such VCs are in a form to which automated provers are well suited.

On the other hand, if a definition hiding quantifiers is expanded in the VC, then these newly exposed quantifiers might, or might not, cause serious additional trouble for an automated prover. For example, in the lucky special case that one of the assumptions, say A_k, expands to the form $\exists y(P(y))$, then there is no problem at all: x_{m+1} can be introduced as a new free variable and A_k can be replaced with $P(x_{m+1})$, i.e., x_{m+1} is simply treated as a witness to the existence of a value that makes P true. The structure of the revised VC remains the same as the original: it is now $\forall x_1, ..., x_{m+1}(\bigwedge A_i \implies C)$.

A more difficult situation is the equally special case where one of the assumptions, say A_k, expands to the form $\forall y(P(y))$. Now, the prover must instantiate y with term(s) appearing in the VC, so the new instance(s) help in the overall proof. SMT solvers use "triggers" to match terms in such a way that the instantiated version(s) might be useful. Sometimes the prover can find an appropriate match automatically, and sometimes it needs a human-suggested trigger [5,15]. A universal algebraic lemma added as an additional assumption in a VC is exactly of this second, somewhat non-trivial, form. However, this is still far less complex a form for an automated prover to handle than an arbitrary quantified statement. Indeed, in some sense this is *the* non-trivial quantified form for which automated provers are tuned to work particularly well.

Though the general idea of providing universal algebraic lemmas about user-defined mathematical function and predicates has been reported in case studies

with interactive proofs [13, 14], it has not previously been systematically and empirically evaluated for use with automated provers for verification conditions.

8 Conclusions

We have presented empirical support for the claim that supplying universal algebraic lemmas about user-defined mathematical functions and predicates is, in general, a better way than expanding definitions to support automated verification of programs; and the approach does not assume that programmers have any knowledge about the intricacies of a back-end theorem prover (such as triggers or proof tactics). We also have demonstrated that the approach can be applied successfully for code whose specifications involve various mathematical theories.

There is clear potential for dependence of these results on specification and programming language features. The VCs we have seen from verification tools for other imperative languages, including Dafny itself and Jahob (for Java), are similar in basic mathematical content to VCs from RESOLVE programs. Yet there is one critical difference as well. RESOLVE programs have value semantics, not reference semantics; hence, there is no possibility for aliasing and no appearance of heap properties in RESOLVE VCs. This makes RESOLVE VCs relatively easier to prove, so our results in one direction should apply across the board: if expanding definitions does not lead to automated proofs of RESOLVE VCs using the same or similar back-end provers as are used for languages with reference semantics, it is unlikely that expanding definitions will lead to successful automated proofs of VCs that involve heap properties *in addition to* the properties of the primary mathematical models used in specifications. In the other direction, as far as we know it remains open to what extent providing universal algebraic lemmas to automated provers rather than expanding definitions has similar value for VCs that also involve heap properties.

Acknowledgment. Jason Kirschenbaum, Ted Pavlic, and Ray McDowell were especially helpful in contributing to this work. The authors are also grateful for the suggestions of Bruce Adcock, Derek Bronish, Paolo Bucci, Harvey M. Friedman, Wayne Heym, Dustin Hoffman, Bill Ogden, and Murali Sitaraman. This material is based upon work supported by the National Science Foundation under Grants No. DMS-0701260, CCF-0811737, and ECCS-0931669. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Sitaraman, M., Weide, B.: Component-based software using RESOLVE. SIGSOFT Softw. Eng. Notes 19, 21–63 (1994)
2. Kirschenbaum, J., Adcock, B., Bronish, D., Smith, H., Harton, H., Sitaraman, M., Weide, B.W.: Verifying Component-Based Software: Deep Mathematics or Simple Bookkeeping? In: Edwards, S.H., Kulczycki, G. (eds.) ICSR 2009. LNCS, vol. 5791, pp. 31–40. Springer, Heidelberg (2009)

3. Sitaraman, M., et al.: Building a push-button RESOLVE verifier: Progress and challenges. Formal Aspects of Computing 23, 607–626 (2011)
4. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
5. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., Frazier, D.: Incremental Benchmarks for Software Verification Tools and Techniques. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 84–98. Springer, Heidelberg (2008)
7. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM 50, 63–69 (2003)
8. Heym, W.D.: Computer program verification: improvements for human reasoning. PhD thesis, The Ohio State University, Columbus, OH, USA (1995)
9. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S.M., Hollingsworth, J.E.: Reasoning about Software-Component Behavior. In: Frakes, W.B. (ed.) ICSR 2000. LNCS, vol. 1844, pp. 266–283. Springer, Heidelberg (2000)
10. Leino, K.R.M.: Specification and verification of object-oriented software. Marktoberdorf International Summer School 2008, lecture notes (2008)
11. Leino, K.R.M.: This is Boogie 2. Manuscript KRML 178 (2008), `http://research.microsoft.com/en-us/um/people/leino/papers.html`
12. Leino, K.R.M., Monahan, R.: Dafny Meets the Verification Benchmarks Challenge. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 112–126. Springer, Heidelberg (2010)
13. Nelson, C.G.: Techniques for program verification. PhD thesis, Stanford University, Stanford, CA, USA (1980)
14. Kaufmann, M., Manolios, P., Moore, J.S. (eds.) Computer-Aided Reasoning: ACL2 Case Studies. Kluwer Academic Publishers (2000)
15. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52, 365–473 (2005)