

To Expand or Not to Expand: Automatically Verifying Software Specified With Complex Mathematical Definitions

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. An attractive strategy for writing human-readable model-based software specifications is to hide complexity, normally including quantifiers, inside mathematical definitions. A mathematical function or predicate may be defined in one of two ways: explicitly, i.e., as direct shorthand for another expression; or implicitly, i.e., via properties (perhaps including inductive ones) that uniquely characterize it. One faced with trying to prove automatically a verification condition (VC) involving a definition of either kind has two options: expand, or unfold, each use of the mathematical function or predicate into its definition; or add to the assumptions used in the proof of the VC certain properties of that mathematical function or predicate. Is it typically easier to prove a VC automatically by expanding definitions or by using the property-based approach? If the latter is preferred, what kinds of properties are most valuable to automated provers? Experience using the SMT solver Z3 to try to prove VCs for a solution to a fully generic sorting benchmark, as well as hundreds of other VCs generated for both clients and implementations of dozens of RESOLVE software components, suggests that providing the automated prover with universal algebraic lemmas involving defined mathematical functions and predicates is superior to the approach of expanding their definitions.

1 Introduction

Any practical language for writing mathematics must support making definitions of new functions and predicates. 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 definitions 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 $\exists k : \textit{integer} (n = 2k + 1)$. The capability to make new definitions gives her the alternative of saying “ $ODD(n)$ ” in all these places; of course, she must say once that $ODD(n)$ is defined as $\exists k : \textit{integer} (n = 2k + 1)$. We note as an important empirical matter that it is typical for a defined function or predicate such as ODD to hide one or more quantifiers in its definition.

In a verified software paradigm as envisioned by Hoare [1], 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 definitions are simply expanded 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 making definitions are limited to the human writers and readers of specifications. The question discussed in this paper is to what extent such complexity can be kept hidden from an automated prover as well, since automated provers (like humans) often have considerable difficulty dealing with quantifiers.¹ The answer is that providing the prover with *universal algebraic lemmas* about defined mathematical functions and predicates is generally far better than expanding their definitions. The primary contribution of the paper is the empirical support it provides for this conclusion. A secondary contribution is the first (to our knowledge) 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.

Section 2 explains why the conclusion that universal algebraic lemmas are very useful in proofs should itself not be surprising. Section 3 describes the tool chain we used for this study and our vision for the related software process. Section 4 explains a non-trivial example, chosen from a verification benchmark suite. Section 5 shows how the VCs for the code in the example are encoded in Dafny [2, 3] to be proved by Z3 [4]. Section 6 elaborates on the two different approaches to proving VCs that involve definitions. Section 7 analyzes the results for the example with each approach, and presents some summary data about results on over 4000 VCs from other code, of which over 800 involve definitions of the kind mentioned above. Section 8 discusses related work. Section 9 summarizes our conclusions.

2 Intuitive Basis for the Approach Used

Sometimes, no knowledge at all about a definition is needed to prove a VC in which it is used [5]; the defined function or predicate can be treated as an uninterpreted symbol. One assumption in a VC might have a form such as $ODD(n) \implies P(x)$, while another assumption is simply $ODD(n)$. The prover concludes $P(x)$ from these two assumptions and proceeds to use that fact in the proof of the conclusion of the VC—without knowing anything about ODD .

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

When certain properties associated with a defined function or predicate such as *ODD* are needed in the proof of a VC, though, they often can be stated as universal algebraic lemmas, e.g., $ODD(n) \iff \neg ODD(n+1)$; or, restated without free variables, $\forall n : integer(ODD(n) \iff \neg ODD(n+1))$.

Intuition for believing that otherwise troublesome quantifiers in VCs might be finessed in this way 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., [6,7]). It is less clear that such lemmas should be so helpful to an automated prover, however, that they should help it produce—fully automatically—proofs of VCs involving complex definitions.

To see why this result is plausible, first consider a VC in which uses of definitions 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 [8] is always $\bigwedge A_i \implies C$. The VC generator adds case splits to make VCs of this form so that they are more human readable. 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, \dots, x_m) in the assumptions A_1, \dots, A_n and the conclusion C are free variables; equivalently, there is an implicit universal quantifier in front: $\forall x_1, \dots, x_m (\bigwedge A_i \implies C)$. 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 $\forall x_1, \dots, x_{m+1} (\bigwedge A_i \implies C)$.

A more difficult situation is the equally special case where one of the assumptions, say A_k , expands 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 [9,4].

A universal algebraic lemma added as an additional assumption in a VC is exactly of the second, somewhat non-trivial, form outlined above. However,

this is still far less complex a form for an automated prover to handle than an arbitrary quantified statement, with arbitrarily nested quantifiers and no structural restrictions. Indeed, in some sense this is *the* non-trivial quantified form for which automated provers are tuned to work particularly well.

3 The Tool Chain and Some Process Implications

At first, the pipeline from specifications and code to proof-of-correctness for this study seems rather byzantine. However, it is not especially unusual given the current state of software verification technology building-blocks, which are now frequently integrated across projects in similar fashion. Our specifications are written in RESOLVE [10] using some of its built-in mathematical theories (booleans, integers, tuples, strings, finite sets, finite multisets, binary trees) and its capabilities for making both explicit and implicit definitions. Code to be verified is also written in RESOLVE. VCs are generated by the OSU RESOLVE verification tools [8] using proof rules described in [11,12]. 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 [2,3]. Dafny translates these “VC programs” into the Boogie intermediate language [13], which in turn generates VCs to be proved by the SMT solver Z3 [4].

Of course, there is the danger of presenting, to the prover, properties about a defined mathematical function or predicate via universal algebraic “lemmas” that are simply false. This is handled in our vision for a verified software paradigm by standard separation of concerns as observed throughout mathematics as well as software engineering. The prover is permitted to assume the universal algebraic lemmas we provide it *only* because its proof of any VC is considered contingent on, or relative to, the validity of these lemmas. The lemmas themselves are meant to be proved separately and not necessarily automatically; the cost of doing so is amortized over many VC proofs because these lemmas are fully reusable in proofs of VCs from other code involving the same mathematical theories and definitions. This is why we suggest that mathematical specialists be available both to assist software developers as they introduce definitions to be used in their specifications, and to prove purely mathematical statements such as these related lemmas—either interactively using a tool such as Isabelle [14] or manually with subsequent automatic proof-checking [15].

This makes our process vision different from that implicitly assumed for, e.g., Dafny or Jahob [16]. In order to use these other tools directly, instead of as back-end components in a tool chain, software developers are expected to understand how proofs of VCs are carried out by attached provers like Z3. Software developers using these systems need to understand (well) notions like SMT-solver triggers and/or interactive prover advice or tactics in order to annotate their software sufficiently that automated proofs can be achieved.

In our view, software developers should not be expected to provide the suggested universal algebraic lemmas, let alone their proofs, without the assistance of a mathematician. Generating and proving such lemmas requires considerable

knowledge about the capabilities of automatic provers that cannot be expected of most software developers. It is clear that it will be quite some time in the future before the typical software developer is educated to tackle such issues alone.

4 Example

To illuminate issues and results regarding what happens when trying to prove VCs—by expanding definitions, or by providing universal algebraic lemmas about definitions—we offer one of the verification benchmark problems proposed in [17]: a generic sorting program in which both the type of the entries to be sorted and the ordering are parameters. We choose to 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., [18], 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 definitions to simplify the specification. Moreover, the code for the merge sort algorithm is not trivial and involves all standard programming constructs including recursion.

```

contract QueueTemplate (type Item)
uses UnboundedIntegerFacility

math subtype QUEUE_MODEL is
  string of Item

type Queue is modeled by QUEUE_MODEL
exemplar q
  initialization ensures
    q = <>

procedure Enqueue (updates q: Queue,
                  clears x: Item)
  ensures
    q = #q * <#x>

procedure Dequeue (updates q: Queue,
                  replaces x: Item)
  requires
    q /= <>
  ensures
    #q = <x> * q

function Length (restores q: Queue)
  : Integer
  ensures
    Length = |q|

function IsEmpty (restores q: Queue)
  : control
  ensures
    IsEmpty = (q = <>)

end QueueTemplate

```

Fig. 1: Queue ADT Specification

The ordered collection to be sorted is a queue. Figure 1 shows the specification of this basic abstract data type (ADT) in RESOLVE, with no mention of sorting. The mathematical model of a `Queue` variable is a mathematical string of the mathematical model of the type parameter `Item`. Initially, a `Queue` variable is an empty string, denoted by `<>`. The operations available to a client program are the usual ones for queues. In their specifications, a few features

call for explanation. The “#” prefix on a parameter name in a post-condition denotes the parameter’s incoming (“old”) value. There are no built-in types in this dialect of RESOLVE, so type **Integer** is imported from another contract, **UnboundedIntegerFacility**. What otherwise might be a boolean-valued function returns **control**, signifying that it can be used in a condition for a loop or selection statement; even **Boolean** is not a built-in type. Parameter modes associated with the formal parameters help in shortening some post-conditions and permitting syntactic sanity checks on some pre-conditions and post-conditions, with the following meanings. In RESOLVE, the “value” of a variable in these situations is considered to be its abstract value (i.e., the value of its specified mathematical model) rather than its representation in memory.

- **updates** means the parameter’s outgoing value is potentially changed from its incoming value, and that the incoming value is potentially important to the operation’s behavior.
- **replaces** means the parameter’s outgoing value is potentially changed from its incoming value, and that the incoming value is *not* important to the operation’s behavior.
- **clears** means the parameter’s outgoing value is an initial value for its type.
- **restores** means the parameter’s outgoing value is the same as its incoming value (though, during execution of the operation body, it might temporarily be changed from that value so long as it is changed back).

```

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

Figure 2 shows an enhancement, or extension, of this contract called **SortExtension**, which specifies an operation to **Sort** a **Queue**. Note that the contract is parameterized by a binary relation **ARE_IN_ORDER** that is restricted to be a total pre-order. The ellipsis in this code is where the definitions in Figure 3 appear. These four definitions are structured in such a way that the two appearing in the post-condition 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:

```

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

- 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.²
- 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 definition’s 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.

We consider now a particular implementation of the **Sort** procedure, **MergeSort**, illustrated in Figure 4. This realization is parameterized 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.

The code realizes the usual recursive merge sort algorithm with two local helper operations, **Split** and **Merge**. Most of the code looks similar to what is ex-

² An implicit definition leads to an existence-and-uniqueness proof obligation that, like others mentioned earlier, is something that might not be expected to be discharged automatically. It would be helpful if implicit definitions were further subdivided into various kinds with special syntactic structures where, for some, existence and uniqueness could be shown once and for all by meta-level proofs. This direction goes beyond the scope of the paper.

```

realization MergeSort (
  function AreInOrder (restores i : Item,
    restores j : Item): control
  ensures
    AreInOrder = ARE.IN.ORDER (i, j)
) implements SortExtension
  for QueueTemplate

uses Concatenate for QueueTemplate

local procedure Split (updates q1 : Queue,
  replaces q2 : Queue)
ensures
  ARE.PERMUTATIONS (q1 * q2, #q1) and
  |q2| <= |q1| and |q1| <= |q2| + 1
variable tmp : Queue
Clear (q2)
tmp :=: q1
loop
  maintains
    ARE.PERMUTATIONS (tmp * q1 * q2,
      #tmp * #q1 * #q2) and
    |q2| <= |q1| and |q1| <= |q2| + 1
    and (tmp /= <> implies |q1| = |q2|)
  decreases |tmp|
  while not IsEmpty (tmp) do
    variable x : Item
    Dequeue (tmp, x)
    Enqueue (q1, x)
    if not IsEmpty (tmp) then
      Dequeue (tmp, x)
      Enqueue (q2, x)
    end if
  end loop
end Split

local procedure Merge (updates q1 : Queue,
  clears q2 : Queue)
requires
  |q1| > 0 and |q2| > 0 and
  IS.NONDECREASING (q1) and
  IS.NONDECREASING (q2)
  ensures
    ARE.PERMUTATIONS (q1, #q1 * #q2) and
    IS.NONDECREASING (q1)
  variable tmp : Queue
  variable q2Item : Item
  Dequeue (q2, q2Item)
  loop
    maintains
      ARE.PERMUTATIONS (
        tmp * q1 * q2 * <q2Item>,
        #tmp * #q1 * #q2 * <#q2Item>) and
      IS.NONDECREASING (tmp * q1) and
      IS.NONDECREASING (tmp * <q2Item> * q2)
    decreases |q1 * q2|
    while not IsEmpty (q1) do
      variable q1Item : Item
      Dequeue (q1, q1Item)
      if not AreInOrder (q1Item, q2Item) then
        q1Item :=: q2Item
      end if
      Enqueue (tmp, q1Item)
    end loop
    Enqueue (tmp, q2Item)
    Concatenate (tmp, q2)
    q1 :=: tmp
  end Merge

  procedure Sort (updates q : Queue)
  decreases |q|
  variable qLength, one : Integer
  Increment (one)
  qLength := Length (q)
  if IsGreater (qLength, one) then
    variable qSplit : Queue
    Split (q, qSplit)
    Sort (q)
    Sort (qSplit)
    Merge (q, qSplit)
  end if
end Sort
end MergeSort

```

Fig. 4: MergeSort Realization

pected in any modern object-based imperative language. However, following the specification of **Split**, the code immediately contains two statements that call for explanation. Every type comes with an operation called **Clear** that re-initializes its argument to an initial value for its type. The primary data-movement operator in RESOLVE, **:=:**, swaps (exchanges) the values of its two operands, which must be simple variables.

As expected in a language supporting automated verification, a loop has a loop invariant and a progress metric. The former is introduced by the keyword **maintains**, and the latter by the keyword **decreases**. In a loop invariant, the “old” values of the variables are those just before the start of the loop, and are denoted using the “#” prefix. (This should not be confused with the use of “#” in a post-condition, where “old” refers to a different time frame, i.e., the start of the operation body.)

5 Translation to Dafny

The OSU RESOLVE tools generate VCs that can be presented to any automated prover via translation from our own XML format into the input format of the prover. In addition, any prover that does not come with the necessary mathematical background must be provided with axioms (and generally some associated lemmas and theorems) of the RESOLVE mathematical theories that appear in the VCs.

As mentioned in Section 3, Dafny is a tool for writing programs annotated with various mathematical and specification statements. It translates these programs into the Boogie intermediate language, which generates VCs and feeds them to Z3 for proof. It would have been possible to translate RESOLVE specifications and 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. In particular, rather than providing Z3 with an axiomatic description of, say, RESOLVE’s string theory, and relying on its core logic capabilities alone, we leverage the fact 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. Operators for RESOLVE strings, i.e., string construction ($\langle \bullet \rangle$), concatenation ($*$), and length ($|\bullet|$), have counterparts for Dafny’s sequences ($[\bullet]$, $+$, $|\bullet|$).

Dafny’s features for making definitions and for writing 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 RESOLVE-generated VC is of the form $\bigwedge A_i \implies C$. This form 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 of the VC generates an `assert` statement. We have used Dafny verifier version 2 and Z3 version 2.15 for this study.

6 Implementation Details

We follow two different paths in attempts to help Z3 prove the resulting VCs. In one, we expand definitions. This entails handing the prover some moderately difficult quantified statements. From a software developer’s point of view, however, this approach seems to be preferred, as no algebraic properties need to be identified and supplied to the prover. In the other approach, we supply some rather shallow and largely unsurprising universal algebraic lemmas about the defined functions and predicates.

6.1 Technique 1: Expand Definitions

In the first approach, each definition is expressed as an `assume` statement in Dafny. This is followed by the VCs encoded in a series of `assume` and `assert`

statements as outlined in Section 5. Figures 5 and 6 show examples of two key VCs that Z3 does not prove. The VC in Figure 5 involves the defined predicate `ARE_PERMUTATIONS` while the one in Figure 6 involves `IS_NONDECREASING`. It

```

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. 5: Key VC in Dafny Syntax

```

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. 6: Key VC in Dafny Syntax

is important to realize 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 when definitions are expanded.

The VC in Figure 5 contains the expanded definitions (as shown in Figure 3) but have been omitted here. The VC also has several assumptions, all but one elided as irrelevant to the proof of the conclusion. To a human, the assertion seems easily provable from the given assumption along with an understanding of concatenation and the definition of `ARE_PERMUTATIONS`. The VC in Figure 6 also has several 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 also holds for the concatenation of two of them arranged in the same order. But here it is less clear how long a reasoning path is required for an automated prover to notice this.

As noted above, Z3 does not prove these two VCs (or several others) when supplied with the expanded definitions. See Section 7 for full results.

6.2 Technique 2: Introduce Universal Algebraic Lemmas

Instead of expanding definitions, we now provide the prover with some simple universal algebraic lemmas about the defined functions and predicates.

It is important to realize that 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, we might say that 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.

1. **forall** $a:\text{seq}\langle T \rangle :: \text{ARE_PERMUTATIONS}(a, a)$
2. **forall** $a:\text{seq}\langle T \rangle, b:\text{seq}\langle T \rangle, c:\text{seq}\langle T \rangle :: \text{ARE_PERMUTATIONS}(a, b) \ \&\& \ \text{ARE_PERMUTATIONS}(b, c) ==> \text{ARE_PERMUTATIONS}(a, c)$
3. **forall** $a:\text{seq}\langle T \rangle, b:\text{seq}\langle T \rangle :: \text{ARE_PERMUTATIONS}(a, b) ==> \text{ARE_PERMUTATIONS}(b, a)$
4. **forall** $a:\text{seq}\langle T \rangle, b:\text{seq}\langle T \rangle, c:\text{seq}\langle T \rangle :: \text{ARE_PERMUTATIONS}((a + b) + c, a + (b + c))$
5. **forall** $a:\text{seq}\langle T \rangle, b:\text{seq}\langle T \rangle :: a == b ==> \text{ARE_PERMUTATIONS}(a, b)$
6. **forall** $a:\text{seq}\langle T \rangle, b:\text{seq}\langle T \rangle :: \text{ARE_PERMUTATIONS}(a, b) ==> |a| == |b|$

Fig. 7: ARE_PERMUTATIONS Lemmas

We encode the lemmas involving `ARE_PERMUTATIONS` and `IS_NONDECREASING` as shown in Figures 7 and 8. An obvious and important question is, “Which lemmas should be provided to the prover?” The lemmas in Figure 7 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 the expression of a current limitation on how they are processed by the prover.

The lemmas added for `IS_NONDECREASING` are more extensive, and can be divided into three types. The most basic set are lemmas 1 through 3 in Figure 8. The second set (4 through 10) are of a different nature, relating various concatenations of strings satisfying `IS_NONDECREASING`. Lemmas 5, 6, and 7 are variations of each other (we show only lemma 5 as the others are similar). We need to enumerate all of these, as Z3 does not infer that $a+(b+c)$, $(a+b)+c$, and $a+b+c$ are equal when they are supplied as arguments to defined functions and predicates. Similarly, Lemmas 8, 9, and 10 state dual-like properties, of which we show only one. The third set (11 through 16) are different still, relating `ARE_IN_ORDER` and `IS_NONDECREASING`.

All of the above-mentioned lemmas were proved interactively with the help of Isabelle used as a proof assistant. This would be the job of a mathematician assisting a software developer under the process vision outlined in Section 3.

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)`
- ...
8. `forall a:seq<T>, b:seq<T>, c:seq<T> :: IS_NONDECREASING(a + c) &&
IS_NONDECREASING(c + b) && c!= [] ==> IS_NONDECREASING(a + c + b)`
- ...
11. `forall a:T, b:T :: ARE_IN_ORDER(a,b) ==> IS_NONDECREASING([a] + [b])`
12. `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)`
- ...
16. `forall a:seq<T>, x:T, y:T :: IS_NONDECREASING([x] + a + [y]) ==>
ARE_IN_ORDER(x,y)`

Fig. 8: IS_NONDECREASING Lemmas

7 Results

A total of 62 RESOLVE VCs were generated from the **MergeSort** code. Using the first technique, expanding definitions, 22 of these VCs remained unproved by Z3. Using the second technique, adding universal algebraic lemmas as described above, all of the 62 VCs were proved. No triggers were provided to help Z3 use any of these lemmas.

	VCs without a definition	VCs with a definition			
		No definition No lemmas	Definition No lemmas	No definition Lemmas	Definition Lemmas
Number of VCs proved / Total number of VCs	(3133/3211)	(392/816)	(558/816)	(761/816)	(640/816)
Percentage of VCs proved	97.6%	48.0%	68.3%	93.1%	78.3%

Table 1: Summary of Empirical Results

We applied the same techniques to over 4000 VCs generated from about 50 RESOLVE components, about 2000 lines of code, including the components involved in an earlier empirical study of different issues [5]. The code includes

arithmetic algorithms over integers, sorting of arbitrary items with arbitrary orders (as shown in the previous sections), manipulations of and representations of stacks, queues, lists, sets, etc. 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. The results are shown in Table 1. Attempts to prove the VCs that contain mathematical definitions were made in four different ways: by supplying neither the definition body nor any lemmas, only the definition body (Technique 1 described above), only the lemmas (Technique 2), and both the definition body and the lemmas.

From Table 1 it is seen that the VCs that do not involve any new definitions of the sort described in this paper, Z3 proves a very high percentage (about 98%). On the other hand, it proves only 68% of the VCs containing mathematical definitions, when the definitions are expanded. Replacing the expanded definitions by appropriate universal algebraic lemmas raises the success rate to 93%. Detailed results showing specifications and code for all components, the VCs generated, the definitions used in specifications, and the lemmas about those definitions are at <http://resolve.cse.ohio-state.edu:8080/archive/vstte12/index.html>.

8 Related Work

To our knowledge, though the general idea of providing universal algebraic lemmas whenever one has new definitions has been reported in case studies with interactive proofs [6, 7], it has not previously been systematically and empirically evaluated for use with automated provers for verification conditions. There is some complementary work. Leino and Monahan [19] discuss automated verification of programs that use definitions of comprehension expressions, such as iterated sum and product. They use Z3 and Simplify as back-end provers and focus on finding matching triggers that help these provers. All their examples involve integers and integer arrays.

9 Conclusions

We have presented empirical support for the claim that supplying universal algebraic lemmas about defined mathematical functions and predicates is an effective way of supporting automated verification of programs whose specifications use such definitions—much better, in general, than expanding the definitions. 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 have also demonstrated that the approach can be applied successfully for a wide range of code whose specifications involve various mathematical theories. Subjectively speaking, Z3 does quite an impressive job overall on the 4000 VCs we presented it in this study.

We have observed that providing certain phrasings and combinations of lemmas can cause Z3's performance to degrade. Providing both expanded definitions and lemmas about those definitions can cause similar degradation, to the point

that VCs proved with one or the other are not proved when both are present. Sometimes a specific lemma may be helpful to prove a VC while other times a more general version of the same lemma may be needed for an apparently similar VC. It would be useful to understand the detailed basis of this behavior—but this might require relatively deep understanding of Z3 that one would hope a software engineer would not need under a successful verified software paradigm.

Z3 can reason about a number of first-order theories, including integers, arrays, and uninterpreted functions. From end-to-end, this means specifications and VCs involving statements about, say, RESOLVE’s strings and integers are translated through statements about Dafny’s sequences and integers and finally into statements about Z3’s arrays and integers. We have not discussed the possibility that any of these translation steps might involve subtle errors of their own. Based on published work to date that uses various tool chains of a similar nature, this generally seems not to be considered a serious practical problem, but there is no doubt it is a latent foundational question.

Finally, 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 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. Hoare, T.: The verifying compiler: A grand challenge for computing research. *J. ACM* **50** (January 2003) 63–69
2. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In Clarke, E.M., Voronkov, A., eds.: *LPAR (Dakar)*. Volume 6355 of LNCS., Springer (2010) 348–370
3. Leino, K.R.M.: Specification and verification of object-oriented software. Marktoberdorf International Summer School 2008, lecture notes
4. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In Ramakrishnan, C.R., Rehof, J., eds.: *TACAS*. Volume 4963 of LNCS., Springer (2008) 337–340
5. Kirschenbaum, J., et al.: Verifying component-based software: Deep mathematics or simple bookkeeping? In: *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering. ICSR '09*, Berlin, Heidelberg, Springer-Verlag (2009) 31–40
6. Nelson, C.G.: Techniques for program verification. PhD thesis, Stanford, CA, USA (1980)
7. Matt Kaufmann, P.M., (eds.), J.S.M.: *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June, 2000. (ISBN 0-7923-7849-0)
8. Sitaraman, M., et al.: Building a push-button RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing* **23** (2011) 607–626
9. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. *J. ACM* **52** (May 2005) 365–473
10. Sitaraman, M., Weide, B.: Component-based software using RESOLVE. *SIGSOFT Softw. Eng. Notes* **19** (October 1994) 21–63
11. Heym, W.D.: Computer program verification: improvements for human reasoning. PhD thesis, Columbus, OH, USA (1995)
12. Sitaraman, M., et al.: Reasoning about software-component behavior. In: *Proceedings of the 6th International Conference on Software Reuse: Advances in Software Reusability*, London, UK, Springer-Verlag (2000) 266–283
13. Leino, K.R.M.: This is Boogie 2. Manuscript KRML 178, 2008. Available at <http://research.microsoft.com/leino/papers.html>
14. Nipkow, T., Wenzel, M., Paulson, L.C.: *Isabelle/HOL: A proof assistant for higher-order logic*. Springer-Verlag (2002)
15. Smith, H., Roche, K., Sitaraman, M., Krone, J., Ogden, W.F.: Integrating math units and proof checking for specification and verification. In: *Proceedings of the Seventh International Workshop on Specification and Verification of Component-Based Systems. SAVCBS 2008* (2008) 59–66
16. Zee, K., Kuncak, V., Rinard, M.: An integrated proof language for imperative programs. In: *ACM Conf. Programming Language Design and Implementation (PLDI)*. (2009)
17. Weide, B.W., et al.: Incremental benchmarks for software verification tools and techniques. In: *Verified Software: Theories, Tools, Experiments*. (2008) 84–98
18. Leino, K.R.M., Monahan, R.: Dafny meets the verification benchmarks challenge. In: *VSTTE'10*, Springer-Verlag (2010) 112–126
19. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: *Proceedings of the 2009 ACM symposium on Applied Computing. SAC '09*, ACM (2009) 615–622