

# Using Isabelle to Help Verify Code That Uses Abstract Data Types

Jason Kirschenbaum  
The Ohio State University  
Columbus, OH 43210, USA  
kirschen@cse.ohio-state.edu

Bruce M. Adcock  
The Ohio State University  
Columbus, OH 43210, USA  
adcockb@cse.ohio-state.edu

Derek Bronish  
The Ohio State University  
Columbus, OH 43210, USA  
bronish@cse.ohio-state.edu

Paolo Bucci  
The Ohio State University  
Columbus, OH 43210, USA  
bucci@cse.ohio-state.edu

Bruce W. Weide  
The Ohio State University  
Columbus, OH 43210, USA  
weide@cse.ohio-state.edu

## ABSTRACT

Verification of programs that use abstract data types (ADTs) is an important piece of the grand challenge of verified software. It is our position that an interactive proof assistant, such as Isabelle, used in a fully automated mode, can be an effective, extensible proof engine for use in the modular verification of software. As technical justification for this position, we describe the modular verification of two implementations of an extension to a queue ADT. One implementation is recursive, while the other is iterative and relies on a stack ADT. The correctness of the implementations is proved by Isabelle automatically, using specification theories from the Resolve mathematical library imported into Isabelle. Isabelle’s viability as a general-purpose VC prover is also discussed.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Formal Methods*; D.2.4 [Software Engineering]: Software/Program Verification—*Correctness Proofs*

## General Terms

Languages, tools

## Keywords

Verification, Isabelle, formal methods, reuse

## 1. INTRODUCTION

The grand challenge for computer scientists to produce a verifying compiler, recently reissued by Hoare [18], has been

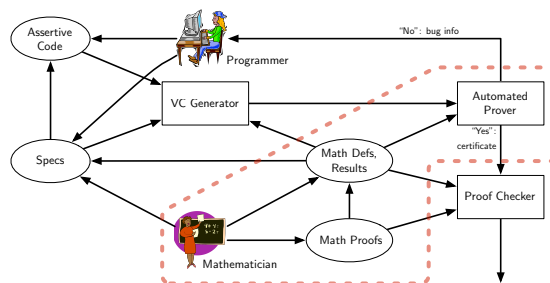


Figure 1: Envisioned framework for verified software

a goal of the community since as early as 1967 [14]. A verifying compiler parses and compiles code, generates verification conditions (VCs) whose validity implies the correctness of the code, and proves or refutes those VCs via automated reasoning methods. Thus any object code generated by the compiler is certified correct relative to a specification of the intended behavior the code implements. Figure 1 diagrams how such a process might work. Programmers write assertive code in an attempt to meet the given specifications. A VC generator then processes the code and the specifications, supplemented with relevant mathematical theories and definitions. An automated theorem prover is fed the resulting VCs, along with the theories, definitions and previously established mathematical results. The theorem prover may rely on specialized decision procedures and/or general-purpose reasoning methods. In the case that the VC is proven, a proof checker can be invoked to confirm the proof to rule out possible errors in the prover, which is both complex and generally treated as a black box.

In this paper, we are concerned with the *automatic* proof of VCs, focusing on the region delineated by the dotted line in Fig. 1. More specifically, this paper focuses on an *extension* of an ADT, and does not investigate the verification of ADT implementations themselves. The extension contains the mathematical specification of a new operation for the ADT, along with one of many possible implementations. In accordance with the usual object-oriented design princi-

ples, the implementation does not use knowledge about the internal data representation of the ADT. Therefore, the operation can be verified by examining only the specifications of any operations called in the code, and the specification that the code purports to implement. Any verifying compiler should only have to perform the proof of the correctness of this code once. This is the basic tenet of modular verification [24].

It is our position that the proof assistant Isabelle [21] can function as an effective, extensible tool to perform the automated verification of software, in a manner similar to SMT solvers [11, 10]. Moreover, we can factor out the mathematical theories needed from the proof engine used for the VC proofs.

In order to justify our position, we present two examples of automated verification for extensions of abstract data types. These examples are from a recently proposed set of benchmarks for automated verification [29]. The mathematical theory involved is that of strings. Next, the advantages of our approach are discussed. Specifically, we illustrate how the disciplined use of a small set of rich mathematical models used in model-based specifications permits the expression of required loop invariants, even where two different ADTs are involved (in this case, stacks and queues). Finally, we turn our attention to the issues that naturally arise as we continue to expand the power of the mathematical theories to specify additional ADTs and extensions using the same approach.

We use the interactive proof assistant Isabelle [21] to prove the VCs. The choice to use an interactive proof assistant rather than a specialized decision procedure for the relevant mathematical theories was motivated by concerns about how the approach scales upwards. The primary drawback to customized decision procedures is that a new one needs to be formulated and proven correct for every new mathematical entity that the code can realize. Of course, interactive proof assistants have their downsides as well, for example the question of whether their failure to establish a result is due to an actual deficiency in the code to be verified, or a limitation in the prover itself. While, in the long term, a hybrid approach may be used in practice, we are interested in the research questions involved in comparing these two approaches.

Our choice of Isabelle, rather than another interactive proof assistant such as PVS [23] or COQ [1], is motivated by several factors. First, one of the members of our team is an expert in Isabelle, which eases the learning curve to use Isabelle (similarly any interactive proof assistant). The second is the popularity of Isabelle/HOL in the area of software verification [2, 28, 5, 7]. Finally, Isabelle/HOL’s ability to add new lemmas to the automated proof tactics was also a consideration.

The language of the specifications and implementations is a dialect of Resolve [12, 22, 4]. This language provides support for the separation of component specification and implementations, with constructs admitted to the language only when they allow proof rules that support modular verification. The language provides a *value* semantics mental model of ADTs. There are no references; aliasing cannot occur [27].

The specifications are model-based, and are manifested as **requires** and **ensures** clauses on each of the operations, thus creating “operation contracts.” Built-in to Resolve are several fundamental mathematical theories that are powerful enough to specify most ADTs. When necessary, additional theories can be defined [12]. The number of theories used is, however, intended and expected to remain small, and Resolve includes the core theories of strings, integers, real numbers, sets, multisets, trees, binary trees, tuples, and booleans.

The specifications shown here are similar to those found in other specification languages. The queue ADT example presented here could be specified in a similar manner in JML [20]. However, the presence of aliasing and other features in Java significantly complicates the specifications; for example, the specification of a **Stack** in JML [20] is far more complex than the Resolve specification of **Stack** (presented in the next section).

Section 2 describes the contracts and implementations of the examples studied, along with the relevant mathematical theory. Section 3 discusses the most interesting VCs necessary for the correctness of the implementations. Section 4 summarizes our experiences with proving correctness of VCs involving string models. Section 5 discusses related work. Finally, Section 6 concludes with possible future directions.

## 2. SPECIFICATION AND IMPLEMENTATION OF THE QUEUE REVERSE PROCEDURE

### 2.1 Resolve String Theory

The contract specifications of the ADT for queue and its reverse extension are in terms of mathematical strings (Fig. 2). These mathematical strings are used as a mathematical model of the behavior of the queue ADT similar to what can be done using sequences in JML [6]. We emphasize that specifications and code are two different entities; specifications are statements in mathematics that may have multiple valid realizations in code, and are not themselves executable.<sup>1</sup>

Informally, strings over a given type *obj* are intended to have a model that is exactly the elements of *obj*<sup>\*</sup>, where <sup>\*</sup> is the Kleene star. However, a theory such as the one in Fig. 2 is self-contained and might have other models; that is, a theory is defined not by exhibiting a particular model, but rather by its axioms. Of course, it is important to know that such a model exists and that we are not describing an inconsistent theory. We briefly describe the process of ensuring that this theory is consistent in Section 2.4.

A few functions are defined for strings: concatenation, length, and reverse. In Fig. 2, we state some simple lemmas without proofs. The proofs of these lemmas follow easily from the axioms and definitions. Once proved, these lemmas can be used in proofs of VCs, exactly like the axioms and definitions.

<sup>1</sup>We write Resolve mathematical theories in a mathematical notation, as shown here. Isabelle proof scripts are shown in the ASCII equivalent. This choice explicitly delineates the proofs from the underlying mathematical theories.

## String Type Signature

$$\begin{aligned} \text{string} &\stackrel{\text{def}}{=} \text{string}(\text{obj}) \\ \Lambda &: \text{string} \\ \text{ext} &: \text{string} \times \text{obj} \longrightarrow \text{string} \end{aligned}$$

## String Axioms

1.  $\text{ext}(s, x) \neq \Lambda$
2.  $\text{ext}(s_1, x_1) = \text{ext}(s_2, x_2) \Rightarrow s_1 = s_2 \wedge x_1 = x_2$
3.  $\forall S \in \mathcal{P}(\text{string}) : (\Lambda \in S \wedge \forall x, s : (s \in S \Rightarrow \text{ext}(s, x) \in S)) \Rightarrow S = \text{string}$

## Function Definitions

1.  $\langle \_ \rangle : \text{obj} \longrightarrow \text{string} \stackrel{\text{def}}{=} \langle x \rangle = \text{ext}(\Lambda, x)$
2.  $|\_| : \text{string} \longrightarrow \mathbb{N} \stackrel{\text{def}}{=} |\Lambda| = 0 \wedge |\text{ext}(s, x)| = |s| + 1$
3.  $* : \text{string} \times \text{string} \longrightarrow \text{string} \stackrel{\text{def}}{=} (s * \Lambda = s) \wedge (s_1 * \text{ext}(s_2, x) = \text{ext}(s_1 * s_2, x))$
4.  $\text{reverse} : \text{string} \longrightarrow \text{string} \stackrel{\text{def}}{=} \text{reverse}(\Lambda) = \Lambda \wedge \text{reverse}(\text{ext}(s, x)) = \langle x \rangle * \text{reverse}(s)$

## Useful Lemmas

1. lemma EmptyNotSingle:  $\Lambda \neq \langle x \rangle$
2. lemma IdofEmpty:  $\Lambda * \alpha = \alpha$
3. lemma LenofSingle:  $|\langle x \rangle| = 1$
4. lemma LenofCat:  $|\alpha * \beta| = |\alpha| + |\beta|$
5. lemma AssocCat:  $\alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma$
6. lemma ReverseofReverse:  $\text{reverse}(\text{reverse}(\alpha)) = \alpha$
7. lemma ReverseofCat:  $\text{reverse}(\alpha * \beta) = \text{reverse}(\beta) * \text{reverse}(\alpha)$
8. lemma LenofReverse:  $|\text{reverse}(\alpha)| = |\alpha|$

Figure 2: String Theory

## 2.2 Specifications

The queue reverse extension uses a queue ADT, which is specified in the `QueueTemplate` component. Figure 3 shows the contract with preconditions and postconditions on each operation written via `requires` and `ensures`, respectively. The mathematical model of a queue is a string of items, and its initial value is  $\Lambda$  which is represented in ASCII as `empty_string`. This component is parametrized by the type of items in the queues. The usual queue operations are all specified in terms of this string model. The parameter modes used include `updates`, `clears`, `replaces`, and `restores`. The `updates` mode indicates that the parameter may be modified by the procedure in accordance with the `ensures` clause. The `clears` mode means the parameter has an initial value for its type upon return. The `replaces` mode indicates that the corresponding argument may be modified but that the incoming value of the parameter has no effect on the behavior of the operation. Finally, the `restores` mode means that the incoming and outgoing values of the parameter are equal. Note that the semantics of the `clears` and `restores` parameter modes each induces a proof obligation that must be discharged in order for an implementation to be verified. In `ensures` clauses, the `#` indicates the old value of a variable. In the `Queue` type declaration, the scope of `exemplar q` is just the `initialization ensures` clause; it introduces a name for an arbitrary object of the new type.

```
contract QueueTemplate (type Item)

  math subtype QUEUE_MODEL is string of Item

  type Queue is modeled by QUEUE_MODEL
  exemplar q
  initialization ensures
    q = empty_string

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

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

  function IsEmpty (restores q: Queue): control
    ensures
      IsEmpty = (q = empty_string)

end QueueTemplate
```

Figure 3: Queue ADT Specification

Finally, the parameter passing for each of the operations is performed via swapping [16]. In the absence of repeated arguments which are ruled out by the syntax of `Resolve` in this dialect, this method of parameter passing is equivalent in behavior to pass by reference.

```
contract QueueReverse enhances QueueTemplate

  procedure Reverse (updates q: Queue)
    ensures
      q = reverse (#q)

end QueueReverse
```

Figure 4: Queue Reverse Specification

## 2.3 Recursive and Iterative Realizations

The functionality described in `QueueTemplate` is extended by `QueueReverse`, which specifies a new procedure operation: `Reverse`. This is shown in Fig. 4. The specification uses the *mathematical* function `reverse`, from the string theory of Fig. 2.

An implementation of the `Reverse` procedure can be done in at least two different ways: recursively, or iteratively using a stack. The stack ADT is specified in a way similar to queue, again using strings as the mathematical model. The `Push` and `Pop` operations both work on the “left” end of the string that models a stack. The stack provides LIFO behavior, while the queue provides FIFO behavior. The contract of the stack ADT is needed for the proof of correctness, of course, and is shown in Fig. 5.

The iterative `Reverse` implementation performs the reversal in two steps, as shown in Fig. 6. First, all of the elements of the queue are moved from the queue to a local stack. Then all of the elements of the stack are popped off and placed in the queue. In this implementation, we must verify two loops. A loop includes both a loop invariant (the `maintains` clause) and a loop progress metric (the `decreases` clause, which is used to prove termination). In a loop invariant, `#` denotes

```

contract StackTemplate (type Item)

  math subtype STACK_MODEL is string of Item

  type Stack is modeled by STACK_MODEL
  exemplar s
  initialization ensures
    s = empty_string

  procedure Push (updates s: Stack, clears x: Item)
  ensures
    s = <#x> * #s

  procedure Pop (updates s: Stack, replaces x: Item)
  requires
    s /= empty_string
  ensures
    #s = <x> * s

  function IsEmpty (restores s: Stack): control
  ensures
    IsEmpty = (s = empty_string)
end StackTemplate

```

Figure 5: Stack ADT Specification

the value of the variable just before execution encounters the loop.

```

realization Iterative implements QueueReverse

  facility StackFacility is StackTemplate (Item)

  procedure Reverse (updates q: Queue)
  variable s: Stack
  loop
    maintains reverse(#s) * #q = reverse(s) * q
    decreases |q|
  while not IsEmpty (q) do
    variable x: Item
    Dequeue (q, x)
    Push (s, x)
  end loop
  loop
    maintains #q * #s = q * s
    decreases |s|
  while not IsEmpty (s) do
    variable x: Item
    Pop (s, x)
    Enqueue (q, x)
  end loop
  end Reverse
end Iterative

```

Figure 6: Iterative Queue Reverse Implementation

The recursive implementation of the `Reverse` procedure is demonstrated in Fig. 7. This implementation removes the first element from the queue, recursively reverses the rest of it, then enqueues the removed element. Since this procedure is recursive, we must provide a metric that decreases in each recursive call to `Reverse` in order to prove total correctness.

These implementations cover several of the standard features of any imperative programming language: (recursive and non-recursive) calls, loops and conditional control structures, and the use of ADTs. The examples show how our tools and techniques for proving VCs process these common language features.

The lemmas needed to prove the correctness of the two implementations of `Reverse` are few, as seen in Fig. 2. We

```

realization Recursive implements QueueReverse

  procedure Reverse (updates q: Queue)
  decreases |q|
  if not IsEmpty (q) then
    variable x: Item
    Dequeue (q, x)
    Reverse (q)
    Enqueue (q, x)
  end if
  end Reverse

end Recursive

```

Figure 7: Recursive Queue Reverse Implementation

do not need to include two mathematical theories, one for stacks and one for queues; string theory is rich enough for both. This mathematical uniformity is especially advantageous for the iterative version. It would be tricky to write both loop invariants without it.

## 2.4 String Theory in Isabelle

Isabelle [21] is an automated proof assistant, meaning it is capable of checking a proof that a user directs. Automated proof assistants can also perform many of the tedious steps needed to produce a proof; in some cases the proof assistant can produce the proof outright, establishing the goal without human guidance.

More specifically, Isabelle has a simplifier that can be invoked to simplify assumptions and goals of a theorem. Isabelle also includes a classical reasoner that can perform many of the logical inference rules automatically. The proof structure of Isabelle is set up in a manner that mimics natural deduction. Assumptions and a goal are presented and simplifications can apply to both. Rules for applying already-proved lemmas and theorems dictate how the assumptions and goals are modified. One can apply forward reasoning and modify the assumptions, apply backward reasoning and modify the goals, or apply both at the same time.

For convenience, many of the proof methods instantiated with common lemmas are performed in Isabelle via the `auto` and `force` commands, commonly referred to as “tactics.” These tactics use both the simplifier and the classical reasoner, the difference being that the `force` method will only succeed or fail, while the `auto` method will return a simplified goal (if possible).

New axiom systems, theories, and theorems can be entered into Isabelle using a built-in meta-level logic. All other axiom systems are implemented on top of this meta-logic. For example, higher order logic (HOL) [21] and ZF set theory [25] are available.

Since string theory is already developed in Resolve, we need only to import that theory into Isabelle. The proofs of the lemmas in string theory can then be factored off from the usage of those lemmas for proving the various VCs. Of course, a theory must also have a witness to the existence of a model that satisfies its axioms. Fortunately, such a model for string theory is readily available, namely the Isabelle `List` type in the HOL theory. More information about the process

used to import the Resolve String theory into Isabelle can be found on the web at <http://www.cse.ohio-state.edu/~kirschen/rsrg/Isabelle.html>

We do not use the plethora of theories available in Isabelle for the automatic proof of VCs, but rather use Isabelle’s proof engine along with *only* the theories already developed for Resolve specifications. By doing so, we are not tied down to any particular proof assistant or theorem proving tool. As long as the tool has an expressive enough proof language, and allows users to add new simplification and proof rules, then it might be used for our purposes.

### 3. VERIFICATION AND RESULTS

```
theory RecursiveQueueReverse_Reverse

imports Main String

begin
...
lemma 4:
"[|
  is_initial((x_2::'obj)) ;
  is_initial((x_5::'obj)) ;
  ~<(x_3::'obj)> o (q_3::'obj string) = empty_string
|]
==>
  reverse(q_3) o <x_3> = reverse((<x_3> o q_3))"

apply ((simp only: simp_thms),clarify?)+?

apply (force+)?

done
...
end
```

Figure 8: A key VC for the verification of the recursive implementation of Reverse

The method of generating VCs [17, 27] is known to be both sound and relatively complete (i.e., relative to the completeness of the mathematics used in the specifications). This method of generating VCs is quite similar to a method described by Barnett *et al.* [3]. At a high level, the generation of VCs involves processing the realization’s code (accumulating facts from the **ensures** clauses of each procedure used) while taking control structure (conditionals and loops) into account. A new VC is generated for each **requires** clause of a called procedure or function that is not syntactically “true,” at each loop invariant and recursive call, and at the end of the operation body.

Lemma #4 (state index: 6, ensures clause)

$$\begin{aligned} & \text{is\_initial}(x_2) \\ \wedge & \text{is\_initial}(x_5) \\ \wedge & \langle x_3 \rangle * q_3 \neq \Lambda \\ \Rightarrow & \text{reverse}(q_3) * \langle x_3 \rangle = \text{reverse}(\langle x_3 \rangle * q_3) \end{aligned}$$

Figure 9: Human readable version of Fig. 8

The VCs are intended not only to be mathematically precise, but also human-readable. At a high level, between each pair of statements in the implementation’s code a new subscript is created for each variable, and a mathematical formula relates the new subscripted variables to the earlier

subscripted variables. Path conditions, facts, and obligations are accumulated and organized into VCs according to the proof rules in [17]. For example, if a variable  $v$  is not changed by a statement  $s$  and the subscript before  $s$  is  $i$ , then the facts known after the statement include  $v_{i+1} = v_i$ . Control statements and loops introduce implications. For each of the possible control paths (e.g., entering or skipping a loop body), we generate a separate VC. This simplifies the task for the prover, as it explicitly does the requisite case analysis. We have found empirically that this format reduces the complexity of the VCs and their proofs, and increases the chance that Isabelle can prove the VCs automatically.

```
..
lemma 4:
"[|
  ~<(x_4::'obj)> o (q_4::'obj string) = empty_string ;
  reverse(empty_string) o (q_0::'obj string)
    = reverse((s_2::'obj string)) o <x_4> o q_4 ;
  (length ((<x_4> o q_4))) > 0 ;
  is_initial((x_3::'obj)) ;
  is_initial((x_5::'obj))
|]
==>
  reverse(empty_string) o q_0
    = reverse((<x_4> o s_2)) o q_4"

apply ((simp only: simp_thms),clarify?)+?

apply (force+)?

done
...
end
```

Figure 10: A key VC for the verification of the iterative implementation of Reverse

One VC that needs to be proved for the recursive implementation of **Reverse** is shown in the raw Isabelle output in Fig. 9<sup>2</sup> and in a more human readable format in Fig. 8. This VC comes from the **ensures** clause of the recursive call to **Reverse** in Fig. 7, the satisfaction of which is the essence of the correctness of the implementation. The VC is part of an Isabelle theory file that includes all of the VCs. The **apply(...)** lines are instructions to Isabelle on how to prove each VC. The first section directs Isabelle to perform basic simplifications, such as propositional simplifications. The second line, **apply (force+)?** directs Isabelle to perform the automated reasoning methods. The **done** line indicates to Isabelle that the person thinks the proof is finished. These lines are all generated *automatically* by the VC generator.

Lemma #4 (state index: 5, loop invariant)

$$\begin{aligned} & \langle x_4 \rangle * q_4 \neq \Lambda \\ \wedge & \text{reverse}(\Lambda) * q_0 = \text{reverse}(s_2) * \langle x_4 \rangle * q_4 \\ \wedge & |\langle x_4 \rangle * q_4| > 0 \\ \wedge & \text{is\_initial}(x_3) \\ \wedge & \text{is\_initial}(x_5) \\ \Rightarrow & \text{reverse}(\Lambda) * q_0 = \text{reverse}(\langle x_4 \rangle * s_2) * q_4 \end{aligned}$$

Figure 11: Human readable version of Fig. 10

<sup>2</sup>The Isabelle versions of the VCs use the  $\circ$  symbol for the  $*$  concatenation symbol in Resolve’s string theory.

	Generation	Proofs
Recursive Time (sec)	0.9	.26
Iterative Time (sec)	1.8	1.37

Table 1: VC Generation and Proof Running Time

In the iterative implementation of `Reverse`, we use a stack to reverse the queue using two loops. Figure 10 is a VC from the loop invariant for the first loop expressed in the Isabelle format; figure 11 expresses the VC in a human readable format. The main string theory lemmas involved here are the associativity of concatenation and the property of concatenation within the `reverse` function.

Isabelle, with the help of the introduced string theory lemmas from Fig. 2, proves both sets of verification conditions automatically. The generation of the VCs and the proofs of the VCs in Isabelle each takes very little time, as seen in Table 1. These timings do not take into account the time for Isabelle to read the Resolve String theory file; the use of Isabelle’s internal tools allow for String theory to be incorporated into an Isabelle executable and bypass the time required for Isabelle to process String theory.

Both implementations of the queue `Reverse` code, string theory in Isabelle, and all VCs are on the web at <http://www.cse.ohio-state.edu/~kirschen/rsrg/Isabelle.html>.

#### 4. LESSONS LEARNED

We have shown that the VCs generated for the reverse extension to a queue ADT are automatically provable by an interactive proof assistant without human advice. While these initial results are positive, there are of course many more issues to address. We now describe several of the issues that have come up as we have explored these and other examples. In this section, we use the term “prover” to mean any tool that attempts to prove VCs without the use of specialized decision procedures.

The first potential complication is the addition of quantifiers in `requires` and `ensures` clauses. For a proof of a universally quantified statement in the conclusion of a VC (e.g.,  $A \Rightarrow \forall x.P(x)$ ), a prover can simply use a fixed (but arbitrary) element of the universe of the quantification for  $x$ , and prove the statement true for that element.

However, when an assumption in a VC involves universal quantification, the natural question is “what term should be used to instantiate the quantified variables?” With this issue, a general proof approach (without the use of a specialized decision procedure) must either never need to instantiate a quantified variable (avoiding the issue), or use a method that instantiates the quantifier “correctly” in many cases (possibly tuned for the types of VCs that are likely to occur). The dual of this is the use of existential quantification. An existential quantification in the assumptions does not cause any problems, whereas an existential quantification in the goal does. Consequently, it is not desirable to have existential quantification in the goal of a VC, although it remains to be seen how often this may arise. One specification design approach is to demand that all `requires` and `ensures` clauses and loop invariants be quantifier-free; quantifiers would be introduced only in mathematical definitions.

This raises the issue of how to include new definitions in existing theories. One approach is to prove algebraic properties (lemmas) involving those definitions. Using those properties, the automated prover would then attempt to verify the VCs. Another approach is to instead unfold the definition immediately and then let the prover verify the VCs using the expanded version of the definition (exposing any quantifiers in the definition to the prover). While the second approach seems easier to achieve at first glance because the requisite algebraic properties need not be identified and proved, the first approach may have benefits by limiting the complexity of the VCs that the prover works with.

For example, one might want to add the definition `IsPermutation(a,b)` to denote that the string `a` is a permutation of the string `b`. `IsPermutation` may be defined via the number of occurrences of an item in a string. `IsPermutation(a * b, b * a)` is a lemma that should—and can—be proved once and then used to prove many VCs. For example, VCs generated for a selection sort algorithm essentially require the prover to deduce `IsPermutation(q1 * (a * <x>),q)` from the assumptions `IsPermutation(q2 * a,q)` and `IsPermutation(q1 * <x>,q2)`. A standard proof involves using lemmas about substitutions, the symmetry of `IsPermutation`, and commutativity of concatenation within the arguments of `IsPermutation`.

#### 5. RELATED WORK

Zee *et al.* [30] have used a hybrid approach of applying both specialized decision procedures and a general proof assistant to prove that code purporting to implement certain data structure specifications is correct. However, the use of Java as a starting language requires that the list specifications use *reference* equality or comparison. Our approach proves properties that depend on the *values* of the objects instead.

Zhang *et al.* [31] describe a decision procedure for queues. Instead of using a special-purpose decision procedure, we use a general-purpose automated proof assistant. The general string theory used for our specifications is slightly simpler than the queue theory developed with the decision procedure, and it is also used to specify the stack ADT used in one of our examples, as well as other Resolve components.

The Why methodology [13] involves a simplified programming language, annotated with logical definitions, axioms, preconditions, post conditions and loop invariants, for which VCs can be generated. A subset of both C (with annotations) and Java (with JML specifications) can be translated into the simplified programming language, such that the VCs generated are claimed to represent the correctness of the original C or Java code. The translation process from C or Java must explicitly capture the memory model of the original source language (C or Java); as a result of using Resolve, we do not need an explicit memory model, simplifying the generated VCs.

SMT solvers such as Yices [11] and Z3 [10] are designed to search for a possible satisfying assignment to a first-order formula by using a SAT solving algorithm (such as DPLL [9, 8]) to find possible satisfying assignments, confirming those assignments via first-order theory-specific satisfiability procedures. SMT solvers are known not to perform well with

quantifiers and the reliance on strictly first-order logic ensures that some predicates may not be definable [19]. We have not yet investigated the relative efficacy of Isabelle compared to any of these SMT solvers.

The Resolve approach for the specifications of a queue ADT is similar to what might be done in JML [20]. For example, the JML specifications for lists use mathematical sequences, similar to our use of strings as the mathematical model for queues and stacks. But with Java, again, the reference/value distinction introduces considerable added complexity.

Resolve superficially resembles the Larch [15] specification and verification discipline. Both approaches employ a programming-by-contract paradigm. Resolve and Larch differ in that Resolve was designed to be used with exactly one programming language and one mathematical specification language, while Larch uses the Larch Shared Language to express mathematical theories (traits) and the Larch Interface Languages (LIL) to express specifications for a particular programming language such as C. Resolve does not have this distinction; the same mathematical language is used both for the creation of mathematical theories and the specification of programmatic operations. Resolve was also designed with the idea of verifiability in mind, so the programming language with its specification language must have a common semantics that allow for proof of soundness and relative completeness of a proof system. Larch was meant to work with several languages whose semantics may be completely different and, indeed, not yet formalized. Also, a key idea of Resolve is that it reuses mathematical theory units as much as possible, while the Larch approach instead tends to reuse a particular trait defined in the Larch Shared Language in each of the interface languages. However, the examples provided for Larch do not show the traits themselves reused within one LIL. For example, a queue and a stack each have a different trait in Larch, while both are modeled by mathematical strings from String theory in Resolve.

## 6. CONCLUSION AND FUTURE WORK

We have described automated verification of two different implementations of an extension of an ADT. We have also discussed lessons learned about possible challenges to achieving verified software by using an automated proof assistant to prove VCs.

In the future, we expect to create implementations that exercise different parts of the string theory development. For example, the use of substring or permutation definitions in the specifications would require the addition of more string lemmas to continue the automated reasoning process. Finally, we expect to add other theories developed for Resolve specifications into Isabelle, such as finite sets and trees.

## Acknowledgments

The authors thank Jeremy Avigad, Harvey M. Friedman, Wayne Heym, Brandon Minter, Bill Ogden, Murali Sitaraman, and Anna Wolf for their assistance. This work was supported in part by the National Science Foundation under grant DMS-0701260. 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.

## 7. REFERENCES

- [1] The Coq Proof Assistant Reference Manual Version v8.1.
- [2] E. Alkassar and M. A. Hillebrand. Formal functional verification of device drivers. In Shankar and Woodcock [26], pages 225–239.
- [3] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 82–87, New York, NY, USA, 2005. ACM.
- [4] P. Bucci, J. E. Hollingsworth, J. Krone, and B. W. Weide. Part III: implementing components in RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19(4):40–51, 1994.
- [5] P. Chalin, P. R. James, and G. Karabotsos. JML4: Towards an industrial grade IVE for java and next generation research platform for JML. In Shankar and Woodcock [26], pages 70–83.
- [6] Y. Cheon, G. Leavens, M. Sitaraman, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, 35(6):583–599, 2005.
- [7] M. Daum, J. Dörrenbächer, M. Schmidt, and B. Wolff. A verification approach for system-level concurrent programs. In Shankar and Woodcock [26], pages 161–176.
- [8] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [9] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
- [10] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
- [11] B. Dutertre and L. de Moura. The Yices SMT solver, 2006. <http://yices.csl.sri.com/tool-paper.pdf>.
- [12] S. H. Edwards, W. D. Heym, T. J. Long, M. Sitaraman, and B. W. Weide. Part II: specifying components in RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19(4):29–39, 1994.
- [13] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *19th International Conference on Computer Aided Verification*, volume 4590/2007 of *LNCS*, pages 173–177, Berlin, Germany, July 2007. Springer-Verlag.
- [14] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.
- [15] J. V. Guttag, J. J. Horning, S. J. Garl, K. D. Jones, A. Modet, and J. M. Wing. Larch: Languages and tools for formal specification. In *Texts and Monographs in Computer Science*. Springer-Verlag, 1993.
- [16] D. Harms and B. Weide. Copying and Swapping: Influences on the Design of Reusable Software Components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.

- [17] W. D. Heym. *Computer Program Verification: Improvements for Human Reasoning*. PhD thesis, Department of Computer and Information Science, The Ohio State University, Columbus, OH, December 1995.
- [18] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
- [19] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–182, New York, NY, USA, 2008. ACM.
- [20] G. Leavens. JML language. <http://www.eecs.ucf.edu/~leavens/JML-release/javadocs>.
- [21] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [22] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben. Part I: the RESOLVE framework and discipline: a research synopsis. *SIGSOFT Softw. Eng. Notes*, 19(4):23–28, 1994.
- [23] S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. PVS: an experience report. In D. Hutter, W. Stephan, P. Traverso, and M. Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, oct 1998. Springer-Verlag.
- [24] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
- [25] L. Paulson. Set theory for verification: from foundations to functions. *Journal of Automatic Reasoning*, 11(2):353–389, 1993.
- [26] N. Shankar and J. Woodcock, editors. *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, volume 5295 of *Lecture Notes in Computer Science*. Springer, 2008.
- [27] M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. D. Heym, S. M. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *ICSR-6: Proceedings of the 6th International Conference on Software Reuse*, pages 266–283, London, UK, 2000. Springer-Verlag.
- [28] A. Starostin and A. Tsyban. Verified process-context switch for c-programmed kernels. In Shankar and Woodcock [26], pages 240–254.
- [29] B. W. Weide, M. Sitaraman, H. K. Harton, B. Adcock, P. Bucci, D. Bronish, W. D. Heym, J. Kirschenbaum, and D. Frazier. Incremental Benchmarks for Software Verification Tools and Techniques. In *Proceedings of VSTTE 2008 (Verified Software: Theories, Tools, and Experiments)*. Springer-Verlag, 2008.
- [30] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. *SIGPLAN Not.*, 43(6):349–361, 2008.
- [31] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. *Inf. Comput.*, 204(10):1526–1574, October 2006.