Impact of Specification Abstractions on Client Verification

Hampton Smith and Murali Sitaraman

Technical Report RSRG-10-02
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

September 2010

Copyright © 2010 by the authors. All rights reserved.

Impact of Specification Abstractions on Client Verification

Hampton Smith
School of Computing
Clemson University
100 McAdams Hall
Clemson, SC
hamptos@clemson.edu

Murali Sitaraman
School of Computing
Clemson University
100 McAdams Hall
Clemson, SC
msitara@clemson.edu

ABSTRACT

Push-button automation is an important milestone for verification systems and a likely requirement for mainstream acceptance of the notion of "verified software". Multiple, logically-equivalent specifications may differ widely from the standpoint of their ability to contribute to verifiable client code. Using the types of problems considered at the VSTTE 2010 competition as motivation, we explore the question of specifying the same programming concept (lists) using completely different mathematical models and in each case examine the provability of client code based on that concept. The ultimate goal is to develop a set of specification patterns that aid software developers in attaining the goal of verified software. Initial results from an experimental exploration are presented along with some hypotheses for best-practices for specification design.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical Verification*

General Terms

Experimentation, Theory, Verification

Keywords

mathematical modeling, mechanical verification, proofs

1. INTRODUCTION

A central problem in the ongoing quest for verified software [10] is the difficulty verifying correctness of software components automatically. There are indeed several success stories of formal verification of non-trivial software (e.g., OS kernel verification [12, 2],) in which proofs are developed interactively with years of manual effort. Other systems that allow for mechanical verification have nontrivial annotation overhead per line of code and manual guidance of the back-end prover to a verification proof (see, for example, [1], [25], [16]). While some assertions in the form of, for example, loop invariants, are mostly unavoidable, until programmers are able to write code without large overtures to the back-end prover, fully verified software development processes are unlikely to gain widespread acceptance.

It is well known that modularity is a powerful tool for reducing the complexity of verification and corresponding proofs. In a modular verification system, only the specifications of subcomponents are used in verification of client systems. Given the prominent role of specification in such a modular verification system, the way in which specifications are expressed have a major impact on the provability of code. Since modularity is maintained, verification effort put into a component need not be repeated: a verified component is verified for all contexts. However, this is only useful insofar as the original component specification has broad applicability in many client contexts. That is: effort spent verifying a component is wasted if the specification of the component is not generically useful as part of a verifiable client. This paper focuses on ways in which specifications contribute (or fail to contribute) to the verifiability of clients.

In traditional software development, multiple, functionally-equivalent programs can have radically different non-functional attributes. With careful design, architects can ensure that a component or system has a desired subset of these attributes. Similarly, multiple, logically-equivalent *specifications* can have radically different non-logical attributes—among which is the property of ease of verification. We seek to demonstrate that a powerful, and oft overlooked, tool in the software engineer's toolbox is how to design and express specifications from the start with this property in mind.

As we have experimented with modular component specifications, we have noticed patterns emerge around idioms that impact the complexity of automated verification. This has led us to hypothesize about a number of possible best practices for specification writing. An active part of our research has become moving these hypotheses from the realm of vague intuition into precise, falsifiable form. Our hope is to begin to catalog such practices as *specification patterns* for verifiable software.

It is important to emphasize that the discussions and analysis in this paper are independent of both programming languages and specification languages, so the results have broad applicability. As its concrete example, this paper explores the impact of alternative conceptualizations of a list abstrac-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAVCBS 2010 Sante Fe, New Mexico USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

tion on provability as expressed and proved using RESOLVE [19], but these conceptualizations can be expressed in any specification notation and the implementations can be written in any imperative language.

We use variants of two of the list-based problems proposed at the VSTTE 2010 competition [22] to illustrate the role of the choice of abstractions in specification. Section 2 of the paper discusses background information and contains a summary of the VSTTE 2010 competition problems relevant for this paper. Section 3 presents alternative mathematical conceptualizations of lists and examines their impact on provability. Section 4 differentiates the effect of a component's specification on client code and the effect of a component's specification on its ability to be implemented. Section 5 considers related work along with some of the solutions to the problems presented at the competition. Section 6 contains our conclusions and directions for continued research.

2. BACKGROUND

2.1 Technical Background

Formal verification systems follow a well established pipeline. The primary inputs of the system are code in a programming language and relevant specifications in a formal specification language[1]. Many specification languages exist. Some, like Z [24], are general purpose and applied to many programming languages. Some, like JML [15] are built on top of existing, industrial languages (in this case, Java). Others like Dafny [16], Spark[3], and RESOLVE are integrated. A detailed discussion of the RESOLVE system along with a comparison with several others may be found in [19].

To verify the correctness of the code, a number of proof obligations will need to be met. Obviously, the postcondition of the code that is under verification needs to be proved. In addition, each operation call with a precondition must be checked; invariants must be enforced; and termination must be established. In data abstraction verification, representation invariants must be verified. Each of these proof obligations is transformed into a mathematical assertion called a Verification Condition (VC). A VC takes the form of an implication. In this paper we'll present them in the following format:

```
A and
B and
C
=====================>
D
```

This should be read "Given A and B and C, prove D."

Finally, VCs may be discharged either by hand, with the use of a proof assistant such as Coq [17], or by an automated prover like Yices [6]. Automated provers come in two primary flavors: SMT solvers, which operate very efficiently on pre-defined, finite, first-order logics; and algebraic provers, which mimic human proving and are more flexible, but slower. Languages such as RESOLVE that support higher order logics require provers of the latter flavor[2].

[1]Along with definitions and results from mathematical developments that form the basis for these specifications.

[2]They could, of course, be equipped with less general solvers where those are adequate, as explained in [19].

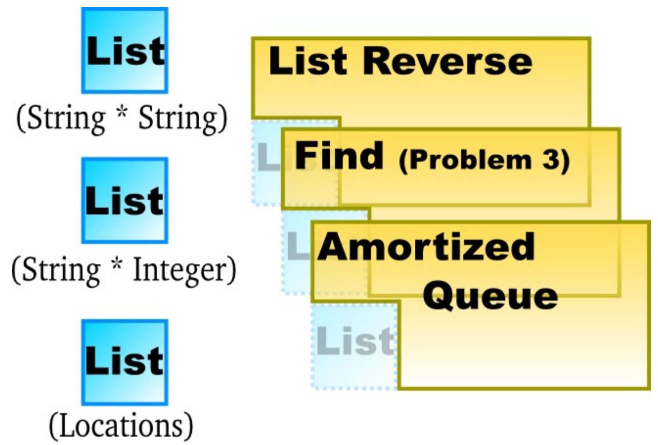

Figure 1: Multiple clients use the `Cursor_List` component and the verification of each client depends on the choice of `Cursor_List` model

The prover (whether human or computer) is supported by mathematical theorems organized into theories. There might be a theory of Integers, Sequence, and others, each containing hundreds of individual theorems. If a system is to remain sound, these theorems must themselves be proved, either by automatic means, or by a mechanically checked proof [21].

2.2 Experiment Background

At the VSTTE 2010 conference, a competition was held allowing teams to submit verified solutions to a set of problems. The problems and proposed solutions are available at [22]. Among these are two problems that involved client code using a linked list. Problem 3 involved searching a linked list for an element with a given value, while problem 5 involved implementing an amortized queue built on top of two linked lists. Verification in both these cases requires some mathematical conceptualization of lists.

Our goal in this paper is to consider several different models of the same data abstraction, `Cursor_List`, which conceptualizes a singly linked list, and analyze how these models—all of which are logically-equivalent—affect the provability of client code, such as those used at the VSTTE 2010 competition.

3. EXPERIMENTS

Our experiments focused on a `Cursor_List`, a list into which the client is provided a cursor that can be moved forward (but not backward) and reset to the beginning. Inserts and deletes occur at the cursor location. To keep the this paper self contained, we used a recursive reversal implementation (a subproblem of problem 5 in the VSTTE competition) as the list client operation. We then analyzed the affect of `Cursor_List`'s mathematical model on ease of verification using the RESOLVE [20] system.

Because the `Reverse()` implementation remained constant, the same eight VCs were generated each time, corresponding to the same eight proof obligations. However, the nature of these VCs differed based on the `Cursor_List` specification that has been used to generate them. This concept is illustrated in Figure 1.

Armed with multiple versions of the `Reverse()` VCs, the final step was to prove them by hand[3] and draw some conclusions about their relative difficulty.

To control for the fact that there is a significant amount of subjectivity in deciding what constitutes a "step" in a proof—after all, each VC could simply be stated as a theorem, making each proof length 1—we established the following guidelines for our theorems:

Be atomic. A theorem should not be considered a single step if it could be decomposed into multiple smaller theorems, unless doing so avoids breaking the next guideline:

Do not introduce functions unnecessarily. While it is certainly true that all of the mathematics expressed in the VCs could be reduced via theorems to set theory or the lambda calculus, then manipulated in that most basic theory, it stands to reason that VCs are best viewed at the level of abstraction that generated them. Consider these theorems:

```
Theorem:
   For all E : Entry, |<E>| = 1;

Theorem:
   For all S, T : String(Entry),
      |S o T| = |S| + |T|;
```

We could certainly use these along the path to proving a fact like `|S o <E>| > |S|`, but by this guideline it would be acceptable to include this fact itself as a theorem, since it it the *most atomic* way of expressing that idea without introducing the `+` function.

3.1 Pointer Model

First, we present a `Cursor_List` specified as it often is in the verification literature: using pointers. We use a theory of abstract `Locations`, where location takes a value from a set with a cardinality that parallels memory capacity. We do not use $\mathbb{Z}$ to model our pointers (a typical alternative) as it might give the illusion that point arithmetic is allowed. The resulting concept is an example of a concept where the mathematical modeling in the specification is influenced by implementation internals.

```
Concept Location_Based_List_Template(
      type Entry);
   uses Std_Integer_Fac, Function_Theory,
      Location_Theory;

   (* Some definitions elided for brevity. *)

   Definition Void: Location;

   Var Entries: Location -> Entry;
   Var Target: Location -> Location;
   Var Is_Used: Location -> B;
      constraints not Is_Used(Void);
      initialization
      for all L: Location,
         Target(L) = Void and
         ((L /= Void) implies not Is_Used(L));

   Type Family List is modeled by (
         First, Cursor, Last: Location);
      exemplar L;
      constraints
         Is_Reachable_from(
            L.First, L.Cursor) and
         Is_Reachable_from (
            L.Cursor, L.Last) and
         (L.Last /= Void implies
            Target(L.Last) = Void);
      initialization
         ensures L.First = Void and
         L.Cursor = Void and L.Last = Void;
      (* finalization clause elided *)

   Operation Advance(updates L: List);
      preserves Entries, Target, Is_Used;
      requires Target(L.Cursor) /= Void;
      ensures L.First = #L.First and
         L.Last = #L.Last and
         L.Cursor = #Target(#L.Cursor);

   Operation Insert(alters E: Entry;
         updates L: List);
      updates Entries, Target, Is_Used;
      ensures
         (there exists New_Pt: Location,
            not Is_Used(New_Pt) and
            Entries(New_Pt) = #E and
            Function_Same_Except_at(
               Entries, #Entries, {New_Pt}) and
            Target(New_Pt, #L.Cursor) and
            (#L.Cursor = Void implies
               L.Cursor = New_Pt and
               (L.Last /= Void implies
                  Target(L.Last) = L.Cursor) and
               Function_Same_Except_at(
                  Target, #Target, {L.Last}) and
                  (L.Last = Void implies
                     Target = #Target) and
                  L.Last = L.Cursor and
                  (L.First = Void implies
                     L.First = L.Cursor)))
         and (#L.Cursor /= Void implies
            L.Cursor = New_Pt and
            (L.First = #L.Cursor implies
               L.First = L.Cursor)) and
         Function_Same_Except_at(
            Target, #Target, {New_Pt, #L.Cursor}));

   (* Other operations elided for brevity. *)

end;
```

In a RESOLVE specification, types are introduced in the context of a Concept, which provides the conceptualizations of those types and the specifications for related operations, but no implementing code. Note that this concept is parameterized by a type called Entry, comparable to a generic in Java. A Family introduces a conceptual type which may have multiple concrete realizations. The clause `exemplar L;` simply introduces a name for the prototypical `Cursor_List` used in the assertions of the Family definition. `#L.First` is RESOLVE notation for the value of `L.First` at the begin-

[3]Most of the resulting VCs are easily dispatched by an automated prover. However, few are equipped to find the *shortest* such proof, a key part of this experiment.

ning of the function call. Also on display are RESOLVE's parameter passing modes, which summarize the affect an implementation is permitted to have on each parameter—parameters that are *updated* will have meaningful incoming values and may be changed in a meaningful way by the operation, parameters that are *cleared* will have meaningful incoming values, but will be change to an initial value by the end of the call, and parameters that are *replaced* have their input values ignored and overridden with result values.

For the purposes of this paper, in the specification of the Insert operation, it is irrelevant whether the inserted entry is specified to be preserved (meaning it remains the same), cleared, or altered (meaning the result value is unspecified). The motivation for avoiding copying (and thus preserving) generic type objects is discussed in [8].

One other principle in [8], however, is relevant: in RESOLVE swapping, not reference or value assignment, is the basic data moving operation and is available on all objects implicitly. So by design, the question of specifying or reasoning about copying a list by reference assignment does not arise in the discussions in this paper.

We define a specific, named `Location`, `Void`, which will serve as the null location. The state space shared by lists is directly modeled in the specification using three shared, conceptual variables, `Entries`, `Target`, and `Is_Used`. The variable `Is_Used` is modeled as a predicate and it indicates whether or not a location is already in use. The variable `Target` is a mapping to the "next linked" location and the variable `Entries` maps a location to the entry. Note that the constraints disallow `Void` from ever becoming allocated and the initialization ensures that all links by default point to `Void`. For more details, we direct the interested reader to [13].

This specification is not fully abstract [23]. The specification of the Advance operation, which interestingly leaves the entire frame unchanged, is straightforward. The specifications of Insert and Remove operations are more involved and they have to include frame-related properties because the conceptual state space is selectively affected. The specification expressions can be simplified with separation logic [18], data refinement [7], or some equivalent. However, verification will still require appropriate frame properties to be proved for the user code. We have not attempted to compare the proof of VCs arising from using this specification with others, because it is much more complex. The existential quantifier, at a minimum, is a hindrance to automated verification, unless user code is documented with suitable witnesses in some form.

In general, introduction of conceptual or shared space adds a non-trivial complexity to specification and corresponding verification. For this reason, it is more appropriate to abstract programming objects along the lines discussed in the next three models, encapsulating these kinds of details inside implementations.

3.2 Before and After Model

We now present a conceptualization of `Cursor_List` that abstracts away most of the details present in the model presented in Section 3.1. In this model, the `Cursor_List` component is modeled as the Cartesian product of two mathematical sequences (called `String`s in RESOLVE's mathematical notation): the first containing those elements *before* the cursor and the second *after*.

As an example of the style of specification, here is a snippet of the `(String * String)` version:

```
Concept Two_Strings_Cursor_List_Template(
      type Entry);
   uses Std_Integer_Fac, String_Theory;

   Family Cursor_List is modeled by
      Cart_Prod
         Before, After: String(Entry);
      end;
      exemplar P;
      initialization
         ensures P.Before = empty_string and
            P.After = empty_string;

   Operation Advance(updates L : Cursor_List);
      requires L.After /= empty_string;
      ensures L.Before =
            #L.Before o <First(#L.After)> and
         L.After = All_But_First(#L.After);

   Operation Insert(clears New_Entry : Entry;
         updates L : Cursor_List);
      ensures L.Before = #L.Before and
         L.After = <#New_Entry> o #L.After;

   Operation Remove(
         replaces Entry_Removed : Entry;
         updates L : Cursor_List);
      requires L.After /= empty_string;
      ensures L.Before = #L.Before and
         L.After = All_But_First(#L.After) and
         Entry_Removed = First(#L.After);

   (* Further operations elided for brevity *)

end;
```

The o operator is string concatenation. `<e>` indicates the string containing the sole element e.The high-level definitions `All_But_First` and `First` have the obvious meanings.

An implementation of Reverse on lists was created using this list specification. Here is the specification and an implementation:

```
Operation Reverse(updates S : Cursor_List);
   requires S.Before = empty_string;
   ensures S.Before = Rev(#S.After) and
      S.After = empty_string;

Procedure Reverse(updates S : Cursor_List);
   decreasing |S.After|;

   Var temp: Entry;

   if After_Length(S) > 0 then
      Remove(temp, S);
      Reverse(S);
      Insert(temp, S);
      Advance(S);
   end;
end;
```

Note that the procedure contains a decreasing clause: RESOLVE demonstrates total correctness using progress metrics such as these. `Rev()` is a mathematical function for reversing a string; it is a definition only and is not backed up by any kind of executable code.

After compiling this with the RESOLVE VC generator, 8 VCs are created, corresponding the various proof obligations in the code. As an example, consider this VC arising from establishing the ensures clause of the Reverse spec (for one path through the code) at the end of the procedure:

```
((((((min_int <= 0) and
(0 < max_int)) and
S.Before = empty_string) and
P_val = |S.After|) and
(|S.After| > 0)) and
Entry.is_initial(First(S.After)))
==================>
(Rev(All_But_First(S.After)) o
      <First((<First(S.After)> o
              empty_string))>) =
   Rev(S.After)
```

The consequent of this implication reduces to:

```
(Rev(All_But_First(S.After)) o <First(S.After)>) =
   Rev(S.After)
```

Which is simply a tautology. We may thus dispatch this VC simply using a few fell-designed theorems. The remaining VCs are, for the most part, simpler than this one.

3.3 List and Position Indicator Model

Another common model is to conceptualize the list as the Cartesian product of a string containing all of the elements in the list and an integer that indicates the index at the insertion point. This `(String * Z)` model leads to a specification of the same operations from before like this:

```
Concept Integer_Pointer_Cursor_List_Template(
      type Entry);
   uses Std_Integer_Fac, String_Theory;

   Family Cursor_List is modeled by
      Cart_Prod
         Entries : String(Entry);
         Insertion_Point : Z;
      end;
      exemplar P;
      constraint
         P.Insertion_Point <= |P.Entries| and
         0 <= P.Insertion_Point;
      initialization
         ensures P.Entries = empty_string and
            P.Insertion_Point = 0;

   Operation Advance(updates L : Cursor_List);
      requires L.Insertion_Point < |L.Entries|;
      ensures L.Entries = #L.Entries and
         L.Insertion_Point =
            #L.Insertion_Point + 1;

   Operation Insert(clears New_Entry : Entry;
         updates L : Cursor_List);
      ensures L.Entries = Left_Substring(
           #L.Entries, #L.Insertion_Point) o
         <#New_Entry> o Right_Substring(
           #L.Entries, #L.Insertion_Point) and
         L.Insertion_Point = #L.Insertion_Point;

   Operation Remove(
         replaces Entry_Removed : Entry;
         updates L : Cursor_List);
      requires L.Insertion_Point < |L.Entries|;
      ensures L.Entries =
           Left_Substring(
              #L.Entries, #L.Insertion_Point) o
           Right_Substring(
              #L.Entries,
              #L.Insertion_Point + 1) and
         Entry_Removed =
           Element_At(
              #L.Entries, #L.Insertion_Point) and
         L.Insertion_Point = #L.Insertion_Point;

   (* Further operations elided for brevity *)

end;
```

Here, `|S|` denotes the length of the string `S`. The definition `Left_Substring(s, x)` returns the first x elements of s, `Right_Substring(s, x)` returns the substring of s starting at element x and continuing to the end, and `Element_At(x)` returns the element at index x.

First, note that the change in specification does not impact any implementation—a working implementation of `Cursor_List` is still a working implementation under either the specification from Section 3.2 or the one in this section.

It is also interesting to note that the model from Section 3.2 leads to a much more succinct specification of at least the `Remove()` operation, while the model in this section leads to a somewhat more succinct `Advance()` operation.

Because the model has changed, the *specification* of `Reverse()` (but not its implementation[4]) must change to follow suit. Here is what the specification looks like under this new model:

```
Operation Reverse(updates S : Cursor_List);
   requires S.Insertion_Point = 0;
   ensures S.Entries = Rev(#S.Entries) and
      S.Insertion_Point = |#S.Entries|;
```

Given these two different mathematical models of a list, we are able to compare how easily they contribute to a verified `Reverse` operation. The number of steps required for a proof are shown in Table 1.

Clearly, the `(String * Integer)` model consistently requires more steps than the `(String * String)` one. For the culprit, consider VC 2[5] (which happens to correspond to establishing termination of `Reverse()`'s recursion), as generated using the former model:

[4]Save for the progress metric, which is a mathematical assertion embedded in the implementation.

[5]Irrelevant conjuncts have been removed for brevity.

Table 1: Proof steps for (String * String) model *vs.* (String * Integer) model.

	Steps	
	(S * S)	(S * I)
VC 1	3	3
VC 2	3	9
VC 3	1	1
VC 4	2	9
VC 5	5	10
VC 6	2	2
VC 7	4	4
VC 8	3	3

```
(|S.Entries| - 0) > 0
=================>
((|Left_Substring(S.Entries, 0) o
      Right_Substring(S.Entries, (0 + 1))| - 0) <
   (|S.Entries| - 0))
```

A full four of the nine required steps are devoted to eliminating spurious zeros and another two determining that the concatenation of `Left_Substring(S.Entries, 0)` adds nothing to the final sequence. Nonetheless, these obvious steps must be taken by the prover to reveal an otherwise straight-forward proof.

3.4 List and Remaining Model

We have observed that often in the literature models are chosen that closely match the internal implementation of a component, even when this model introduces additional complexity in the form of constraints on the model. Presumably, this eases verification of the component implementation, which is not, in and of itself, an invalid strategy if component verification is the sole goal. However, we hypothesize that more constrained models will complicate a verified component's ability to be used as part of client code that is itself verifiable, an essential property in any verification system that is to scale [20].

As a specific example we consider [25] where a `Cursor_List` is represented as a `(String * String)`. However, unlike our previous model with this signature, the first string contains the *entire* list while the second string contains those elements after the cursor. Thus, for a list `L`, those elements *before* the cursor may be derived by considering only the first `|L.All| - |L.After|` elements of `L.All`[6]. We created a similarly specified `Cursor_List` using RESOLVE, yielding this specification:

```
Concept Jahob_Cursor_List_Template(
      type Entry);
   uses Std_Integer_Fac, String_Theory;

   Family Cursor_List is modeled by
      Cart_Prod
         All, Remaining: String(Entry);
      end;
      exemplar P;
      initialization
         ensures P.All = empty_string and
            P.Remaining = empty_string;

   Operation Advance(updates L : Cursor_List);
      requires L.Remaining /= empty_string;
      ensures L.All = #L.All and
         L.Remaining =
            All_But_First(#L.Remaining);

   Operation Insert(clears New_Entry : Entry;
         updates L : Cursor_List);
      ensures L.Remaining =
            <#New_Entry> o #L.Remaining and
         L.All = Left_Substring(#L.All,
               |#L.All| - |#L.Remaining|) o
            <#New_Entry> o #L.Remaining;

   Operation Remove(
         replaces Entry_Removed : Entry;
         updates L : Cursor_List);
      requires L.Remaining /= empty_string;
      ensures L.Remaining =
            All_But_First(#L.Remaining) and
         L.All = Left_Substring(#L.All,
               |#L.All| - |#L.Remaining|) o
            All_But_First(#L.Remaining) and
         Entry_Removed = First(#L.Remaining);

   (* Further operations elided for brevity *)

end;
```

[6]The reality of the model is significantly more complex and for the full details we direct the reader to [11]. However, this simplification suffices for our purposes here.

Because the implementation of the component in [11] is as a single linked list with a pointer to the beginning and a pointer to the current cursor location, this model almost exactly matches the internal implementation details of the component: the first sequence corresponds to those nodes in the view of the linked list defined by starting at the "first" pointer, and the second sequence corresponds to those nodes in the view of the linked list defined by starting at the "cursor" pointer.

Note the implicit relationship enforced by such a configuration: `L.All` must be the concatenation of some sequence `S` with `L.After`.

As before, this necessitates reconceiving our `Reverse()` specification:

```
Operation Reverse(updates S : Cursor_List);
   requires S.All = S.Remaining;
   ensures S.All = Reverse(#S.All) and
      S.Remaining = empty_string;
```

The difficulty of proving each VC resulting from using this model with the `Reverse()` client code is summarized in Table 2 against the results for our original, unconstrained, `(String * String)` model.

Consistent with our intuition, the constrained version requires significantly more steps for two of the VCs (VC 3 and VC 5, corresponding to establishing the precondition on the recursive call to `Reverse()` and establishing the final correctness of `Reverse()` on non-empty input, respectively.)

While this small experiment is insufficient to draw any broad conclusions, we find some support to warrant further investigation.

Table 2: Proof steps for unconstrained (String * String) model *vs.* constrained.

	Steps	
	Unconstrained	Constrained
VC 1	3	3
VC 2	3	3
VC 3	1	4
VC 4	2	2
VC 5	5	8
VC 6	2	2
VC 7	4	4
VC 8	3	3

4. IMPLEMENTATION VERIFICATION

Ultimately, list implementations themselves have to be verified against the list specification, regardless of how it is conceptualized. Assuming that a list is represented internally in a form close to the mathematical modeling given in section 3.1 (with a structure that includes first, cursor, and last pointers), it may be easy to verify it against the "pointer" modeled specification. To verify against other specifications, abstraction functions (or relations) that relate the internal representations with the abstract models would be necessary. Verification of such data abstraction implementations, in general, will involve multiple mathematical theories. However, such verification needs to happen only once for a component implementation. Verification of much software will be at the client end—the focus of the paper—so the specifications should be tuned to ease that verification task.

The complexity of the list specification in version 3.1 should be ideally moved down to the specification of a pointer concept, as explained in [13]; once such a pointer concept and a suitable mathematical specification of lists (such as those in Sections 3.2, 3.3, or 3.3) are available, then it becomes possible to implement the list concept using the pointer concept and contain the verification complexity to that component.

5. RELATED WORK

While, to our knowledge, this is the first experimental exploration of alternative specification formulations in the literature, Hatcliff et al. [9] evaluate multiple specification *systems* (i.e., choices made at the language level) with respect to frame properties. This work includes interesting general discussion comparing the systems. Each is evaluated is in terms of three criteria: abstraction, which measures to what degree heap properties are given in a device-independent way; reasoning, the degree to which clean, automatic reasoning is encouraged; and framing, the degree to which disjoint state spaces affected by code may be identified and expressed. Examples are provided for each style in a different language that exemplifies that specification style, and general discussion follows.

Among the systems discussed in detail [9] are Dafny[16] and Spec#[5], both of which are built on Boogie[4]. The former is an experimental research language, while Spec# is a superset of C# augmented with a specification language.

Another system mentioned in this paper is Jahob: an attempt to verify programs written in the Java language, capturing all Java complexity. It targets a large range of prover backends. Perhaps most relevant to this paper are the exciting result from the Jahob team in [25], in which linked data structures were fully verified using automatic means. However, by contrast to the goals of our research, sizable annotations and significant reasoning about the back-end provers were required on the part of the programmer in order to accomplish this.

The reasoning motivation for specifying linked list behavior with an abstraction removed from implementation details is the topic of [14]. Though it does not contain a specification, such as the one in Section 3.1, it illustrates potential difficulties in specifying a splice procedure and its invariant. The recent work in [7] is also a detailed discussion of client-end difficulties in reasoning in the presence of behavior, such as the one in Section 3.1., but the (partial) solution proposed there concerns how client programs need to be written and not mathematical models used for abstract specification.

6. CONCLUSION

The results of our experiment are promising and suggest a comprehensive evaluation over multiple concept specifications. In addition, the variability in specifications is certainly not limited to choice of model and may include use of functional vs. implicit style alternatives and equivalent, but alternative logical statements, etc., and the impact of such considerations on verification needs to be studied.

Much of the complication inherent in experiments such as the one in this paper currently arise from the necessity to prove VCs by hand to count steps carefully. This is necessary since time taken by itself is not often a sufficient metric. Because specifications are designed to be less-than-ideal, the required length of proofs often exceeds the capabilities of our available automated provers (as algebraic proving is necessarily exponentially complex in either time or space and the application of heuristics complicates complexity metrics.) With additional prover developments, we hope to be able to categorize VCs mechanically.

Ultimately, we hope to arrive at programming-independent specification design and expression guidelines for software component developers, so that avoidable obstacles to automated verification are systematically eliminated.

7. ACKNOWLEDGMENTS

We would like to thank current and past members of the RSRG at Clemson, Ohio State, Denison, and Virginia Tech. This research is supported by NSF grants DMS-0701187 and CCF-0811748.

8. REFERENCES

[1] E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, and A. Starostin. The Verisoft approach to systems verification. In N. Shankar and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 209–224. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-87873-5_18.

[2] E. Alkassar, W. Paul, A. Starostin, and A. Tsyban. Pervasive verification of an OS microkernel. In G. Leavens, P. O'Hearn, and S. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 71–85. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15057-9_5.

[3] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin / Heidelberg, 2006. 10.1007/11804192_17.

[5] M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 144–152. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69149-5_16.

[6] B. Dutertre and L. D. Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[7] I. Filipović, N. Torp-smith, and H. Yang. To appear in formal aspects of computing: Blaming the client: On data refinement in the presence of pointers.

[8] D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Trans. Softw. Eng.*, 17(5):424–435, 1991.

[9] J. Hatcliff, G. T. Leavens, K. Rustan, M. Leino, P. Müller, M. Parkinson, J. Hatcliff, G. T. Leavens, K. Rustan, M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. 2009.

[10] C. Hoare, J. Misra, G. T. Leavens, and N. Shankar. The verified software initiative: A manifesto. *ACM Comput. Surv.*, 41(4):1–8, 2009.

[11] Examples of data structures verified in Jahob, lara.epfl.ch/dokuwiki/data_structure_examples.html, Sept. 2010.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, New York, NY, USA, 2009. ACM.

[13] G. Kulczycki. *Direct Reasoning*. Phd dissertation, Clemson University, School of Computing, Jan. 2004.

[14] G. Kulczycki, M. Sitaraman, B. W. Weide, and A. Rountev. A specification-based approach to reasoning about pointers. In *Proceedings of the Fourth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2005)*, pages 55–62, 2005.

[15] G. T. Leavens, A. L. Baker, and C. Ruby. Jml: a java modeling language. In *In Formal Underpinnings of Java Workshop (at OOPSLA'98*, 1998.

[16] K. Leino and R. Monahan. Dafny meets the verification benchmarks challenge. In G. Leavens, P. O'Hearn, and S. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 112–126. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15057-9_8.

[17] LogiCal Project. *The Coq proof assistant reference manual*, 2004. Version 8.0.

[18] M. Parkinson. The next 700 separation logics. In G. Leavens, P. O'Hearn, and S. Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15057-9_12.

[19] M. Sitaraman, B. Adcock, J. Avigad, D. Bronish, P. Bucci, D. Frazier, H. Friedman, H. Harton, W. Heym, J. Kirschenbaum, J. Krone, H. Smith, and B. W. Weide. Building a pushbutton RESOLVE verifier: Progress and challenges. *Formal Aspects of Computing (to appear)*, 2010.

[20] M. Sitariman and B. Weide. Component-based software using RESOLVE. *SIGSOFT Softw. Eng. Notes*, 19(4):21–22, 1994.

[21] H. Smith, K. Roche, M. Sitaraman, J. Krone, and W. F. Ogden. 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)*, pages 59–66, 2008.

[22] Vstte'10 verified software competition, www.macs.hw.ac.uk/vstte10/competition.html, Sept. 2010.

[23] B. W. Weide, S. H. Edwards, W. D. Heym, T. J. Long, and W. F. Ogden. Characterizing observability and controllability of software components. *Software Reuse, International Conference on*, 0:62, 1996.

[24] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[25] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 349–361, New York, NY, USA, 2008. ACM.