Specification and Verification with References

Bruce W. Weide and Wayne D. Heym
Computer and Information Science
The Ohio State University
Columbus, OH 43210
+1-614-292-1517
{weide,heym}@cis.ohio-state.edu

ABSTRACT

Modern object-oriented programming languages demand that component designers, specifiers, and clients deal with references. This is true despite the fact that some programming language and formal methods researchers have been announcing for decades, in effect, that pointers/references are harmful to the reasoning process. Their wise counsel to bury pointers/references as deeply as possible, or to eliminate them entirely, hasn't been heeded. What can be done to reconcile the practical need to program in the languages provided to us by the commercial powers-that-be, with the need to reason soundly about the behavior of component-based software systems? By directly comparing specifications for value and reference types, it is possible to assess the impact of visible pointers/references. The issues involved are the added difficulty for clients in understanding component specifications, and in reasoning about client program behavior. The conclusion is that making pointers/references visible to component clients needlessly complicates specification and verification.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Languages, Methodologies.

D.2.4 [**Software/Program Verification**]: Correctness proofs, Formal methods, Programming by contract, Reliability.

General Terms

Design, Reliability, Languages, Verification.

Keywords

Java, Pointers, References, Specification, Verification.

1. INTRODUCTION

A well-known "folk theorem" in computing circles is that nearly every problem can be solved with one more level of indirection. Like most folklore, this claim is partially true—a fact not lost on programming language designers, who have consistently delivered not only computational models, but a variety of language constructs, to help us more easily write programs that use indirection.

The belief is a dangerous one, however, which has been noted many times over the past few decades. Writing programs more easily is one thing. Reasoning more easily about their behavior is quite another. As early as 1973, Tony Hoare remarked of pointers that "their introduction into high-level languages has been a step backward from which we may never recover" [10]. In 1976, Dick Kieburtz explained why we should be "programming without pointer variables" [15]. And in 1978, Steve Cook's seminal paper on the soundness and relative completeness of Hoare logic [3] identified aliasing (of arguments to calls, i.e., even in a language without pointer variables) as the key technical impediment to modular verification. There have been recent papers (e.g., [19, 23]) showing how it is *technically* possible to overcome such problems, but apparently only at the cost of even further complicating the programming model that a language presents to a software engineer.

Why do we need another paper about this issue? The consequences of programming with pointers have been examined so far primarily in the context of programming language design and formal methods. We take a position in the context of the human element of specification and verification:

> *Making pointers/references visible to component clients needlessly complicates specification and verification.*

In supporting this position, we rely in part on another, and far older, bit of folklore: "Occam's Razor", a.k.a. the Law of Parsimony. It holds that simpler explanations of phenomena are better than more complex ones. The phenomena of software behavior are entirely of our own making, giving us ample opportunity to control the intellectual complexity and comprehensibility of specifications and reasoning based on them.

Throughout the paper (and with apologies to C++ gurus, as noted in Section 5.1) the terms "pointer" and "reference" are used interchangeably. The point, so to speak, is that from the standpoint of specification and verification difficulties they amount to the same thing. Code examples use Java notation. The reader is also assumed to be familiar with the basis for standard model-based specifications but not with any particular specification language; RESOLVE [27] is used for specification examples, but the notation is explained right here.

Section 2 discusses the difference between value and reference variables, which might seem so well known as to go without saying. (The reason for saying it anyway is detailed in Section 5.2.) Section 3 describes the serious impact of this distinction on the complexity of behavioral specifications, and Section 4 describes the impact on modular verification. Section 5 discusses related work. Section 6 presents our conclusions.

2. VALUES VS. REFERENCES

Popular object-oriented languages, including C++, Eiffel, and Java, share a bizarre feature. They create a dichotomy between two kinds of types and, therefore, two kinds of variables:

- ***Value variables***, which stand for values of the built-in types (***value types***) such as *boolean*, *char*, and *int*.
- ***Reference variables***, which stand for references to objects whose values are of types (***reference types***) introduced through interfaces and classes.

Why is this dichotomy "bizarre"? It clearly is not intuitive, which is obvious if you have ever tried to explain and justify

it to students. Parsimony certainly suggests having only value variables or only reference variables, not both.

Knowing Hoare's hints on programming language design and recognizing the elegance of some purely functional programming languages, the C++, Eiffel, and Java designers must have preferred to have only value variables, all other things being equal. But all other things are not equal. For one thing, there is the folk theorem about indirection. In fact, the use of indirection is a little like the use of tobacco: an addictive bad habit. Modern programming languages have contributed to the problem by making indirection harder and harder to avoid and programs using indirection easier and easier to write. Reference variables are everywhere in Java yet carry no syntactic baggage at all! So surely it would be considered sacrilege to remove easy indirection from any modern imperative language—even though the effect of indirection, when truly appropriate as it is occasionally, could be provided by a small set of library components offering comparable power and performance profiles to language-provided pointers [12, 14].

Of course, tradition is not the reason these popular languages distinguish between values and references. Language designers simply failed to discover another way to make programs *efficient* in terms of execution time and storage usage [11]. Value variables can be represented with small chunks of storage that can easily be copied, leaving *x* and *y* completely independent in code following the assignment statement here:

```
int x;
int y;
...
y = x;
```

If user-defined types were value types that behaved like *int*s, then this kind of code could be terribly inefficient. For example, suppose *x* and *y* were value variables in the following Java code—remember they are not—and so would remain independent in code following the assignment statement:

```
SetOfInt x = new SetOfInt ();
SetOfInt y = new SetOfInt ();
...
y = x;
```

The assignment would then entail deep copying of a *SetOfInt* object representation, which presumably would take time linear in the size of the set *x*. Overriding the assignment operator to make a deep copy is recommended practice for C++ programmers who use the Standard Template Library [24], precisely because this leaves *x* and *y* independent of each other following the assignment. The Java assignment operator, on the other hand, cannot be overridden. An optional *clone* method is supposed to make a deep copy (but it doesn't, in fact, even for the collections in the popular java.util package).

Having reference variables directly addresses the performance problems involved in copying large data structures because:

- the representations of all variables remain small, i.e., the size of one pointer each, although every reference variable still refers to an object whose representation is a potentially large data structure; and
- the assignment statement is fast for both value and reference variables.

Most of the rest of this paper discusses the price paid for following this road to efficiency: complications in specification and verification, and therefore in understanding and reasoning about program behavior. The appendix (adapted from [14]) briefly explains the *swapping paradigm*, an alternative approach that permits the same efficiency to be achieved without introducing references into the language model, specifications, or programmer reasoning. The purpose of the appendix is to suggest that there are other solutions to the apparent reasoning vs. efficiency trade-off, i.e., that the choice is not limited to a pure functional programming paradigm (reasoning over efficiency) or the standard object-oriented programming paradigm (efficiency over reasoning).

3. IMPACT ON SPECIFICATION

Simply introducing reference types into a language model makes it harder for clients to understand the specified behavior of components—if such behavior were carefully specified, which in practice (e.g., Java component libraries) it is not. This section illustrates the additional complication by describing a reference type in a model-based specification language, RESOLVE, that is designed for specifying value types [27]. That is, there should be no syntactic sugar through which the specification language might mask the fact that there is a reference type. This approach allows an apples-to-apples comparison of the underlying "intellectual load" introduced by value vs. reference types, both on component specifiers and on clients of those specifications.

Wouldn't it be fair to (also?) select a specification language that is designed to handle reference types, and use it to try to specify value types? Not really. Variables in traditional mathematics stand for values; they do *not* stand for references to objects that have values. In other words, a hypothetical specification language that is designed to hide references behind syntactic sugar must still, in the final analysis, "mean" (i.e., have its semantics) in the domain of traditional mathematics. The verification conditions arising in correctness proofs must be stated in traditional mathematics in order that proofs can be carried out. "Desugaring" from references to values is, therefore, ultimately required. It is only in the desugared version of this hypothetical specification language that we could really compare the relative difficulties values and references pose for specification writers and readers.

3.1 Defining Mathematical Models for Types

Let's start with a simple case: specifying the mathematical model for a built-in value type. For example, for type *int* in Java the obvious mathematical model is a mathematical integer constrained to be within some bounds. In RESOLVE notation, this is expressed as follows:

```
type int is modeled by integer
  exemplar i
  constraint
    -2147483648 <= i <= 2147483647
```

The *exemplar* clause introduces an arbitrary name for a prototypical variable of the new type, and the *constraint* clause is an assertion that describes such a variable's value space. So, the meaning of this specification is that in reasoning about Java code such as that shown earlier using *int* variables, you should think of the values of *x* and *y* as being mathematical integers like 1372 and –49 (i.e., not as strings of 32 bits).

3.1.1 Value Type Specification

A similar scenario arises for a type such as *SetOfInt* whose mathematical model is more complex and whose representation is potentially large. For example, if *SetOfInt* were a value

type in the earlier Java code—remember it is not—then you would want to think of the values of *x* and *y* as being sets of numbers like {1, 34, 16, 13} and {2, –9, 45, 67, 15, 16, 942, 0}. The mathematical model specification would look like this:

```
type SetOfInt is modeled by
    finite set of integer
  exemplar s
  constraint
    for all k: integer where (k is in s)
      (-2147483648 <= k <= 2147483647)
  initialization ensures
    s = {}
```

The *initialization* clause says that when a new *SetOfInt* variable is declared, its value is the empty set.

3.1.2 Reference Type Specification

Unfortunately, life is not so simple in Java: *SetOfInt* is a reference type. In order to reason soundly about what your programs do, you must think of the values of *x* and *y* as being references to objects whose values are sets of numbers like {1, 34, 16, 13} and {2, –9, 45, 67, 15, 16, 942, 0}. That is, the fact that this is a reference type must be made explicit in the type's mathematical model specification. How can this be done?

Without syntactic sugar to hide references, the obvious approach (known to many others) is to model the mapping of references to sets of integers as a mathematical function whose scope is global to all *SetOfInt* variables. In RESOLVE, you can say this using abstract *state variables* that may be accessed and updated in (the specification of) any method associated with any variable of the type being specified. An appropriate mathematical model can be expressed as follows; there are other ways to do it but this is the simplest one we know:

```
state variables
    last: integer
    objval: function from integer to
              finite set of integer
  constraint
    for all r: integer
      (for all k: integer
          where (k is in objval(r))
        (-2147483648 <= k <= 2147483647))
  initialization ensures
    for all r: integer (objval(r) = {})

type SetOfInt is modeled by integer
  exemplar s
  initialization ensures
    last = #last + 1  and
    objval = #objval  and
    s = last
```

The state variable *last* is an abstraction of the address held in a *SetOfInt* variable. Its purpose is to ensure that a newly constructed *SetOfInt* object is independent of all others. The starting value of *last* does not matter because, each time a new *SetOfInt* object is constructed, the value of *last* is incremented ("#" before a variable name denotes the old value). Since *last* is an abstract variable, there is no need to worry about eventual overflow.

The value *null*, however, is an annoying problem: there must be some way to tell it apart from other values. This can be handled in the above model with a minor change:

```
  initialization ensures
    last = 0  and  ...
```

in which case null is modeled by 0. Throughout the rest of this paper, however, we ignore the possibility of null references. There are two reasons. First, we are trying to evaluate how little additional trouble is necessarily entailed by having reference types. Allowing null references only makes method specifications messier, i.e., what happens for null and what happens for non-null values of all the method parameters that are of reference types. Second, it might be possible in principle to have a language in which there were reference types but no null references. Java is used here for illustration, but we don't want to limit observations about reference types to Java.

The state variable *objval* is an abstraction of the mapping between references and the values of the objects they refer to. Again, *objval* is an abstract mathematical variable, so there is no problem that it (or, for that matter, *last*) has a value from a mathematical domain that is manifestly too large to represent. In this specification, we decided to initialize *objval* so every possible reference is mapped to an empty set. The illusion is that there is an infinite pool of objects whose values are empty sets of integers, and that every time a new *SetOfInt* object is constructed, one of these pre-formed objects is selected from that pool. There are other ways to model the situation, of course, but none is any simpler or cleaner when written out.

It is already evident that the mathematical machinery involved in modeling the reference type is significantly more complex than that needed to model the corresponding value type. But this is only part of the problem; there remains the issue of specifying the behavior of methods.

3.2 Defining Method Behavior

Let's consider a method to add an *int* to a *SetOfInt*:

```
public void addInt (int i);
```

3.2.1 Value Type Specification

If *SetOfInt* were a value type in Java—remember it is not—then the specification for *addInt* might look like this:

```
evaluates i
updates self
requires
  i is not in self
ensures
  self = #self union {i}
```

Before the precondition (*requires* clause) and the postcondition (*ensures* clause), the lists of variables classify each variable in scope as either unchanged (*restores* or *evaluates* list) or potentially modified (*updates* or *replaces* list). "Restores" means that the abstract value of the parameter undergoes no net change from call to return, but it might be modified temporarily while the method is operating. Because *i* is passed by value in Java, and the corresponding actual parameter is treated as an expression, *i* is listed as having *evaluates* mode.

3.2.2 Reference Type Specification

Here is what happens because *SetOfInt* is really a reference type:

```
evaluates i
restores self, last
updates objval
requires
  i is not in objval(self)
ensures
  objval(self) = #objval(self) union {i} and
  for all r: integer where (r /= self)
    (objval(r) = #objval(r))
```

Note that *self* is not changed because it is a reference. But the *SetOfInt* object it refers to has its value (i.e., *objval*(*self*)) changed. The last clause of the postcondition says that no other *SetOfInt* object has its value changed.

All the other public methods for *SetOfInt* have specifications with the same flavor as *addInt*. So, all of this is "boilerplate":

```
restores self, last
updates objval
ensures
  for all r: integer where (r /= self)
    (objval(r) = #objval(r))
```

By making these oft-repeated specification clauses implicit with a wave of the hand, it is possible to create a specification language with enough syntactic sugar to simplify the *look* of a specification for a reference type. In ESC/Modula-3 [19], for example, variables not in a "modifies" list are preserved, and the value of a referenced object (e.g., *objval*(*self*)) can be listed as though it were a variable name, so the short version of the above statements is (in RESOLVE-like syntax) just:

```
updates objval(self)
```

This does not materially change the intellectual task of understanding the *meaning* of the specification, however. And as noted in Section 4, the underlying additional complication of references reveals itself once you start relying on that specification to try to reason about client code that uses *SetOfInt*.

3.3 Assignment

It is instructive to specify the behavior of the Java assignment operator, prototypically of the following form:

```
lhs = rhs;
```

3.3.1 Value Type Specification

If *SetOfInt* were a value type in Java—remember it is not—then the specification would be:

```
evaluates rhs
replaces lhs
ensures
  rhs = lhs
```

Note that the "=" in the specification is not itself an assignment operator, but denotes the assertion of ordinary mathematical equality between the mathematical models of *lhs* and *rhs*. We have written "rhs = lhs" rather than the equivalent "lhs = rhs" to emphasize this, any ambiguity being removed by the specification that *rhs* is merely evaluated. The confusing use of "=" as an assignment operator is an unfortunate design choice that crept from Fortran back into C after having been nearly eradicated by ":=" in Algol-like languages.

3.3.2 Reference Type Specification

Interestingly, the assignment specification looks virtually identical for *SetOfInt* as a reference type, the only difference being that *last* and *objval* are also listed as being unchanged:

```
evaluates rhs
restores last, objval
replaces lhs
ensures
  rhs = lhs
```

Maybe there is some comfort in knowing that the assignment operator does "the same thing" for value and reference variables. Of course, the only reason it does "the same thing" is that the mathematical model for a reference type makes the value of a reference variable explicit and distinct from the value of the object it refers to. The assignment operator simply copies the value of the (value or reference) variable on the right-hand side to that on the left-hand side.

4. IMPACT ON MODULAR VERIFICATION

It is widely acknowledged that practical verification must be modular, a.k.a. compositional. Factoring of the verification task cuts along the lines of programming-by-contract [22]. That is, a component implementation is verified against its specification once and for all, out of the context of the client programs that might use it. The legitimacy of client use of a component implementation is gauged during verification of the client, based on knowledge of only the component specification, i.e., without "peeking inside" the separately-verified component implementation and without reverifying any part of it on a per-use basis.

The primary verification issue for software with references stems from the possibility of aliasing: having two or more references to the same object. Aliasing can arise either from reference assignment (the case considered here) or from parameter-passing anomalies (the case Cook considered in his study of Hoare logic [3]; see also [13, 17]). The challenge here is to discover how the specification of *SetOfInt* in Section 3 might be used in modular verification of a client of *SetOfInt*, if the client program could execute a reference assignment.

Let's consider a relatively simple situation where the client program is a main program having two "helper" operations *P* and *Q* with specifications not shown:

```
import Section3.SetOfInt;
class Client {
  private static void P (SetOfInt si) {
    ...
  }
  private static int Q (int i) {
    ...
  }
  public static void main (...) {
    int j, k;
    SetOfInt s1 = new SetOfInt();
    SetOfInt s2 = new SetOfInt();
    ...
    P(s1);
    ...
    k = Q(j);
    ...                              // point A
    P(s2);                           // point B
    ...
  }
}
```

Suppose this program uses no other classes or constructs that might cause modular verification problems, so the focus is entirely on the impact of using *SetOfInt*. In other words, suppose *main*, *P*, and *Q* could be verified independently except for any effects introduced by using *SetOfInt*.

4.1 Value Type Verification

If *SetOfInt* were a value type in Java—remember it is not—then variables of this type could be passed from *main* to *P* without fear that modularity might be compromised. The specification of *SetOfInt* as a value type makes this clear. There are no state variables in that specification and, consequently, no shared

state would be introduced among *main*, *P*, and *Q* as a result of their common visibility over the *SetOfInt* class. For example, suppose the intended behavior of *P* were this:

```
updates si
ensures
  si = #si union {13}
```

You would be able to reason about the correctness of the body of *P* independently of the bodies of *main* and *Q* because there would be nothing *P*'s body could do to the values of any variables in the program other than the argument passed for the formal *si* in a given call. The same would be true of the bodies of *main* and *Q*. Reasoning would remain modular even with this user-defined type in the picture—*if* it were a value type.

4.2 Reference Type Verification

In truth, *SetOfInt* is a reference type. But suppose, in a fit of wishful thinking, you decided that it didn't matter that much and made the simplification of thinking of *SetOfInt* as a value type. Given the specification above, you might expect *P* to have the following body:

```
if (! si.contains (13)) {
  si.addInt (13);
}
```

The problem with your thinking would be that *P* has visibility over the reference type *SetOfInt*, including the abstract state variables *last* and *objval*. Through them *P* might do other things. For example, *P* might copy and save the reference *s1* that *main* passes in the first call, and then quietly change *objval*(*s1*) through that alias during the next call. If you erroneously thought of *SetOfInt* as a value type, then it would seem that the value of *s1* changed spontaneously between the points labeled "A" and "B" in *main* even though the variable *s1* was not even mentioned in the statement executed between them. In reality, of course, what was changing was *objval*(*s1*); but by hypothesis you were oblivious to the abstract state variable *objval* and were thinking of *si* as a value variable— a "no-no".

So, the following might be the body of *P*. It also seems to satisfy the specification above in terms of its effect on *si*, if you treat *si* as a value variable and thereby ignore *objval*. Here, *Alias* is a simple class with two static methods, *saveTheAlias* and *theAlias*, which copy an *Object* reference and return the copy, respectively. The point is that nowhere outside the body of *P* is there even a hint that an alias is being kept inside it.

```
if (Alias.theAlias () != null) {
  ((SetOfInt) Alias.theAlias ()).clear ();
}
Alias.saveTheAlias (si);
if (! si.contains (13)) {
  si.addInt (13);
}
```

In reasoning about the body of *main*, how could you predict the strange behavior resulting from this code without examining the body of *P*—and thereby giving up modular reasoning? The key to salvaging modularity is to realize that the specification of *SetOfInt* as a reference type involves two abstract state variables, *last* and *objval*, that are visible throughout *main*, *P*, and *Q*. From the reasoning standpoint, there are variables in this program that are global to *main*, *P*, and *Q*, although the syntax of Java does a great job of hiding them.

Now *main* still can be verified independently of *P* and *Q* despite sharing *last* and *objval* with them. The specifications of *P* and *Q* simply must describe their effects on the abstract state variables *last* and *objval* as well as on their explicit parameters. *P*'s specification should be changed to this:

```
evaluates si
restores last
updates objval
ensures
  objval(si) = objval(#si) union {13}  and
  for all r: integer where (r /= si)
    (objval(r) = #objval(r))
```

Knowing only that *P* preserves *last* does not allow the verifier of *main* to be sure that *P* cannot create an alias by copying *si* and then changing the object value later. But the "nothing else changes" clause in the postcondition prevents a correct body for *P* from doing anything funny with an alias (like the second body above) even if it saves one.

Another possibility is that maybe the above specification isn't really what is wanted! Perhaps the weird implementation of *P* is correct according to the programmer's intent, and the problem is specifying what *P* is supposed to do. Such a situation also can be handled in this specification framework.

This example shows why it is critical for sound reasoning that a programmer not imagine and/or hope that reference variables are sort of like value variables. They aren't.

Can *Q* be verified independently of *main* and *P* despite sharing *last* and *objval* with them? Here, *main* and *P* can manipulate *last* and *objval* by executing any series of *SetOfInt* method calls. It turns out that *Q* cannot see the effects of those manipulations even if it declares and uses *SetOfInt* variables of its own—and vice versa. But the basis for this claim is not clearly evident from the specification of *SetOfInt*. It is a consequence of a special "non-interference" property that arises from the way the *SetOfInt* specification uses the abstract state variables: Neither of two methods declaring their own *SetOfInt* variables but otherwise not communicating with each other can detect changes that are made by the other to the abstract state variables. So, curiously, *Q* can be verified independently of *main* and *P* in this case even if its specification does not include a "nothing else changes" clause.

5. RELATED WORK

Following the early papers cited in Section 1, there have been some interesting recent episodes in the literature on programming language design, specification, and verification. They suggest a fundamental struggle between acknowledging the folklore about the importance and power of indirection, and the reasoning problems arising from its use. We briefly review two language designs, the cases of C++ and Java, in Sections 5.1 and 5.2, respectively. Other researchers have investigated some of the specification and verification difficulties arising from pointers. We briefly discuss their work in Section 5.3.

5.1 C++

C++ makes a distinction between pointers and references, as explained by Bjarne Stroustrup, the creator of C++ [30]:

> A *reference* is an alternative name for an object. The main use of references is for specifying arguments and return values for functions in general and for overloaded operators... [T]he value of a reference cannot be changed after initialization; it always refers to the object it was initialized to denote.

That is, references were introduced into C++ primarily to simplify parameter passing and overload resolution. These programming language concerns had nothing to do with trying to address the reasoning problems that arise from using pointers. Indeed, C++ still has pointers, too.

The decision to complicate C++ by not only introducing references, but making them different from pointers in a rather subtle way, might seem to be another "step backward". But other language features combine with references to give the C++ programmer the flexibility to change the default programming model from reference-oriented to value-oriented. That is, it turns out it is quite possible in C++ to keep pointers and references from bubbling up through component (class) interfaces where they must be faced by clients reading specifications and verifying client code. One of these extra features is the ability to override the assignment operator and copy constructor so they make deep copies, not merely copies of references.

The problem is that there is a performance penalty for making deep copies, as discussed earlier. Luckily, the flexibility of C++ does not stop there. It is also possible to make both the assignment operator and copy constructor private, so they are simply unavailable to clients of a class.

We have taken advantage of the latter feature (and several others) to create a disciplined style of programming in C++, the RESOLVE/C++ discipline [14, 33], in which adherence to many rules of the discipline is compiler-checked by C++ itself. The bottom line is that you can program in C++ using what are technically reference variables yet maintain the *illusion* that you have only value variables. To achieve this, we introduced the swap operator [9] to replace the private assignment operator and copy constructor. Then we designed a large library of class templates [25] whose formal specifications allow clients to reason modularly about client code [14, 33]. The RESOLVE/C++ discipline has been shown to be rather easily understandable and usable by introductory CS students [20, 28, 29] and has been shown to result in dramatically good code quality when used to build a commercial software system [14]. See the appendix for a brief discussion of the key idea behind the discipline, i.e., the swapping paradigm.

5.2 Java

By the time Java was born, Sun Microsystems apparently sensed that people were worried about the "safety" of their programming languages. Thus, the conservatism of Java's design was heavily stressed. In the first paragraph of *The Java Language Specification*, James Gosling, Bill Joy, and Guy Steele wrote [7]:

> Java is intended to be a production language, not a research language, and so, as C. A. R. Hoare suggested in his classic paper on language design, the design of Java has avoided including new and untested features.

Some of the early literature about Java also argued that it did not have certain old and well tested but known-to-be-dangerous features—like pointers. For example, consider this passage written by Gosling and Henry McGilton in their 1996 white paper on *The Java Language Environment* [8]:

> [P]ointers are one of the primary features that enable programmers to put bugs into their code. Given that structures are gone, and arrays and strings are objects, the need for pointers to these constructs goes away. Thus the Java language has no pointers.

Later, it became clear that this claim was a bit of an overstatement, or at least that it could be considered correct only in the legalistic sense that Java does not have pointer *syntax*. Of course, it has pointers almost everywhere, but it calls them references. The potential for confusion was addressed by Sun Microsystems itself in its on-line Java FAQ [31]:

> How can I program linked lists if there are no pointers?
>
> [Answer:] Of all the misconceptions about the Java programming language, this is the most egregious. Far from not having pointers, object-oriented programming is conducted in the Java programming language exclusively with pointers. In other words, objects are only ever accessed through pointers, never directly. The pointers are termed "references" and they are automatically dereferenced for you.
>
> > "An object is a class instance or an array. The reference values (often just references) are *pointers* to these objects." Java Language Specification, section 4.3.1. [emphasis is in the original text]
>
> Any book that claims Java does not have pointers is inconsistent with the Java reference specification.

Interestingly, then, some of Hoare's general advice about programming language design was heeded by the Java designers. But his specific warning about pointers was ignored, early claims to the contrary notwithstanding. By the way, what is the correct answer to the FAQ question, "How do I program linked lists?" You don't; you use java.util.List, or similar.

5.3 Specification and Verification

In the 1970s, several researchers addressed pointer specification and verification in the context of the precursors to object-oriented languages, notably Pascal. The culmination of this effort was reported in a 1979 paper by David Luckham and Nori Suzuki [21], where the modeling of the state of memory was made explicit in specifications and verification conditions in a slightly different way than we have done it. They introduced a mapping from the reference variable's textual name, not its mathematical model value (integer in our case), to the data value it pointed to. They would write the type-specific state variable we call *objval* in our example as *P#SetOfInt*, for "pointer to *SetOfInt*". Special notation also was introduced for dereferencing a pointer-to-*SetOfInt* variable *s* when writing assertions, i.e., *P#SetOfInt s* .

An important missing ingredient in this early work—apparently because Pascal lacked user-defined types with hidden representations—was any use of abstraction in explaining the behavior of new types. For example, in our *SetOfInt* specification as a reference type, as a client you may think of *objval*(*s*) as being a mathematical set of integers. In the Luckham/Suzuki style of specification, you would see not only the top-level reference complication but the pointers to the nodes in the (unhidden) data structure that represented the set. In other words, in 1979 and in Pascal, client component specifications for user-defined types exhibited all the complexity of specifications of reference types in Java, and then some. This was technically acceptable from the formal standpoint of verification but could not be used to give a fair comparison between specifying reference types and specifying value types because specifying reference types this way was even uglier than it needed to be, with no simplifying abstractions.

In 1980, George Ernst and Bill Ogden [5] considered similar specification and verification issues in Modula, which had a module construct with hidden exported types. They, therefore, needed to consider the question of how it was possible to hide reference types behind abstract specifications. They showed it was technically possible to hide references in module specifications through the use of some syntactic sugar in the specification language and an appropriate abstraction function in the module implementation. But the complexity moved over the horizon and into the proof rules:

> The only conceptual difficulty with the verification rules presented in this paper is that they do not prevent a procedure from side-effecting certain instances of abstract types which are not parameters to a call on it... [T]o verify a module, we must verify everything prescribed by the rule ..., but we must also verify that the side-effecting ... cannot occur. Developing such a rule is a non-trivial task ... beyond the scope of this paper.

One problem with showing that "the side-effecting ... cannot occur" is that it *can* occur according to the Modula language definition by assignment of a reference variable; even worse, some programmers *want* it to occur and write programs that way, and these programs might be correct, as noted in the example of Section 4.2. This means that hiding the complexity of references in a proof obligation stating that there is no aliasing causes a completeness problem.

In 1994, Ernst and Ogden, along with Ray Hookway, published a verification method for ADT realizations that handled "shared realizations" [6], including heap storage. Their approach to modeling references was essentially identical to our approach for reference types, with syntactic sugar hiding the abstract state variables that recorded the "serial number" of the last object constructed and the mapping from reference values to data values. A value-type specification was possible, with reference details arising only within the proof of the realization, because the only source of possible aliasing in the language was within realization code, i.e., not from client assignment of references. Moreover, the paper contained another caveat about the example used for specification and verification with a shared realization:

> The example does not use heap memory, because it would require extensive use of pointers, which would unnecessarily complicate both specification and verification...

So, the fundamental problem of how to verify programs with reference types seems technically solvable by making sure that the abstract state variables associated with reference variables "follow them around" throughout the proof. Everyone seems to agree that the introduction of references seriously complicates both specification and reasoning, though.

Other work that is directly related involves specification and verification of standard object-oriented software, where reference types are considered something we just have to learn to live with. The primary group in this area includes Gary Leavens, K. Rustan M. Leino, Peter Müller, and Arnd Poetzsch-Heffter, who have worked on similar issues both separately and in combination. They have tackled the problem of specifying behaviors of components involving pointers and references, and have dealt with potential aliasing both from copying of references and from parameter passing anomalies [4, 16, 17, 18, 19, 23]. Others (e.g., [2]) also have proposed ways to limit yet not eliminate aliasing through clever linguistic mechanisms. But these approaches have yet to be shown understandable by the real programmers they are supposed to enable to deal with references, and there is little evidence so far that this will be easily achieved. In other words, despite impressive technical advances that could contribute to the survival of reference types in our languages, these ideas still need to be validated against their ultimate objective: to become practically useful and comprehensible by real programmers.

Of course, we also have published some prior work in this area [13, 26, 32, 33, 34], including evidence of the comprehensibility and practical effectiveness of the RESOLVE discipline, which simply eliminates reference types [14, 28]. And Manfred Broy, like us, has generally suggested designing components for ease of specification, as opposed to writing "post-mortem" specifications for previously-designed components [1]. This would suggest simply avoiding reference types: the advice we'd get from Occam, too, were he still around.

To summarize, we were unable to find any work directly comparing the intellectual loads involved in specifying value types vs. reference types, or using such analysis to compare the difficulty in reasoning about client programs using them.

6. CONCLUSIONS

Adding references types to value types, as in Java, significantly and needlessly complicates standard model-based specifications and the modular verification they help enable. Technically, everything can be made to "work" with reference types. Reasonably concise and even plausibly comprehensible model-based specifications can be designed that account for the behavioral peculiarities arising from references.

By comparing the complexity of mathematical models and method specifications in a language that has no syntactic sugar to mask references, though, it becomes obvious that reference types introduce a substantially greater intellectual load than value types for both specifier and client. Writing specifications for reference types suggests obvious ways in which syntactic sugar can shorten the specifier's typing time, while still acknowledging a distinction between value types and reference types. Yet it is unlikely that such sugaring can in any way simplify the specifier's thinking or the client's ability to understand the specified behavior of reference types.

In general, modular verification remains possible in the face of reference types *if* the abstract state variables needed to specify a reference type are considered part of the state space of all units that have visibility over that type. For verification purposes the abstract state variables used in specifying a reference type can be treated like additional ghost parameters to all calls involving one or more explicit parameters of that type. Reasoning is, of course, far more complicated with these extra variables in the picture than it would be with value types only. But technically you can still have modular verification with reference types if there are no other language constructs that thwart modularity.

It remains common practice to encode indirection in Fortran by using arrays and integer indices as though they were dynamically-allocated storage pools and pointers into them. These arrays and integers are passed among subroutines as parameters, or sometimes placed in named common blocks that are visible to a selected subset of the subroutines in a program. All the subroutines that manipulate these arrays must agree on how they are using them in order to work correctly together. Over the years, programming language designers have simplified the syntax required to do this sort of thing, to the point

where in Java there is almost no syntax at all associated with indirection. But the underlying logic of programming with references in Java is the same as the logic of programming with arrays and indices in Fortran.

Is it, then, a good idea to hide the sharing of global state by making such language "advances"? This sharing is clearly evident in Fortran programs, but not in Java programs. But shared state introduced through references can remain hidden only from the *minds* of programmers who program without specifications and who never try to verify their programs. If specification language designers try to take the same road then they, too, will find they can hide this shared state only from the minds of programmers who never try to verify programs. Eventually, the emperor's thin disguise will reveal itself to all—although perhaps not before some catastrophic software failures cause more people to take a serious look at whether programming languages should offer constructs that are so well known to complicate specification and verification.

7. ACKNOWLEDGMENTS

Murali Sitaraman and Gary Leavens and their students, as well as the members of the OSU Reusable Software Research Group, have provided much food for thought through various personal and electronic discussions of some of the issues mentioned here. Scott Pike and the anonymous referees helped in many other ways, too, especially by suggesting how to focus the paper on its real thesis.

We gratefully acknowledge financial support from the National Science Foundation under grant CCR-0081596, and from Lucent Technologies. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation or Lucent.

8. REFERENCES

[1] Broy, M. *Experiences with Software Specification and Verification Using LP, the Larch Proof Assistant*. Research Report 93, Compaq Systems Research Center, Palo Alto, CA, 1992.

[2] Clarke, D.G., Potter, J.M., and Noble, J. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, ACM Press, 1998, 48-64.

[3] Cook, S.A. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing 7*, 1 (1978), 70-90.

[4] Egle, R. *Evaluating Larch/C++ as a Specification Language: A Case Study Using the Microsoft Foundation Class Library*. TR #95-17, Department of Computer Science, Iowa State University, Ames, IA, 1995.

[5] Ernst, G.W., and Ogden, W.F. Specification of abstract data types in MODULA. *ACM Transactions on Programming Languages and Systems 2*, 4 (1980), 522-543.

[6] Ernst, G.W., Hookway, R.J., and Ogden, W.F. Modular verification of data abstractions with shared realizations. *IEEE Transactions on Software Engineering 20*, 4 (1994), 288-207.

[7] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996.

[8] Gosling, J., and McGilton, H. *The Java Language Environment: A White Paper.* Sun Microsystems, Inc., 1996; http://java.sun.com/docs/white/langenv/ viewed 8 August 2001.

[9] Harms, D.E., and Weide, B.W. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering 17*, 5 (1991), 424-435.

[10] Hoare, C.A.R. *Hints on Programming Language Design*. Stanford University Computer Science Department Technical Report No. CS-73-403, 1973. Reprinted in *Programming Languages: A Grand Tour*, E. Horowitz, ed., Computer Science Press, Rockville, MD, 1983, 31-40.

[11] Hogg, J., Lea, D., Holt, R., Wills, A., and de Champeaux, D. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, April 1992. http://gee.cs.oswego.edu/dl/aliasing/aliasing.html viewed 8 August 2001.

[12] Hollingsworth, J.E. and Weide, B.W. Engineering 'unbounded' reusable Ada generics. In *Proceedings of 10th Annual National Conference on Ada Technology*, 1992, ANCOST, 82-97.

[13] Hollingsworth, J.E. Uncontrolled reference semantics thwart local certifiability. In *Proceedings of the Sixth Annual Workshop on Software Reuse*, 1993.

[14] Hollingsworth, J.E., Blankenship, L., and Weide, B.W. Experience report: Using RESOLVE/C++ for commercial software. In *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, 2000, ACM Press, 11-19.

[15] Kieburtz, R.B. Programming without pointer variables. In *Proceedings of the SIGPLAN '76 Conference on Data: Abstraction, Definition and Structure*, 1976. ACM Press.

[16] Leavens, G.T., and Cheon, Y. Extending CORBA IDL to specify behavior with Larch. In *OOPSLA '93 Workshop Proceedings: Specification of Behavioral Semantics in OO Information Modeling*, 77-80; also TR #93-20, Department of Computer Science, Iowa State University, Ames, IA, 1993.

[17] Leavens, G.T., and Antropova, O. *ACL — Eliminating Parameter Aliasing with Dynamic Dispatch*. TR #98-08a, Department of Computer Science, Iowa State University, Ames, IA, 1998.

[18] Leavens, G. T., Baker, A. L., and Ruby, C. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, eds. H. Kilov, B. Rumpe, and I. Simmonds, Kluwer Academic Publishers, Boston, MA, 1999.

[19] Leino, K.R.M., and Nelson, G. *Data Abstraction and Information Hiding*. Compaq SRC Rep. #160, 2000.

[20] Long, T.J., Weide, B. W., Bucci, P., Gibson, D. S., Hollingsworth, J., Sitaraman, M., and Edwards, S. Providing intellectual focus to CS1/CS2. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, 1998, ACM Press, 252-256.

[21] Luckham, D.C., and Suzuki, N. Verification of array, record, and pointer operations in Pascal. *ACM Transactions on Programming Languages and Systems 1*, 2 (1979), 226-244.

[22] Meyer, B. *Object-oriented Software Construction*. Prentice-Hall, New York, 1988; second edition, 1997.

[23] Müller, P., and Poetzsch-Heffter, A. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems*, eds. G.T. Leavens and M. Sitaraman, Cambridge University Press, 2000, 137-159.

[24] Musser, D.R., Derge, G.J., and Saini, A. *STL Tutorial and Reference Guide, Second Edition*. Addison-Wesley, Upper Saddle River, NJ, 2001.

[25] RESOLVE/C++ Component Catalog Home Page. http://www.cis.ohio-state.edu/~weide/sce/rcpp/RESOLVE_Catalog-HTML viewed 8 August 2001.

[26] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J.E. Reasoning about software-component behavior. In *Software Reuse: Advances in Software Reusability (Proceedings Sixth International Conference on Software Reuse), LNCS 1844*, ed. W. Frakes, 2000, Springer-Verlag, 266-283.

[27] Sitaraman, M., and Weide, B.W. Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes 19*, 4 (1994), 21-67.

[28] Sitaraman, M., Long, T.J., Weide, B.W., Harner, J., and Wang, C. A formal approach to component-based software engineering: education and evaluation. In *Proceedings 2001 International Conference on Software Engineering*, 2001, IEEE, 601-609.

[29] Software Component Engineering Course Home Page. http://www.cis.ohio-state.edu/~weide/sce/now viewed 8 August 2001.

[30] Stroustrup, B. *The C++ Programming Language, 3rd edition*. Addison-Wesley, Reading, MA, 1997.

[31] Sun Microsystems, Java "Frequently Asked Questions". http://java.sun.com/people/linden/faq_b.html viewed 8 August 2001.

[32] Weide, B.W., Edwards, S.H., Harms, D.E., and Lamb, D.A. Design and specification of iterators using the swapping paradigm. *IEEE Transactions on Software Engineering 20*, 8 (1994), 631-643.

[33] Weide, B.W. *Software Component Engineering*. OSU Reprographics, Columbus, OH, 1996.

[34] Weide, B.W. "Modular regression testing": Connections to component-based software. In *Proceedings Fourth ICSE Workshop on Component-Based Software Engineering*, 2001, IEEE, 47-51.

9. APPENDIX: THE SWAPPING PARADIGM

How do you make some variable (say, *y*) get the value of another variable (say, *x*)? For example, suppose *x* and *y* are variables of type *int*, a value type whose mathematical model is a mathematical integer, as discussed in Section 3.1. Obviously, you use an assignment statement:

```
y = x;
```

What if *x* and *y* are variables of a value type *VT*, where *VT*'s mathematical model is relatively complex and its representation data structure is probably large? Suppose, for example, that *VT* is *SetOfInt*, whose mathematical model is a mathematical set of mathematical integers, as discussed in Section 3.1.1. There are now two options for data movement, neither of which is especially attractive:

1. Consider the assignment operator for *SetOfInt* to perform deep copy, so that after the assignment statement we can think of both *x* and *y* as having the same abstract value. Logically, *x* and *y* must behave independently, too, so changes to *x* do not side-effect the value of *y* and vice versa. This can be terribly inefficient, because without using fancy data-structure-specific tricks that frequently do not apply, the assignment operator must take time linear in the size of *x*'s representation. Big sets simply take a long time to copy and hence to assign.

2. Do not view *x* and *y* as value variables, but as reference variables; i.e., change their type from value type *VT* to reference type *RT*, and think of *x* and *y* as references to objects whose values are sets of integers. This fixes the efficiency problem but at the cost of a distressing non-uniformity in reasoning about program behavior: Some variables denote values and others denote references. It also means that the assignment operator creates aliases, which complicates formal specification and reasoning about program behavior, as explained in Section 4.2.

Approach #2 has been codified into most modern languages, notably Java. It is actually far worse than #1 from certain software engineering standpoints. One reason is that the programmer now must be aware that variables of some types have ordinary values while variables of other types hold references to objects (it's the objects that have the values). For template components this creates a special problem. Inside a component that is parameterized by a type *Item*, there is no way to know *before instantiation time* whether an assignment of one *Item* to another will assign a value or a reference. Of course, this can be "fixed" as it is in Java, by introducing otherwise-redundant reference types such as *Integer* to wrap value types such as *int*. Actual template parameters can then be limited to reference types. This is really ugly, though. And there is still the issue of the complication caused by references for specification and verification, as seen in Sections 3 and 4.

Figure 1 summarizes the data movement dilemma faced by someone who wants efficient software about whose behavior it is easy to reason. The conclusion is that this is only attainable

by sticking to built-in value types—not incidentally, the only types available when the assignment operator was introduced into programming languages—or, at best, by inventing only new user-defined types that admit "small" representations.

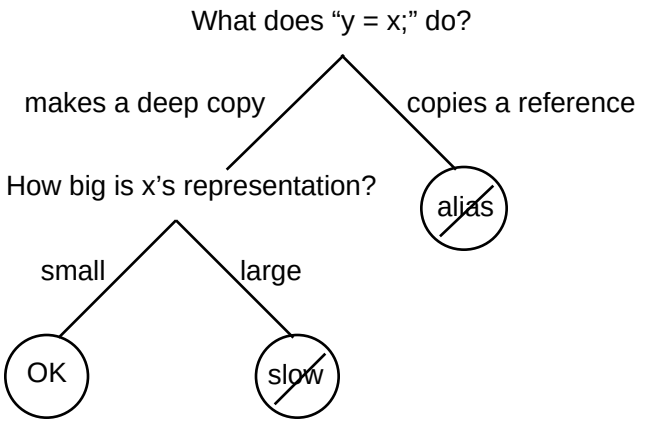

Figure 1: The Data Movement Dilemma

Again, how do you make some variable (say, *y*) get the value of another variable (say, *x*)? There is no inherent requirement that the value of *x* must not change as a result of the data movement process. Realizing this opens the door to other possibilities. The new value of *y* must be the old value of *x*, but the new value of *x* might be:

- the old value of *x* (to get this behavior we use assignment, which works well if *x*'s representation is small); or
- undefined; or
- a defined, but arbitrary and unknown, value of its type; or
- some particular value of its type, e.g., an initial value; or
- the old value of *y*.

It is beyond the scope of this appendix to analyze the pros and cons of all the possibilities beyond the first one, which is unsatisfactory as a general approach to data movement. Suffice to say that leaving *x* undefined complicates reasoning, although not nearly as much as allowing aliasing; and that leaving *x* with either an arbitrary or a distinguished value of its type is actually quite a reasonable thing to do. However, the last approach— *swapping* the values of *x* and *y*—is both efficient and safe with respect to modular reasoning, and it results in remarkably few changes to how most programmers write imperative code [9, 33].

You need to get used to a few new idioms when adopting the swapping paradigm, e.g., for iterating through a collection [32]. The biggest effect of the swapping paradigm, however, is on the design of component interfaces. Consider, for example, a *Set* component (parameterized by the *Item* type it contains) with operations *add*, *remove*, etc. What should *add*(*x*) do to the value of *x*? The analysis of this question parallels the analysis of the data movement dilemma as the question was phrased above. The conclusion is that *add* should *consume x*, i.e., it should leave *x* with an initial value of its type.

How can this be accomplished? A direct implementation of the *Set* component declares a new variable of the parametric type *Item* in the body of *add*, e.g., the data field in a new node that is to be inserted in a linked list of nodes. This variable is then swapped with *x*. Swapping simultaneously puts the old value of *x* into the *Set*'s representation data structure, where it needs to be; and sets the new value of *x* to the initial value for its type that was originally in the data field of the node.

What if there are no pointers in the language, though? In an implementation of the *Set* component that is layered on top of a provided *List* component, for example, the *add* operation simply inserts *x* at the appropriate place into the *List* that represents the *Set*. If the insertion operation for *List* also is designed using the swapping paradigm, so it consumes its argument just like *add* does, then this call does exactly what is needed.

In other words, in both these situations, the code that you would have written if using assignment for data movement is changed in just one respect: assignment of *x* to its place in the *Set*'s representation is replaced by swapping *x* with its place in the *Set*'s representation.

Our experience is that a family of components such as those in the RESOLVE/C++ component catalog [25] can be designed according to the swapping paradigm to compose in such a way that programming with swapping is substantially similar to programming with assignment statements. But the resulting components offer efficiency and/or reasoning advantages over similar components designed in a traditional fashion.

Let's be clear that we still use the assignment operator with built-in value types. There is nothing wrong with the following statement from either the efficiency or reasoning standpoints, assuming that *x* and *y* are variables of some value type with a "small" representation:

```
y = x;
```

The possibly surprising empirical observation that has been substantiated by commercial application development is that, with swapping, there is rarely a need for such a statement when *x* and *y* have user-defined types. You can have value types and efficiency at the same time.

The main advantages of the swapping paradigm are, then:

- The swapping paradigm is easy for imperative-language programmers to learn and apply.
- All types are value types, which allows for understanding of specifications and modular reasoning that are complicated significantly if reference types creep in.
- All pointers and references can be hidden deep within the bowels of a few low-level components and remain invisible to a client programmer layering new code on top of them.
- If these low-level components have no storage leaks, then client programs have no storage leaks, and client programmers do not have to worry about where to invoke **delete** in, e.g., C++, because they simply *never* invoke it. In the case of a garbage-collected language, e.g., Java, there is no need for the complications of general garbage collection because there are no aliases and all collection takes place at predictable times.

Other questions often asked about the interactions between the swapping paradigm and other programming language and software engineering issues, such as the role of function operations, assignment of function results to variables, parameter passing, etc., are discussed in [9].