

Efficient Reusable Components with Value Semantics

Gregory W. Kulczycki
(Third Year PhD Student)

Department of Computer Science
Clemson University
Clemson, SC, USA
gregwk@cs.clemson.edu
<http://www.cs.clemson.edu/~resolve>

1 Introduction

Nearly 30 years after Tony Hoare criticized the introduction of pointers into high-level languages [3], our software systems (and components) are riddled with complex reference semantics. Formal methods advocates see references—and aliasing in particular—as the primary obstacle to modular verification [4], and typical examples of formal specifications unrealistically assume value semantics [10]. Many software engineers and educators recognize the benefit of value semantics for component construction and computer science education [7, 5]. On the other hand, practicing programmers view things differently. They see references as a necessary—if sometimes confusing—tool for implementing efficient data structures. And they point to difficulties of developing simple specifications for simple software components (such as those in Java or STL) as evidence that formal methods cannot be applied to real world software.

This debate raises a fundamental design dilemma for reusable software components: If components do not have accurate and understandable behavioral specifications, their potential for black-box reuse is limited; if components are not efficient, programmers will continue to build and use custom components. The contribution of this paper is in helping to bridge the gap between understandable specifications and efficient implementations. It proposes a multifaceted solution to the reusable component design dilemma.¹

2 Problem

This paper illustrates the reusable component design dilemma with a simple, specific example, but the same issues exist in developing any reusable component in any language. Consider a *SymbolTable* class whose interface is similar to (but slightly different from) the Java *Map* interface.² Informally, a *SymbolTable* is

¹ This research is funded in part by NSF grant CCR-0113181.

² see <http://java.sun.com/products/jdk/1.2/docs/api/java/util/Map.html>

a data type that maps *Keys* to *Values*. The informal description of the *put* operation is given below.

```
public void put(Object key, Object value);
```

Associates the specified value with the specified key in this symbol table. If the table previously contained a mapping for this key, an exception is thrown.

2.1 A Naive Symbol Table Specification

The symbol table concept has been formally specified by Wing in [10] using various specification languages. We reference the Z specification of the type and the *put* operation here.

$$\text{SymbolTable} = \text{Key} \mapsto \text{Value}$$

$\begin{array}{l} \textit{put} \\ \hline st, st' : \textit{SymbolTable} \\ k : \textit{Key} \\ v : \textit{Value} \\ \hline k \notin \text{dom } st \\ st' = st \cup \{k \mapsto v\} \end{array}$

The *SymbolTable* data type is formally described as a partial map from *Keys* to *Values*. Variables affected by the *put* operation are declared above the central dividing line of the Z schema. Unprimed variables indicate values at the beginning of the operation, while primed variables indicate values at the end. The assertions below the line specify relationships among the variables. The line $k \notin \text{dom } st$ is a precondition requiring that the key k not be in the domain of the symbol table st . The line $st' = st \cup \{k \mapsto v\}$ is a postcondition ensuring that the new symbol table contains all the mappings in the old symbol table in addition to a new mapping from k to v .

At first glance, it may appear that the Z specification reasonably captures the behavior of the *put* operation for the Java *SymbolTable* class. Unfortunately, it fails to capture important reference behavior. Consider the following client code.

```
SymbolTable st = new SymbolTable();
Image tomImg = new Image("tommy.gif");
st.put("tommy", tomImg);
tomImg.mirror();
```

Assume that the operation *mirror* modifies an image so that it becomes a mirror image of what it used to be. If the original image in "tommy.gif" is TOMMY, what is the value of the symbol table st after this code fragment has executed? According to the specification the value of st will be $\{\text{"tommy"} \mapsto$

$\boxed{\text{TOMMY}}$. But if the code is run in Java, the value of st will be {"tommy" \mapsto $\boxed{\text{YMMOT}}$ }. In Java, when the object $tomImg$ is passed to the put procedure, an alias of the object is created, and the symbol table st retains the alias. Then when the underlying value of $tomImg$ is updated, the underlying value of st is updated as well.

2.2 Accounting for Reference Behavior

An accurate specification of $SymbolTable$ and its put operation must model the $SymbolTable$ to reflect the fact that the objects it stores are references rather than values. The following specification meets this condition.

$$\begin{array}{l}
 \text{Pointer} : \mathbb{PN} \\
 \hline
 \text{GlobalReferenceMappings} \\
 \text{last} : \text{Pointer} \\
 \text{contents}_K : \text{Pointer} \leftrightarrow \text{Key} \\
 \text{contents}_V : \text{Pointer} \leftrightarrow \text{Value} \\
 \hline
 \text{SymbolTable} = \text{Pointer} \leftrightarrow \text{Pointer} \\
 \hline
 \text{put} \\
 \Delta \text{GlobalReferenceMappings} \\
 \text{st}, \text{st}' : \text{SymbolTable} \\
 \text{k}, \text{v} : \text{Pointer} \\
 \hline
 \forall j : \text{dom st} \bullet \text{contents}_K(\text{k}) \neq \text{contents}_K(j) \\
 \text{contents}'_K = \text{contents}_K \cup \{\text{last} + 1 \mapsto \text{contents}_K(\text{k})\} \\
 \text{contents}'_V = \text{contents}_V \cup \{\text{last} + 2 \mapsto \text{contents}_V(\text{v})\} \\
 \text{last}' = \text{last} + 2 \\
 \text{st}' = \text{st} \cup \{\text{last} + 1 \mapsto \text{last} + 2\} \\
 \hline
 \end{array}$$

This revised specification contains two $GlobalReferenceMappings$ function variables, which represent pointers to the actual data types Key and $Value$. The new variables and new model allow the specifier of the put operation to write pre and postconditions that capture the reference behavior that occurs during the operation. The precondition $\forall j : \text{dom st} \bullet \text{contents}_K(\text{k}) \neq \text{contents}_K(j)$ requires that no pointers to keys in st are aliased to k . The next three lines reflect the changes made to variables in $GlobalReferenceMappings$. When pointer variables k and v are passed into the procedure, two new pointers are created as aliases to k and v (the specification accounts for this by updating the function variables contents_K and contents_V). The variable $last$ maintains the last pointer (an integer) that has been used, which ensures that each new reference will be modeled by a unique integer. Finally, the last line in the put schema ensures that the symbol table st will be updated by adding a mapping from the alias of k to the alias of v .

Not only is this specification significantly more complex than the first, but the specification for the operation put updates data external to the component.

This accurately reflects the behavior of the implementation, but it leaves little hope for modular reasoning and verification. A practical, scalable approach that permits both understandable specifications and efficient implementations is the focus of my dissertation.

3 Solution

In the proposed approach, reusable component design delegates the complexity of reference behavior to low-level components and allows programmers to use value semantics for the vast majority of components in a system. Efficient mechanisms are used for data movement and parameter passing that do not introduce aliasing. As validation of this approach, the author is designing a compiler to analyze a collection of reusable components developed in a fully integrated programming and specification language that reflects the principles outlined below.

3.1 Efficiency and Value Semantics

A successful solution to the reusable component design dilemma must make the distinction between references and values disappear at the specification level, so that reasoning and specification are simplified. However, the solution must also permit references in implementations to make them efficient. The key to making this all happen lies in avoiding unnecessary aliasing—something we were unable to do in the *put* operation above. Specifically, languages need mechanisms for data movement and parameter passing that are efficient yet avoid aliasing. An example of such a mechanism appears below. Suppose that two variables x and y are implemented as references, as in the **Before** column of Table 1. The table illustrates three different options for ensuring that x will obtain the value of y . The first option is Java assignment, which copies references. This option makes x an alias for y , which is efficient but complicates reasoning. The second option is an assignment that makes a deep copy. This option avoids aliasing, but can be very inefficient if the representation of y is large. The third option swaps the values of x and y . It is efficient and it avoids aliasing. The argument for swapping as the primary means of data movement has been made in [2].

Table 1. Objective: transfer y 's value to x .

Approach	Before	After
$x = y$ (Java)		
$x := y$ (deep copy)		
$x :=: y$ (swap)		

The problem with parameter passing is similar to the problem with data movement, though some consider it more serious since aliases can be introduced even in languages without pointers (by repeating arguments as parameters). Fortunately, the swapping paradigm works here as well, and in most cases, the effects of call-by-swapping are indistinguishable from call-by-reference [9]. The full implications of swapping and other alternative mechanisms for data movement are not discussed here due to space constraints.

3.2 Making References Rare: Component Design Strategies

A proper component library can reduce the need for references and simplify software construction. The library should adhere to the principle that all components are created equal. In particular, types that are built in to the language should be provided by components that are constructed according to the same semantic framework as custom components, though syntactic sugar may be provided for built-in components as a convenience to users. The Java programming language violates this principle by offering built-in types with value semantics, while custom types have reference semantics. A consistent view of component types simplifies reasoning for the programmer.

Given a sufficiently rich component library, the need for indirection should be rare. The functionality of data structures that are commonly implemented with references should be provided by pre-existing components in the library—pointer-based implementations of lists and trees should be standard parts of the library. This requires that performance specifications are provided with implementations just as behavioral specifications are provided with interfaces. As an example, the Java component library provides a linked-list class whose performance specifications are given informally, so Java programmers no longer need worry about building a custom linked-list.

Though reference use can be drastically reduced by proper design, it is occasionally necessary. Reference behavior should be provided by a small number of library components that offer the same performance benefits of pointers but do not hide their inherent complexity (a language should *not* provide syntactic sugar for references). There are benefits to offering more than one component to capture common pointer functionality. For example, a referencing component that did not allow cyclic pointer structures could be implemented using reference counts, so that objects could be deallocated when their count dropped to zero. Such a component would be used for one-way list and tree implementations. Components such as bidirectional lists and graphs would require a more general referencing component that would not enjoy the performance advantages of the acyclic one.

The approach described above is meant to bridge the gap between specifiers and programmers, so that understandable specifications can coexist with efficient implementations of software systems and components. To completely close this gap, specification languages and programming languages should be integrated into one system, so that modular verification can become practical. The cement for this process lies in the programming and mathematical type systems of the

integrated language, which must complement one another to be effective. The author intends to validate the ambitious goals addressed in this paper with a compiler for the integrated RESOLVE language, which adheres to the principles outlined above.

4 Related Work and Conclusions

Since the introduction of references into high-level languages, there have been warnings against them. Hoare derides them in [3], Stepanov advocates value semantics for component libraries in his preface to [7], and Koenig argues for a rethinking of how C++ is taught in [5] based partly on the difficulty students have with pointers. Aliasing is identified as the primary problem with references in [4]. Though attempts have been made to accurately model programs written in popular programming languages with specification languages such as Larch [1] and JML [6], the specifications are necessarily complex. An approach to reasoning about software using RESOLVE techniques is given in [8], and the swapping paradigm is discussed in [2, 9]. The best hope for reusable components and modular reasoning involves the principles of component design outlined above with efficient data movement mechanisms that preserve value semantics.

References

1. J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, 1993.
2. D. E. Harms and B. W. Weide. Copying and swapping: Influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424–435, May 1991.
3. C. A. R. Hoare. Hints on programming language design. In C. A. R. Hoare and C. B. Jones, editors, *Essays in Computing Science*. Prentice Hall, New York, 1989.
4. J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
5. A. Koenig and B. Moo. Teaching standard C++. *JOOP*, 11(7):11–17, 1998.
6. G. T. Leavens, A. A. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12. Kluwer, 1999.
7. D. R. Musser, G. J. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Boston, 2nd edition, 2001.
8. M. Sitaraman, S. Atkinson, G. Kulczycki, B. W. Weide, T. J. Long, P. Bucci, W. Heym, S. Pike, and J. E. Hollingsworth. Reasoning about software-component behavior. In *Procs. Sixth Int. Conf. on Software Reuse*, pages 266–283. Springer-Verlag, 2000.
9. M. Sitaraman, G. W. Kulczycki, W. F. Ogden, B. W. Weide, and G. T. Leavens. Understanding and minimizing the impact of reference-value distinction on software engineering. Technical report, Department of Computer Science, Clemson University, 2002.
10. J. M. Wing. A specifier’s introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.