Copying and Swapping: Influences on the Design of Reusable Software Components

Douglas E. Harms, *Member, IEEE*, and Bruce W. Weide, *Member, IEEE*

***Abstract*—The only built-in mechanism for data movement in most modern programming languages is the assignment statement for copying the value of one variable to another. In the context of reusable software components, this reliance on copying leads to two classes of difficulties: components whose implementations are inherently inefficient, and client programs that are hard to reason about. A powerful substitute for copying as the primary data movement primitive is a *swapping* mechanism that exchanges the values of two variables. Reusable components and client programs designed with a "swapping style" of programming have many advantages over designs based on the traditional "copying style," including improved execution efficiency, higher reliability, and enhanced reusability.**

***Index Terms*—Abstract data type, assignment statement, formal specification, object-oriented programming, parameter passing, pointer, reusable software component.**

I. Introduction

ONE of the foundations of most modern programming languages is the ability to make copies of data. Examples of the implicit use of copying are the ordinary assignment statement and call-by-value and call-by-value-result parameter passing. In fact, the "copying style" is so ingrained in our languages and normal programming style that it is sometimes difficult even to see it, let alone to avoid it. Consider, for example, that assignment statements pervade the examples in computer science textbooks. Each such statement describes copying.

A. Copying Versus Swapping

What is the copying style? It is designing and programming under the assumption that copying, especially in the form of the assignment statement, is a reasonable data movement primitive for *any type* of data—not just for built-in types such as integers, but also for complex user-defined types.

This paper is specifically concerned with the impact of the copying style of programming on the design of generic reusable software components. Not surprisingly, published designs and commercial implementations for such components (as in [1], [2], [14]) generally are based on the copying style. A typical example of a reusable component is a generic module that provides a type together with operations to manipulate variables of that type; e.g., a list module. Clients using this component may want to copy lists (e.g., by assigning one list variable to another). Moreover, implementers of the component may want to copy list elements (e.g., by assigning one variable of the list element type to another). In fact, because of the way the functional behavior of such a module is usually designed, such copying may be unavoidable if the module is to be used effectively and/or implemented correctly.

Here we argue that a simple alternative to copying as a data movement primitive—swapping (exchanging) the values of two variables—has potentially significant advantages in the context of the design of generic reusable software components. Specifically, we claim that generic module designs based on a "swapping style" are superior to designs based on copying, both in terms of execution-time efficiency and with respect to the likelihood of correctness of client programs and module implementations. Furthermore, designs based on swapping are more reusable, in an important sense, than traditional designs.

B. The Case for Swapping

The case that the swapping style leads to improved reusable component designs rests on two fundamental arguments. One is based on a simple analysis of program efficiency. The other relies on an understanding of the purpose of abstraction in program design. Specifically:

1) It is important that operations exported by a generic module should execute efficiently, regardless of what parameter type is substituted when the reusable component is instantiated. For example, inserting an integer into list of integers should be efficient. So should inserting a stack of integers into a list of stacks of integers. Traditional designs based on the copying style permit the former but not the latter. Designs based on the swapping style are efficient in every case.
2) It is important to be able to reason both formally and informally—rigorously in either case—about the behavior of implementations of reusable components and about the client programs that use them. Considering variables to stand for abstract values, independent of their underlying representations, is crucial to this reasoning ability. Attempts to address the efficiency issue 1) in the copying style of programming lead to copying of pointers to representation data structures, to replace copying the abstract values those structures represent. This shortcut compromises abstract reasoning.

Manuscript received June 2, 1989; revised January 2, 1991. Recommended by T. Ichikawa. This work was supported in part by the National Science Foundation under Grant CCR-8802312.

D. E. Harms is with the Department of Mathematics and Computer Science, Muskingum College, New Concord, OH 43762.

B. W. Weide is with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210.

IEEE Log Number 9143150.

The swapping style solves the efficiency problem in a way that preserves the reasoning power afforded by abstraction.

The main body of the paper contains specific arguments and examples to support these positions. Section II sketches the framework for our arguments about abstraction and reasoning, and defines several key terms. Section III discusses the problems inherent with the copying style of programming. Section IV proposes swapping as an alternative to copying as a data movement primitive and uses it to explain a swap statement and a function assignment statement, as well as parameter passing. Section V follows with a simple but illustrative example of generic reusable component design in the swapping style and investigates comparative advantages vis à vis the copying style. Section VI relates the ideas to work of other researchers, and Section VII summarizes our conclusions and recommendations.

C. Design Guidelines and Programming Language Issues

The suggestions developed here for how generic reusable software components should be designed may be considered guidelines that could be followed in nearly any programming language. However, because the influence of programming language features on reusable component design seems so strong, we also discuss how minor changes to existing algorithmic languages could discourage the copying style and encourage the swapping style with relatively little impact on the way most people program. This argument is not the main theme of the paper, but it suggests that copying is inappropriate as a built-in and usually implicit operation in programming languages that otherwise encourage software reuse. A swap primitive would be a better choice.

II. Framework and Terminology

The primary objective of this work is to encourage more effective design and implementation of reusable software components. We therefore assume a language that supports *modules* which encapsulate and export hidden types along with procedures and functions to manipulate variables declared to be of those types. A *hidden type* is a type whose concrete representation is hidden from the client program that uses it. Modules are typically *generic* in that they have one or more module parameters that specialize their provided types; these parameters are usually other hidden types. A client program that uses a module *instantiates* it, statically, by providing actual values for module parameters. As an example, the StackTemplate module to be discussed in Section III provides a hidden type called "stack." It has a type parameter "item" which is the type of items in a stack. When this module is instantiated the actual type of the items must be supplied. All types are known at compile time and type checking is static. A prototypical language having these properties is Ada [4].

The purpose of the remainder of this section is to lay the groundwork for treating abstraction in conjunction with types in this kind of language. The connection between types and abstraction is important in order to understand the significance of swapping.

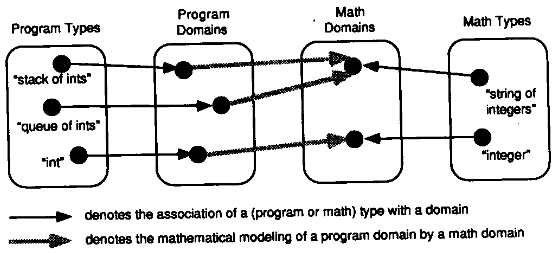

Program Types
Program Domains
Math Domains
Math Types
"stack of ints"
"queue of ints"
"int"
"string of integers"
"integer"
denotes the association of a (program or math) type with a domain
denotes the mathematical modeling of a program domain by a math domain

Fig. 1. Program and mathematical types.

A. Types and Domains

There is hardly a consensus in the computer science literature about the meaning of the word "type" [3]. We take a simple but effective approach that distinguishes between program types and mathematical types, and consequently between program variables and mathematical variables. The explanation that follows is left at an intuitive level, because the technical details [7] are not necessary for understanding the rest of this paper. But the ideas here are crucial. Later arguments may seem either cryptic or wrong to a reader who clings to other definitions, such as the notions of type that are often used in the object-oriented programming community.

A (program or mathematical) *type identifier*, or simply *type*, is a character string. Each type names a collection of values called a *domain*. Domains and the values they contain are purely abstract and can be considered to exist independently of the programs that name them with types. On the other hand, a type is a local name in a separately compilable program unit.

Fig. 1, for example, applies in the context of some particular program unit. In it, "integer" is a (mathematical) type that names the domain defined by axioms of integer theory, while "string of integers" is a (mathematical) type whose associated domain is defined by axioms of mathematical string theory—which is a generic theory in the sense that it is parametrized by another (mathematical) type. Similarly, the (program) types "int" and "stack of ints" and "queue of ints" name domains defined by some external program specifications. The solid thin arrows from type names to domains in Fig. 1 denote the type-to-domain mappings that are set up as the result of type declarations in the program unit being considered.[1]

A (program or mathematical) *variable identifier*, or simply *variable*, is a character string that stands for some value. Every variable has a corresponding type, and a variable's value is always from the domain of the variable's type. By these definitions, types are to domains as variables are to values. For instance, if "x" is a (mathematical) variable of the (mathematical) type "integer," then "x" stands for some value in the domain defined by integer theory. Conventionally, we write this as "x : integer."

[1]In principle, the type "integer" could be associated with a domain other than what we usually consider to be the mathematical integers. But doing so would needlessly complicate the discussion and repudiate long-standing mathematical tradition. Here, we are not concerned with the language used to define domains nor with how the type-domain association is established.

Of course, type is a useful concept even without programs. Program specifiers [23] often say things about variables of mathematical types by writing assertions in a formal language, such as the first-order predicate calculus with equality, augmented with function and predicate symbols appropriate for the theory that defines the type's domain. For instance:

```
∀ x : integer, x + 1 > x
```

says that the result of adding one to any integer is larger than that integer. (The 0-ary function symbol "1" and the common infix notations for the binary function "+" and the binary predicate ">" are used for readability.)

What role do types play here? They simply provide a method for determining whether an assertion is meaningful. The + is defined to be of the syntactic form "integer $\times$ integer $\rightarrow$ integer." Similarly, < is defined as a predicate on a pair of integers. If the arguments to + and < are of the wrong types, we do not have to worry about the meaning of the assertion or whether it is true or false; it is simply nonsense because it is structurally incorrect. In this view, the whole reason for having types and not simply domains (sets of values) is to permit type-checking. If an argument of type T_1 is expected somewhere that an argument of type T_2 is used, type-checking succeeds if T_1 and T_2 name the same mathematical domain.

As with their mathematical counterparts, program types are used so the structure of program statements can be checked against expected usage. If a procedure expects arguments of particular types, then it is easy for a compiler to check statically that it is called appropriately. Once again, type-checking is by a simple rule: two program types are equivalent if they name the same program domain.

Without distinguishing types from domains we would be left with two choices: do without types altogether and try to define the meanings of structurally erroneous assertions and programs, or try to define equivalence between domains. The former approach needlessly complicates the logic in mathematics and leads to an easily avoidable class of run-time errors in programs. The latter approach has serious problems dealing with isomorphic domains. Separating types from domains simplifies the situation considerably.

B. Mathematical Modeling and Abstraction

The above view of types is indeed simple compared to some treatments [3]. But it is valuable because it sheds considerable light on a problem that is critical to the remainder of this paper. We argue that the main source of confusion about types in programming arises not from programming itself, but from attempts to *reason* about programs, either formally or informally. The natural tendency is to mix up program and mathematical types or even to suppose they are the same thing. The use of algebraic specification techniques to define program types or their domains has contributed to this confusion by implicitly treating program variables as mathematical variables, and by treating "axioms" involving program functions as though they defined a mathematical theory. Some programming language and reusable component designers have recognized this problem [12], [14], [16], but many others have not.

Reasoning about the behavior of programs is simplified by establishing a mathematical domain as an explicit *mathematical model* of each program domain. This modeling is expressed as a mapping (shown in the middle of Fig. 1) that defines a precise correspondence between any program value and the abstract mathematical value of its model. For instance, a program variable of type "int" may be modeled as a mathematical variable of type "integer" and, as noted in Section III, a program variable of type "stack of ints" may be modeled as a mathematical variable of type "string of integers."

Note that it is essential for different program domains to be able to have the same mathematical model. Fig. 1 shows both queues and stacks modeled as mathematical strings. Because the program types "queue of ints" and "stack of ints" do not name the same program domain, a compiler can flag a type mismatch if a programmer attempts to, say, push an "int" onto a "queue of ints." But that programmer is still free to reason about the mathematical models of stacks and queues as though they were strings, without fear of a mathematical type mismatch. For example, a correct program with no type mismatches might have a loop invariant stating that the concatenation of a variable of type "stack of ints" and another of type "queue of ints" is equal to a third of type "stack of ints." There is no mathematical type mismatch if this assertion is treated as a relationship among the *models* of the program variables, which are all of mathematical type "string of integers."

To summarize, we may—indeed we must—reason about the values of program variables as though they were mathematical variables. Only a mathematical model can ever be of interest to a client programmer because a program variable's concrete representation is hidden. For example, a variable of type "int" in a client program must be treated as a symbol that stands for a mathematical value of type "integer." It must never be thought of as a sequence of 32 bits or whatever the concrete representation of an "int," nor as a memory location that contains the concrete representation. Such low-level views would flagrantly violate information hiding and abstraction.

Note also how the use of explicit mathematical modeling helps illuminate the subtle distinction between information hiding and abstraction. *Information hiding* is a technique whereby concrete representations used within a module are kept secret from a client. Most modern languages have constructs to support information hiding. *Abstraction* is a technique used to explain to a client what must be revealed so he/she knows how to use a module; i.e., the mathematical model needed to reason about client programs. Most modern languages have no constructs to support this sense of abstraction because they lack the machinery to set up mathematical models as the basis for explaining program behavior. Section III postulates a language with formal specification constructs in order to better explain the benefits of swapping.

C. Programming Language Issues

Many modern languages generally support the framework outlined above. Invariably, they also carry extra baggage yet lack certain important features. Therefore, while continuing to relate the ideas that follow to existing languages, we hereby imagine ourselves to be designing for and programming in a simple hypothetical language with the following features:

- There are no built-in program types. Even such primitive types as "int" and "char" must be provided by module instances. In most languages these types are built-in. The hypothetical language is similar to Alphard [16] in this respect.
- There is a program unit in which behavioral specification of a module is provided separately from the code for its realizations (implementations). Something like the notation in [12] or [16] would be suitable for this purpose. Here we present examples in a simple dialect of RESOLVE, a language we have developed for defining, building, and incorporating generic reusable software components [7], [8], [15], [17], [22].
- A single client program may create more than one instance of the same generic module. It may also use multiple instances of the same abstract module that have different implementations; e.g., for performance reasons.

The proposed language features and design decisions discussed here have been strongly influenced by the overall goal of creating *efficient* and *verifiable* generic modules, the achievement of which would be an important technical contribution leading to more widespread acceptance of reusable software components in existing languages. But there is no reason the ideas cannot be applied to virtually any language in which reusable software components can be built and used, including object-oriented languages.

III. Problems with Copying

Consider an assignment statement "x := y," where x and y are variables of the same type.[2] The meaning of this statement is that, after it it executed, the variables x and y (treated now as mathematical variables) have the same abstract mathematical value, and furthermore that the value of y has not changed. In implementation, this usually means making a copy of the concrete representation of y and placing it in x. Parameter passing by value and by value-result also imply copying in the same sense.

Although the ability to make a copy of a variable's value is occasionally necessary, reliance on it to the point that copying is implicit—and in some languages even essential in order to program at all—is unwarranted. There are three major problems with copying:

- Physically copying a large representation data structure is expensive in execution time.
- Copying a pointer is not tantamount to copying the data structure it points to; it compromises abstraction and therefore complicates reasoning about program behavior and often results in programs that are incorrect and difficult to debug.
- The copying style leads to designs for generic modules that inherently have no efficient implementations.

[2]From now on, we usually do not distinguish explicitly between mathematical and program types and variables when we use the words "type" and "variable," as the meaning is usually clear from the context. We also do not enclose type identifiers in quotation marks, for the same reason.

These points are explained in the following subsections.

A. Physically Copying a Large Data Structure Is Expensive

Inefficiency in copying large data structures is especially important in the kind of programming considered here. Languages that support generic reusable components providing hidden types encourage the development of modules that encapsulate just such complex data structures as the concrete representations of variables' values. The time required to make a physical copy of a data structure is generally linear in the size of the structure. This means copying simple values, such as ints or chars, is fast, whereas copying records, arrays, or linked structures is potentially quite expensive in terms of execution time.

If a client programmer knows code is copying large data structures—even though reusable components are not supposed to reveal anything about the representations of hidden types, including their size—and if there is no alternative, then the client may be willing to pay the price. But as suggested above for the cases of assignment statements and some kinds of parameter passing, copying is often implicit.

B. Copying a Pointer Does Not Replace Copying the Data Structure

One might be content with this inefficiency in principle, but the designs of modern programming languages suggest it is impossible to ignore it in practice. Most languages and most programmers attempt to sidestep the inefficiency of copying large data structures by copying pointers to them. These pointers may appear explicitly through pointer types or implicitly through call-by-reference parameter passing to augment or replace call-by-value and/or call-by-value-result. This approach still relies on copying. The apparent advantage is that small data items (i.e., pointers) are being copied in place of the large data structures they point to.

Although copying pointers may reduce the amount of work that must be done, it also introduces a serious problem: it compromises abstraction, which complicates reasoning about the behavior of the client program. Copying a pointer to a variable's concrete representation does not produce the same effect in the abstract mathematical model view as copying the data structure it points to. The immediate effect of an assignment is the same as before. Specifically, the assertion "x = y" is true just after the assignment statement "x := y," because the concrete representations of x and y are identical, so their abstract values must be equal. But the aliasing introduced by copying the pointer means that a subsequent change to either x or y will affect the abstract value of the other.

This effect significantly complicates reasoning about the program's behavior. Ordinary abstract mathematical variables are independent. Writing an assertion that does not mention a variable gives no information about that variable. But the mathematics of programs with aliasing is quite different. Now, in reasoning about the effect of any statement, we must account for the possibility that other variables—ones not mentioned in the current statement or even the current procedure—may have their abstract values changed as a side-effect of executing the statement. Everyday practical difficulties arising from this include obscure program bugs and wildly unexpected behavior that most programmers have experienced, much to their dismay. It has been noted [10] that "the fundamental sin in the use of pointers is to make them assignable."

Despite this general condemnation, aliasing itself is not necessarily all bad. The problem stems from *visible* aliasing, where the effects of the alias are observable in the future behavior of the client program. If aliasing can be carefully controlled so a client program cannot tell there is aliasing, the objection does not apply. This point is also made in [13], [21], among other places. But visible aliasing has long been viewed as a source of serious difficulties for programmers. For instance, Hoare [9] discusses several problems associated with assignment of pointers and notes that "their [pointers] introduction into high-level languages has been a step backward from which we may never recover." We hope this conclusion was too pessimistic, but there is little doubt that careless copying of pointers is a dangerous practice.

There is a further difficulty with assignment statements in some languages: It is not always clear just what they will do. For example, consider an Ada package that defines a private (not limited private) type T. Suppose a client declares two variables x and y of type T. What is the effect of "x := y"? Because the representations of x and y are hidden, the client programmer does not know whether the assignment will copy the data structure representing y or just a pointer to that data structure. The former occurs if T is declared as a record, say, and the latter if T is a pointer to that record. Either kind of copying causes problems—the first is expensive in execution time, and the second complicates reasoning about program correctness. But the client programmer cannot tell which problem will occur, because the declaration of the representation is in the private part of the package specification and is not (supposed to be) seen by the client.

If copying is going to be done at all, then, it must have a consistent meaning. For the remainder of this paper we assume the overriding consideration is the ability to reason about program behavior, especially correctness. What good is a program that gets the wrong answer quickly? Program variables must be treated as mathematical variables from the standpoint of reasoning about programs. This generally rules out simply copying a pointer to the representation data structure as a correct implementation of copying a variable's value. If a variable's value is to be copied, the implementation must respect the rule that the copying cannot introduce aliasing that would later result in two variables' abstract values becoming implicitly linked.

C. Reusable Component Designs Based on Copying are Inherently Inefficient

There is at least one more problem with copying. Because copying is second nature, most existing reusable software components have been designed in such a way that they cannot possibly be implemented as efficiently as they could have been with a slightly different design. As an illustration, consider the following specification for a generic module that provides a hidden type "stack." The mathematical model for a stack is a mathematical string of elements of (the mathematical model of) the parameter type "item." The notation is from mathematical string theory. The symbol "Λ" denotes the empty string; "s ∘ x" is the string obtained by extending string s with item x. The precondition for an operation, if any, is written in a **requires** clause, where each variable in the assertion stands for the abstract mathematical model of the corresponding program variable at the time the operation is invoked. The postcondition is written in an **ensures** clause. In the postcondition a variable name with a prefixed "#" stands for the abstract mathematical model of that variable at the time the operation is called, while the variable name by itself signifies the value of the variable's model when the operation completes.

In the specification that follows, the top of the stack is viewed as the last (rightmost) item in the string that models it. The specification includes no *a priori* limit on how large a stack can get, because this "ideal" stack module illustrates the points of the paper without the need for the complication of a size limit. Close inspection of the design reveals that procedure "push (s, x)" does not change x, and that functions "top (s)" and "isempty (s)" do not change s.

```
module StackTemplate (type item)
    provides
        type stack is modeled by
            string (item) = Λ -- initially
        procedure push (s : stack, x : item)
            ensures s = #s ∘ x and x = #x
        procedure pop (s : stack)
            requires s ≠ Λ
            ensures ∃ x : item, #s = s ∘ x
        function top (s : stack)
            returns x : item
            requires s ≠ Λ
            ensures s = #s and
                (∃ r : string (item),
                s = r ∘ x)
        function isempty (s : stack)
            returns empty : bool
            ensures s = #s and
                (empty iff s = Λ)
end StackTemplate
```

Essentially, this module design, without a formal specification, appears in many modern data structures texts (e.g., [5], [13]), presumably as an example of a good module design. It is no doubt used in hundreds of programs and has been for years, and we certainly cannot credit the above authors with inventing it. The question at hand is simply, what is wrong with the design?

For one thing, there is no reason there should be any restrictions on type item and in fact none is specified. It can be replaced by *any* type, including int, char, array of chars, or even stack of ints. But without knowing anything about this type in the realization of this module, how can "top(s)" be implemented efficiently? Its specification demands that s not be changed and that the top value of stack s be returned as the function's result. This means the realization code must make a copy of the top value of s. Because an item may be a type whose concrete representation is large and inefficient to copy, though, the top function may execute very slowly. Consider as an example the time taken when s is of type stack of lists of ints.

Another problem with the design stems from the semantics of "push (s, x)." It effectively places a copy of x onto stack s, leaving the argument associated with formal parameter x unchanged. Again, because item may be any type, the copy operation for type item may be very expensive. This situation is tolerable if item is restricted to simple types such as int and char, as it is in virtually all textbooks. But if there are no restrictions on type item, then the inherent inefficiency designed into the push procedure is problematical—especially if a more efficient design is possible. The next section introduces a simple alternative design for the StackTemplate that does permit efficient implementation of all the stack operations.

To summarize, there are major problems with applying the copying style to design of generic reusable components. If copying is necessary at all, copying abstract mathematical values of hidden types is desirable because this permits easy reasoning about the behavior of client programs. But if copying an abstract mathematical value is implemented by copying the concrete representation of that value, copying can be costly in execution time (generally linear in the size of the representation data structure). This performance penalty is especially insidious if the copying is implicit, as with some assignment statements or call-by-value parameter passing. Copying pointers to variables' concrete representations is generally the only efficient means of trying to copy abstract values, but copying pointers usually has the nasty side-effect of thwarting the reasoning process by subverting abstraction and by introducing visible aliasing in the client program.

IV. Swapping as an Alternative to Copying

This section proposes a mechanism and minor language changes that would promote a style of programming in which it is common to swap the values of two variables rather than copy one to the other. This proposal has two important characteristics:

- The only primitive for data movement, i.e., the only primitive way to change the value of a variable, is to swap the values of two variables. There is no primitive to copy the value of a variable and assign it to another variable.
- The language does not provide the client programmer with pointers. Implementations of modules providing hidden types may involve pointers, and the language implementation itself certainly may use them, but a client program can be written and understood without knowing about pointers at all.

Swapping is interesting for many reasons, as discussed below. The importance of hiding all pointers is that, by building module implementations as clients of lower-level modules, entire hierarchies of modules can be constructed, none of which explicitly mentions pointers. All variables of all types can be treated in the abstract way outlined earlier. Of course at some very low level some modules will need to be built in another language (or in the "system programming dialect" of the hypothetical language), because some types—those primitive to most languages—will have to be implemented directly on top of the hardware. This low-level code may of course involve explicit use of pointers and even copying of pointers. But if client programs are developed without pointers and without the possibility of visible aliasing, it is much easier to reason about their correctness [10], [11].

The following subsections discuss these language characteristics and justify the claim that reusable component design and programming with the swapping style is a desirable alternative to design and programming using the traditional copying style.

A. The Swap Primitive

All data movement in the hypothetical language is explained in terms of a single primitive called swap (illustrated in Fig. 2), which exchanges the values of two variables of matching types. If the swap primitive were considered a generic procedure, it would be specified as:

```
procedure swap (x : item, y : item)
    ensures x = #y and y = #x
```

Considering swap as a procedure, however, causes a problem. All data movement should be explained in terms of swapping, and a procedure call itself causes data movement: the values of the arguments must be communicated to the formal parameters at the time of the call, and back again when the procedure returns. Section IV-C explains parameter passing in terms of swapping. Swap is therefore considered to be a built-in operation available for every type, not a procedure.

B. The Swap Statement and the Function Assignment Statement

One way to change a variable's value in the hypothetical language is with a swap statement. A swap statement is of the form "x :=: y" where x and y are variables of matching types. It simply interchanges the values of x and y. This is a direct use of the swap primitive.

Although the hypothetical language does not permit variable-to-variable assignment, it does have a *function assignment statement.* This statement takes the form "y := f(· · ·)," where y is a variable and f is a function whose return value is of a matching type. An example from the module in Section III is "y := top (r)" for y of type item and r of type stack of item.

The abstract behavior of a function assignment statement can be explained to a client as follows, using the above example. First, the code for top is started. During its execution there is a local variable x of type item whose value will be the return value of the function; see the first line of the specification. The code for top (it does not matter how) makes x equal to the top item in r. Return from the function causes x to be swapped with y, the target of the function assignment. At this point the original value of y—now in local variable x within function top—is no longer accessible to the client program. Finally, the code for top terminates and execution resumes in the client program.

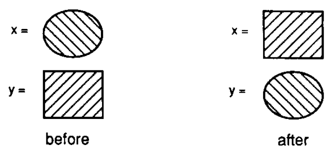


Fig. 2. Programmer's view of swapping.

One use of a function assignment statement achieves the effect of the usual assignment statement "x := y" in other languages. A programmer may simply write "x := replica (y)" where replica has the following specification:

```
function replica (y : item) returns z : item
    ensures y = #y and z = y
```

There are two differences between this way of copying a variable's value and the ordinary assignment statement. One is that copying using the replica function is explicit; the client programmer knows there is some execution-time cost for copying. The other is that copies may be made only for variables for which there is a replica operation. It is not necessary to assume that copying is permitted for every type.

C. Parameter Passing

A third and final way to achieve data movement in the hypothetical language is by passing parameters to procedures and functions. This may cause implicit copying in languages that use call-by-value or call-by-value-result. Call-by-reference, while an efficient *implementation* of parameter passing, must be *explained* in terms of pointers or memory addresses. Such an explanation violates the abstraction needed to hide the representations of program variables from clients. This section shows how the effect of the call-by-reference mechanism can be explained in terms of the swap primitive, thereby preserving abstraction throughout the language definition.

For the moment, suppose a language that has only procedures, no functions. All arguments to procedures must be simple variables because there are no built-in types such as arrays, for example. Now consider the following operational description of the effect of a procedure call. First, all the formal parameters are initialized. Then each argument is swapped with the corresponding formal parameter. Then the body of the procedure is executed until it is ready to return. Finally, each argument is again swapped with the corresponding formal parameter, and execution continues in the calling program.

We term this behavior *call-by-swapping*. Obviously, it differs from call-by-value because it permits arguments to be changed by the call. It differs from call-by-value-result because every parameter is passed in constant time regardless of its type; there is no copying. It differs from call-by-reference because its explanation does not violate abstraction by talking about pointers or memory addresses.

Most importantly, call-by-swapping differs from these other parameter passing mechanisms because it purports to describe only what parameter passing acts like, not how it is implemented. Indeed, it is not important whether swapping actually occurs at the time of the call and the return. All that matters is that the language should implement parameter passing by some method that behaves as the above description suggests. For example, under the following conditions the effect of call-by-swapping is obtained by using call-by-reference as its implementation:

- No variable may be passed as an argument to any procedure if that variable is visible inside the procedure body.
- No two arguments to a procedure call may be the same variable.

Call-by-reference achieves the desired result under these conditions, because the formal parameters do not really need to be initialized and swapped with the arguments at calling time. The argument values are inaccessible to the called procedure except through the formal parameters (and the formals are of course inaccessible to the caller, which is suspended during the call). Similarly, no swapping is really necessary upon return. The temporary aliasing introduced with call-by-reference can cause no difficulty because it is not detectable by the client. Note that, in order to prevent well-known kinds of anomalous behavior due to aliasing that can otherwise result from procedure calls, a language might do well to enforce the above compile-time restrictions in any case.

It is easy to generalize call-by-swapping to handle functions and to deal with calls where the arguments are themselves function invocations [7]. The approach is to use the same ideas as in the definition of the function assignment statement described in Section IV-B. The following again assumes this more general setting.

D. Advantages of the Swap Primitive

The swap primitive has three valuable properties. First, unlike copying, it is implementable uniformly and executable in constant time for any type. To a programmer, each variable has an abstract mathematical value, as depicted in Fig. 2. But it is possible for a language implementation to represent each variable with a pointer to a data structure that represents its abstract value and to keep this pointer invisible to the programmer. Swapping the abstract mathematical values of two variables can be implemented simply by exchanging these two pointers, as shown in Fig. 3. The time required for this is constant—three MOVE instructions in a typical machine language. Neither the code to swap nor the time to execute it is related to the sizes of the data structures that represent the abstract values. Swapping two stacks requires the same code and the same amount of time as swapping two ints.

Of course, a compiler need not represent all types using pointers. The important characteristic of the representation is that all variables occupy just the amount of storage required for a pointer. For instance, if pointers are 32 bits in a particular architecture, then all variables must be represented in 32 bits. Any program type whose concrete representation uses no more storage than that required for a pointer can be represented directly. Others are represented indirectly. The implementation of swapping is simply to exchange the 32-bit words that represent the values, whether those words are pointers to data structures or whether they can be interpreted directly as the abstract value representations.

A second advantage of swapping is that it never introduces aliasing, even though the representations of variables may involve pointers. Recognizing the potentially harmful effects of visible aliasing makes this property particularly attractive.

Third, with swapping and the proposed variable representation it becomes possible to reuse object code for generic modules; i.e., realizations of generic modules need not be recompiled for each module instance. For example, consider a generic module (such as the StackTemplate module discussed in Section III) where type item is a parameter to the module. The object code produced by a compiler for a swap involving two variables of type item within a realization of this module is independent of the actual type used to instantiate it. This is because the object code to swap two variables consists of exchanging two words, and this works regardless of the type involved. With care, the object code for a module can be shared among all instances of it that are instantiated even with different types. The technique is explained more fully in [8].

E. Keeping Pointers Hidden

The hypothetical language does not provide the programmer with pointers. It is important to realize that the use of pointers in the discussion of the swap primitive was to describe a possible language implementation, not a language definition. This use of pointers is completely hidden from a client programmer. In fact, the only reason pointers were even mentioned was to justify the performance claims. The language can be completely defined without them.

In the absence of pointers, many traditionally difficult programming bugs are eliminated. But it might also seem that traditional linked structures, such as linked lists, cannot be implemented without explicit use of pointers. Fortunately, this is not the case [10]. For example, [8], [15], [21] provide a description and formal specification of a module that permits implementation of a large class of traditional linked structures without explicit mention of pointers; [11] provides other examples. Experience has shown that pointers *per se* are not necessary for a language to be useful, even when one of the major uses of the language is to implement and use complex data structures. With careful use of encapsulation and modularity, all pointers can be hidden in the language implementation and in a few low-level reusable modules that form the basis for all higher-level representations, never to be seen by a client programmer.

V. Improved Reusable Component Design

We now return to the question of generic reusable component design. This section uses the StackTemplate example and a related QueueTemplate to illustrate the swapping style of design and programming.

A. Stacks Revisited

One problem noted in Section III is that the designs of the current generation of generic reusable software components

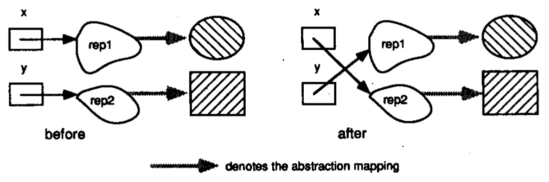


Fig. 3. Implementation view of swapping.

rely heavily on copying. For example, a stack module is often designed with separate top and pop operations, which forces the implementer of the former to make a copy of a value of type item. This type may be expensive to copy.

Possibly recognizing this, some authors (e.g., [19]) replace these two operations with a single operation, called "pop (s,x)," having the following specification:

```
procedure pop (s : stack, x : item)
    requires s ≠ Λ
    ensures #s = s ∘ x
```

The advantage of this definition from the standpoint of reusability is that the implementation of pop need not copy a value of type item. It can simply remove the top value from s and return it in x. If a client program does not need a copy, then it does not pay for making one.

While this design is an improvement over the original version, it still leaves a problem with the "push (s,x)" operation, whose specification demands that the implementer place a copy of x on top of s. This, too, needs to be respecified. We propose the following:

```
procedure push (s : stack, x : item)
    ensures s = #s ∘ #x and item.init (x)
```

The meaning of the assertion "item.init (x)" is that x is an initial value for type item. Notice that in the StackTemplate we specified that a variable of type stack would initially equal the empty string. We assume each type has such an initial value specification. Elsewhere (e.g., [7], [16], [22]) it has been argued that every type should have an associated initial value and that a compiler should generate a call to a type-specific initialization procedure for each variable at the beginning of the block in which it is declared. Similarly, it should generate a call to a type-specific finalization procedure at the end of the block where the variable is active. In the case of the hypothetical language, the initialization procedure should allocate storage for the variable's representation (if it cannot be squeezed into the single pointer space which is allocated automatically), and it should initialize that storage to represent an initial value for the variable's type. The finalization procedure should reclaim any dynamically allocated storage. C++ [18] supports such automatic initialization and finalization, but most languages do not.

With the new specification, push can be efficient for any type item, assuming item initialization is fast. The realization code moves—it does not copy—the concrete representation of x onto the top of s (at the concrete level by moving the pointer to x's possibly complex data structure). It also sets x to an initial value for type item and then returns. This process takes place using only a couple of swap statements. Pointers to data structures remain hidden even within the realization of the StackTemplate.

It is important that in many client programs which use stacks, these new definitions of pop and push will do precisely what is needed. No copying is normally needed. The whole point of using a stack is to save some values temporarily so they can be used later in LIFO order. When a value is placed on a stack for safe keeping, a copy of it is not needed in the client program, so there is no point in wasting the time to make a copy.

But what if some client program does need to get a copy of the top value of a stack without removing it from the stack? Or what if it needs to push a value onto a stack, while retaining a copy of that value for other purposes? This is no problem. The client simply builds upon (reuses) the new StackTemplate module along with a replica function for type item, which would have been needed to implement the original version of the module in any case:

```
function top (s : stack) returns x : item
    variable top_value : item
    pop (s, top_value)
    x := item_replica (top_value)
    push (s, top_value)
end top

procedure push_a_copy (s : stack, x : item)
    variable top_value : item
    top_value := item_replica (x)
    push (s, top_value)
end push_a_copy
```

There are several advantages to this design from the standpoint of reusability. First, a client program pays for copying only if it needs copying. Most clients will not. Second, by implementing operations having the original specifications as *secondary operations* on top of the *primary operations* provided by the new basic-functionality StackTemplate module, no efficiency is lost (up to a small constant for call overhead) compared to the implementation of the original specification. Reversing the choices of the primary and secondary operations would remove this advantage. Third, the implementation of each secondary operation can be proved correct from its formal specification and the formal specification of the StackTemplate operations. This means these operations will work regardless of the representation of type stack. A different realization of the StackTemplate can be substituted in a completely plug-compatible fashion and the secondary operations will continue to work correctly. In principle, they need not even be recompiled, as noted earlier.

We conclude that the revised module design, which eschews copying entirely, is "more reusable" than the original design found throughout the computing literature and in all texts we know of. This illustrates that, surprisingly, even the simplest generic components still offer a challenging design task to the reusable software community. We have used the swapping style to design dozens of generic reusable components. In every case our modules exhibit the same advantages over their classical counterparts as we have demonstrated here for stacks. The approach that substitutes swapping style for copying style in operation design is both general and straightforward.

B. Programming in the Swapping Style

The two short pieces of code above for top and push_a_copy look like what someone might write in the copying style. The only apparent difference is, for example, "x := top_value" is replaced by "x := item_replica (top_value)." We have found such similarity between client programs written in the two styles to be the general rule.

A couple of new programming idioms are frequently encountered though. They are illustrated by the example of implementing a replica function for a FIFO queue. Suppose we have a specification for the QueueTemplate, shown below, which is virtually identical to that of the (revised) StackTemplate. The only difference between the LIFO stack and the FIFO queue is whether items are removed from the same end of the string to which they were added, or from the other end.

```
module QueueTemplate (type item)
    provides
        type queue is modeled by
          string (item) = Λ -- initially
        procedure enqueue
          (q : queue, x : item)
            ensures q = #q ∘ #x
              and item.init (x)
        procedure dequeue (q : queue)
            requires q ≠ Λ
            ensures #q = x ∘ q
        function isempty (q : queue)
              returns empty : bool
            ensures q = #q
              and (empty iff q = Λ)
end QueueTemplate
```

We recommend as a reusable component design guideline that if a module providing a type does not provide a replica function, then it should be possible to copy a variable of that type by invoking operations the module does provide (i.e., the primary operations). This guideline may be considered a weak test of the functional completeness of the primary operations. For example, to copy the value of queue q1 to queue q2, begin by swapping q1 with a temporary queue that is initially empty. Then loop through the temporary queue, doing the following to each item: dequeue it; copy it; enqueue the original item in q1 and enqueue its copy in q2. The loop terminates when the temporary queue is empty, at which time the two copies of the original q1 are in q1 and q2. The following code uses this method:

```
procedure queue_replica (q1 : queue)
      returns q2 : queue
    variable c : queue, x1 : item, x2 : item
    q1 :=: c
    -- let q1' = q1 and let c' = c at this
    -- point, before the loop
    -- loop invariant: q1 = q2 and q1 o c
    -- = q1' o c'
    while not isempty (c) do
        dequeue (c, x1)
        x2 := item_replica (x1)
        enqueue (q1, x1)
        enqueue (q2, x2)
    end while
end queue_replica
```

There are a few interesting things to note about this code. First, a local variable always starts out with an initial value for its type, so initially q2 and c are both empty queues. The way to force any variable (in this case q1) to an initial value for its type is to swap it with an unmodified local variable.

Second, at the end of the procedure every local variable is finalized and the space occupied by its representation is reclaimed. Recall that upon return from this routine, the value of q2 will be swapped with the target of the function assignment statement in which queue_replica is invoked. Therefore q2 will equal the old value of the target variable at the time q2 is finalized. That old value is not "overwritten," as it is with ordinary assignment. This means there is no need for garbage collection, although garbage collection techniques might be applied to storage reclamation at variable finalization time if desired.

Two new programming idioms are apparent in this code. One is the use of a temporary variable to hold the values in the original q1. It starts out as an empty queue, is swapped with q1, and is again empty at the end of the loop. This property leads us to call such a variable a *catalyst* rather than simply a temporary variable, because it is necessary to make the code work, but there is no net change in the variable's value from the beginning of the code to the end. The analogy to a catalyst in a chemical reaction is clear.

It is striking how often the "no net change" property of local variables arises in programs written with software components designed in the swapping style. (The local variable top_value also has this property in top and push_a_copy, as do x1 and x2 in queue_replica.) This is important because finalizing catalyst variables is generally easier and faster than finalizing arbitrary local variables. An initial value for a type is generally chosen by the designer to be something whose representation is easy to construct and therefore, presumably, easy to reclaim.

The other idiom seen here is also typical of processing structured data values using designs based on swapping. It consists of swapping the original input structure with a catalyst variable, dismantling the catalyst item-by-item, doing something with each item as it is removed (in this case copying it and inserting the replica into a duplicate structure), and incrementally rebuilding the original input structure. Both the original input structure and the catalyst experience no net change during this process.

Except for the heavy use of catalyst variables as temporary holding tanks for the reasons noted above, and limited but explicit copying, code written in the swapping style tends to be virtually identical to code written in the copying style for the same task. For example, we have implemented Kruskal's algorithm to find the minimum spanning tree of a graph. The code is about 20 lines long because it is built on generic reusable components we call a Serializer Template (which captures the concept of sorting and selection), an Equivalence Relation Template (which permits dynamic processing of equivalence relations), and a Graph Template. Considerable additional experience—with several hundred undergraduate students and over a hundred practicing programmers in industry to whom we have taught the swapping style—suggests that learning to write code based on swapping is easy. Programmers who are used to copying variable values with assignment statements quickly adjust to the new style.

VI. Related Work

Of course, we are not the first to realize the problems inherent to copying as a means of data movement. For example, the designers of Fortran realized that passing copies of arguments to subroutines would be inefficient. Their solution was to pass all parameters by reference, introducing implicit pointers into the language definition and making reasoning about program behavior more difficult. Many others (e.g., [6], [9], [10]) have also noted the problems with aliased pointers.[3] Yet over the years programming languages have generally continued to provide pointers and to make it easy for programmers to alias them.

Alphard [16] and Ada [4] are noteworthy with respect to copying. In Alphard, the assignment (i.e., copy) operator is not a language primitive, but instead must be defined as a procedure within a **form** (essentially a generic module). Pointers are not explicitly provided by the language, and it is possible to create truly generic modules. However, Alphard does not offer alternatives to the copying style of design or coding. For instance, although copying is not a primitive in Alphard, it is still the only mechanism suggested for data movement. There is no recommendation for an alternative such as swapping. Similarly, parameters are passed by value or by reference. These characteristics suggest that Alphard does not encourage anything other than traditional component designs and that the potential inefficiency of copying was not a major issue to its designers.

Ada addresses some of the problems inherent to copying by allowing the declaration of **limited private** types. Variables declared to be of a type that is **limited private** cannot be copied using the built-in assignment primitive. Instead, the package implementing the type must provide a copy routine if variables of that type are allowed to be copied. Although this feature solves some of the most glaring problems associated with copying, it does not suggest any efficient alternatives. Published designs of generic packages that provide hidden types (such as those in [2]) often rely on the built-in assign-

[3]Hoare's criticism was the earliest of these. The cited paper is a recent version of a less accessible 1973 paper.

ment operator for parameter types and preclude the possibility of a type parameter that is **limited private**. For example, it is not possible to create a stack of stacks of integers using the design guidelines of [2], because the type provided by the stack package is **limited private**, and the item type in a stack is not allowed to be **limited private**. It is even suggested that if a client program needs stacks of stacks, it should declare a new intermediate type that is a pointer to a stack in order to achieve this effect.

Language and component designers have realized some of the problems inherent to copying and to pointers. To our knowledge, though, none has addressed the problems of the copying style by completely avoiding implicit copying, or by providing general-purpose alternatives to copying and suggesting how they might be used to advantage.

We also are not the first to discover advantages of swapping. For example, [10] discusses an "exchange assignment" that is implemented by swapping pointers that are hidden from the client program. "Pointer rotation" [20] is generalized swapping. However, previous discussions of swapping (or rotating) pointers have dealt only with their application to linked structures such as lists and trees. We know of no prior discussion of the advantages of swapping to nonlinked structures, nor has swapping been proposed as a general alternative to copying. Finally, we are not aware of any other attempts to use swapping to give a uniform explanation of all data movement from function assignment to parameter passing.

VII. Conclusions

The following suggestions have been found to be valuable in guiding the design of generic reusable software components:

- Do not assume types that are parameters to generic modules are primitive and therefore inexpensive to copy. Allow for the possibility that they may have large data structures as their concrete representations.
- In a reusable component that provides a new type, include as primary operations those procedures and functions that are necessary to be able to perform any "interesting manipulations" with variables of that type. Among the manipulations a client clearly might wish to perform are checking the **requires** clause of every primary operation and copying a value of the provided type (assuming the ability to copy the item type).
- Subject to the previous guideline, include as primary operations *only* those procedures and functions whose efficient implementation depends on having direct access to the representation of the provided type. All other operations can and should be implemented as secondary operations using calls to the primary operations.

Some specific corollaries of these general suggestions include:

- When designing a primary operation to insert an item into a composite structure (e.g., push for a stack, enqueue for a queue, insert for a list, store for an array, etc.), define its behavior to permit an implementation that can swap it in. Do not force the component implementer to insert a copy of the item, because that should be a secondary operation for a well-designed reusable component.
- When designing a primary operation to inspect or remove an element from a composite structure (e.g., top or pop for a stack, dequeue for a queue, remove or get_value for a list, fetch for an array, etc.), define its behavior to permit an implementation that can swap it out. Do not force the component implementer to return a copy of the element, because that operation should be secondary.

It is interesting to note what happens when these guidelines are applied to design of an array abstraction. Both corollaries above suggest the *same* primary operation for storing a value into an array and fetching a value from it. This single procedure—given an array, an index into it, and a value of the item type—swaps that value with the value currently stored in the specified position of the array. The usual store and fetch operations are secondary operations that are implemented using this "swap_indexed_item" procedure and a copy operation for the item type.

Generic reusable software components developed using the swapping style have many potential advantages over components developed using the traditional copying style. Algorithms are potentially more efficient when copying (which is often costly) is avoided. With swapping it is easy to design unconstrained generic components in which no restrictions are placed upon the type of items in a complex structure. It becomes easy to reuse the generic module object code among all instances of a module, thereby reducing both the amount of object code for a complete system and the time necessary for recompilation. Finally, keeping pointers hidden from a client programmer means programs become potentially more reliable, because reasoning about their behavior is easier and because programs can no longer contain bugs resulting from visible aliasing and dangling references.

Acknowledgment

The authors are pleased to thank the members of the Reusable Software Research Group at The Ohio State University for many helpful discussions. They especially thank B. Ogden for suggesting many of the central ideas of this paper, and S. Zweben, W. Hegazy, J. Krone, C. Nicholas, T. Pittel, M. Sitaraman, and M. Stovsky for other helpful input regarding its form and content. The anonymous reviewers also provided valuable feedback to help improve the paper.

References

[1] E. V. Berard, *Creating Reusable Ada Software.* Frederick, MD: EVB Software Engineering, 1987.

[2] G. Booch, *Software Components with Ada.* Menlo Park, CA: Benjamin/Cummings, 1987.

[3] S. Danforth and C. Tomlinson, "Type theories and object-oriented programming," *Comput. Surveys,* vol. 20, no. 1, pp. 29–72, Mar. 1988.

[4] *Reference Manual for the Ada Programming Language,* Ada Joint Program Office, U.S. Dept. Defense, ANSI/MIL-STD-1815A-1983. Washington, DC: U.S. Government Printing Office, 1983.

[5] M. B. Feldman, *Data Structures with Modula-2.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

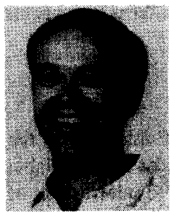
[6] M. Ganapathi and G. O. Mendal, "Issues in Ada compiler technology," *Computer,* vol. 22, no. 2, pp. 52–60, Feb. 1989.

[7] D. E. Harms, "The influence of software reuse on programming language design," Ph.D. dissertation, Dept. Comput. Inform. Sci., Ohio State Univ., Columbus, Aug. 1990; available from University Microfilms, Inc.

[8] W. A. Hegazy, "The requirements of testing a class of reusable software modules," Ph.D. dissertation, Dept. Comput. Inform. Sci., Ohio State Univ., Columbus, June 1989; available from University Microfilms, Inc.
[9] C. A. R. Hoare, "Hints on programming language design," in *Programming Languages: A Grand Tour,* E. Horowitz, Ed. Rockville, MD: Computer Science Press, 1987, pp. 31–40.
[10] R. B. Kieburtz, "Programming without pointer variables," *ACM SIGPLAN Notices,* vol. 8, no. 2, pp. 95–107, 1976.
[11] J. Krone, "The role of verification in software reusability," Ph.D. dissertation, Dept. Comput. Inform. Sci., Ohio State Univ., Columbus, Aug. 1988; available from University Microfilms, Inc.
[12] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development.* Cambridge, MA: MIT Press, 1986.
[13] J. J. Martin, *Data Types and Data Structures.* Englewood Cliffs, NJ: Prentice-Hall, 1986.
[14] B. Meyer, *Object-Oriented Software Construction.* Cambridge, England: Prentice-Hall International, 1988.
[15] T. Pittel, "Pointers in RESOLVE: Specification and implementation." M.S. thesis, Dept. Comput. Inform. Sci., Ohio State Univ., Columbus, June 1990.
[16] M. Shaw, Ed., *ALPHARD: Form and Content.* New York: Springer-Verlag, 1981.
[17] M. Sitaraman, "Mechanisms and methods for performance tuning of reusable software components," Ph.D. dissertation, Dept. Comput. Inform. Sci., Ohio State Univ., Columbus, Aug. 1990; available from University Microfilms, Inc.
[18] B. Stroustrup, *The C++ Programming Language.* Reading, MA: Addison-Wesley, 1986.
[19] D. F. Stubbs and N. W. Webre, *Data Structures with Abstract Data Types and Modula-2.* Monterey, CA: Brooks/Cole, 1987.
[20] N. Suzuki, "Analysis of pointer 'rotation,'" *Commun. ACM,* vol. 25, no. 5, pp. 330–335, May 1982.
[21] B. W. Weide, "A new ADT and its applications in implementing 'linked' structures," Dept. Comput. Inform. Sci., Ohio State Univ., Columbus, Tech. Rep. OSU-CISRC-TR-86-3, Jan. 1986.
[22] B. W. Weide, W. F. Ogden, and S. H. Zweben, "Reusable software components," in *Advances in Computers,* vol. 33, M. C. Yovits, Ed. New York: Academic, 1991, to be published.
[23] J. M. Wing, "A specifier's introduction to formal methods," *Computer,* vol. 23, no. 9, pp. 8–24, Sept. 1990.

Douglas E. Harms (S'87–M'88) received the Bachelor's degree in mathematics and computer science from Muskingum College, New Concord, OH, in 1979, and the M.S. and Ph.D. degrees in computer science from The Ohio State University in 1983 and 1990, respectively.

After working in the Advanced Development Department of NCR Corporation in Cambridge, OH, for several years, he joined the faculty of Muskingum College, where he is currently an Assistant Professor of Computer Science. He also teaches in the computer science curriculum for AT&T's Kelly Education Center. His current research interests include software engineering, especially in the areas of reusable software, program specification, program verification, and programming language design.



Bruce W. Weide (S'73–M'78) received the B.S.E.E. degree from the University of Toledo, Toledo, OH, in 1974 and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, in 1978.

He joined the Department of Computer and Information Science at The Ohio State University in 1978, where he is currently an Associate Professor. His research interests center on technical aspects of software reuse, including formal specification, verification, and component design methodology. He is also the primary designer of MacSTILE®, a commercial schematic diagramming tool that supports system design with reusable components.