A Class of Programming Language Mechanisms to Facilitate Multiple Implementations of the Same Specification

Murali Sitaraman*
Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506
E-mail: murali@cs.wvu.wvnet.edu

Modern programming languages include mechanisms to permit details of specification and implementation of a software component to be separated. This separation is the key to attractive performance compromises, because it allows several components with different performance characteristics for the same abstract functional behavior.

A software system may involve more than one implementation of a specification either because different regions were developed independently or for performance reasons. Alternatively, a system may evolve and require changes in its performance, reliability, or hardware, and such changes might be accomplished by switching the implementations associated with some abstractions used in the system. To facilitate construction and use of multiple implementations of the same specification, we show that an important class of programming language mechanisms — not present in languages such as Ada — are essential. We propose ways to enhance Ada with these mechanisms and discuss some interesting ramifications.

1 Introduction

The importance of separating the specification of a reusable software component from its implementation details has been widely recognized, and several programming languages supporting this idea have been designed and developed. A good specification must capture the abstract functionality of a component without any implementation bias [26], and must permit several implementations with different performance characteristics. The functionality of a client of a component should depend only on the abstract behavioral specification of the component, so any proofs of correct functioning depend only on this specification [6, 17].

The importance of permitting multiple component implementations for the same abstract specification has been observed by designers of modern programming languages [1, 2, 7, 14, 18, 21, 22, 25, 26, 27]. Given a specification, normally it is not possible to develop one implementation that has best performance in all respects. One component using certain data structures and algorithms may provide most efficient implementations for some operations described in the specification of the component, while another may provide more efficient implementations for other operations. The average case performance of one component may be better whereas another may be better in the worst case. One implementation may be more space-efficient and another may be more time-efficient. One may be more "predictable," and hence, attractive for real-time applications, and another may be more efficient, but not necessarily predictable. Different implementations may be better suited for different hardware architectures. The inevitability of performance compromises argues for the need to have several concrete components available for the same abstract specification.

Availability of multiple component implementations from different component manufacturers with varying price/performance characteristics plays an important role in software component industry views, such as the ones discussed in [4, 13, 15, 25]. This is true of every mature industry from cosmetics to computer hardware, and a successful software industry will likely follow suit.

There are two different ways in which a *large* client system may come into contact with more than one implementation during its lifetime. For one thing, a software system may intentionally or accidentally use more than one implementation of the same abstraction simultaneously. For performance reasons, the same software system may intentionally choose to use two different implementations with two instances of a reusable abstraction (possibly in different regions of the system for different purposes) [8, 19]. It is also possible that accidentally, the different regions of a software system developed independently may have used different

---

* Murali Sitaraman's name appears as S. Muralidharan in some of his previous publications.

0-8186-2585-6/92 $03.00 © 1992 IEEE

implementations of the same abstraction [10]. A programming language therefore should permit a software system to simultaneously use different implementations of an abstraction.

Performance-related evolution is another independent reason for how a software system might come in contact with multiple implementations of an abstraction. [14] notes that programming language mechanisms should permit such evolution to be graceful. While the fundamental abstractions used in a software system may not change during the system's lifetime, typically it has to be upgraded to meet new performance, reliability, or hardware requirements. Such requirements can be readily satisfied by plugging in new implementations (for the abstractions used by the system) that satisfy these modified requirements. Switching to different implementations in such cases must be possible with only minimal revalidation and recompilation of the system. Ideally, only the specific declarations where implementations are changed must be recompiled and all uses of the declared instances must remain unaffected.

Statically and strongly typed, imperative programming languages such as Ada [1], Alphard [18], CLU [2], and Modula-2 [27] do not include mechanisms for facilitating the construction and use of multiple component implementations, as explained in this paper. Using Ada as a representative of this class, we show how a minimal set of mechanisms can be added to facilitate these possibilities, without introducing inheritance, subtypes, or dynamic binding as essential features. Object-oriented programming features, such as those in [14], do provide better support for multiple implementations than Ada, though some key questions raised here have not been addressed in that community. Our goal in this paper is to extend Ada along its own lines to support multiple implementations. The proposed mechanisms allow functionality and performance of software components to be controlled independently, by allowing for separation in specification, analysis, and validation of these two aspects.

The paper is organized as follows. Section 2 introduces an appropriate framework for our discussions. Section 3 introduces a basic set of mechanisms to facilitate the use of multiple implementations in Ada. Section 4 gives an example providing a stronger rationale for the proposed basic mechanisms and demonstrating a need to independently parameterize specifications and their implementations. Section 5 outlines the need and mechanisms for performance parameterized components. Section 6 exemplifies a representation-independent technique for translation of abstract data values. Section 7 discusses related work. Section 8 presents our conclusions.

2 A framework for software development based on reuse

A framework for a component-based reuse technology for software development that parallels that of the traditional engineering disciplines is discussed in [19, 25]. Here we explain this framework, for it provides a useful focus for discussion. Figure 1 shows how we consider a software system to be structured for our purposes. (This structure is close in spirit to the "3C reference model" [5, 23].)

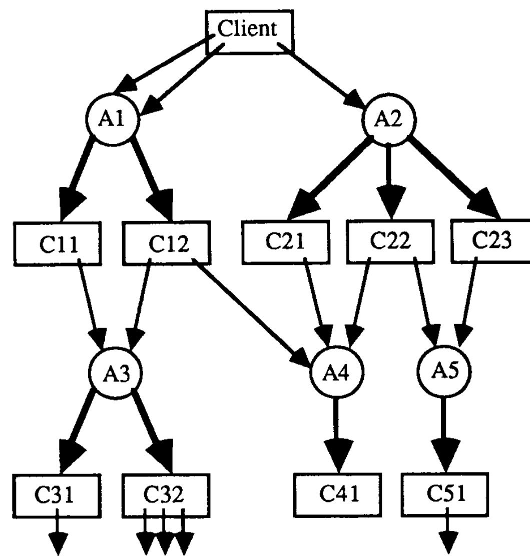

Figure 1 — A software model with abstract and concrete components

The circles denote *abstract components* and the rectangles denote *concrete components.* A thick arrow from an abstract component to a concrete component means that the latter implements the former. A thin arrow from a concrete component to an abstract component means that an instance of that abstract component is used by the concrete component. The same client may use more than one instance of an abstraction. Throughout this paper, we consider this small window on a much larger software system from the point of view of a representative concrete component called Client. A *component* (or part) is (in programming language terms) a module, package,

class, or other similar entity, and it typically provides a procedural or data abstraction. An abstract component (usually called specification) explains functionality, and a concrete component (also called a body or implementation or realization) for that abstract component implements the functionality. It is important to consider abstract and concrete components as separate, but related, entities.

For example, suppose Client is a piece of code that involves searching and sorting. Then A1 might be a searching abstraction and A2 a sorting abstraction. C11 might be code that implements A1 using a binary search tree and C12 code that implements A1 using a hash table. The key idea — by now well known to software engineers [17] — is that in order to *program* Client one should need to understand only the abstract components describing the visible behavior of searching and sorting abstractions (i.e., A1 and A2). Any proofs of correctness of Client will depend only on these abstractions. In order to *execute* Client, however, one must also choose concrete components that implement the searching and sorting abstractions (i.e., one of C11 or C12 and one of C21, C22, or C23). The performance of Client will depend on all the concrete parts used in it, directly or indirectly.

For the discussions in this paper, it does not really matter what languages are used in expressing the abstractions and their implementations, or how the concrete components are constructed for a given abstract component. The emphasis here is on a set of minimal language features, not available in languages such as Ada, essential for integrating the idea of multiple implementations in the design of reusable components, and not on any one specific programming language.

3 Mechanisms for naming and referencing package bodies

Figures 2 and 3 show how the elements of the structure in Figure 1 can be presented by enhancing Ada with mechanisms for naming and referencing package bodies and specifications independently. Figure 2 shows a package specification A1 and two bodies C11 and C12 for A1. Each body has an independent performance specification. The syntax and notations that can be used for specifying performance are important, but we will not discuss these issues in this paper. Figure 3 shows a client that uses two different bodies with two instances of the same abstraction A1.

```
package A1 is
  ...
end A1;

package body C11 for A1 is

  ...
  performance
  ...
end C11;

package body C12 for A1 is
  ...
  performance
  ...
end C12;
```

Figure 2 — Two concrete components for an abstract component

```
package P1 is new A1
           with body C11;

package P2 is new A1
           with body C12;
```

Figure 3 — Two declarations in the same client program

It is important to note that if the client switches the body associated with a package, only that specific declaration needs to be recompiled. For example, a change in the declaration of P1 from C11 to C12 (or C13) in Figure 3 should result only in the declaration of P1 to be recompiled. None of the modules that use P1 need to be recompiled or revalidated, because the specification of P1 has not changed. This is an important advantage in the construction of large software systems. (The declarations P1 and P2 may or may not be within the same module of the client program.)

With the modified syntax, type equivalence rules remain the same as in standard Ada. If A1 provides a type T, P1.T and P2.T are not equivalent even if the two package declarations are structurally identical (i.e., they use the same specification and the same body with the same arguments).

Are there alternatives to the proposed enhancements?

The need for the proposed enhancements must be clear. Without them, it is not possible to capture the structure in Figure 1. In particular, it is not possible to construct a client such as the one in Figure 3 in Ada, as it stands now, where the same client uses different bodies with different instances of an abstraction. [1, 8, 19]. This is because two bodies for the same package specification need to be linked with the client program.

The ability to use multiple implementations simultaneously in the same client system is only one of the reasons for introducing the syntax along the lines shown in Figures 2 and 3. Numerous other reasons for the proposed enhancements will become clear later in this paper.

4 Need for generic parameters to package bodies

The simple syntax introduced in the last section, unfortunately, is not sufficiently powerful. In particular, there is a need for independently parameterizing a specification and its bodies. Here we consider a detailed example to illustrate the problem and our solution.

Specification of the map abstraction

Figure 4 shows an implementation-independent Ada package specification to abstract the associative searching problem: Given a search value and a map, access the information associated with the search value in the map.

```
generic
   type D is  limited  private;
   type R is  limited  private;
   ...
package Partial_Map_Template

   type Partial_Map
         is  limited  private;
   -- Partial_Map is modeled by
   -- D -> <r: R, defined: Boolean>

   procedure Initialize
      (m: in out Partial_Map);
   -- ensures ∀ d: D, ~m(d).defined

   procedure Finalize
      (m: in out Partial_Map);

   procedure Swap
      (m1, m2: in out Partial_Map);
   -- ensures m1 = #m2 and m2 = #m1

   procedure Define
      (m: in out Partial_Map;
      d: in out D; x: in out R);
   -- requires ~m(d).defined
   -- ensures (m = #m, except
   --    m(#d).r = x) and m(#d).defined
   --    and D.Init(d) and R.Init(x)

   procedure Undefine
      (m: in out Partial_Map;
      d: in out D; x: in out R);
   -- requires m(d).defined
   -- ensures (m = #m, except
   --    ~m(#d).defined) and
   --    x = #m(#d).r and d = #d

   procedure Undefine_Any_One
      (m: in out Partial_Map;
      d: in out D; x: in out R);
   -- requires ∃ d: D,
   --     s.t. m(d').defined
   -- ensures (m = #m, except
   --    ~m(d).defined) and #m(d).r = x

   procedure Is_Defined
      (m: in out Partial_Map;
      d: in out D; def: out BOOLEAN);
   -- ensures m = #m and d = #d and
   --    def = m(d).defined

   procedure Is_Undefined_Everywhere
      (m: in out Partial_Map;
      empty: out BOOLEAN);
   -- ensures m = #m and
   --  empty = (∀d': D, ~m(d').defined)

   private
      type Partial_Map_Rep;
      type Partial_Map
         is access Partial_Map_Rep;
end Partial_Map_Template;
```

Figure 4 — An Ada package specification for the map abstraction

The Ada package specification in Figure 4 provides an abstract data type Partial_Map. Formal model-based specifications of the operations are presented in a dialect of RESOLVE [9, 19, 25]. [6] explains how such specifications can be used in mathematically proving the functional correctness of clients using Ada packages. The specification can be expressed in any formal specification language such as those found in [26], or it can even be informal. What is important is to see that the associative searching problem can be specified without bringing in details of any one implementation.

The type Partial_Map is modeled as a total mathematical function from D (for domain) to an ordered pair of R (for range) and Boolean. Each operation on a Partial_Map is explained using standard symbols from the theory of functions. Two clauses are used in the formal specification of each operation: a **requires** clause and an **ensures** clause. The requires clause states what must be true of the arguments passed to the operation. If the requires clause holds when an operation is called, the ensures clause will hold when it terminates for a correct implementation of the operation. In the ensures clause, the notation "#x" for a parameter x denotes the incoming value of the parameter when the operation is called and "x" denotes its value when the operation returns. In the requires clause, the variables always denote the incoming parameter values.

The package specification Partial_Map_Template presents no implementation bias, following some of the design guidelines for Ada packages discussed in [5, 9, 16]. These guidelines argue for the need to define an initialize, a finalize, and a swap operation on every abstract data type. Initialize allocates storage and creates an appropriate initial value for a type, while finalize reclaims the allocated storage. These two operations are essential in order for a client to use different implementations uniformly. [9] explains the problems with "copying" as a primitive data movement mechanism for abstract data types, and proposes "swapping" as an alternative, leading to the inclusion of the Swap operation.

The need for most other operations must be clear. Define permits a new definition to be added to the map. It has been specified so that the incoming domain and range values do not have to be copied. Undefine removes the definition at a given domain value and returns the corresponding range value. Is_Defined returns true only if the map is defined on the given domain value. The one operation that is a bit unusual is Undefine_Any_One. This operation combined with Is_Undefined_Everywhere permits a client to iterate over all definitions in a partial map. Without this operation, for example, an equality-testing operation on Partial_Maps cannot be implemented efficiently, or at all. A careful reader will observe that the operations listed in the specification are orthogonal, i.e., no operation can be efficiently implemented using a combination of others. Notice also that every requires clause in the specification can be tested by a client.

In the Ada package specification in Figure 4, generic (parameteric) procedures have been omitted. Most implementations of this specification will need some such procedures on the generic types D and R to be brought in as parameters. This is the topic of the next sub-section.

Possible implementations

The Partial_Map_Template can be implemented in a variety of ways, resulting in different performance behaviors. Any known search technique can be used in creating an implementation of this abstraction. Linear search and binary search, (balanced) binary search trees, and hash tables are examples of such techniques that can be used in creating a package body for Partial_Map_Template. Careful performance analysis of possible search implementations reveals interesting compromises to be made [12, 19]. The map abstraction is, of course, not unique in this respect. For moderately complex abstractions, it is unlikely that there will be one best implementation.

Let us consider two bodies of Partial_Map_Template — one that uses a binary search tree to represent the encapsulated type Partial_Map and another that uses a hash table to represent the Partial_Map. In the body that uses a binary search tree, say Binary_Search_Tree body, the places where a map is defined are kept in the tree, based on some ordering of the domain values. Each entry in the tree is a domain-range pair. To write this implementation, a procedure to order values of the generic parameter type D is needed as a generic package parameter. In the other body, Hashed_Search, the type Partial_Map is represented using a hash table with chaining to handle collisions. Given a domain value, a hash function is applied on it. The function produces an index into the hash table, and the domain value and the corresponding range value are stored in that location in the table. All domain values that hash to the same location in the table are chained. To write this implementation, the implementer needs the client to supply the hash function as a parameter. Clearly, different bodies of a generic package specification may need different implementation-dependent procedures as parameters. Figure 5 introduces generic procedures as parameters for bodies to solve this problem. Figure 6

shows a client declaration that uses Binary_Search_Tree for an instance of Partial_Map_Template.

```
generic
   with procedure Are_Ordered[1]
      (x, y: in out D;
      ordered: out BOOLEAN);
   ...
package body Binary_Search_Tree
      for Partial_Map_Template is
   ...
end Binary_Search_Tree;


generic
   Table_Size: NATURAL;
   with procedure Hash_Operation
      (d: in out D;
      hash_index: out NATURAL);
   with procedure Are_Equal
      (x, y: in out D;
      equal: out BOOLEAN);
   ...
package body Hashed_Search
      for Partial_Map_Template is
   ...
end Hashed_Search;
```

Figure 5 — Different generic parameters for two bodies of Partial_Map_Template

```
package Int_Float_PM is new
 Partial_Map_Template(INTEGER, FLOAT)
 with body Binary_Search_Tree(<=,..);
```

Figure 6 — A client of Partial_Map_Template using the Binary_Search_Tree body

In Figure 6, if the body associated with Int_Float_PM is changed to Hashed_Search, only the package declaration of Int_Float_PM must be recompiled. None of the modules using Int_Float_PM has to be recompiled or revalidated, because its specification has not changed.

The problem and solution discussed here reinforce the need for mechanisms for independently naming and referencing package bodies (presented in the previous section). Without the ability to separately name specifications and bodies, it is not possible to parameterize them independently either.

Are there good alternatives to the proposed solution?

It is important to understand that the generic parameters for bodies introduced in the last section are unavoidable. Without them, it is impossible to write those bodies. The question here is whether such parameters can be made parameters of the specification, thus avoiding the need for independent parameters for bodies.

Can the designer of the specification Partial_Map_Template create a super-general specification that includes as generic parameters every procedure that may be needed in building every implementation of this specification? An attempt such as this can only be an expensive but futile exercise — it is unlikely that a specifier can conceive of every possible implementation. In addition, a client will be burdened by being forced to supply a large set of parameters, even though only a few of those will really be used, depending on the implementation. Creation of a specification that is super-general is clearly an inferior solution.

Another alternative is to create a different package for each different body, each having generic parameters appropriate for its body. For example, Partial_Map_Template_1 and Partial_Map_Template_2 could be two packages with identical specifications, except for their generic parts. Following this strategy, for every new package body to be created for the map abstraction, a new package specification must be designed. This situation is far from ideal. The problems here are similar in nature to attempts to simulate generics in a language without a language mechanism. The generics in Ada, for example, can be simulated by copying the specification of a package and replacing the generic types with specific types. This can be done manually or with the assistance of a "smart" compiler. However, the need for language support to have a single generic specification, that can be appropriately instantiated, has been argued by several programming language designers, and most of these arguments can be extended to the present case. Having multiple copies of the same specification leads to

---

[1] We have omitted formal specifications of generic procedure parameters for simplicity; details can be found in [6, 19]. Note also that procedures other than Are_Ordered may be needed as parameters (e.g., a procedure to swap or replicate a value of the type D). However, for the point we want to make here, these details are not essential. Are_Ordered has been made a procedure because D is a limited private type, and therefore, cannot be an "in" parameter to any formal generic procedure [1].

problems of consistency, the need to update duplicates in case of modifications to a specification, etc. Switching a body associated with an instance will result in expensive recompilation and revalidation of the entire client system. With the proposed solution this will not be the case.

The generic procedure parameters discussed here serve a completely different purpose from the parameters discussed in [6], or [11, 14] in the context of "constrained genericity," and they should not be confused. In that context, parameters are needed for explaining the specifications. The parameters discussed here vary depending on the implementation, and have nothing to contribute to the abstract behavior described in the specification.

5 Performance parameterization

Ada permits constants, types and operations as generic parameters to package specifications. In this paper, we have argued already for constants and operations as parameters to bodies. The mechanisms discussed so far are the basic requirements to facilitate multiple implementations. However, these mechanisms can be extended in several ways [19]. Here, we consider one such extension.

In [19], we introduce an entirely different type of parameter — bodies as parameters to other bodies. In a successful reuse scenario, new bodies typically will be constructed by reusing existing abstractions and their bodies. For example, the Hashed_Search body discussed in the last section may use the declarations in Figure 7 to represent a Partial_Map. Here, List_Template is the specification of a reusable, generic package that provides the abstract data type List; Pointer_Based is one of its bodies. The builder of Hashed_Search has arbitrarily chosen Pointer_Based for the package LP.

```
...
type DR_Pair is record
   d: D;
   r: R;
end;
package LP is
   new List_Template(DR_Pair)
   with body Pointer_Based;
type Partial_Map_Rep is
   array (1..Table_Size) of LP.List;
...
```

Figure 7— Internals of Hashed_Search

It is possible to imagine several versions of Hashed_Search, all of them *identical*, except that each uses a different body for List_Template. The performance of Hashed_Search will be clearly influenced by the performance of the body chosen for LP. In general, if a body uses m package specifications and if each of these specifications has n different bodies, then a total number of n^m versions can be built. All these versions are identical, except for the bodies they reuse. It is possible to parameterize this aspect of performance. Figure 8 shows a performance-parameterized version of Hashed_Search that includes a body for List_Template as a parameter.

```
generic
   Table_Size: NATURAL;
   with procedure Hash_Operation
      (x: in out D;
      hash_index: out NATURAL);
   with procedure Are_Equal_D_Values
      (x, y: in out D;
      equal: out BOOLEAN);
   with body B for List_Template;
   ...
package body
   Parameterized_Hashed_Search
   for Partial_Map_Template is
      ...
      package LP is
         new List_Template(D)
         with body B;
      ...
end Parameterized_Hashed_Search;
```

Figure 8 — Different generic parameters for different bodies of Partial_Map_Template

To instantiate a package with body Parameterized_Hashed_Search, a client should pass a body of List_Template as an argument. Additional discussion on this topic can be found in [20, 19].

6 The translation problem

Sometimes, when a client software system uses different representations of a data abstraction (e.g., Partial_Map) simultaneously, there is a need to translate a value from one representation to another. Figure 9 shows part of the code to translate a value of Partial_Map between two representations.

```
package Int_Float_PM1 is new
 Partial_Map_Template(INTEGER, FLOAT)
 with body Binary_Search_Tree(<=,..);

package Int_Float_PM2 is new
 Partial_Map_Template(INTEGER, FLOAT)
 with body Hashed_Search(100,...);
   ...
      m1: Int_Float_PM1.Partial_Map;
      m2: Int_Float_PM2.Partial_Map;
      d: INTEGER;
      x: FLOAT;
   ...
   while not Int_Float_PM1.
      Is_Undefined_Everywhere(m1) loop
      Int_Float_PM1.Undefine_Any_One
                              (m1, d, x);
      Int_Float_PM2.Define(m2, d, x);
   end loop;
   ...
```

Figure 9 — A representation-independent translator for Partial_Maps

A careful observation of the translator code will indicate that this code can be used to translate a Partial_Map from any one representation to any other. It is indeed possible to write a single generic package that provides a reusable translator. This translator can then be instantiated to translate from one given representation to another. Solutions to the translation problem involving a canonical representation such as in [10], or using other representation-dependent methods will require construction of O(n) different translators, where n is the number of different representations.

The time complexity of the translator in Figure 6 is (at least) proportional to the number of defined points in the map. A translation approach based on a canonical representation is no better in this respect. In fact, the translator presented here is more efficient, because it "moves" a value of Partial_Map rather than replicating it [9]. It should be noted, however, that a replicating reusable translator can be readily constructed, if needed, by first replicating the original Partial_Map and then translating the replicated value. As long as a collection abstract data type includes operations to access every member, a representation-independent translator can be written. This requirement is satisfied by natural designs for most abstract data types such as those found in [2, 3, 24].

Communication overheads become an important issue in translation, when representations are distributed over a network. It is possible to reduce the impact of these overheads on the generic translation approach using clever message passing methods. Details are beyond the scope of this paper.

7 Related work

Object-oriented languages, such as Eiffel [14] and C++ [21], support some form of specification inheritance, and this mechanism can be used to permit the same client program to use more than one implementation of a specification. In our example, Partial_Map_Template can be a parent and Binary_Search_Tree and Hashed_Search can be two heirs of it. The same client can now use both Binary_Search_Tree and Hashed_Search. Thus, object-oriented languages provide some support for multiple implementations of an abstraction, if designers follow certain conventions such as constructing a deferred class for every class. [11] highlights some complex issues in developing multiple implementations of the same abstract class using inheritance.

The mechanisms presented in this paper bind bodies to instances statically. This is because we want them to be in the same spirit as other mechanisms in Ada. It will be interesting to study the influences of allowing these bindings to be dynamic on the efficiency, flexibility, and verifiability of software systems. To our knowledge, this is an open problem.

[8] views the process of going from a specification to an implementation as a transformation problem, and notes the importance of multiple transformations from a specification. However, no language mechanisms are proposed.

ML, a functional language, includes mechanisms for independent naming of signatures and structures, and for passing structures as parameters [22]. We note in [19], however, that passing structures or packages as parameters to specifications may be appropriate, but passing them as parameters to bodies can have undesirable consequences on software verification.

8 Conclusions

We noted at the outset that the idea of multiple implementations for the same specification is not new. Designers of most modern programming languages have observed the importance of this idea. Our contribution in this paper is in identifying the need for and then presenting a small set of programming language mechanisms to

facilitate the use of multiple implementations. Using the mechanisms presented here, functionality and performance of software systems can be separated, specified, analyzed, and verified [19].

It is only reasonable to expect any programming language aimed at enhancing software reuse to include sufficient mechanisms for specifying and implementing an abstraction as fundamental as Partial_Map_Template. Designs of programming languages such as Ada, Alphard, CLU, and Modula-2 emphasize the role of multiple implementations for the same abstract specification, but do not include all mechanisms needed to use this important technique. Though some object-oriented languages include support for a subset of these mechanisms in different forms, the problems raised here and the solutions proposed here are new.

We have argued, using realistic examples, for the need to name specifications and implementations separately, for the need to distinguish parameters to specifications and implementations, and for the ability of clients to associate different implementations with different instances of the same abstractions. To support these possibilities, we have demonstrated that some basic language mechanisms are essential. We have outlined how the proposed solutions can be extended to produce performance-parameterized implementations and reusable translators. Mechanisms proposed in this paper are currently under implementation at the West Virginia University.

Acknowledgments

It is a pleasure to thank the anonymous referees for their valuable comments which have helped clarify details of this paper. I would like to thank Bruce Weide for his support throughout the evolution of this research work. I would also like to thank the other members of the Reusable Software Research Group at The Ohio State University, including Joe Hollingsworth, Bill Ogden, and Stu Zweben, and members of the group at West Virginia University.

References

1. *Reference Manual for the Ada Programming Language*, Washington, D.C, January, 1983.

2. B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, McGraw-Hill (1986).

3. G. Booch, *Software Components with Ada*, Benjamin/Cummings (1987).

4. B.J. Cox, "Planning the Software Industrial Revolution", *IEEE Software 7*, 6 (1990).

5. S. Edwards, *An Approach for Constructing Reusable Software Components in Ada*, Tech. Rept. P-2378, IDA Paper, Institute for Defense Analyses, Alexandria, VA, September, 1990.

6. G.W. Ernst, R.J. Hookway, J.A. Menegay, and W.F. Ogden, "Modular Verification of Ada Generics", *Computer Languages 16*, 3/4 (1991), 259-280.

7. J. Goguen, "Reusing and Interconnecting Software Components", *IEEE Computer 19*, 2 (1986), 16-28.

8. D. Gries and D. Volpano, "The Transform — a New Language Construct", *Structured Programming 11*(1990), 1-10.

9. D.E. Harms and B.W. Weide, "Copying and Swapping: Influences on the Design of Reusable Software Components", *IEEE Trans. Soft. Eng. 17*, 5 (1991), 424-435.

10. M. Herlihy and B. Liskov, "A Value Transmission Method for Abstract Data Types", *ACM Trans. Prog. Lang. and Systems 4*, 4 (1982), 527-551.

11. C. Horn, "Conformance, Genericity, Inheritance, and Enhancement", In *Proc. European Conference on Object-Oriented Programming*, June 1987, pp. 223-233.

12. D.E. Knuth, *The Art of Computer Programming*, Addison-Wesley (1968).

13. M.D. McIlroy, "Mass-Produced Software Components", in *Software Engineering Concepts and Techniques*, J.M. Buxton, P. Naur and B. Randell, eds., Petrocelli/Charter (1976), , 88-98.

14. B. Meyer, *Object-oriented Software Construction*, Prentice-Hall (1988).

15. S. Muralidharan and B.W. Weide, "Reusable Software Components = Formal Specifications + Object Code: Some Implications", In *Proc. Third Annual Workshop: Methods and Tools for Reuse*, June 1990.

16. S. Muralidharan and B.W. Weide, "Should Data Abstraction Be Violated to Enhance Software Reuse?", In *Proc. Eighth Nat. Conf. on Ada Technology*, March 1990, pp. 515-524.

17. D.L. Parnas, "A Technique for Software Module Specification with Examples", *Comm. ACM 15*, 5 (1972), 330-336.

18. *ALPHARD: Form and Content*, M. Shaw, ed., Springer-Verlag (1981).

19. M. Sitaraman, *Mechanisms and Methods for Performance Tuning of Reusable Software Components*, Ph.D. dissertation, The Ohio State University, 1990.

20. M. Sitaraman, "Parameterized Performance Control of Systems Built from Reusable Software Components", In *Proc. Third Int. Conf. on Software Engineering and Knowledge Engineering*, June 1991, pp. 224-229.

21. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley (1986).

22. M. Tofte, *Four Lectures on Standard ML*, Technical Report, Computer Science, Edinburgh University, UK, 1989.

23. W. Tracz, "The Three Cons of Software Reuse", In *Proc. Third Annual Workshop: Methods and Tools for Reuse*, June 1990.

24. B.W. Weide, *A Catalogue of OWL Conceptual Modules*, Tech. Rept. OSU-CISRC-TR-86-2, Department of Computer and Information Science, The Ohio State University, Columbus, OH, January, 1986.

25. B.W. Weide, W.F. Ogden, and S.H. Zweben, "Reusable Software Components", in *Advances in Computers*, M.C. Yovits, eds., Academic Press, Vol. 33 (1991), 1-65.

26. J.M. Wing, "A Specifier's Introduction to Formal Methods", *IEEE Computer 23*, 9 (1990), 8-24.

27. N. Wirth, *Programming in Modula-2*, Springer-Verlag (1982).