

On Tight Performance Specification of Object-Oriented Software Components*

Murali Sitaraman
Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506-6330
E-mail: murali@cs.wvu.edu

Abstract

Most modern designs of a software component include two separate pieces: functionality specification and implementation. When the specification is formal, this separation permits verification of reusable software components to be modular — essential for verification to be local, scalable, and hence, practical. In this paper, we explain the role of a third piece — an implementation-dependent, performance specification — for a component. Introduction of this piece permits performance (e.g., execution time bounds) specification to be expressive (tight) while leaving functionality specification fully abstract and verification to be modular.

1 Introduction

The importance of separating the formal behavioral specification of a component from its implementation details is widely acknowledged in the software engineering community [1, 28]. For reusable software components this separation is especially crucial because it permits specification-based, modular verification — verification that a component implementation meets its specification without the need for client context and verification of the client without the need for the component implementation [2]. Essentially, modular verification permits a component to be verified once in its lifetime when it is entered into a library, and never again when it is reused. Such “quality verification cost-savings” are noted to be at least as significant as those from improved productivity [10]. (Modular verification of components in popular programming languages, however, does entail careful component design [9].)

This paper considers performance specification and modular verification issues introduced by the availability of multiple implementations to produce the same implementation-independent, functional behavior. The need for avoiding implementation bias in the specification and the importance of multiple component implementations have been noted by most language designers [5, 6, 7, 15, 23, 24, 28]. Given a functionality, normally it is not possible to develop one implementation that is best in all respects. Inevitable trade-offs such as those in the performance of different operations provided by a component, or between average and worst-case time behaviors, or between time and space argue for the need to have several components to provide the same abstract specification of functionality. Clients then have the opportunity to choose from and switch among these implementations for performance reasons. (Other than for performance reasons, implementations may need to be switched for such reasons as software evolution, maintainability, and portability [21].)

To specify performance of different implementations, oftentimes the abstract model used in the specification of functionality is not sufficiently expressive. While the abstract models usually permit expression of loose performance constraints, specification of tighter constraints demand intermediate models that are more implementation-oriented. In addition, different classes of implementations may need different intermediate models for tight specification of performance. In general, there is no one model that is suitably abstract for functionality specification and at the same time is sufficiently expressive for performance specification of all implementations. *Expressiveness* (and hence, tightness) is the first reason for why implementation-dependent specification module(s) in between abstract functionality specification and implementations are needed.

Implementations of *generic* software components provide an additional motivation for the separation [21]. Different implementations of generic data abstractions need different module-level generic parameter operations (on the generic type), and these implementation-dependent

* This research is funded in part by ARPA Grant DAAH04-94-G-0002, NASA Grant 7629/229/0824 and NSF Grant CCR 9204461.

operations (and related mathematical definitions) belong in the implementation-dependent specification module.

Separating implementation-independent functionality and implementation-dependent performance specifications results in important verification cost savings as well. When performance aspects of a client component change (such as when it switches among functionally plug-compatible component implementations), ideally only its performance should need to be re-verified; assertions generated from functionality verification should be reusable.

The rest of the paper is organized into the following sections. Section 2 describes a framework for separating implementation-independent and implementation-dependent aspects of a component specification. Section 3 presents examples to illustrate the conflict between tightness of performance specification and modular verification, and explains the need for implementation-independent specification module(s). Section 4 has a summary of related work. Section 5 has our conclusions.

2. The Framework

This section describes a framework for a component-based reuse technology for software development that parallels that of the traditional engineering disciplines. Figure 1 shows how we consider a software system to be structured starting from a representative component, named Client. This structure is close in spirit to the "3C reference model" described in [3, 25], and is an extension of the RESOLVE framework for functionality [20].

In Figure 1⁰, circles denote implementation-independent specifications of functionality, rectangles denote implementations, and the "tombstones" denote implementation-dependent specifications of performance (in general, constraints). A1 is the functionality specification of a concept used by Client (denoted by a thin arrow). P1 and P2 are implementation-dependent specifications for two (not necessarily disjoint) sets of implementations for A1. P1 and P2 depend on A1, and are expressed in the context of A1 as can be seen from the solid arrows. Implementation C11 satisfies constraints P1; Though C12 and C13 are both satisfy P2, only C12 satisfies the tighter constraints P2¹. These

⁰ This Figure does not include a direct notion of concept or constraints inheritance; our more general framework and proof system do include the use of specification-based inheritance. Also, constraints can be seen as inheriting functionality specification and an implementation can be seen as inheriting both specifications.

¹ In the discussions in the rest of this paper, we consider only one implementation-dependent constraints specification module between a concept and an implementation. In general,

implementations themselves may have been built layered using other components such as one providing the concept A2.

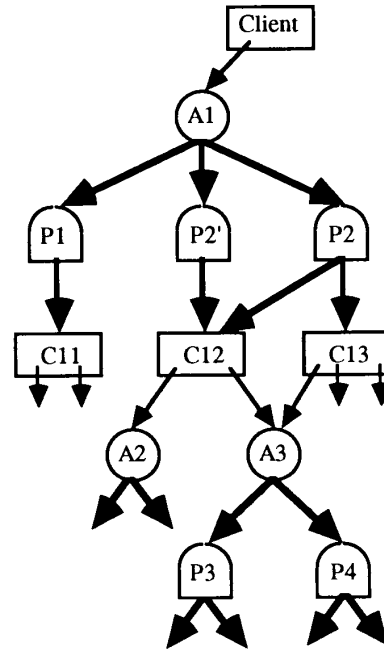


Figure 1 — Implementation-Independent and -Dependent Specifications and Implementations in a Software System

A language for construction of software components that can be verified for functionality and performance in a modular fashion must somehow support each "piece" in Figure 1. Details of a formal verification system for functionality and performance based on the framework are given in [13]. A summary of the essential syntactic extensions for Ada, for example, can be found in [21].

Reuse of a component in this setting is based on both the specifications of functionality and performance. The implementation is treated as a black box; any information within an implementation cannot be and need not be accessed by a user program or in verification of the user program. All client adaptations are through parameters. In the figure, functional correctness of Client is verified based on A1; verification of performance correctness is additionally based on P1 or P2 depending on the client's choice of constraints and implementations.

between a concept and a set of its implementations, there may be a network of such constraints specifications.

3. Implementation-Dependent Performance Specification Module

This section presents examples, and motivates implementation-independent and implementation-dependent specifications.

3.1 Example Specification of Implementation-Independent Functionality

Figure 2 shows a skeleton of an implementation-independent, specification of a data abstraction for Integer sets. We have intentionally used a non-generic version here; a generic version of the specification is in Section 3.3. (The complete specification of a more general concept `Partial_Map_Template` can be found in [21, 28].)

```
concept Int_Searching_Machine_Facility
  context
    global context
      Standard_Boolean_Facility
      Standard_Integer_Facility

  interface
    type Int_Set is
      set of integer
    exemplar s
    initialization
      ensures "s = {}"

    operation Add_Item (
      alters s: Int_Set
      preserves x: Integer)
    requires "x is not in s"
    ensures "s = #s union {x}"

    -- other operations: Remove_Item,
    -- Remove_Any_One_Item, Size
    ...

    operation Is_Member (
      preserves s: Int_Set
      preserves x: Integer
    ): Boolean
    ensures Is_Member iff
      (x is in s)
end Int_Searching_Machine_Facility
```

Figure 2 - An Implementation-Independent, Functionality Specification

The specification in Figure 2 has been expressed using RESOLVE notations [8, 20, 22, 27]; RESOLVE also includes a framework and a discipline for component construction. For the present discussion, it does not actually matter which formal specification approach is used in expressing functionality. It can be expressed in

any formal specification language such as those discussed by Wing in [28]. Only the form of the specifications and the types of information presented in the specification are of importance for the current paper; syntactic details can be safely ignored.

The implementation-independent specification in RESOLVE, denoted by the key word **concept** has two parts: the **context** section lists imports and the **interface** section describes the exports provided by the component to the client. RESOLVE encourages the use of **parametric context** in general over **global context**; the first idea in the paper is more easily explained without considering parametrization and hence the present concept is not parameterized.

It is important to note that the specification in Figure 2 has been presented without any implementation bias. Here the type `Int_Set` is modeled as a mathematical set of mathematical **integer**. (This modeling has nothing at all to do with the representation that may be used to implement a set.) Each set operation has been explained using symbols from set theory. Initially, every set is modeled by the empty set as specified in the initialization assertion. Two clauses are used in the formal specification of each operation: a **requires** clause (pre-condition) and an **ensures** clause (post-condition). 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 incoming parameter values.)

The parameter modes **alters** and **preserves** are specification notations. Making **x** to be a **preserves** parameter has the same effect as adding the conjunction "**x = #x**" in the post-condition. The value of an **alters** parameter is provided in the post-condition.

3.2 Tight Specification of Implementation-Dependent Performance Constraints

`Int_Searching_Machine_Facility` 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 ordered linear search, (balanced) binary search trees, and hash tables are examples of such techniques that can be used in creating an implementation for the set concept. Careful performance analysis of possible implementations reveals interesting compromises to be made [11].

For purposes of illustration, we limit the discussion in this paper to two implementations: (1) an ordered list-based implementation and (2) an unordered list-based

implementation. These implementations are indeed quite similar from a performance perspective, but they are sufficient to illustrate the issues. (Though these examples involve only linear or constant time expressions, specification and verification based on more complex time expressions pose no special problems [12].) Consider the client code for `Int_Searching_Machine_Facility` in Figure 3, where 's' is an Integer `Int_Set`, 'x' is an Integer, and 'answer' is a Boolean.

```
Assume s = {1, 2, 4, 7, 9} and x = 4
answer := Is_Member(s, x)
Confirm answer = true
```

Figure 3 — Example Client Code

It is easy to see that the above assertive code is functionally correct. It is also possible to show that the execution time (one aspect of performance) for the above code is $O(|s|)$ where $|s|$ is the size of the Integer set, irrespective of whether ordered or unordered implementation is chosen by the client.

While worst-case bounds are adequate for some client applications, real-time applications demand tighter bounds. Much attention in the real-time systems community has therefore focused on tightness issues [4, 16]; More theoretical real-time research has also addressed tightness because it is a completeness issue [13, 17].

If performance is specified formally for a (class of) realizations then it becomes possible to formally prove the implementations meet those performance bounds [12, 13]. Tight verification of performance, however, requires tight specification of reused components.

Tight Bounds for Ordered Implementations: Specification of tighter execution time bounds, in general, depends on details of the chosen implementation for `Int_Searching_Machine_Facility`. However, basing client reasoning on implementation details will compromise modular reasoning. Introduction of an intermediate specification module provides a solution to both these tightness and modularity problems.

The intermediate, implementation-independent specification shown in Figure 3, denoted by the key word **constraints** has two parts: the **context** section lists imports and the **interface** section describes the exports provided by a class of realizations. These are in addition to the imports and exports listed in the concept for which the constraints are being specified. (The use of **parametric context** for a constraints specification module is exemplified in the next sub-section.)

Figure 3 shows tight constraints specification for a class of order-based implementations. In this specification, a local mathematical definition "LEQ_SET" has been introduced. LEQ_SET is a subset of the set that models a store 's' and it includes items in 's' that satisfy

the relation R with the given item 'x'; R is the ordering used within the implementation.

```
constraints Order_Based for
  Int_Searching_Machine_Facility is
  context
    local context
      -- definition of ordering R
      math operation LEQ_SET (
        s: set of integer
        x: integer
      ): set of integer
      definition {y: integer |
        y is in s and R(y, x)}

  interface
    operation Add_Item (
      alters s: Int_Set
      preserves x: Integer)
    time expression
      A * |LEQ_SET(s, x)| + B

    -- other operations: Remove_Item,
    -- Remove_Any_One_Item, Size
    ...
    operation Is_Member (
      preserves s: Int_Set
      preserves x: Integer
    ): Boolean
    time expression
      D * |LEQ_SET(s, x)| + E
  end Order_Based
```

Figure 4 - Tight Constraints Specification for a Class of Ordered Implementations

Performance in Figure 4 is shown to include only a **time expression** that specifies the execution time interval for an operation; in general, space and other performance constraints will also belong in the interface section. All constants in the figure are intervals. The time interval may be hard such as <3.4ms, 4.2 ms> in which case satisfaction of the constraints may require certain hardware and compiler choices for the implementation. If the interval is expressed in terms of parameters (including hardware aspects), then the implementation is less restricted.

Based on this specification, the time taken for the client code in Figure 3 can be shown to be $3D + E$ (instead of the upper bound $5D + E$) because item 4 is third when the usual less-than-or-equal "ordering" of Integers is used to store items in this implementation. Such modular verification is possible from the (functionality and performance) specification-based proof rules such as for procedure calls given in [12].

Expression of tight, implementation-dependent performance for the ordered-list implementation is relatively simple because its performance can be expressed using sets — the same mathematical model used in its functionality specification. The *conventions* used in the assumed ordered list implementation have been apparently designed so that there is a single representation value *corresponding* to an abstract store value, i.e., the correspondence is “invertible.” For the present case, given a set of items, there is exactly one way of ordering the items (by the definition of R which is a total ordering). The “invertibility” property is what makes tight, implementation-dependent performance specification readily possible based on the implementation-independent set model. In general, strict invertibility is not required, as long as all representation values corresponding to a given abstract value are “performance equivalent.”

Tight Bounds for Unordered-List Implementation(s): Tight performance specification of the unordered list representation and verification based on that specification are considerably more complex. This is because for a given abstract set value, there are several corresponding representation values which are *not* performance-equivalent. For an Integer Set variable with the abstract set value {1, 2, 4, 7, 9}, the corresponding representation value may be any permutation of the 5 items. The permutation depends on the order in which the items were added to the set; the exact time taken to search for an item in turn depends on that permutation. For this reason, unless the constraints specification introduces an “intermediate” abstract model (e.g., mathematical strings) which preserves the order in which the items are added to the Int_Set, tight performance specification is not possible. An explanation of the issues involved in introducing an intermediate model in the constraints specification is beyond the scope of this paper, and are detailed in [13].

Different classes of implementations, in general, will need different intermediate models for expressive performance specification. Binary search tree-based implementations, for example, will need different models for tight specification. While these intermodels are usually more abstract than the representations used in the implementations themselves (e.g., pointers), they are definitely not sufficiently abstract for functionality specification. Separation of implementation-dependent (performance) specification and implementation-independent (functionality) specification is the only solution for keeping the latter *fully abstract* [27] and the former expressive for performance. It must be noted that assertions generated from functionality verification cannot be entirely reused in verification of performance when intermediate models are introduced.

On the surface, though there is not much difference in the performance between the ordered list and unordered list

implementations based on upper bounds listed in Figures 3 and 4, the ordered-list implementation is clearly easier to specify, use, and verify when tightness is a concern. In fact, it turns out that even though a “binary search tree” implementation of Int_Searching_Machine_Facility is more efficient than the ordered list implementation, its performance is not tightly predictable unless the implementation follows additional unusual conventions.

3.3 Generic Constraints Specifications

The interface of the constraints specification module constrains component developers to provide additional guarantees such as for performance. When generic software components are implemented, the developers may also have additional requirements of the clients. These requirements are specified in the parametric part of the context section. The context section for the generic Searching_Machine_Template (generalization of Figure 2) is shown in Figure 5. In the interface section, **math[Item]** denotes the mathematical model of the parametric type Item.

```

concept Searching_Machine_Template
context
  global context
    Standard_Boolean_Facility

  parametric context
    type Item

interface
  type Set is set of math[Item]
  ...
end Searching_Machine_Template

```

Figure 5 - A Generic Functionality Specification

Different implementations a generic concept will usually require different implementation-dependent, module-level parameters from the client, and will ensure different performance behaviors in turn. For example, implementations of Searching_Machine_Template based on a hash table search will need a hash function on the generic type Item as a parameter. Another class of implementations using binary search trees or ordered lists will require as a parameter an operation to determine the order of Item values. Importantly, different implementations of the same functionality specification will need different generic operation parameters [21]. (Because the client provides the generic type Item, only she can provide these operations.)

These implementation-dependent requirements and guarantees are stated in the **context** section of the **constraints** specification module. The **parameteric**

context section lists the items that must be supplied by a client (in addition to the concept parameters) to use one of the implementations satisfying these constraints. Figure 3 shows an implementation-dependent specification module for the class of implementations of `Searching_Machine_Template` such as ones based on an ordered list representation for a Set.

```

constraints Order_Based for
  Searching_Machine_Template is
context
  parametric context
    math operation R (
      x: math[Item]
      y: math[Item]
    ): boolean
    definition
      forall z: math[Item],
      R(x, x) and
      (R(x,y) and R(y,x)
       implies x = y) and
      (R(x,y) and R(y,z)
       implies R(x,z)) and
      (R(x,y) or R(y,x))

    operation Are_Ordered (
      preserves x, y: Item
    ): Boolean
    ensures Are_Ordered iff
      R(x, y)
    time expression Exp(x,y)

  local context
    ...

  interface
    ...
end Order_Based

```

Figure 5 - Parameterized Constraints Specification for a Class of Generic Ordered Implementations

The parameters section notifies the user that she needs to pass an operation that satisfies the specification — syntax and semantics — of the formal operation parameter `Are_Ordered`. The component in turn will satisfy the performance (time, in this example) listed in the interface section. The performance of operations provided by the component such as `Add_Item` will, of course, depend on and be parameterized by performance of the operations provided by the user such as `Are_Ordered`.

In writing the formal specification of `Are_Ordered`, we need a *mathematical definition* of `R` on (the mathematical model of) the type `Item`. The definition of `R` itself is the first parameter which forces the user to pass a total

mathematical ordering on the type `Item`; this definition must in turn be ensured by the operation that the user supplies as the second parameter. This example illustrates that languages desiring to support formal, modular verification of reusable software components must allow mathematical entities to be defined and passed as arguments, in addition to programming objects and operations. For other examples of parameterized constraints specifications, see [13, 21].

4. Related Work

To our knowledge, no previous research effort has tackled the expressive specification and modular verification of performance of object-oriented software components. Most related efforts have restricted themselves to a subset of the following sub-topics: languages, functionality or time specification, functionality or time verification. The importance of modularity in (functionality) verification and the impracticality of non-modular verification are emphasized in [2].

Language efforts have concentrated mostly on separation of functionality and implementation details. Though most language designers have noted the importance of multiple implementations for a given specification [6, 15, 24, 28] and parametrized programming [5], neither multiple implementations nor the idea of a “performance” module have been adequately addressed. Efforts in the formal specification community have also mainly concentrated on functionality. Wing provides an excellent survey of the current research efforts in [28].

Mary Shaw has noted the importance of and an approach for performance specification and verification as early as in [19]; while this technical report itself concentrates on statement-level verification within a module, it concludes with a list of key research issues to be addressed in module-level performance verification of generic data abstractions and multiple implementations.

An excellent summary of real-time verification issues can be found in [17, 26]. Alan Shaw’s system [18] that extends the Hoare system to handle time verification is a practical time verification system, but is based on worst-case bounds. More recently, achieving tighter time bounds for sequential parts of concurrent programs using execution-based (instead of verification) approaches has received attention. Most theoretical work, though concentrate on concurrency and limit reasoning to variables to simple ones such as Integers, note the importance of establishing tight bounds for sequential parts at least implicitly [17]. Nirkhe and Pugh explain a partial evaluation-based approach whereby loops and recursive calls are unrolled to achieve tighter time estimates [16]. Gupta and Gopinath note that scheduling of sequential processes based on worst-case run-time

estimates results severe under-utilization of processor resources, and a provide a run-time technique for establishing better than worst-case bounds [4]. Our concentration on generic data abstractions, multiple implementations, modularity and relatively complete verification distinguishes our contributions from most of these efforts.

Research on automated implementation synthesis has considered using performance constraints to limit possible alternatives [14]. When there are more than one intermediate, performance specification module between a functionality specification and one of its implementations, the issues can also be related to the use of "refinements" [23]; however, the purpose of the intermediate modules are quite different.

Though we have restricted our attention to time in this paper, other performance constraints such as space bounds raise interesting questions as well [2]. Space verification is important to avoid "memory leaks" and thus permit modular, local verification [9]. It is possible to handle space in the framework described in this paper. Verification of space, however, is subtly different from that of time because utilized space may both increase and decrease during the execution of a program.

5. Conclusions

Modular verification is important for software reuse to succeed as a viable idea. Performance verification of software components must be modular, yet tight. This paper outlines the essential elements of a system for modular verification of functionality and performance, which includes separation of implementation-independent functionality and implementation-dependent performance specifications of software components. The separation is noted to be either inevitable and/or profitable. While tight performance specification (leaving functionality specification to be fully abstract) makes implementation-dependent specification module(s) necessary, potential verification savings from re-verification of only performance aspects in the face of implementation changes makes the separation profitable.

Acknowledgments

It is a pleasure to thank the anonymous referees for their valuable comments which have helped clarify details of this paper. My thanks are also due to Joan Krone, Bill Ogden, Bruce Weide, Stu Zweben, and other members of the Reusable Software Research Groups at The Ohio State University and West Virginia University.

References

- [1] *Software Reusability, Volume 1: Concepts and Models, Volume 2: Applications and Experience*, Eds. T. Biggerstaff and A.J. Perlis, Addison-Wesley (1989).
- [2] G. W. Ernst, R. J. Hookway, and W. F. Ogden, "Modular Verification of Data Abstractions with Shared Realizations," *IEEE Trans. Software Engineering* 20, no. 4, April 1994, 288-307.
- [3] B. Frakes, L. Latour, and T. Wheeler, "Descriptive and Prescriptive Aspects of the 3Cs Model — SETA Working Group Summary," *Third Annual Workshop — Methods and Tools for Reuse*, Syracuse, 1990.
- [4] R. Gupta and P. Gopinath, "Correlation Analysis Techniques for Refining Execution Times of Real-Time Applications," *Proc. RTOSS'94*, Seattle, WA, 1994, 54-58.
- [5] J. Goguen, "Reusing and Interconnecting Software Components", *IEEE Computer* 19, 2, February 1986, pp. 16-28.
- [6] D. Gries, and D. Volpano, "The Transform — a New Language Construct", *Structured Programming* 11, 1990, pp. 1-10.
- [7] J. V. Guttag and J. J. Horning, *Larch Languages and Tools for Formal Specification*, Springer-Verlag 1993.
- [8] 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.
- [9] J. Hollingsworth, *Software Component Design for Reuse: A Language Independent Discipline Appl to Ada*, Ph. D. Thesis, The Ohio State Unvers 1992; available via anonymous FTP from ftp.cis.ohio-state.edu in directory pub/tech-repo
- [10] J. C. Knight, *Issues in Certification of Reusab. Parts*, Technical Report TR-92-14, Dept. of Computer Science, University of Virginia, May 1992.
- [11] D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1968.

- [12] J. Krone and M. Sitaraman, "On Modularity and Tightness," *Procs. of the 10th IEEE Real-Time Workshop on Operating Systems and Software*, May 1993.
- [13] J. Krone and M. Sitaraman, *Expressive Specification and Modular Verification of Performance of Generic Data Abstractions*, Technical Report 94-4, Dept. of Statistics and Computer Science, West Virginia University, Morgantown, July 1994.
- [14] R. D. McCartne, "Synthesizing Algorithms with Performance Constraints," *Procs. 6th National Conf. on AI*, Seattle, WA 1987, 149-154.
- [15] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.
- [16] V. Nirkhe, "A Partial Evaluator for the Maruti Hard Real-Time System," *Proc. 12th RTSS*, San Antonio, TX, December 1991, 64-73.
- [17] Proceedings of the REX Workshop "Real-Time: Theory in Practice," J. W. de Bakker, C. Huizing, W. P. de Roever, G. Rosenberg, eds., *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1991, 618-639.
- [18] A. Shaw, "Reasoning About Time in Higher Level Language Software", *IEEE Trans. Soft. Eng.* 15, 7 (1989), 875-889.
- [19] M. Shaw, *A Formal System for Specifying and Verifying Program Performance*, CMU-CS-79-129, Computer Science, CMU, Pittsburgh, PA, 20pp.
- [20] Special Feature on RESOLVE, *ACM Software Engineering Notes* 19, October 1994.
- [21] M. Sitaraman, "A Class of Mechanisms to Facilitate Multiple Implementations of A Specification", In *Proc. Fourth IEEE Int. Conf. on Computer Languages*, April 1992, 272-281.
- [22] M. Sitaraman, L. W. Welch, and D. E. Harms, "On Specifications of Reusable Software Components," *International Journal of Software Engineering and Knowledge Engineering* 3, No. 2, June 1993.
- [23] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.
- [24] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Menlo Park, CA, 1986.
- [25] W. Tracz, "Where Does Reuse Start?", *ACM Software Engineering Notes* 15, no. 2, 1990, 42-46.
- [26] Special issue on Real-Time Specification and Verification, *IEEE Trans. on Soft. Eng.*, September 1992.
- [27] B. W. Weide, W. F. Ogden, and M. Sitaraman, "Improving Reusability by Recasting Algorithms As Objects", *IEEE Software* 11, September 1994.
- [28] J.M. Wing, "A Specifier's Introduction to Formal Methods", *IEEE Computer* 23, 9 (1990), 8-24.