INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Mechanisms and methods for performance tuning of reusable software components

Sitaraman, Muralidharan, Ph.D.

The Ohio State University, 1990
MECHANISMS AND METHODS FOR PERFORMANCE TUNING OF REUSABLE SOFTWARE COMPONENTS

A Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of the Ohio State University

By

Muralidharan Sitaraman, B. E. (Honors), M. E. (Distinction)

* * * * *

The Ohio State University
1990

Dissertation Committee
Dr. Bruce W. Weide
Dr. Wolfgang Kuechlin
Dr. Neelam Soundararajan
Dr. Stuart H. Zweben

Approved by

B. W. Weide, Adviser
Department of Computer and Information Science
To My Parents
ACKNOWLEDGMENTS

It is a pleasure to acknowledge the contributions of Dr. Bruce Weide throughout this research. I express my sincere appreciation for his often insightful comments, guidance, and constant encouragement and enthusiasm. I would also like to thank my committee members Dr. Wolfgang Kuechlin, Dr. Neelam Soundararajan, and Dr. Stuart Zweben for their careful reading of the dissertation and their helpful suggestions. Thanks are also due to the members of the Reusable Software Research Group for several useful discussions. In particular, I offer my sincere gratitude to Dr. William Ogden for his constructive criticisms that helped shape several key ideas found in the dissertation. I thank my father and my brothers, Suresh and Vijay, for first motivating this important undertaking and then for their patience and support during my moments of frustration. I thank Susan, my fiancee, for her understanding and for her willingness to endure with me the vicissitudes of this endeavor.
VITA

June 7, 1960 — Tuticorin, TN, India

1983 B. E. (Honors), University of Madras, Madras, India

1985 M. E. (Distinction), Indian Institute of Science, Bangalore, India

1985 - 1986 Software Engineer, Madras Computer Laboratories, Madras, India

1986 - present Graduate Teaching Associate, Department of Computer Science, The Ohio State University, Columbus, OH, USA

PUBLICATIONS


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


FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in data structures and algorithms, programming languages, software engineering, and some aspects of artificial intelligence and computer science theory
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>VITA</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
</tbody>
</table>

## I. INTRODUCTION
- 1.1 The Likely Nature of a Software Components Industry | 3 |
- 1.2 Controlling Factors That Influence Performance of Software |
  - 1.2.1 Three Performance Factors | 5 |
  - 1.2.2 Controlling the Major Performance Factors | 9 |
- 1.3 The Thesis |
  - 1.3.1 Characterization of a Reusable Software Component | 10 |
  - 1.3.2 Methods for Tuning Performance of a Reusable Component | 16 |
- 1.4 Outline of Dissertation | 19 |

## II. FRAMEWORK FOR SUPPORTING A COMPONENT INDUSTRY | 20 |
- 2.1 Conceptualizations |
  - 2.1.1 The Parameters Section | 24 |
  - 2.1.2 The Auxiliary Section | 24 |
  - 2.1.3 The Interface Section | 25 |
  - 2.1.4 The Description Section | 29 |
- 2.2 Realizations |
  - 2.2.1 Using a Component | 31 |
  - 2.2.2 Program Type Equivalence | 33 |
2.3 Implementation Independence in Specifications ............................................ 34
2.4 Extending Components with Additional Operations ............................... 36
   2.4.1 Primary and Secondary Operations .................................................. 36
   2.4.2 Mechanisms for Developing Reusable Secondary Operations ............. 39
2.5 Support for Construction of Efficient Components ............................... 44
   2.5.1 Problems with Copying As a Basic Data Movement Mechanism .......... 44
   2.5.2 Swapping As a Primitive Data Movement Mechanism .................... 45
   2.5.3 Influences of Swapping on the Design of Reusable Components ......... 47
2.6 Object Code Generation ......................................................................... 47
2.7 An Example to Illustrate Performance Compromises ............................. 49
   2.7.1 A Conceptualization for the Associative Searching Problem ............... 50
   2.7.2 Analysis of Performance Characteristics ......................................... 55
   2.7.3 A Realization Based on Linear Searching ........................................ 56
   2.7.4 A Realization Based on Ordered Searching ..................................... 65
   2.7.5 A Realization Based on Binary Search Trees ................................. 68
   2.7.6 A Realization Based on Hashing .................................................... 69
   2.7.7 Standard and Sub-Standard Conceptualizations ............................... 73
   2.7.8 Performance Compromises ............................................................ 79
2.8 Other Languages for Constructing Reusable Software Components ......... 79
   2.8.1 Ada .............................................................................................. 80
   2.8.2 CLU ............................................................................................ 84
   2.8.3 Eiffel ........................................................................................... 86
2.9 Summary .................................................................................................. 88

III. PARAMETRIZING PERFORMANCE OF SOFTWARE COMPONENTS ...... 89
3.1 The Basic Idea and Mechanisms for Performance Tuning ..................... 90
   3.1.1 An Example .................................................................................. 90
   3.1.2 Passing Parameter-less Realizations as Parameters to Realizations ....... 93
3.2 General Mechanisms for Constituent Performance Tuning ................. 96
3.2.1 An Example ................................................................. 96
3.2.2 Realization Forms .......................................................... 102
3.2.3 Mechanisms for Passing Realizations with Selected Forms As Parameters ......................................................... 105
3.2.4 Enhancing The Generality of a Tunable Realization .............. 111
3.2.5 On Passing Realizations Without Forms As Parameters ......... 116
3.2.6 A Comprehensive Example Showing Performance
Parametrization ........................................................................ 118
3.3 Reshaping of Realizations ...................................................... 119
3.4 Summary ............................................................................. 123

IV. DISTRIBUTED EXECUTION OF SOFTWARE COMPONENTS .......... 124
4.1 An Example ......................................................................... 126
4.2 Elements of the Proposed Execution Method ......................... 131
  4.2.1 Distribution on Physical Machines ..................................... 131
  4.2.2 Distribution on Logical Machines ....................................... 132
  4.2.3 The Proposed Execution Method ....................................... 133
  4.2.4 Parallel Execution ........................................................... 142
  4.2.5 Discussion ....................................................................... 148
4.3 Why Representations of Data Values Should Not Be Moved ........ 149
  4.3.1 Review of Representation Movement Methods in Other
       Distributed Systems ............................................................. 149
  4.3.2 Why Representations of Data Values Should Not Be Moved .... 153
4.4. Summary ............................................................................ 158

V. CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH ....... 159
  5.1 Contributions .................................................................... 160
  5.2 Directions for Future Research ............................................. 164
  5.3 Conclusions ...................................................................... 166

APPENDIX ................................................................................ 167

BIBLIOGRAPHY ........................................................................... 176
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 — Client, Component and Constituents</td>
<td>6</td>
</tr>
<tr>
<td>2.1 — A Conceptualization of Stack_Template</td>
<td>23</td>
</tr>
<tr>
<td>2.2 — Performance Information for Stack_Real_1</td>
<td>31</td>
</tr>
<tr>
<td>2.3 — Example Client Code</td>
<td>32</td>
</tr>
<tr>
<td>2.4 — A Specification of Stack_Reverse Operation</td>
<td>40</td>
</tr>
<tr>
<td>2.5 — Instantiation and Use of Stack_Reverse_Capability</td>
<td>41</td>
</tr>
<tr>
<td>2.6 — One Possible Realization of Stack_Reverse</td>
<td>42</td>
</tr>
<tr>
<td>2.7 — A Conceptualization for Associative Searching</td>
<td>53</td>
</tr>
<tr>
<td>2.8 — A Realization Based on Linear Search for Partial_Map_Template</td>
<td>58</td>
</tr>
<tr>
<td>2.9 — A Client of Linear_Search</td>
<td>60</td>
</tr>
<tr>
<td>2.10 — Information Supplied to a Client of Linear Search</td>
<td>64</td>
</tr>
<tr>
<td>2.11 — Information Supplied to a Client of Ordered_Search Realization</td>
<td>66</td>
</tr>
<tr>
<td>2.12 — A Client of Ordered_Search</td>
<td>67</td>
</tr>
<tr>
<td>2.13 — Information for a Client of Binary_Search Realization</td>
<td>68</td>
</tr>
<tr>
<td>2.14 — Information Supplied to a Client of Hashed Search</td>
<td>71</td>
</tr>
<tr>
<td>2.15 — A Conceptualization for Unique Id Assignment</td>
<td>75</td>
</tr>
<tr>
<td>2.16 — A Realization Using a Facility from Id Assignment</td>
<td>76</td>
</tr>
<tr>
<td>2.17 — A Client of C_Real</td>
<td>77</td>
</tr>
<tr>
<td>2.18 — Specification of a Generic Package Providing the Encapsulated Type Stack</td>
<td>83</td>
</tr>
<tr>
<td>3.1 — A Fragment of Linear Search for Partial_Map_Template</td>
<td>91</td>
</tr>
<tr>
<td>3.2 — A Performance Tunable Realization for Partial_Map_Template</td>
<td>94</td>
</tr>
<tr>
<td>3.3 — A Client of Tunable_Linear_Search</td>
<td>95</td>
</tr>
<tr>
<td>3.4 — A Conceptualization for a Two-Dimensional Array</td>
<td>98</td>
</tr>
<tr>
<td>3.5 — A Realization of AC_2D_Array_Template Using Partial_Map_Template</td>
<td>101</td>
</tr>
<tr>
<td>3.6 — An Example Realization Form</td>
<td>103</td>
</tr>
</tbody>
</table>
3.7 — Usage of a Realization Form ................................................................. 103
3.8 — Another Realization Form for Partial_Map_Template ............................... 104
3.9 — A Realization Form for AC_2D_Array_Template ....................................... 107
3.10 — A Tunable Realization of AC_2D_Array_Template Using Partial_Map ...... 109
3.11 — A Client of AC_2D_Array_Template .................................................. 110
3.12 — Modification of Tunable_AC_2D_Array_Real .......................................... 111
3.13 — A Realization Form for AC_2D_Array_Template ...................................... 113
3.14 — Modification of Tunable_AC_2D_Array_Real .......................................... 114
3.15 — A Client of AC_2D_Array_Template Using Tunable_Linear_Search ........... 115
3.16 — A Modified Realization Form for AC_2D_Array_Template ..................... 117
3.17 — Creating a Specific Version from a Tunable Version ............................... 120
3.18 — Another Realization Form for Partial_Map_Template ............................... 121
3.19 — Reshaping a Realization to Suit a Specific Form .................................... 122
4.1 — Specification of Queue_Template ......................................................... 128
4.2 — Specification of Queue_Reverse Operation .............................................. 128
4.3 — One Possible Realization of Queue_Reverse ........................................... 129
4.4 — A Tunable Component That Uses Queue_Reverse_Capability .................... 130
4.5 — An End User ......................................................................................... 131
4.6 — Distribution of Components on Logical Machines .................................... 134
4.7 — Distribution of Facilities on Logical Machines ........................................ 135
4.8 — Remote Call to an Initialize Operation .................................................. 137
4.9 — Remote Call to a Secondary Operation .................................................. 140
4.10 — Remote Call to a Primary Operation ..................................................... 141
6.1 — A Conceptualization for Integer_Template ............................................. 169
6.2 — A Conceptualization for Record_2_Template .......................................... 170
6.3 — A Conceptualization for Array_Template ............................................... 173
6.4 — A Conceptualization for One_Way_List_Template .................................... 175
CHAPTER I
INTRODUCTION

Software demand and costs have increased steadily to the point where the software industry is expected to become the largest industry in the U.S. [Yourdon 86]. Software costs amount to 85% of the total cost of a typical computer system. However, the quality of software products has not kept pace with the cost, and software products generally are not delivered on time [Brooks 82]. Consequently, two of the most important problems facing software engineers are how to improve the quality of their software products, and how to improve the productivity of people who build them. A mainstay of traditional engineering — reusability — attacks both these problems.

There is, indeed, considerable room for the transfer of general purpose engineering design and management techniques to software design and development [McIlroy 76, Spector 86]. At the same time, there are notable differences between software and products in other engineering disciplines, and these pose several technical and non-technical obstacles to the development of reusable software [Parnas 85, Stovsky 90].

In our research in the Reusable Software Research Group at The Ohio State University, for the last several years we have been investigating solutions to many of the technical impediments to constructing software largely from reusable software components. For widespread software reuse, a software component should have a formal specification,
certification of correctness, and efficient performance [Ogden 90, Weide 86a]. We have been developing a conceptually robust framework addressing several related issues including these. Our framework is the basis for a programming system and paradigm called RESOLVE (for REusable SOftware Language with Verifiability and Efficiency).

This dissertation addresses the question of tuning the performance of a reusable software component to specific needs. Usually different applications of a reusable component have different performance requirements for that component, and often these requirements conflict. In general, it is not possible to construct components with performance characteristics satisfying every application's requirements. For widespread reuse, software components should be adaptable to a diverse set of applications with varying performance requirements.

What is needed is the ability for users of a component to tune its performance to their specific needs. One way to provide this ability is to supply the component's source code to its users, who can then modify it to alter performance [Weiser 87]. This approach, though it allows complete performance control for users, also thwarts the very idea of software reuse because it results in significant re-coding, re-compilation, re-validation, and maintenance costs. In other words, the ability to tune performance to specific needs should not come at the cost of software quality and productivity — the primary goals of software reuse. Source code modification is therefore not a desirable solution to this problem.

A more appropriate approach to the performance tuning problem that gives clients control over performance at little or no cost is suggested by the “software component industry” view proposed in the dissertation. We characterize the likely nature of a software component industry that will evolve when large-scale software reuse becomes technically
feasible, and argue that a reusable software component in this industry will be sold in an object-code form (meant to be readable only by a machine) along with performance information and a formal, implementation-independent specification. At the outset, this view poses the intriguing question of how to tune the performance of a component, but it also suggests an answer, as demonstrated in this dissertation.

1.1 The Likely Nature of a Software Components Industry

We posit a software components industry that resembles the current electronic components industry. A “client” programmer has one or more catalogs of standard reusable components. As in electronics, these catalogs will be offered by different “providers” or “manufacturers” of plug-compatible reusable parts. Each component in a catalog has a formal description of its structural interface and abstract functional behavior, sufficient to explain what it does and how it might be incorporated into a client’s system. The component is marketed to a client in a “sealed chip” form not revealing the internals, along with some price performance information. Typically, the component also has a certification that it reliably behaves according to its specification.

Many of the parts in a typical catalog are quite general — in fact “standard” in the sense that they have the same structural interfaces and functional behaviors as corresponding parts in other manufacturers’ catalogs. Parts having identical functionality provided by different manufacturers might differ in implementation details and therefore in performance and/or price, however. A client is able to substitute freely a component provided by one manufacturer for a corresponding component provided by another manufacturer.
This vision of a software component industry may seem so old [McIlroy 76, Cox 86] and/or obvious as to go without saying. On the other hand, it does not seem to be the underlying model in circles where source code is assumed as the basis for reuse (e.g., [Berard 87], [Krueger 89]). We claim source code reuse would neither be viable nor desirable under the above scenario. In our view, a marketed reusable software component will have the following characteristics:

- It will have a formal, implementation-independent, machine-processable yet human-understandable specification of its functionality.

- It will be sold in an object code form with a certification of correctness (that the given object code reliably implements the associated specification) and additional performance/price information.

The primary rationale for the proposed view of a component is the observation that traditional engineering disciplines have identified and used a similar notion of reusable components. This analogy to traditional engineering counterparts, however, is not the only justification for the above proposition. There are several other compelling reasons discussed later in this chapter.

Defining precisely the difference between source code and object code is an important legal as well as technical issue. **Source code** is a precise statement of concrete representational and algorithmic details written in a high-level language (e.g., Ada [DoD 83] or RESOLVE [Weide 90]) and it is meant to be both machine-readable and human-readable. **Object code** is the result of translation of source code by a compiler and is meant to be machine-readable
only. Looking at object code for a component, a human cannot easily decipher the details contained in its source code.

1.2 Controlling Factors That Influence Performance of Software

This section describes three factors that significantly affect a component's performance and discusses what would be involved in tuning these factors to suit specific performance needs. The view of the reusable component industry proposed here suggests an approach that permits a client to do such tuning without having to pay a big price for it.

1.2.1 Three Performance Factors

A software component is typically constructed using other available components which have been built using still other components. This hierarchical nature of component construction is illustrated in Figure 1.1. In this figure, a circle stands for an abstraction (explained with an implementation-independent specification) and a square stands for a component implementing that abstraction. A thick arrow pointing from a circle to a square indicates that the square is a component implementing the abstraction represented by the circle. There can be arrows from a single circle to several squares because multiple components can provide the same abstraction. For example A1 is an abstract specification provided by the components C11 and C12. A thin arrow from inside a component to an abstraction indicates that the component uses an instance of the abstraction. The same component may also use more than one instance of an abstraction. Before execution, one of the many components that implement the abstraction must be chosen. For example, the component C11 uses abstractions A2 and A3. C11 has a choice of components for each of these abstractions, and so forth through arbitrarily many layers down to the primitive
components that are implemented directly on the underlying virtual machine or physical hardware.

Figure 1.1 — Client, Component and Constituents
In the figure, "Client" is the name of the component currently under construction; specifically it is a client of the abstraction A1, i.e., it needs to use abstraction A1. What factors related to A1 can influence Client's performance?

*Algorithms and Data Structures Used in Implementing a Component.* The data structures and algorithms used in implementing an abstraction obviously affect the performance of that implementation. For example, an abstraction that involves sorting a set of values can be implemented using any of a number of sorting algorithms such as quicksort, mergesort, selection sort, and so on. Each different algorithm results in different performance characteristics, e.g., different average and worst-case time complexities [Knuth 68]. The differences in performance resulting from different algorithms are often differences in orders of magnitude.

For some clients, the average case may be of more interest, while to other clients the worst-case may be crucial. Furthermore, if a component provides more than one operation then there may be no algorithm that produces best performance for each of these operations. It is more likely that one algorithm has better performance for some subset of operations and another provides better performance for a different subset of operations.

In Figure 1.1, the component Client uses the abstraction A1. Both components C11 and C12 implement this abstraction. The client programmer can choose one of these components based on his or her performance requirements. It is important that the client code that uses the services of A1 *need not change* no matter which component that provides A1 is selected for use.
Implementations of the Constituents of a Component. Another factor that significantly affects the performance of a client is the performance of the constituents of a component used by the client. In Figure 1.1, let us assume that Client uses C11 for the abstraction A1. The constituents of C11 affect Client's performance as well. C11 uses abstractions A2 and A3. The components chosen to implement A2 and A3 affect C11's performance, and hence the performance of Client. There are two choices for each of the abstractions A2 and A3, and therefore as many as four performance possibilities for Client even after the decision to use C11. The constituents of C11 can also affect Client's performance by orders of magnitude, and hence Client should be able to tune this aspect of performance.

Hardware Architecture Used for Component Execution. The third major factor that affects the performance of a component is the hardware architecture where it is executed. A program may be required to execute on a hardware architecture for which some constituent components might not have been specially designed. How the hardware architecture is used (e.g., how the constituent components are assigned to machines in a distributed hardware environment) affects the execution efficiency of the component. A client must be able to control this aspect of performance as well.

Other Factors. There are factors other than the three discussed above which can affect a component's performance. For example, the specific programming language statements used in coding an algorithm will impact the performance of a component. Clever compiler optimizations can often produce more efficient code than a typical programmer can write manually. But these factors are lower-order effects when compared to the influences of the three primary factors discussed above. Furthermore, considerable attention has been given to these issues in the past, and most production compilers now include switches to turn on
various optimizations. Therefore there is not much need for clients to be able to control these factors through programming language mechanisms.

1.2.2 Controlling the Major Performance Factors

Though it certainly seems desirable that a client be able to control each significant aspect of performance, at what cost is this to be achieved? While supplying source code of a component to clients and allowing them to modify it makes performance tuning very expensive in terms of software quality, productivity, and long-term maintenance, the component industry view where only object code for the component is marketed raises the following issue.

The proposed view of a component industry suggests two distinct communities (rather, roles) of programmers: developers of reusable components and clients who use the components. The developer of one component may also be the client of another component. More importantly, the developer and client of a component may be two different persons.

While only the developer of a component knows about all the implementation aspects of the component contained in the source code, only a client knows which aspects of performance of the component are crucial to his or her application. It is the developer who can apparently exercise the most control over the performance of the component since the source code is available to him or her. It is, however, the client who needs to tune the performance to his or her needs. Thus, neither the developer nor the client has all the information needed to tune the performance of a component to a particular application.
In this context, what are the programming language mechanisms and methods needed for allowing clients control over the three major aspects of performance of a component at little cost? This is the question addressed by this dissertation. Given these mechanisms and methods, the client must still make certain decisions, such as which components (and which of its constituents) should be used to construct his or her component. The client must also decide questions such as how the components should be assigned to the machines in a distributed hardware architecture. The answers to these questions are, of course, based on the client’s performance needs, and may not be readily obvious. Research into these issues needs to be done. However, this dissertation does not attempt to show how a client can make such decisions. It is concerned with the mechanisms and methods for permitting client control over performance.

1.3 The Thesis

This dissertation defends the following thesis:

There are programming language mechanisms and methods to permit a client to control effectively the factors that significantly influence the performance of a reusable software component developed in the software component industry scenario.

The overall approach to providing performance tuning is based on the proposed component industry view. This view carefully separates the specification of an abstraction from implementation details, and requires that the specification be implementation-independent to permit multiple components (with different performance characteristics) to realize the same specification. Support for the component industry view directly provides a method to tune the first of the three performance factors discussed earlier — flexibility for a client to
choose among algorithms and data structures implementing an abstraction in order to match his or her performance needs. Methods for controlling the other two factors are also greatly influenced by the industry view and the framework discussed in this dissertation for effectively supporting the construction of reusable components in this industry.

The formality of the specification and the certification of correctness required of reusable components play an important role in the design of the mechanisms and methods proposed in this dissertation. Therefore we first examine the reasons for the proposed characterization of a marketed reusable component and industry. These reasons serve as motivation for the proposed solution to the other two factors of the performance control problem.

1.3.1 Characterization of a Reusable Software Component

Analogy to the traditional engineering disciplines, though it is the primary rationale for the proposed view, is not the only one. There are several other reasons for a marketed reusable software component to have a formal, implementation-independent specification of functionality and to be marketed in object-code form with a certification of correctness and additional performance information. There are also many important implications of the industry view. Both aspects are discussed below.

Formal Specification and Correct Implementations. What could be worse than not reusing software? For one thing, inappropriate software could be reused — software whose actual behavior is misunderstood by a potential client. For another thing, incorrect software could be reused — software whose actual behavior is simply not what the specification says it is supposed to be. It is most critical then, that a reusable software component have a
formal abstract functional specification which unambiguously states what the component does [Krone 88]. [Meyer 85] also discusses several problems with informal specifications.

Informal arguments about the correctness of component implementations can be made with only informal behavioral descriptions of the components. But any rigorous treatment of certification of logical correctness, certainly by mathematical verification but even by testing [Gourlay 83, Hegazy 89], requires formal specification [Guttag 78, Liskov 86, Bjørner 87]. [Krone 88] discusses proof rules for all the statements in a close ancestor of RESOLVE, and these rules have been used to show the correctness of several non-trivial reusable components. A suitable specification approach permits completely modular formal verification and testing.

Verification of correct functioning of a component is one place where the analogy between a software component and its traditional engineering counterparts does not hold. In the electronics industry, even when a component has been tested most rigorously, it might break down because of some physical defects arising during manufacturing or use. However, a piece of software can be mathematically proved to meet its specification, if the specification is formal. Once proved to meet its specification, a component will remain correct all its lifetime. This feature of software has important implications for the economic feasibility of formal verification of correctness, which impacts several issues addressed in the dissertation.

In the reusable component industry, we can even imagine a "standards" organization to certify that a marketed component does indeed meet its specification. Clients (at least careful ones) use only components that have the stamp of approval from the standards
organization. The approval may be based on formal testing rather than formal verification in which case it might provide a reliability factor for the component.

*Comprehensible Specifications.* Having formal specifications does not preclude having informal, natural language descriptions of behavior, too. Inevitably, though, a natural language description is ambiguous or unclear [Meyer 85]. The formal specification is always the true basis for the contract between the component provider and the client. Therefore, it is important to have a specification approach and language that are formal, yet highly readable and accessible to a typical client programmer. In the component industry view, the specification of a reusable component is read far more frequently than it is written. This "read-mostly" nature of specifications suggests some characteristics of such an approach. The specification author may be assumed to be an expert in writing formal specifications, but the client may not be so well-versed in formal methods. A specification approach that supports intuition and isn’t full of cryptic symbols, but is perhaps a bit more verbose than a minimal set of axioms, say, is desirable. RESOLVE uses a specification approach based on these principles.

*Implementation-Independent Specifications.* A client of a software component certainly needs both functional and performance information about the component. The specifications we have discussed so far refer only to functionality. Separating the functionality from performance information helps in reusing the same functional specification for components using different implementations and hence having different performance characteristics. It also helps in separating proofs of functional and performance correctness of software components. Hereafter unless stated otherwise, specification always means specification of *functionality*; *performance* specification is explicitly qualified.
In the component industry, abstract specifications of functional behavior for standard components can be used by multiple manufacturers (via a licensing agreement from the original designer, with the blessing of a standards organization, or whatever) to supply multiple components with the same specification. Price-performance criteria drive client purchases. This works in much the same way as when electronics component manufacturers provide both CMOS and TTL versions of a single kind of IC.

This argument suggests the need for an approach to specification that is truly abstract, so a client relies on no implementation-oriented details such as record formats or data representations. While some current specification languages and methods (e.g., ANNA [Luckham 87]) permit such abstraction, they also permit a component designer to write specifications directly in terms of implementation details. In RESOLVE, abstractions are specified using well-known mathematical theories, and are completely devoid of implementation details. It is not possible to write implementation-oriented descriptions of functional behavior in RESOLVE because programming notions like records and array are not part of the specification language.

Implementation independence of the specification makes it possible for a client to freely switch among the several components with that specification. Such replacements do not affect the functionality of the client program, meaning it does not have to be re-compiled or re-verified. This developmental independence is of great significance for the construction and maintenance of large software systems [Parnas 72].

Object Code Versions of Marketed Components. In the analogous electronics component industry, manufacturers do not publish or sell clients the masks for their implementations
of standard ICs, even though they literally give away catalogs containing interface specifications. They generally sell only sealed packages into which a client need not and should not look. We expect a software components industry to follow suit, assuming technical barriers to this approach are conquered.

A client should need to know only the specifications of a component to use it, and there is no reason why the source code of the component is needed. Advocates of source code reuse suggest that a reusable component may not provide some functionality as efficiently as a client may need, and hence the client should be able to access the source code to modify it to suit his or her needs [Meyer 88, Weiser 87]. In [Muralidharan 90], we have argued that in the context of a software component industry where components are designed for reuse and several components are available for providing the same functionality with different performance, this is usually not the case, and any inefficiency that results from this approach is only by a small constant factor. In general, modifications needed to the source code of a component to enhance its functionality or improve its efficiency (by better than a constant factor) are non-trivial, and they usually result in complete re-coding of the component.

We already know that object code reuse can work based on our experience with using library functions and procedures. The Macintosh toolbox, for example, demonstrates that reuse is possible in the object code form at levels of granularity higher than simple functions and procedures [Apple 85]. There is no compelling reason to believe that reliance on source code is essential for software reuse [Muralidharan 90].

There is also an important non-technical, legal reason for the above position. The internal data structures used by a component provider to represent various types and objects, as
well as the algorithms to manipulate them, are not patentable under current law. Therefore, it has recently become common for companies to register copyrights for object code versions of their programs and to maintain that the source code contains separately protected trade secrets [Tomijima 87]. Presumably such secrets would lose their protected status if provided to a client in a clearly revealed form, e.g., as source code. However, the current legal situation is so unstable that it is dangerous to buy source code as well as sell it, as this may limit the algorithms or data structures that the client can use in his or her own programs in the future without fear of being sued for royalty or other payments [Samuelson 88].

1.3.2 Methods for Tuning Performance of a Reusable Component

This section provides an outline of the mechanisms and methods proposed in the dissertation for allowing clients control over the three major factors that influence the performance of a component. The view of the software component industry being the basis of the proposed solutions, the dissertation first discusses how RESOLVE can effectively support each significant requirement of the component industry. Programming language mechanisms for separating the functional abstraction of a component from implementation details (and hence performance details), thus allowing several components with the same abstract specification are discussed. RESOLVE mechanisms also permit a client of an abstraction to freely choose and switch among the many components providing that abstraction.

Factoring Algorithmic Performance. Support for a software component industry directly provides control over this aspect of performance — flexibility for a client to choose a component to match his or her performance needs from among functionally equivalent
ones. Such flexibility is available to a client at little cost, with changes not to the functionality of the client but only its performance.

Tuning the Performance of Constituents of a Component. For clients to be able to control the performance of the constituents of a component, they must know what the constituents are and how their performance affects that of the component. In Figure 1.1, let us assume that the client programmer (builder of the component Client) has chosen to use C11 for the abstraction A1. To control the constituents used by C11, what must the client know? For one thing, the abstractions used by C11 (i.e., A2 and A3) must be known.

The dissertation proposes programming language mechanisms whereby the constituents used by a component can be made parameters to that component. For example, the parameters of the component C11 reveal that C11 uses A2 and A3, and that clients need to supply implementations for these abstractions as parameters to C11. Clients can now choose specific components for each of these abstractions, controlling the performance of the component within the bounds of the hidden algorithms used in C11. This approach definitely provides clients with more than ordinary object code of the component. The constituents used in the component are supplied as well. However, the manner in which these constituents are used (which forms the bulk of the source code) still remains hidden though clients can get some idea of their usage from performance statements that relate the constituents' performance to the component's performance. The proposed programming language mechanisms minimize the information that should be known to clients in order to have control over this aspect of performance.

Tuning a Component to Client Hardware Architecture. To tune a component's performance to a particular hardware architecture, the client must know performance-related
details of the architecture, such as the speed of each machine in a distributed architecture and the communication delays between each pair of these machines. What should be known about the component? The constituents of the component must be known, and more than that how much these constituents interact must be known.

The methods for allowing clients to choose constituent components can be used by clients to get (at least some of) the desired information. The parameters of a component describe the constituents, and the performance statement of how the constituents affect the component’s performance provides information about interaction among the constituents. No additional programming language mechanisms are proposed to give clients any more than this information because it is not clear if any additional information is needed. Clients can use this information along with the knowledge about the hardware architecture, and distribution methods such as in [Sadayappan 87] or automatic distribution tools, to assign components to the machines in the hardware architecture. This dissertation does not discuss how the assignment can be done. See [Welch 90b] for details.

What the dissertation does is to propose a method for execution, given some distribution. This method permits maximal distribution of representations of data values used by a program, and thus allows components to manipulate them in parallel effectively exploiting the potential for performance resulting from distribution. The method also provides an answer to the important question that arises from permitting clients to distribute components arbitrarily: How can the different components communicate data values without moving the representations of these values? Movement of representations is difficult or impossible in the component industry view proposed in the dissertation where only object-code versions of components are marketed and hence representational details are hidden. The method shows that by creating representations that come into existence
during execution on appropriate machines and allowing them to be distributed, they will never need to be moved. The data distribution resulting from the method permits extraction of parallelism at desired granularity in a uniform manner.

1.4 Outline of Dissertation

The remainder of the dissertation is organized as follows. Chapter II discusses a framework for supporting the software component industry view. As noted earlier, support for the industry directly provides clients the flexibility to choose one of the many available components for an abstraction. Chapter III discusses flexible programming language mechanisms for allowing a component, and even constituents of a component, to be parametrized. Chapter IV proposes a method that is independent of the distribution of hardware in the underlying architecture for the efficient execution of a software component. Chapter V presents conclusions and directions for future research.
A critical requirement for the successful evolution of a software component industry is a conceptually robust framework that promotes effective specification, design, and implementation of reusable software components. RESOLVE is a programming language and paradigm specifically designed for this purpose. The primary objective of this chapter is to show that RESOLVE permits the possibility of several components for an abstraction and allows clients of that abstraction to choose from among these components one that best suits their performance requirements. This chapter also discusses some other aspects of RESOLVE that are pertinent to this dissertation.

In RESOLVE, every software component has a formal, implementation-independent specification and one or more concrete implementations of the specification. The specification of an abstract component is termed a conceptualization. The human-readable source code, which includes concrete data structures and algorithms to implement a given conceptualization, is termed a realization. Each marketed RESOLVE component will therefore have a conceptualization, and a translated (object-code) version of some realization for it.

RESOLVE conceptualizations and realizations usually have parameters, i.e., they are generic. Conceptualizations are formal, which makes it possible to prove formally that a
realization correctly implements the corresponding conceptualization [Krone 88]. They are also implementation independent, and hence several different realizations are possible for the same abstraction. A RESOLVE conceptualization typically specifies only the functional behavior of a component and does not include any performance constraints. Different realizations of the same conceptualization have the same functional behavior, but can have different performance characteristics. A client of a RESOLVE component therefore depends only on the conceptualization of the component for its functionality. However, the performance of the client depends on which particular realization he or she chooses to use.

This chapter discusses how the RESOLVE framework provides all the technical machinery to support a software component industry. The chapter is organized into eight sections. The first four sections relate to specification, starting with methods for writing formal specifications in section 2.1 for a reusable component in RESOLVE. Section 2.2 discusses how a client can use a RESOLVE component. Section 2.3 discusses the mathematical modeling used in RESOLVE specifications that makes it possible to have implementation independent conceptualizations and hence, many implementations of the same conceptualization. Section 2.4 shows how reusable components can be extended for added functionality.

The remainder of the chapter deals primarily with realizations. Section 2.5 discusses some aspects of RESOLVE especially conceived for making reusable software components efficient. Section 2.6 addresses some questions related to object code generation for reusable components. Section 2.7 provides a non-trivial example conceptualization, and discusses several realizations of it. It demonstrates the need for programming language support for multiple implementations of a specification. Section 2.8 provides a brief comparison of the RESOLVE approach to others with respect to reuse, and section 2.9 has a summary of the chapter. For a RESOLVE language report, see [Harms 90].
2.1 Conceptualizations

A RESOLVE software component typically provides a type and some operations. Figure 2.1 shows the conceptualization for a component that provides a generic type Stack and the operations Push, Pop, and Is_Empty for manipulating variables of type Stack.

```plaintext
conceptualization Stack_Template
  parameters
    type Item
  end parameters
  auxiliary
    math facilities
      String_Theory is String_Theory_Template (math[Item])
      renaming
        String_Theory.String as String
        String_Theory.Lambda as Lambda
        String_Theory.Post as Post
      end renaming
    end math facilities
  end auxiliary
  interface
    type Stack is modeled by String
    exemplar s
      initially "s = Lambda"
    end Stack
    operation Push
      parameters
        alters s : Stack
        consumes x : Item
      end parameters
      ensures "s = Post(s, x)"
    end Push
    operation Pop
      parameters
        alters s : Stack
        produces x : Item
      end parameters
      requires "s ≠ Lambda"
      ensures "#s = Post(s, x)"
    end Pop
```
operation Is_Empty returns control
parameters
preserves s : Stack
end parameters
ensures Is_Empty iff "s = Lambda"
end Is_Empty
end interface

description
Stack_Template provides the type family "Stack of Item," where Item is any type. In the formal specifications above, an abstract stack is a string of Items, with the top of the stack at the right end of the string. Initially, every stack is empty.

• "Push(s, x)" pushes x onto stack s. Since x is a consumes parameter, it has an initial value for its type upon return.

• "Pop(s, x)" pops the top element off stack s and returns it in x.

• "Is_Empty(s)" returns yes if and only if s is an empty stack.
end description
end Stack_Template

Figure 2.1 — A Conceptualization of Stack_Template

It is important to observe the formality and representation independence of the conceptualization in Figure 2.1. The conceptualization does not specify any particular concrete representation for the type Stack, such as an array or a linked list. Instead, the type Stack is explained as being modeled by a mathematical string, and the operations on Stack variables are explained using assertions in mathematical string theory. The foremost requirement to make multiple components possible for the same specification is that the specification should be free of all implementation detail. The explicit use of mathematical modeling makes this possible. This modeling is discussed in greater detail in section 2.3. The remainder of this section explores the three formal sections of a RESOLVE conceptualization using the Stack_Template for illustration.
2.1.1 The Parameters Section

The Stack_Template conceptualization is generic because the type of elements contained on a stack (i.e., Item) is a parameter to the conceptualization. Conceptualizations can also have mathematical functions and program facilities as parameters, the details of which will be discussed later in section 2.4.

2.1.2 The Auxiliary Section

The abstraction provided by a conceptualization in RESOLVE is typically explained using well-known mathematical theories. In the conceptualization Stack_Template, the explanation of the type Stack and the operations uses mathematical string theory, in particular the type String and functions Lambda and Post from this theory. (Other mathematical theories are used in the examples, discussed later in this dissertation.) The auxiliary section in the conceptualization provides a syntactic slot for bringing in all the mathematical machinery essential in describing an abstraction, including mathematical theories.

The formal definition of a mathematical theory is contained within a theory module. A theory typically defines a mathematical type and functions involving variables of that type. In this example, the axioms of string theory come from String_Theory_Template. This is a generic theory — the type of elements in the string is a parameter to the module. The module provides a mathematical type String, and among others, definitions for the empty string (constant function Lambda) and Post. Function Post takes a string of some type and an element of that type as arguments, and yields the string in which the element is attached.
to the right end of the string. The details of theory modules are beyond the scope of this dissertation and are not needed in order to understand it.

Before a theory can be used in a conceptualization, it has to be instantiated. An instance of a theory module is called a *math facility*. An example is the instance String_Theory of String_Theory_Template using $\text{math[Item]}$ as a parameter, where $\text{math[Item]}$ is the mathematical model of the type Item.

Theory modules should not be confused with conceptualizations such as Stack_Template. Theories do not have any realizations. They simply define mathematical theories.

We have been careful to distinguish mathematical types from (program) types in the discussion above, without providing definitions for these terms. For this dissertation, precise definitions of these terms are not essential and they can be found in [Harms 90]. A mathematical type is provided by a theory module (possibly generic) and before it can be used the theory module has to be instantiated. Two mathematical types are considered equivalent if they have the same structure. Conceptualizations provide program types and they have to be instantiated as discussed in section 2.2 before program types can be used. Unless qualified, a type always refers to a program type. When a mathematical type is meant, it is made explicit. Section 2.3 discusses the idea of modeling a program type with a mathematical type in more detail.

2.1.3 The Interface Section

The *interface* section of a conceptualization lists the type(s) and operations provided by the conceptualization.
Specification of Types. The type declaration names the type (or the family of types, if the conceptualization is generic) provided by the conceptualization and shows its mathematical model. To make assertions about a prototypical variable of this type, a name is provided for an exemplar. In this case, the name s is the exemplar for Stack variables.

Every type declaration must include an initially clause. The initially clause in Figure 2.1 asserts that the initial value for every variable of type Stack is conceptually the empty string (denoted by Lambda). The empty string models the empty stack. In RESOLVE, every variable is given an initial value specified in the conceptualization of its type when it is declared. A variable gets this value before it is used in any executable statement, and this value satisfies the initially clause. Just before the scope of a variable ends, it is finalized, i.e., the space occupied by its representation may be reclaimed. Because this (generally) has no abstract effect — the variable is finalized only after its useful life is over — a typical conceptualization contains no specification of the effect of finalization.

Specification of Operations. RESOLVE specifications of the operations provided in the interface are both syntactic and semantic. An operation may or may not return a value of any type. It may also return a special value called a control. For example, the Is_Empty operation in Figure 2.1 returns a control value, i.e., yes or no. The decision to include control values in RESOLVE is based on the design goal of keeping the language as orthogonal and as small as possible. In RESOLVE, there are no built-in types. Even the conventionally built-in types such as booleans, integers, arrays, and records are not part of the language. These types are defined through conceptualizations just like any other type. In this respect, RESOLVE is similar to Alphard [Shaw 81].
Alphard, however, includes booleans, and RESOLVE does not. The absence of built-in booleans raises the obvious problem in explaining the semantics of control statements such as `while...endwhile` and `if...endif`. Control-valued operations have been included in RESOLVE just for this purpose. However, there is no type "control," i.e., no variables can be declared to have a control value. Only control operations can return control values and only they can be used in the conditional expressions in `while...endwhile` and `if...endif` statements.

The effect of every operation is formally defined by two assertions — an _ensures_ clause (post-condition) and a _requires_ clause (pre-condition). If the requires clause is true when the operation is invoked, the operation guarantees that the ensures clause will be met when it returns. A requires clause may be omitted from the specification if it is identically true. Within the ensures clause of an operation with parameter `x`, "#x" denotes the value of `x` when the operation was invoked (i.e., the value sent by the invoker), and "x" denotes the value when the operation returns (i.e., the value returned to the invoker).

For example, the specification of the operation `_Pop_` (alters `s: Stack`, produces `x: Item`) requires that before the operation is called it must be true that the stack `s` is not empty. It ensures (if the requires clause is met) that after the operation is executed, the value of `s` returned with the value of `x` that is returned attached to its right end will be the same as the value of `s` that was passed by the invoker. The value passed for the variable `x` by the invoker is not used by the procedure at all.

Parameters to operations can have four different _modes_ and these serve as useful specification notations:
• "alters x" denotes that the value of parameter x may be used and changed by the operation; the exact way x is changed is specified in the operation's ensures clause.

• "produces x" indicates that the incoming value of x is not used by the operation; the value of x when the operation returns is specified in the ensures clause.

• "preserves x" indicates that the ensures clause implicitly contains the assertion "and x = #x," which means that the value of x returned is the same as what was passed; however, the value of x may change temporarily during the execution of the operation.

• "consumes x" indicates that the ensures clause implicitly contains "and T.init(x)," i.e., the value of x when the operation returns is an initial value of type T (the type of x).

In the specification of the operation Push (alters s: Stack, consumes x: Item), the value of x returned is equal to an initial value of the type Item, which is implicitly specified by use of the consumes parameter mode.

The ensures clause of an operation will be true upon return only if the requires clause was true when the operation was invoked. If the requires clause is not met, the operation can do anything. The caller of the operation is not guaranteed anything at all. The reason for this meaning has to do with efficiency — there is no reason to penalize those clients of the operation who do not violate the requires clause. If every operation is forced to check the
validity of its requires clause before it is executed, there is a run-time penalty for everyone including those who do not violate the requires clause.

It is essential to notice that the component design style described here does not preclude the defensive programming methods advocated elsewhere [Liskov 86]. It is easy to see that a safe Pop operation which always checks for the validity of its requires clause can be implemented using the Pop operation described here and the Is_Empty operation. This is not a special characteristic of Stack_Template. Our component design guidelines, such as those discussed in [Weide 90], ensure that this is usually the case — if an operation on a component has a requires clause, it must be possible to check the validity of that clause using some combination of the other operations provided by the component.

2.1.4 The Description Section

The description section provides an informal and intuitive explanation of the abstraction described formally in the interface. The purpose of this section is not to provide information not specified in the formal part, but only to serve as an additional aid to human understanding. A client of a conceptualization must rely only on the formal sections for its functionality.

2.2 Realizations

A realization of a conceptualization defines appropriate data structures and algorithms for correctly implementing the types and operations specified. (Recall that in a mature software component industry, the source code of a realization will be available only to the developer of the realization. A translated machine-readable form of the realization will be marketed to
a client along with the corresponding conceptualization.) The declarations, control structures, statements, and assertions that can be used in RESOLVE realizations are discussed in [Harms 90]. Example realizations are provided later in this chapter. For the component industry viewpoint, all that is essential to understand at this point is that there may be several realizations (possibly with different performance characteristics) for the same conceptualization, and that clients can use components without having access to their implementation details.

For example, let Stack_Real_1 and Stack_Real_2 be two realizations of the conceptualization in Figure 2.1. Figure 2.2 shows the average time and space complexities of the operations provided by the realization Stack_Real_1. Assume that the performance details of Stack_Real_2 are identical to Stack_Real_1 except that the finalization operation takes linear time (in the size of the stack being finalized) rather than constant time. In fact, the only aspect of performance of Stack_Real_1 that might be surprising is that the finalization operation executes in constant time. General amortization techniques for providing constant time implementations of initialization and finalization operations for several non-trivial data structures are discussed in [Harms 89a]. Since it is possible to have a realization that can provide constant time implementations for all the operations of Stack_Template, the need for another realization for performance compromise is not clear from this example. An example where different realizations provide better implementations for different sets of operations is discussed in section 2.7. All that is important for now is that the two stack realizations Stack_Real_1 and Stack_Real_2 have some performance differences, and that they may use different representations for variables of type Stack.

A component marketer will essentially provide performance information such as shown in Figure 2.2 with the object code version of the realization, in addition to supplying the
conceptualization of the component. This kind of information (i.e., not source code) is all that is needed for a client to use a component, as demonstrated in the next few sections.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Finalization</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Push</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Pop</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Is_Empty</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

**Figure 2.2 — Performance Information for Stack_Real_1**

2.2.1 Using a Component

A reusable software component is typically constructed using other available reusable components. The developer of one component may therefore be the client of several other components as noted in Chapter I. Before the types and operations provided by a component can be used within a client program, the client must create an instance of the component by plugging in actual values for all the parameters (if any) to the conceptualization and choosing one of the many realizations of this conceptualization. A component instance created in this way is called a *facility*. Just as a mathematical type comes from a math facility, a program type comes from a (program) facility.

Figure 2.3 shows a client code fragment using facilities created from Integer_Template and Stack_Template. (The conceptualization Integer_Template may be found in Appendix. It provides the type Int and operations on Int variables.) As noted earlier, there are no built-in
types in RESOLVE. The client has to create a facility from Integer_Template in order to use integers. Int_Real_1 and Int_Real_2 are two different realizations of Integer_Template. The client code also uses two instances of Stack_Template with the realizations Stack_Real_1 and Stack_Real_2. The client has passed the type IF1.Int as a parameter to that generic conceptualization to create the two stack facilities.

```plaintext
facilities
    IF1 is Integer_Template
       realized by Int_Real_1
       renaming
           IF1.Int as Pounds
       end renaming
    IF2 is Integer_Template
       realized by Int_Real_2
       renaming
           IF2.Int as Seconds
       end renaming
    IF3 is Integer_Template
       realized by Int_Real_1
       renaming
           IF3.Int as Feet
       end renaming
    ISF1 is Stack_Template (IF1.Int)
       realized by Stack_Real_1
    ISF2 is Stack_Template (IF1.Int)
       realized by Stack_Real_2
end facilities
...
local variables
    i1 : Pounds
    i2 : Seconds
    i3 : Feet
    s1 : ISF1.Stack
    s2 : ISF2.Stack
end local variables
...
maintaining "...
while not ISF1.Is_Empty(s1) do
    ISF1.Pop (s1, i1)
    ISF2.Push (s2, i1)
endwhile
...
```

Figure 2.3 — Example Client Code
It is essential to notice, of course, that the client code in Figure 2.3 can be written without access to the source code of the realizations Int_Real_1, Int_Real_2, Stack_Real_1, or Stack_Real_2. The client depends only on the conceptualizations Integer_Template and Stack_Template for its functionality. The performance, however, depends on which realizations are used when facilities are declared in the client. Section 2.3 elaborates on these issues.

2.2.2 Program Type Equivalence

Unlike mathematical types, a program type from one facility is never considered equivalent to a type from another facility even if the facility declarations happen to be identical. In Figure 2.3, Pounds, Seconds, and Feet are all different types and variables of these types cannot be mixed.

There are at least two significant reasons for not equating type equivalence with structure equivalence for program types. In computer programming it is sometimes important to distinguish, for example, length and weight, though both of them may have the same structure. If a programmer creates two types (using facility declarations such as IF1 and IF3 in Figure 2.3) having the same structure, there is good reason to believe that he or she wants to avoid mixing variables of these types. The programming language should certainly not preclude such a possibility.

Another reason is based on our philosophy that a conceptualization specifies what a module does (i.e., functionality) while a realization determines how a conceptualization is implemented (i.e., performance). If, for example, the types IF1.Int and IF3.Int were
considered equivalent based on their structures, changing the realization of IF3 from Int_Real_1 to Int_Real_3 (another realization of Integer_Template), will make the client program illegal, and this is undesirable. Changing realizations should only alter the performance characteristics of a client, not its functionality, and therefore certainly not whether the client program compiles correctly.

2.3 Implementation Independence in Specifications

The foremost requirement to make possible multiple components with the same specification is that the specification should be free of all implementation detail. The use of mathematical modeling in RESOLVE conceptualizations makes this possible. In RESOLVE, every program type is explicitly modeled by a mathematical type. In Figure 2.1, the family of program types Stack is modeled with the family of mathematical types String. All references to program variables within a mathematical assertion (such as initially, requires and ensures clauses) actually denote the corresponding mathematical variables of the mathematical type modeling the program type. For example, in the requires clause of the operation Pop, its parameter stack s is treated as a mathematical String.

This modeling makes it possible for a client of Stack_Template to reason about the value of a program variable of type Stack as though it were a mathematical String. Consider the example client code shown in Figure 2.3. The values of variables s1 and s2 are treated as mathematical Strings of mathematical Integers. Let #s1 and #s2 denote the values of s1 and s2 just before the while loop is first encountered, and s1 and s2 denote the values just after the loop finishes execution. It is possible to prove the following assertion at the end of the loop:
\[ \text{Cat}(s_2, s_1) = \text{Cat}(\#s_2, \text{Reverse}(\#s_1)) \]

where \text{Cat} and \text{Reverse} are functions from \text{String\_Theory\_Template} with the obvious meanings. It is possible to formally prove the correctness of the client code using the proof rules associated with \text{RESOLVE} statements and declarations [Krone 88], and assertions in the conceptualizations used by the client. The proof will need only the assertions found in the requires and ensures clauses of the operations \text{Push}, \text{Pop}, and \text{Is\_Empty} in \text{Stack\_Template}, and not anything from the implementations \text{Stack\_Real\_1} or \text{Stack\_Real\_2}. In other words, the proofs will be independent of the particular realizations chosen for use with the facility declarations. If the client code in Figure 2.3 is modified so that realization \text{Stack\_Real\_3} (a possible third realization for \text{Stack\_Template}) is used for the facility ISF1, the functional correctness of the client code will remain unaffected. That is, reasoning about a client program is independent of the representations (or realizations) of the program types used by that program.

The developmental independence for clients from the implementations of components stems from total adherence to the principles of abstraction and information hiding in \text{RESOLVE} conceptualizations. Such independence has several important implications for the construction and maintenance of large software systems. Since a client program is dependent only on the mathematical models described in the implementation-independent conceptualizations, a client does not have to be re-coded, re-compiled, or re-verified even if the realizations used by the client program are changed.
2.4 Extending Components with Additional Operations

Sometimes an operation needed by a client is not among those specified in the conceptualization providing the type on which it operates. As examples, consider operations Stack_Revers (which reverses its parameter stack) and Stack_Replica (which produces a copy of its parameter stack). They each operate on a parameter of the type Stack provided by Stack_Template. Two questions arise in this context. What are some guidelines to ensure that a component provides all desired functionality but does not "overdo" it? How can we construct new operations layered on top of already available components so that these new operations themselves are reusable? Answers to these questions are based on an identification of a small but yet comprehensive set of fundamental operations on a type.

2.4.1 Primary and Secondary Operations

The operations whose implementations can access the representation of a type are termed the primary operations on that type. In the example conceptualization Stack_Template, Push, Pop, and Is_Empty are primary operations on the type Stack.

[Weide 90] suggests guidelines for deciding on the appropriate primary operations for a module. Together they should provide the basic functionality needed on an abstraction. In general, the primary operations are orthogonal, i.e., one of the operations cannot be implemented efficiently using a combination of others. The efficiency consideration here has to do with orders of magnitude and not constant factors — an operation generally should not be made a primary operation just because making it a primary operation will result in a constant time improvement in its efficiency. There may be more than one set of
primary operations which provide the desired functionality on an abstraction. In this case some implementation considerations may influence the selection of one set over another.

Every primary operation has at least one parameter of the type for which it is a primary operation. Its implementation, by definition, can access the representations of such parameters. However, it does not have access to the representations of any other types of variables passed as parameters. For example, the implementation of the operation Push can access the representation of its parameter s directly. It cannot access the representation of x, its other parameter.

Operations that must be implemented without access to representations of any of their parameters are termed secondary operations. These operations can be constructed using a combination of primary (and other secondary) operations. For example, the Stack_Revers e operation can be constructed as a secondary operation as shown in Figure 2.6. It should be clear that any implementation of this operation as a secondary operation can depend only on the conceptualization Stack_Template and not on the implementation details found in any of its realizations, because, by definition, an implementation of a secondary operation cannot access the representations of its parameters. This observation implies that it is possible to construct a Stack_Revers e operation that reverses a stack in some representation containing, say, elements of type Item, into another stack in any other representation (of course, containing elements of type Item). In this sense the Stack_Revers e operation is highly reusable — reusable across all realizations of Stack_Template. The next section demonstrates the details of how this can be done in RESOLVE.

This distinction between primary and secondary operations does not seem to be widely recognized. For example, compare the design of a component in Appendix that provides
lists with designs discussed in [Meyer 88, Booch 87]. Designs suggested by these authors do not distinguish between a constant time efficiency improvement and orders of magnitude efficiency improvement, and hence, when an operation should be made primary and when it should not be is not as clear as suggested here. The failure to recognize this important distinction results in designs of reusable components with a large number of primary operations. The class that provides lists in [Meyer 88] has 31 distinct operations! Each new implementation of this class must (directly or indirectly using source code reuse) provide code for all 31 operations.

Secondary operations provide the ability to extend reusable components with new operations using a layered approach. C++ [Stroustrup 86], Eiffel [Meyer 88], Smalltalk [Goldberg 80], and other object-oriented language designers have proposed programming language mechanisms (e.g., implementation or code inheritance) for extension of reusable components. But these rely on the availability of the source code for the components. If source code of components is not marketed, as we have argued, then such mechanisms will be futile.

One of the common objections to the layered approach to extension is that the basic component may not provide a sufficient set of primary operations, and hence it may not be possible to write the desired secondary operation. In the context of software components designed for reuse (as the components marketed in a reusable component industry will be), this will usually not be the case. Another objection comes from arguments of inefficiency. It is indeed possible that an operation if implemented as a primary operation will be more efficient (in order of magnitude) than otherwise. (An example for this case is discussed in the next section.) However, such efficient implementations generally are not possible using mechanisms that allow extensions (but not modification) of components based on
source code such as implementation inheritance. These mechanisms can only improve efficiency by a constant factor and not in order of magnitude, and they are neither desirable nor can they be used in a successful software component industry [Muralidharan 90].

There is a technical problem related to layering of secondary operations, however. To extend components with secondary operations, facilities are needed as parameters to conceptualizations. The next sub-section discusses the need for facility parameters and shows how to use them in specifying the operation Stack_Reverse.

2.4.2 Mechanisms for Developing Reusable Secondary Operations

In RESOLVE, every reusable abstraction or operation must be provided by a component. Even if a component provides a single operation, such as the Stack_Reverse, it must have a conceptualization which specifies what the operation does.

Figure 2.4 shows the conceptualization of a component that provides the Stack_Reverse operation. Because every type must come from a facility, where is the facility that provides type Stack to be declared? This facility (which is a program facility and not a math facility) cannot be declared in the conceptualization itself. Recall that a program facility associates a particular realization with an instance of a conceptualization, and therefore, permitting a program facility to be declared in a conceptualization will make the conceptualization dependent on some realization, i.e., the conceptualization will not be implementation-independent. The solution is to use an instance of Stack_Template as a parameter to the conceptualization. In general, whenever a conceptualization needs to refer to a particular program type (unlike the type Item that was a generic type parameter in Figure 2.1), the type has to be brought into the conceptualization through a facility parameter. The need for
(program) facilities to be passed as parameters is a consequence of the fact that program
types are not considered equivalent based on structure. If a conceptualization needs a
specific mathematical type, it can use math facility declarations because mathematical type
equivalence is based on structure.

Figure 2.4 shows the specification of Stack_Reverse.

```
conceptualization Stack_Reverser_Capability
  parameters
    facility SF1 is Stack_Template
    facility SF2 is Stack_Template
  restrictions
    SF1.Item = SF2.Item
  end restrictions
  end parameters

auxiliary
  renaming
    SF1.String_Theory.Reverse as Reverse
  end renaming
end auxiliary

interface
  operation Stack_Reverse
    parameters
      consumes s1: SF1.Stack
      produces s2: SF2.Stack
    end parameters
  ensures "s2 = Reverse(s1)"
  end Stack_Reverse
end interface

description
...
end description
end Stack_Reverser_Capability
```

Figure 2.4 — A Specification of Stack_Reverse Operation

The conceptualization shown in Figure 2.4 is very general — it can be used to reverse a
stack from any facility of Stack_Template to any other. The restrictions on the parameter
list have an obvious purpose. The types of elements on the two stacks must be the same, i.e., SF1.Item must be the same as SF2.Item. (Recall that Item was the name of the generic type parameter to Stack_Template.) Stack_Reverse can be used to reverse any stack containing elements of type Item into another containing elements of type Item.

Figure 2.5 shows a client that uses the secondary operation Stack_Reverse. The client, of course, does not need the source code of any of the realizations, and hence does not need that of Stack_Rev_Real_1. However, this realization has been shown in Figure 2.6 to demonstrate that it is possible to write a reusable secondary operation.

```plaintext
... facilities
  IF is Integer_Template realized by Int_Real_1
  ISF1 is Stack_Template(IF.Int) realized by Stack_Real_1
  ISF2 is Stack_Template(IF.Int) realized by Stack_Real_2
  SRCF is Stack_Reverse_Capability(ISF1, ISF2) realized by Stack_Rev_Real_1
end facilities
...
local variables
  s1 : ISF1.Stack
  s2 : ISF2.Stack
end local variables
...
SRCF.Stack_Reverse(s1, s2)
...
```

Figure 2.5 — Instantiation and Use of Stack_Reverse_Capability
realization Stack_Rev_Real_1 for Stack_Reverser_Capability

realization auxiliary
renaming
  SF1.Item as Item
  SF1.String_Theory.Cat as Cat
end renaming
end realization auxiliary

interface
operation Stack_Reverser
parameters
  consumes s1: SF1.Stack
  produces s2: SF2.Stack
end parameters
ensures "s2 = Reverse(s1)"
begin
local variables
  x: Item
  s: SF2.Stack
end local variables
s := s2
maintaining "Cat(s1, Reverse(s2)) = #s1"
while not SF1.Is_Empty(s1) do
  SF1.Pop(s1, x)
  SF2.Push(s2, x)
endwhile
end Stack_Reverser
end interface
end Stack_Rev_Real_1

Figure 2.6 — One Possible Realization of Stack_Reverser

A realization is written in the context of the corresponding conceptualization, and therefore all the names and declarations found in the conceptualization are available for use in the realization. When a realization is written, a RESOLVE environment brings in the corresponding conceptualization parts for easy reference [Harms 90]. The realization writer cannot alter these parts. The developer of Stack_Rev_Real_1, for example, need not rewrite the operation interface for Stack_Reverser. Instead it will be automatically brought in by the RESOLVE editor.
In the implementation of Stack_Reverse, notice that s2 is initialized by swapping it with a local variable s that has an initial value. The maintaining clause provided just above the while statement in this realization describes the invariant for that loop. This assertion along with the specification in Stack_Template can be used in showing the correctness of this realization [Krone 88, Hoare 69].

The implementation of Stack_Reverse in Figure 2.6 takes time linear in the number of elements on its parameter stack. It is possible to provide a constant-time implementation of Stack_Reverse (as well as Stack_Replica) if it is a primary operation. RESOLVE provides a mechanism for adding primary operations to conceptualizations, termed an enhancement. However, Stack_Reverse implemented as an enhancement will require a specific representation for the type Stack. Every other primary operation on Stack must also be implemented based on this representation. (This is the reason why mechanisms for source code extension, such as code inheritance, generally will not be useful.) Constructing an efficient enhancement for a type is usually more expensive than writing it as a secondary operation. In addition, it cannot be reused across all representations of the type.

In the component industry, efficient implementations of most useful operations on a type will be available directly from its conceptualization or as enhancements. If an operation is not available, but it is possible to implement it efficiently as a secondary operation, then it must be implemented as a secondary operation. If this is not the case, a trade-off has to be made between the cost involved in constructing the operation as an enhancement and the inefficiency resulting from making it a secondary operation.
Enhancements are not discussed in detail here because they are not relevant to the topic of this dissertation.

2.5 Support for Construction of Efficient Components

Performance of reusable software components being the central theme of this dissertation, how efficiently such components can be constructed in a given programming paradigm is a related issue. In this section we discuss one particular programming construct in RESOLVE whose essential purpose is to facilitate construction of efficient components. The data movement mechanism “swapping” introduced in this section as an alternative to copying of data values also has important implications to the manner in which RESOLVE components can be distributed — the topic of Chapter IV in this dissertation.

2.5.1 Problems with Copying As a Basic Data Movement Mechanism

Copying of data values through assignment statements and parameter passing to procedure calls has become a standard feature of programming languages. Even languages advocating software reuse provide these methods of copying as built-in features.

The first problem with copying is one of efficiency. Copying a variable generally takes time proportional to the size of the representation of its value. If reusable software components are widely used, it will not be uncommon to see the usage of, say, variables having values such as stacks of stacks of queues of integers. Copying such variables is tremendously inefficient. Sometimes it may be essential to make a copy. But usually a copy of a variable is made simply because copying is the only primitive data movement mechanism that is available in the language.
The commonly suggested solution to this problem is to use indirection. The semantics of copying is defined as sharing, and when a copy of a variable has to be made, for example through an assignment statement, a pointer to the variable (or the address of the variable) is copied in place of the value of the variable. Since pointers typically have a constant size, such copying will only take a constant time — time not dependent on the size of the representation of a variable's value. Sharing and copy ing are definitely not the same. Sharing creates an alias to a variable's value and copying does not. If variables \( x \) and \( y \) share the representations of their values, altering the value of one of them also alters the value of the other. Such aliasing complicates program verification efforts [Hoare 87, Harms 89b], and makes informal reasoning about program behavior even more risky than usual.

In RESOLVE, we have proposed swapping as a built-in primitive data movement mechanism. Swapping is efficient but does not introduce aliasing.

### 2.5.2 Swapping As a Primitive Data Movement Mechanism

In RESOLVE, there is no built-in assignment operation for any type. Instead, there is a predefined \texttt{swap} operation (written as \texttt{":=:"}) available for every type [Harms 89b]. This operation has two parameters of the same type, and exchanges the values of the two.

Swapping two values of a type, in general, is apparently expensive. However, this need not be the case as explained here. In RESOLVE's implementation, every variable is represented indirectly using a pointer. To RESOLVE programmers, however, these pointers are not visible. To them every variable has a value from the domain of its type.
Because all variables are represented indirectly, swapping two variables' values can be achieved by simply exchanging the pointers to these two variables. Exchanging pointers to two variables takes only constant time — time independent of the values that are swapped. Since the participating pointers are only exchanged, problems that would otherwise result from copying pointers do not arise.

Along with swapping, RESOLVE also includes a function assignment operation (written as ":="). The left hand side of this assignment is, of course, a variable, but the right hand side can only be an operation that returns a value of the type of the variable on the left hand side. The right hand side of this statement cannot be a variable. Function assignment does not involve copying, either, but can be handled as a special case of swapping [Harms 89b].

Parameter Passing By Swapping. The problems with copying and the advantages of swapping extend to the way in which parameters are passed between callers and operations. Parameter passing by value uses copying and hence is inefficient. Parameter passing by reference complicates formal specification of operations, because the notion of addresses has to be introduced in the specification language and in the specification of each operation. In RESOLVE, therefore, parameters to procedure calls are passed by a new mechanism, termed "call by swapping" [Harms 89b].

Conceptually, when a procedure is invoked, each actual parameter is swapped with the formal parameter (on call and return) if the actual parameter is a variable. The value of the actual parameter is assigned (transferred) to the formal parameter at calling time if the actual parameter is a function call; no transfer occurs on return in this case.
All parameters are passed in constant time since swapping is a constant time operation. Thus values represented by large data structures can be passed efficiently between the caller and the called procedure. (It must be emphasized again that the pointers in the above discussion are an aspect of RESOLVE implementations, and are not part of RESOLVE. They are not visible to the programmer.)

2.5.3 Influences of Swapping on the Design of Reusable Components

Efficiency being one of the vital characteristics of a reusable component, it is essential that designs of reusable components must permit efficient implementations. To give an example, the specifications of the operations Push and Pop in Figure 2.1 do not require implementations of these operations to make a copy of the item that is pushed on or popped off the stack. The specifications of a reusable stack abstraction found in most textbooks (e.g., [Booch 87]) usually require copies of the items to be made for Push and Pop operations. This is but one example of how thinking of efficiency (an example being swapping values rather than copying) at the time of constructing specifications creates interesting component designs even for the simplest abstractions. Section 2.7 provides another interesting example.

2.6 Object Code Generation

This section discusses the need for generality of the object code generated for reusable components. The object code for a realization of Stack_Template, for example, must be independent of the Item passed as a parameter to Stack_Template at instantiation time. Different facilities of Stack_Template using the same realization must be able to share the
object code of that realization. [Hegazy 89] discusses how object code is generated for RESOLVE components so it can be shared among different instances.

Several Ada compilers compile the source code and create separate object codes for each instance of a generic package [Ganapathy 89]. This situation is clearly unacceptable in a component industry where only object code versions of components are sold.

Another issue relates to how a RESOLVE compiler must represent variables of a type. For example, consider the client program shown in Figure 2.3. How can variables s1 and s2 be represented, because the source code of Stack_Real_1 and Stack_Real_2 (and hence the representations of a stack variable in these realizations) are not known? The only reasonable solution is to represent these variables indirectly through a pointer. This makes constant-time swapping possible. It also provides another important advantage — if, for example, the realization of the facility ISF1 is changed from Stack_Real_1 to Stack_Real_3, object code for the client program need not be re-generated. This observation has important implications for parametrizing realizations for performance, as discussed in the next chapter.

Advocates of C++ ([Stroustrup 86], for example) claim that the use of indirect representation for variables as suggested above will lead to inefficient execution due to dynamic storage allocation/deallocation and additional memory references to access indirect representations. Based on this reasoning C++ permits representation details of provided types to be part of interface headers (as do Ada and Modula-2), and loses several advantages of limiting such details to implementations. For most representations built on top of pointers, dynamic storage allocation is inevitable. Using amortization techniques, it is possible to have efficient storage allocation and deallocation for certain non-trivial data
structures [Harms 89a]. For the case of simple representations such as standard integers, we have already argued that indirection in representations can be eliminated by optimizing compilers. Therefore, dynamic storage allocation/deallocation is not a significant source of inefficiency. In [Muralidharan 89a], we have shown that indirection in representations actually results in a penalty of one memory reference per operation call for a parameter of a type whose actual representation is comparable in size to a pointer (e.g., an Int, represented in the standard way). Gannon et al. have reached a similar conclusion [Gannon 87]. For most types, such as those discussed in [Booch 87], there need be no difference whether or not the representations are known at compile-time of clients, i.e., indirection in representations does not result in any penalty.

One more question regarding the generality of object code remains to be answered. How is it possible to use the same object code across different client hardware architectures? Either the object code should be marketed for a standard virtual machine (one designed specifically for this purpose), and then clients can either use a translator or a simulator; or a different version of object code should be marketed for each different client hardware.

2.7 An Example to Illustrate Performance Compromises

In this section we see a concrete example of how the software component industry view is effectively supported by RESOLVE. We discuss a conceptualization for the associative searching problem and several realizations of this conceptualization with different performance characteristics. This abstraction provides an excellent example of how a non-trivial specification can be presented without implementation-dependent details. The realizations are built using other reusable components, thereby demonstrating the layering inherent in this approach to software reuse.
The general searching problem is to determine if a particular value is among a given set of values. In most searching applications (e.g., searching for a record with a certain key value in a database), it is not sufficient to find out if the search value is among the values being searched, but it is also important to retrieve some information associated with it. We term this the associative searching problem: Given a key value, to retrieve the corresponding value associated with the key from a store which contains a number of pairs of the form <key, associated value>.

A good abstraction of the associative searching problem permits several interesting implementations. We discuss four such realizations. While the realizations of abstractions presented so far do not need implementation-specific parameters, the realizations discussed in this section illustrate the need for such parameters. Parameters to a conceptualization are uniformly used by all realizations of it, whereas realization parameters are specific to an implementation. Sometimes, a particular realization requires some operations which should not appear in the conceptualization. Such operations, possibly a different set for each realization of the same conceptualization, can be parameters to realizations, as explained in this section.

2.7.1 A Conceptualization for the Associative Searching Problem

Figure 2.7 shows a conceptualization Partial_Map_Template that abstracts the associative searching problem. In general, the searched key will be from a set containing a large number of values (possibly infinite), and only relatively few of these will have related information associated with them. When the association from key values to related information is viewed as a mathematical function, it is generally true that the function is
defined only on some values on its domain. The conceptualization Partial_Map_Template captures this abstraction.

The generic theory module Tuple_2_Theory_Template provides an ordered pair and two projection operations. Function_Theory_Template provides the type Function. The domain and range of the function are parameters to the module. Delta is a function with a function and a set of values from the domain of that function as parameters. It asserts (returns true) that the given function remains unchanged except for the set of domain values passed as the other parameter. For example, in the ensures clause of Make_Defined, the use of Delta asserts that the function m is unchanged except at the domain value #d.

```plaintext
conceptualization Partial_Map_Template
parameters
type Domain
type Range
end parameters

auxiliary
math facilities
  Boolean_Theory is Boolean_Theory_Template
  renaming
  Boolean_Theory.Boolean as Boolean
end renaming

Tuple_2_Theory is
Tuple_2_Theory_Template(Boolean, math[Range])
renaming
  Tuple_2_Theory.Tuple as Range_Pair
  Tuple_2_Theory.Projection_1 as Defined
  Tuple_2_Theory.Projection_2 as Range_Value
end renaming

Function_Theory is
Function_Theory_Template(math[Domain], Range_Pair)
renaming
  Function_Theory.Function as Function
  Function_Theory.Delta as Delta
end renaming
end math facilities
end auxiliary
```
interface
type Partial_Map is modeled by Function
  exemplar m
  initially "forall d: math[Domain], not Defined(m(d))"
end Partial_Map

operation Make_Defined
  parameters
    alters m: Partial_Map
    consumes d: Domain
    consumes r: Range
  end parameters
  requires "not Defined(m(d))"
  ensures "Delta(m, {#d}) and Defined(m(#d)) and
          Range_Value(m(#d)) = #r"
end Make_Defined

operation Make_Undefined
  parameters
    alters m: Partial_Map
    preserves d: Domain
    produces r: Range
  end parameters
  requires "Defined(m(d))"
  ensures "Delta(m, {d}) and not Defined(m(d)) and
          r = Range_Value(#m(d))"
end Make_Undefined

operation Swap_Entry
  parameters
    alters m: Partial_Map
    preserves d: Domain
    alters r: Range
  end parameters
  requires "Defined(m(d))"
  ensures "Delta(m, {d}) and Defined(m(d)) and
          Range_Value(m(d)) = #r and
          r = Range_Value(#m(d))"
end Swap_Entry

operation Is_Defined returns control
  parameters
    preserves m: Partial_Map
    preserves d: Domain
  end parameters
  ensures Is_Defined iff "Defined(m(d))"
end Is_Defined

operation Is_Undefined_Everywhere returns control
  parameters
    preserves m: Partial_Map
  end parameters
  ensures Is_Undefined_Everywhere iff
  "forall d: math[Domain], not Defined(m(d))"
end Is_Undefined_Everywhere
operation Make_Any_One_Undefined
parameters
  alters m: Partial_Map
  produces d: Domain
  produces r: Range
end parameters
requires "exists x: math[Domain], Defined(m(x))"
ensures "Defined(#m(d)) and Delta(m, {d}) and not Defined(m(d)) and r = Range_Value(#m(d))"
end Make_Any_One_Undefined

description
... description
end Partial_Map_Template

Figure 2.7 — A Conceptualization for Associative Searching

The partial function abstracted in Partial_Map_Template is modeled by a total function whose domain is the same as the domain of the partial function, but whose range is the Cartesian product of boolean values and the range of the partial function. If the partial function is defined on a domain value, then its model maps it to a (boolean value, range value) pair such that the boolean value is true. If the partial function is undefined on a domain value then its model maps it to a pair such that the boolean value is false. The abstraction does not permit (without violating any requires clause) accessing the range values corresponding to the domain values where the partial function is undefined. To make the conceptualization generic, the types domain and range have been made parameters to the conceptualization.

Initially the partial function is undefined everywhere. The operation Is_Undefined_Everywhere tests if the function is undefined everywhere. Make_Defined defines the function at a given domain value and maps it to a particular range value, also passed as a parameter. Make_Undefined removes and returns the range value associated
with a given domain value, and thus makes the function undefined at that point. Swap_Entry can be used to change the range value associated with an already defined domain value, as well as to evaluate the partial map there. Is_DEFINED is a control that tests if the function is defined at a given domain value. Make_Any_One_Undefined undefines the function at some domain value on which it was previously defined, and returns the domain value and the associated range value. (Without this operation, Replica and Are_Equal cannot be implemented efficiently as secondary operations on Partial_Map. It is therefore suggested as a consequence of one of several component design guidelines.)

There are some interesting observations to be made regarding the conceptualization Partial_Map_Template. The conceptualization has been explained in terms that are totally independent of specific implementation details. The operations listed in the conceptualization are orthogonal — no operation can be efficiently implemented using a combination of others. Also note that every requires clause in the conceptualization can be tested using the control operations. The operations have been carefully specified to permit efficient implementations. For example, none of the operations requires a copy of a domain or a range value to be made by a realization.

The Partial_Map_Template can be implemented in a variety of ways, resulting in different performance behaviors. Any implementation of this abstraction has to store explicitly only those pairs that correspond to domain values where the partial function is defined, because the conceptualization prohibits access to range values corresponding to domain values where the function is undefined. The four realizations discussed here illustrate the need for operations, mathematical functions, and facilities as parameters to realizations. But, first we introduce some terminology for analyzing the performance of a realization.
2.7.2 Analysis of Performance Characteristics

The "performance of a component" includes the performance of each of the operations it provides, and should be supplied to a client in terms of the parameters to that component. Execution time and the amount of space used by an operation are both important aspects of performance. Because time is usually more important, we use execution time as the only measure of performance for the discussions in this dissertation.

Execution time of an operation is equal to the time to execute the code of the operation plus the time for passing parameters between the caller and callee plus some constant time that includes costs such as transferring control to the code for the operation. In RESOLVE, parameters are passed by swapping [Harms 89b]. Because call-by-swapping takes constant time regardless of the size of the argument that is passed, all the parameters to a given operation are passed in total time which is some constant.

Execution time for a RESOLVE operation invocation is therefore the execution time for each of the operations called by that operation multiplied by the number of times that operation is called, plus a constant time for call/return overhead. Time for executing an operation is generally a function of its parameters. The worst case time complexity of an operation gives an upper bound on the execution time of that operation. The average time complexity of an operation gives the average execution time of that operation over some distribution of inputs to the operation.
2.7.3 A Realization Based on Linear Searching

Figure 2.8 shows part of a realization `Linear_Search` for `Partial_Map_Template`. In this implementation, a partial function is represented by a one-way list of (domain value, range value) pairs for the points where the partial function is defined. The assertion `correspondence` precisely describes how this representation corresponds to the mathematical model in the conceptualization. The assertion `conventions` explains the idea that in this implementation, the same pair will not be stored in two places in the one-way list. These assertions are used in proving that `Linear_Search` correctly implements `Partial_Map_Template`. (The proof is not shown here, however.)

The facility declarations set up a type "one-way list of (domain, range) pairs" to represent a `Partial_Map`. (See Appendix for the conceptualizations of `One_Way_List_Template` and `Record_2_Template`. `Pointer_Based` is a realization of `One_Way_List_Template`. This realization provides constant time implementations of each of the operations on `One_Way_List_Template`, including initialize and finalize operations [Harms 89a]. The Standard realization of `Record_2_Template` also provides constant time implementations for all its operations. We do not need any more information about these realizations in order to use them.)

```plaintext
realization Linear_Search for Partial_Map_Template
realization parameters
operation Are_Equal_Domain_Values returns control
parameters
    preserves x: Domain
    preserves y: Domain
end parameters
ensure Are_Equal_Domain_Values iff "x = y"
end Are_Equal_Domain_Values
end realization parameters
```
realization auxiliary
facilities
RF is Record_2_Template(Domain, Range)
realized by Standard
renaming
RF.Pair as DR_Pair
RF.Projection_1 as Domain_Val
RF.Projection_2 as Range_Val
end renaming

LF is One_Way_List_Template(RF.Record)
realized by Pointer_Based
renaming
LF.String as String
LF.Lambda as Lambda
LF.Cat as Cat
LF.Pre as Pre
LF.Member as Member
LF.Pair_Of_Strings as Pair_Of_Strings
LF.Left as Left
LF.Right as Right
end renaming
end facilities

local operations Search_List
operation Search_List
parameter
alters 1: LF.List
preserves d: Domain
end parameters
ensures ~(Cat(Left(1), Right(1)) =
Cat(Left(#1), Right(#1))) and
((exists p: DR_Pair, exists a: String,
Pre(p, a) = Right(1) and Domain_Val(p) = d)
iff Right(1) /= Lambda)"

begin
local variables
p: RF.Record
temp_d: Domain
end local variables
LF.Reset(1)
maintaining ~(Cat(Left(#1), Right(#1)) =
Cat(Left(1), Right(1))) and
not Member(Left(1), d)"
while not LF.At_Right_End(1) do
LF.Remove_Right(1, p)
RF.Swap_Entry_1(p, temp_d)
if Are_Equal_Domain_Values(d, temp_d) then
RF.Swap_Entry_1(p, temp_d)
LF.Add_Right(1, p)
return
else
RF.Swap_Entry_1(p, temp_d)
LF.Add_Right(1, p)
LF.Advance(1)
endif
endwhile
end Search_List
end local_operations
end realization auxiliary

interface

  type Partial_Map is modeled by Function
  is represented by LF.List
  realization exemplar real_m for m

  correspondence "(forall p: DR_Pair,
    forall d: math[Domain],
    forall r: math[Range],
    Domain_Val(p) = d and Range_Val(p) = r and
    Defined(m(d)) and Range_Value(m(d)) = r iff
    Member(Cat(Left(real_m), Right(real_m)), p))"

  conventions "not (exists p: DR_Pair,
    exists a, b: String,
    Cat(a, b) = Cat(Left(real_m), Right(real_m))
    and Member(a, p) and Member(b, p))"

end Partial_Map

...operation Swap_Entry
  parameters
    alters m: Partial_Map
    preserves d: Domain
    alters r: Range
  end parameters
  requires "Defined(m(d))"
  ensures "Delta(m, {d}) and Range_Value(m(d)) = #r and r
  = Range_Value(#m(d))"

begin
  local variables
    p: RF.Record
  end local variables

  Search_List(m, d)
  LF.Remove_Right(m, p)
  RF.Swap_Entry_2(p, r)
  LF.Add_Right(m, p)
  and Swap_Entry

end interface
end Linear_Search

Figure 2.8 — A Realization Based on Linear Search for
Partial_Map_Template
The figure shows an implementation only for the operation Swap_Entry. To help implement Swap_Entry, a local operation Search_List has been written. Search_List looks for a pair in a given list with a given domain value. If such a pair is in the list, the "cursor" of the list is positioned just in front of that pair. The cursor is at the end of the list otherwise. Implementation of Search_List necessarily involves a comparison of domain values for equality, and this operation cannot be written locally since the type Domain is a parameter to the generic conceptualization Partial_Map_Template. The actual types Domain and Range will be supplied by a client at the time of instantiation of the conceptualization, and hence it is the client who must supply an equality-testing operation on the type Domain.

Of course, it is not sufficient to specify only the syntactic interface of formal realization parameters such as Are_Equal_Domain_Values. The operation passed by a client must also match certain semantic requirements. The formal parameter Are_Equal_Domain_Values is expected to return the value true only when its two arguments are equal. Obviously, it would be disastrous to allow a client to pass any control operation that has two parameters of the type Domain.

Figure 2.9 shows a client of Partial_Map_Template using Linear_Search realization. The equality-testing operation is not available from Integer_Template as a primary operation. (See Appendix for the Integer_Template conceptualization.) An equality-testing operation on integers will be needed so often that it is easy to imagine that an enhancement of Integer_Template providing this operation and an implementation of it will be readily available. Just to demonstrate that, in general, operations to be passed as parameters to realizations may not be readily available and that a client may have to construct them, the equality-testing operation has been shown implemented as a secondary operation. Observe
that the declaration of the operation Are EQUAL Int appears after its use. This is allowed because the scope of the name Are EQUAL Int is the entire module of Figure 2.9.

... realization auxiliary facilities
  IF is Integer_Template realized by Standard
  RNF is Real_Number_Template realized by Standard
  PMF is Partial_Map_Template(IF.Int, RNF.Real)
    realized by Linear_Search(Are_EQUAL_Int)
end facilities

local operations
  operation Are_EQUAL Int returns control
    parameters
      preserves x: IF.Int
      preserves y: IF.Int
    end parameters
    ensures Are_EQUAL Int iff "x = y"
  begin
    if Less_Than_Or_Equal(x, y) then
      if Less_Than_Or_Equal(y, x) then
        return yes
      else
        return no
      endif
    else
      return no
    endif
    end Are_EQUAL Int
  end local operations
end realization auxiliary...

Figure 2.9— A Client of Linear_Search

Why Parameters to Realizations Should Not be Made Parameters to the Corresponding Conceptualization. It is important to understand why the operation Are_EQUAL Domain _Values should not be a parameter to the conceptualization Partial_Map_Template. The idea of testing equality of domain values is an aspect of the particular implementation Linear_Search and has nothing to do with the abstraction described in Partial_Map_Template. There are other realizations that do not need this
operation, but that may need other operations. The realization Ordered_Search for Partial_Map_Template shown in Figure 2.11 does not use an equality testing on domain values. Instead it needs another operation Are_Ordered_Domain_Values as a parameter and this is not needed by Linear_Search. The realization Hashed_Search in Figure 2.14 needs still other operations as parameters.

The alternative to having realization parameters would be to make all such operations parameters to the conceptualization. This approach is neither desirable nor even viable. The conceptualization of an abstraction should contain no implementation details. An equality-testing operation is an idea related to a particular implementation and has nothing to do with the abstraction presented in the Partial_Map_Template, and hence it does not logically belong at the conceptualization level. Making it a parameter to the conceptualization will only clutter the presentation of the abstraction. In fact, operations are never needed as parameters to conceptualizations, and RESOLVE precludes making them parameters to conceptualizations.

**Performance Characteristics of Linear Search.** The marketed version of Linear_Search will be in object code form along with the information shown in Figure 2.10. We will analyze the time complexity of the operation Swap_Entry in detail to illustrate how these performance figures can be computed by a developer. Because our interest here is limited to time complexity expressed in orders of magnitude, we will ignore the constant terms involved in this analysis. Refer to Figure 2.8 for this analysis.

Let $T_{\text{Swap_Entry}}(m, d, r)$ denote the execution time of the operation Swap_Entry. It can be expressed as follows:
The Initialize and Finalize operations on the type of the local variable \( p \) are automatically invoked. The realization Pointer-Based for the facility LF and the realization Standard for RF supply constant time implementations for each of the provided operations, and hence the time complexity of \( \text{Swap\_Entry} \) can be re-written as follows (in order of magnitude):

\[
T_{\text{Swap\_Entry}}(m, d, r) = T_{\text{RF\_Record\_Initialize}}(p) + T_{\text{Search\_List}}(m, d) + T_{\text{LF\_Remove\_Right}}(m, p, r) + T_{\text{RF\_Swap\_Entry\_2}}(p) + T_{\text{LF\_Add\_Right}}(m, p) + T_{\text{RF\_Record\_Finalize}}(p)
\]  

(2.1)

Note that the complexity of \( \text{Swap\_Entry} \) is independent of one of its parameters: the range value \( r \). Now consider the complexity of \( \text{Search\_List} \). Let \( D = \{ x : \text{math}[\text{Domain}] \mid \text{Defined}(m(x)) \} \). \( T_{\text{Search\_List}}(m, d) \) can be expressed after ignoring constant-time operation invocations by \( \text{Search\_List} \) as follows:

\[
T_{\text{Search\_List}}(m, d) = \Theta(T_{\text{Search\_List}}(m, d))
\]

(2.2)

A careful observation of the \( \text{Search\_List} \) operation in Figure 2.9 reveals that the value of \( \text{temp\_d} \) finalized at the end of that operation is really an initial value of the type Domain. This is not an accident. In general, there are few implementations of operations where arbitrary values have to be finalized because of the programming method that uses swapping. Noting that the time complexity of \( \text{Swap\_Entry} \) is the same as \( \text{Search\_List} \), it can be written as below:
\[ T_{\text{Swap_Entry}}(m, d) = O(T_{\text{Domain.Initialize}} + \sum_{x \in D} T_{\text{Are_Equal_Domain_Values}(d, x)} + T_{\text{Domain.Finalize}(\text{temp}_d)}) \] (2.4)

where temp_d satisfies the condition Domain.Init(temp_d). Observe that the execution time for Swap_Entry has been expressed totally in terms of the parameters to that operation and the parameters to the realization Linear_Search. (Initialize and Finalize operations on the conceptualization parameter type Domain (and Range) are implicit parameter operations to all realizations, and the performance of a realization, in general, will depend on these operations as well. Because Swap is a constant time operation on every type it will not appear in performance expressions when only orders of magnitude are of interest).

The analysis of the time complexity of Swap_Entry exemplifies that it is possible to rigorously specify the performance of an operation provided by a component in terms of the performance of the operations it uses. To keep the presentation simple, Figure 2.10 shows the performance of each operation provided by Linear_Search tabulated only in terms of \( n \), the number of domain values on which the Partial_Map passed as a parameter to that operation is defined. The performance information shows average (for the usual distribution assumptions) and worst-case time complexities of each operation in orders of magnitude.

It is important to note that the performance of Linear_Search depends as well on the realizations used for the facilities declared in it. Recall that the realizations used in Linear_Search (Standard for Record_2_Template and Pointer_Based for One_Way_List_Template) provide constant-time implementations for all their operations. The Finalize operation for a Partial_Map, for example, takes constant time only because the realization Pointer_Based provides constant time finalization of a list. The operations
Make_Undefined and Is_Defined call Search_List once each and their performance is similar to that of Swap_Entry. Make_Defined is a constant-time operation because the pair for a new definition can be added anywhere in the list.

\[
\text{realization Linear Search for Partial Map Template}
\]

\[
\text{parameters}
\]

\[
\text{operation Are Equal Domain Values returns control}
\]

\[
\text{parameters}
\]

\[
\text{preserves x: Domain}
\]

\[
\text{preserves y: Domain}
\]

\[
\text{and parameters}
\]

\[
\text{ensures Are Equal Domain Values iff "x - y"}
\]

\[
\text{end Are Equal Domain Values}
\]

\[
\text{end realization parameters}
\]

Let \( n \) be the number of domain values on which the partial map \( m \) that is passed as a parameter to each of the operations is defined. The time complexities are independent of the other parameters to these operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Avg. Time</th>
<th>Worst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize ( m )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Finalize ( m )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Is_Undefined_Everywhere ( m )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Make_Defined ( m, d, r )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Make_Undefined ( m, d, r )</td>
<td>( \Theta(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Swap_Entry ( m, d, r )</td>
<td>( \Theta(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Is_Defined ( m, d )</td>
<td>( \Theta(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Make_Any_One_Undefined ( m, d, r )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
</tbody>
</table>

**Figure 2.10— Information Supplied to a Client of Linear Search**

Performance information as it has been presented in Figure 2.10 is neither formal nor complete. If performance information is expressed precisely in a formal language, then there is no reason in principle why performance correctness of clients (that a client meets a certain performance specification) cannot be proved. The performance information shows
only average and worst-case time complexities in orders of magnitude and does not include space complexities, and in this sense is incomplete. However, it is possible for a programming language to provide the machinery for formally stating performance information and proving performance correctness. These issues are not addressed further in this dissertation.

2.7.4 A Realization Based on Ordered Searching

We next discuss a variation of the realization Linear_Search. Just as in Linear_Search, the implementation Ordered_Search represents a Partial_Map by a one-way list of (domain value, range value) pairs for the domain values where the partial function is defined. But the pairs in the list always remain ordered based on domain values. The main difference between the two implementations is in how Search_List is implemented. Search_List searches a given ordered list for a pair with a given domain value only as long as the ordering holds, and not always to the end of the list. Unlike Linear_Search, the implementation of the operation Make_Defined also calls Search_List. The implementations of all other operations are identical to Linear_Search.

Figure 2.11 shows the information supplied to a client of Ordered_Search. The performance behavior of this realization actually appears inferior to Linear_Search because Make_Defined is no longer a constant time operation. However, the average time complexity of other operations such as Is_Defined is better than Linear_Search by a constant factor. This fact is not reflected in the figure because complexities have been expressed in orders of magnitude. Based on the performance information contained in Figure 2.11, no client of Partial_Map_Template may choose Ordered_Search over
Linear Search. Ordered Search has been included here only to illustrate some issues related to passing parameters to realizations.

```
realization Ordered_Search for Partial_Map_Template
realization parameters
definition R(x, y: math[Domain]): Boolean

operation Are_Ordered.Domain.Values returns control
parameters
preserves x: Domain
preserves y: Domain
end parameters
ensures Are_Ordered.Domain.Values iff "R(x, y)"
end Are_Ordered_DOMAIN.Values
end realization parameters

constraints
"forall x, y, z: math[Domain], 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))"
end constraints
```

Let n be the number of domain values on which the partial map m that is passed as a parameter to each of the operations is defined. The time complexities are independent of the other parameters to these operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Avg. Time</th>
<th>Worst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize(m)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Finalize(m)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Is_Undefined_Everywhere(m)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>Make_Defined(m, d, r)</td>
<td>Θ(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Make_Undefined(m, d, r)</td>
<td>Θ(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Swap_Entry(m, d, r)</td>
<td>Θ(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Is_DEFINED(m, d)</td>
<td>Θ(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Make_Any_One_Undefined(m, d, r)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>

Figure 2.11— Information Supplied to a Client of Ordered_Search Realization
For the same reasons equality-testing is a parameter operation to Linear_Search, an ordering operation is needed as a parameter to Ordered_Search. It is easy to write the syntactic interface for this ordering operation Are_Ordered_Domain_Values. How can it be specified?

Mathematical Definitions as Realization Parameters. What is needed here is a mathematical definition of the ordering computed by the operation parameter. This definition should also be a parameter to the realization. Figure 2.11 shows how this definition can be formally written. The syntax for R, the ordering relation, is shown in the parameter section. The mathematical type Boolean used in the definition R comes from the auxiliary section of the conceptualization Partial_Map_Template. The semantics of R is explained in the auxiliary section of the realization with constraints. They state that R is a total order. A client of Ordered_Search should pass a mathematical definition as well as an operation as parameters in order to use this realization, as shown in Figure 2.12.

```plaintext
... realization auxiliary
facilities
RNF is Real_Number_Template realized by Standard
IF is Integer_Template realized by Standard
PMF is Partial_Map_Template(IF.Int, RNF.Real)
   realized by
      Ordered_Search(IF.Math_Leq, IF.Less_Than_Or_Equal)
end facilities
end realization auxiliary
...
```

Figure 2.12— A Client of Ordered_Search

The implementation of the Search_List operation apparently will also need an equality-testing operation since domain values have to be compared for equality to stop searching at
the right pair. However, an equality-testing operation is not needed as a parameter. It can be built locally because \( R(x, y) \) and \( R(y, x) \) implies that \( x \) equals \( y \).

### 2.7.5 A Realization Based on Binary Search Trees

**realization** Binary_Search_Tree for Partial_Map_Template

**realization parameters**

definition \( R(x, y: \mathit{Domain}) : \mathit{Boolean} \)

**operation** Are_Ordered_Domain_Values returns control parameters

preserves \( x: \mathit{Domain} \)

preserves \( y: \mathit{Domain} \)

**end** parameters

ensures Are_Ordered_Domain_Values \( \iff \) "\( R(x, y) \)"

**end** Are_Ordered_Domain_Values

**end** realization parameters

**constraints**

"forall \( x, y, z: \mathit{Domain}, R(x, x) \) 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 \( R(y, x) \)"

**end** constraints

Let \( n \) be the number of domain values on which the partial map \( m \) that is passed as a parameter to any of the operations is defined. The time complexities are independent of the other parameters to these operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Avg. Time</th>
<th>Worst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize(( m ))</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Finalize(( m ))</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Is_Undefined_Everywhere(( m ))</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Make_Defined(( m, d, r ))</td>
<td>( \Theta(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Make_Undefined(( m, d, r ))</td>
<td>( \Theta(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Swap_Entry(( m, d, r ))</td>
<td>( \Theta(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Is_Defined(( m, d ))</td>
<td>( \Theta(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Make_Any_One_Undefined(( m, d, r ))</td>
<td>( \Theta(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>

**Figure 2.13** — Information for a Client of Binary_Search Realization
A realization that stores (domain, range) pairs in a balanced binary search tree can provide more efficient (and balanced time) implementations for the operations provided by Partial_Map_Template. This realization also needs an ordering on the domain values with specification identical to that of Ordered_Search. Figure 2.13 shows the information supplied to a client for this realization. Because the binary tree can be kept balanced, the worst case complexities are as good as the average time complexities.

2.7.6 A Realization Based on Hashing

Figure 2.14 shows a realization Hashed_Search for the Partial_Map_Template. We briefly discuss the details of this implementation so it is possible to justify the performance information in Figure 2.14. The type Partial_Map is represented using an array to store all the (domain value, range value) pairs, for the domain values where the partial function is defined. A pair is stored in the array at the index returned by a hash operation on the domain value. The hash operation may map more than one domain value to the same array index, resulting in a collision. We therefore chain all the pairs that map to the same array index using a one-way list. Given a good hash operation [Knuth 68], the domain values map uniformly over the array indices, and thus the length of any chain tends to be near the number of defined domain values (n) in the partial map divided by the size of the array (Number_of_Buckets). The realizations of Record_2_Template, One_Way_List_Template, and Array_Template used in Hashed_Search provide constant time implementations for all operations.
realization Hashed_Search for Partial_Map_Template
realization parameters
  facility IF is Integer_Template
  renaming
    IF.Integer as Integer
    IF.Int as Int
end renaming

definition Math_Number_Of_Buckets: Integer

operation Number_Of_Buckets returns size: Int
  ensures "size = Math_Number_Of_Buckets"
end Number_Of_Buckets

definition Math_Hash_Fn (x: math[Domain]): Integer

operation Hash_Opn returns h: Int
  parameters
    preserves x: Domain
  end parameters
  ensures "h = Math_Hash_Fn(x)"
end Hash_Opn

operation Are_Equal_Domain_Values returns control
  parameters
    preserves x: Domain
    preserves y: Domain
  end parameters
  ensures Are_Equal_Domain_Values iff "x = y"
end Are_Equal_Domain_Values

end realization parameters

constraints
  Math_Number_Of_Buckets > 0
end constraints

Let n be the number of domain values on which the partial map m that is passed as a parameter to any of the operations is defined. The time complexities are independent of the other parameters to these operations (except Hash_Opn).
Let b = Math_Number_Of_Buckets. For a "good" Math_Hash_Fn, a = n/b (see [Knuth 68]).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Avg.</th>
<th>Worst</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize(m)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td></td>
</tr>
<tr>
<td>Finalize(m)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td></td>
</tr>
<tr>
<td>Is_Undefined_Everywhere(m)</td>
<td>Θ(b)</td>
<td>O(b)</td>
<td></td>
</tr>
<tr>
<td>Make_Defined(m, d, r)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td></td>
</tr>
<tr>
<td>Make_Undefined(m, d, r)</td>
<td>Θ(a)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>Swap_Entry(m, d, r)</td>
<td>Θ(a)</td>
<td>O(n)</td>
<td></td>
</tr>
</tbody>
</table>
This realization needs a hash operation as a parameter. The operation Hash_Opn, given a
domain value, returns an integer to be used as an index into the array. The realization also
needs a parameter to indicate the number of buckets to be used in hashing, i.e., the size for
the array used in the implementation.

There are two questions regarding the specification of these operations. One of them is to
ensure that these operations behave like mathematical functions. For example, the
Hash_Opn must return the same integer value whenever the same domain value is passed
as a parameter. The operation Number_Of_Buckets should return the same integer
whenever called. A problem here is that operations can have state information, and
therefore when called at different times with the same argument values might return
different values. There are other problems as well. For example, the operation “sqrt” on a
real number is usually considered a program function, though its specification might allow
it to return a positive or negative value, and thus it would not qualify as computing a
mathematical function. In general, operations that may syntactically look like program
functions need not behave like mathematical functions. For the Hashed_Search realization
to work, it is vital that Hash_Opn return the same index into the array whenever called with
the same domain value as parameter. To ensure that Hash_Opn and Number_Of_Buckets
do behave correctly, also included are two mathematical definitions Math_Hash_Fn and
Math_Number_Of_Buckets as formal realization parameters.
The other question relates to the type of the values returned by Hash_Opn and Number_Of_Buckets. The next section addresses this question.

_Need for Facilities as Parameters to Realizations._ Whenever an operation is a parameter to a realization, the formal parameters to the operation must have certain types. Consider the function Are_Equal_Domain_Values which is a parameter to Hashed_Search. This function has two formal parameters, both of which are of the type Domain, a parameter to the conceptualization Partial_Map_Template. In this case, there is no need to introduce a new type as a parameter to the realization. In fact, all parameters to the formal operation parameters of the three realizations discussed earlier are of the type Domain, and hence there is no need for introduction of any new types at the realization parameter level.

One of the parameters of the implementation Hashed_Search is Hash_Opn. This operation returns a value of a type not known in the conceptualization Partial_Map_Template. In this example, Hash_Opn returns an Int. Where does this type Int come from? Because the client is going to supply the actual hash operation, only the client knows the type of the value returned by the hash operation. Therefore the client must also pass that type as a realization parameter. The way in RESOLVE to communicate a specific type to an implementer is to use a facility parameter. Therefore, a facility of Integer_Template is a parameter to the realization. The reasoning is similar to the reasoning for facility parameters to RESOLVE conceptualizations — that a client and the developer of a component should be using the same program type.

The need for facility parameters to realizations is tied to the need for operations as parameters to realizations. Facilities are not needed as parameters to a realization if it does not have any operations as parameters. In the example discussed here, a facility of
Integer_Template is needed as a parameter because the type Int is not a built-in type in RESOLVE. Operations passed as parameters to realizations usually do not (in our experience) need any types other than the traditionally built-in types such as integers, real numbers, or records. This suggests that facilities passed as parameters to realizations are most often from a small set of conceptualizations including Integer_Template, Real_Number_Template, and Record_Template. This is a fortuitous phenomenon, since passing facilities from certain conceptualizations raises an interesting side-issue. The next section explains the problem and proposes to distinguish "standard" from "non-standard" conceptualizations. Only facilities from "standard" conceptualizations should be passed as parameters to realizations.

2.7.7 Standard and Sub-Standard Conceptualizations

The conceptualizations discussed so far have no "shared data." It is possible, however, to design conceptualizations in RESOLVE so that all operations provided by a facility of that conceptualization share state information. (This is done by declaring math variables in the auxiliary section.) Most well-designed conceptualizations do not have this property, but a few may need to use the capability. For example, the important conceptualization Pointer_Template [Pittel 90] uses it.

Figure 2.15 shows another example: a conceptualization that captures the idea of generating unique identifiers. The variable Id_Set is shared among all operations provided by an instance of this conceptualization.
conceptualization Unique_Id_Assgn_Template
parameters
  facility IF is Integer_Template
end parameters

auxiliary
  math facilities
    Number_Theory is Number_Theory_Template
    renaming
      Number_Theory.Integer as Integer
    end renaming
    Set_Theory is Set_Theory_Template(Integer)
    renaming
      Set_Theory.Set as Set
      Set_Theory.MakeSet as MakeSet
      Set_Theory.Member as Member
      Set_Theory.Union as Union
      Set_Theory.Diff as Diff
      Set_Theory.Size as Size
      Set_Theory.EmptySet as EmptySet
    end renaming
  end math facilities

  math variables
    Id_Set: Set
  end math variables

  initially "Id_Set = EmptySet"
end auxiliary

interface
type Id is modeled by Integer
  exemplar x
    initially "x = 0"
end Id

operation Is_Assigned returns control
parameters
  preserves x: Id
end parameters
  ensures Is_Assigned iff "x /= 0"
end Is_Assigned

operation Assign_Id
parameters
  alters x: Id
end parameters
  requires "x = 0"
  ensures "x /= 0 and not Member(#Id_Set, x) and
            Id_Set = Union(#Id_Set, MakeSet(x))"
end Assign_Id
operation Return_Id
    parameters
    alters x: Id
    end parameters
    requires "x /= 0"
    ensures "x = 0 and Id_Set = Diff(#Id_Set, MakeSet(#x))"
end Return_Id

operation Id_Count returns count: IF.Int
    parameters
    end parameters
    ensures "count = Size(Id_Set)"
end Id_Count

end interface
description
...
end description
end Unique_Id_Assgn_Template

Figure 2.15 — A Conceptualization for Unique Id Assignment

A sub-standard conceptualization has shared conceptual state variables which can be accessed and changed by a client of the conceptualization using one or more of the provided operations. A conceptualization which has a facility parameter from a sub-standard conceptualization is also sub-standard. A facility from a sub-standard conceptualization is termed a sub-standard facility. Unique_Id_Assgn_Template is a sub-standard conceptualization. Stack_Template, Partial_Map_Template, and the ones in Appendix are all examples of standard conceptualizations; i.e., they are not sub-standard.

We are now ready to see why facilities from sub-standard conceptualizations cannot be passed as parameters to realizations. Figure 2.16 shows a realization C_Real that fakes as a parameter a facility TF of Unique_Id_Assgn_Template. C_Real implements the conceptualization C_Template. One of the operations provided by C_Template is P. The specification of P contained in C_Template, of course, cannot and does not state anything
about TF which is a realization parameter. The implementation of P in C_Real calls the
operation Assign_Id from the facility TF, which is a sub-standard facility as noted above.

```plaintext
realization C_Real for C_Template
  realization parameters
    facility TF is Unique_Id_Assgn_Template
    ...
  end realization parameters

realization auxiliary
  facility BF is Boolean_Template realized by Standard
  ...
end realization auxiliary

interface
  ...
  operation P
    parameters
    ...
    end parameters
    ensures "..."
  local variables
    x: TF.Id
    b: BF.Bool
  end local variables
  begin
    ...
    if BF.Is_True(b) then
      TF.Assign_Id(x)
    endif
    end P
    ...
  end interface
end C_Real
```

**Figure 2.16 — A Realization Using a Facility from Id Assignment**

Figure 2.17 shows a client of C_Template. Set_Real is a realization of
Unique_Id_Assgn_Template. The client has chosen to use C_Real with the facility
declaration CF of C_Template, and therefore, has passed an instance of
Unique_Id_Assgn_Template as a facility parameter to C_Real.
Consider one of the client operations Q shown in Figure 2.17. Q calls operations from both TF and CF. In particular, Q has operation calls from TF and CF interleaved as shown. What can the client assert about the values of count1 and count2? Will they be equal? This question cannot be answered without knowing the details of C_Real, and in our component industry view it will not be available. What we have is a problem of incompleteness. There is no way for C_Real to communicate with a client how it might affect the shared data of the facility TF, and a client cannot make any assumptions about what C_Real might do with it. C_Real cannot extend the specification of P given in C_Template to say how TF is affected by P because permitting it to do so would mean the
behavior of client would be changed by changing realizations of C_Template, thereby destroying modularity of verification.

The problem here is not that a sub-standard facility (facility from a sub-standard conceptualization) is passed as a parameter, but that it is not possible to specify how that facility’s conceptual state will be affected, and hence a client cannot know how it is affected. A facility from a sub-standard conceptualization can be passed as a parameter to a conceptualization because then it is possible to explain in the conceptualization how that facility’s state is affected by each of the provided operations. The problem arises only when a sub-standard facility needs to be passed as a parameter to a realization. A solution to this problem must ensure that standard and sub-standard conceptualizations are syntactically distinguished and that only standard facilities are allowed as parameters to realizations.

The proposed solution leaves behind one question. What can be done when one of the operations needed as a parameter to a realization requires the type provided by a sub-standard facility? We know of no practical component design in which this seems necessary or desirable, but disallowing such operations to be passed as parameters to realizations may make it impossible to construct certain interesting realizations. One solution is to make the sub-standard facility (that provides the type needed in specifying an operation) a parameter to the conceptualization. This solution clearly violates our design goal of keeping implementation details separate from conceptualizations. Unfortunately, there may not be any better solution for this special case. Most conceptualizations are standard, and rarely is there a need to design a sub-standard conceptualization. Therefore the problem presented in this section may not have any serious impact on reusable software component designs in practice, but it is something that needs to be considered in the
language design. The issue arises solely because of the requirement of modular verification, by the way, and does not seem to have been addressed in any other programming language.

2.7.8 Performance Compromises

The example discussed in this section clearly illustrates the need for implementation independence in the specification of an abstraction, multiple implementations for an abstraction, and also the need for separate parameters to implementations. The performance behaviors of the four realizations of Partial_Map_Template discussed here are different, and in fact, no one realization has dominant performance. Still other realizations are possible for Partial_Map_Template with other performance behaviors. Partial_Map_Template is not unique in this respect. Most interesting abstractions do permit different implementations with different performance. Programming language mechanisms that permit the construction of abstract specifications and multiple implementations provide clients of abstractions with the flexibility to make an appropriate choice to match specific performance needs. As seen in the next section, existing languages fall short of meeting this objective on several critical grounds.

2.8 Other Languages for Constructing Reusable Software Components

Several languages have been developed over the last decade for supporting data abstraction and constructing reusable software components, including Ada [DoD 83] and ANNA [Luckham 87], Alphard [Shaw 81], C++ [Stroustrup 86], CLU [Liskov 81] and Larch/CLU [Wing 87], Eiffel [Meyer 88], and Modula-2 [Wirth 82]. These languages were designed with goals different from RESOLVE and hence the mechanisms and
methods in them are in some respects significantly different. The most important difference from the standpoint of issues addressed in this dissertation is in language support for multiple implementations of the same abstraction. This section elaborates on this issue.

Most of the above languages support separation of the specification of an abstraction from its implementation. However, most do not recognize the need for language support for multiple implementations of an abstraction. The languages that recognize this need do not support it adequately. It is possible to have support for multiple implementations in Ada and Modula-2 to a limited extent if certain design guidelines are followed. Though CLU and Alphard provide this support without the need for programmer conventions, they fall short of other reasonable requirements. Mechanisms in object-oriented languages like Eiffel and C++ to facilitate multiple implementations of an abstraction are fundamentally different from RESOLVE, though they are effective. These mechanisms, however, introduce other problems. In the following sections, we discuss Ada, CLU, and Eiffel as representative languages illustrating previous approaches and their problems.

2.8.1 Ada

Generic packages in Ada can be used to construct reusable software components [Booch 87]. An Ada package has a *package specification* and a *package body* which are separately compilable. A client of a package can only see and depend on the specification of a package, not its body. The specification of a package providing an encapsulated type, such as Stack, includes a declaration of the type as a *private* or a *limited private* type along with the syntactic interface for each of the operations provided by the package. (When an encapsulated type is provided as a private type, built-in operations for assignment and equality-testing are implicitly available for use by clients. When a type is declared as a
limited private type, every operation must be explicitly provided by the package.) The specification also contains a representation for the encapsulated type in the *private* part. The corresponding package *body* contains code for implementing the operations.

In Ada, only one package body can be associated with a specification, and in this sense, the *language* does not directly support multiple implementations of an abstraction as does RESOLVE. Some of the impediments to having multiple implementations can be overcome if certain guidelines are followed by developers and clients of Ada packages [Muralidharan 89b]. However, as we see in this section, these guidelines and use of certain clever environment tricks can only lead to limited support for multiple implementations, and in any case cannot fully compensate for lack of language support.

One of the fundamental problems with Ada as a language for software reuse is that Ada package specifications, unlike RESOLVE conceptualizations, describe only syntactic interfaces. For the designers of Ada, formal specification of functionality and certification of correctness of packages were not issues [Ichbiah 83]. This has led to an “add-on” approach. For example, ANNA is a specification language specially designed for annotating Ada programs [Luckham 87]. The purpose of ANNA specifications, however, is different from RESOLVE specifications in that they are used for run-time assertion checking rather than formal verification of correctness. There are numerous missing constructs and technical difficulties with using ANNA for verification [Krone 88].

Representation independence is a fundamental requirement of a specification if it has to be implemented in several different ways. However, an Ada package specification is forced to reveal the representation of the type it encapsulates in its private part, thus preventing the use of different representations for the same type. While referring to the private part,
Feldman [Feldman 85] notes that "unfortunately Ada requires that the data structure implementing a data type appear somewhere in the specification part of a package, and not in the 'hidden' part of a package, as we would like in the ideal." He also notes that "For reasons having to do with how compilers for the language will be implemented, Ada compels us to write the private part in the package specification. Clearly, it would be preferable to hide those details away in the body."

This problem can be cured by using guidelines such as those discussed in [Muralidharan 89b]. The recommended conventions include that an encapsulated type must always be declared as a limited private type (not a private type), must be represented in the private part indirectly through an access type (a pointer), and must have two particular operations on this type: Initialize and Finalize. A client must always call the Initialize operation on a variable of an encapsulated type before using it in any other operation, and must call the Finalize operation before exiting the scope of that variable. The specification of a package providing the type Stack following these (and a few other) design guidelines has been shown in Figure 2.18.

For the specification shown in Figure 2.18, the actual representation data structure used for the type Stack is known only within the package body. Even though Ada allows only one body to be associated with a specification, it is possible to have several bodies satisfying the specifications (possibly using different data structures for the type Stack) and a clever environment can allow a client the choice to link any one of them with a particular client program. Apparently, then, it is possible to have multiple implementations of an abstraction in Ada if certain guidelines are followed.
Figure 2.18 — Specification of a Generic Package Providing the Encapsulated Type Stack

There are at least two limitations to this solution. A single client program cannot use two different implementations of the same specification because only one of the different bodies of a package can be linked with the client. This problem arises because Ada does not allow independent naming of specifications and bodies. Another problem is that all implementations of a specification must have the same generic parameter lists because
parameters can be associated only with a specification and not independently with a body. The Partial_Map_Template and the four realizations of it discussed in the last section clearly demonstrate the necessity for realization parameters separate from conceptualization parameters. To force every possible parameter that may be needed by a package body to be a parameter to the package specification has obvious problems.

The limited extent to which multiple implementations can be used in Ada depends on how well package designers and clients adhere to certain conventions. Conventions are no substitute for language-enforced constraints. It seems inevitable that the conventions will be violated when they are most important. In fact, none of the packages discussed in standard Ada text books such as [Booch 87] follow the guidelines presented in [Muralidharan 89b].

Modula-2 [Wirth 82] suffers from all the problems discussed in the context of Ada.

2.8.2 CLU

CLU supports data abstraction by separating its interface specification from an implementation, termed a cluster [Liskov 81]. A CLU cluster can be specified formally using Larch/CLU, a specification language, and it is claimed to be possible in principle to prove correctness of CLU programs [Liskov 86, Wing 87], though we do not know of any published examples. The specification of a CLU cluster using Larch/CLU is independent of representational details. In CLU, just as in RESOLVE, every variable of an abstract type is represented indirectly using a pointer. However, in CLU, this pointer (called an object reference, but still a pointer in terms of abstract behavior) is visible to clients.
Designers of CLU observed the importance of allowing several implementations for the same interface specification [Liskov 77, Liskov 81]. Unfortunately, they did not see the need for language support. Referring to the usage of one of the several implementations by a client of an abstraction, they note [Liskov 77] “We imagine a process of binding together modules into programs prior to execution, at which time this selection would be made.” They go on to suggest how a CLU library might permit several implementations of an interface specification to coexist, of which one can be chosen by a client. This solution is identical to the one we discussed in the context of Ada. Because the interface specification and a cluster cannot have separate names or separate parameter lists, the two problems we raised in the context of Ada — that a single client program cannot use two different implementations of an abstraction, and that every implementation of an abstraction is forced to have identical parameter lists which are parameters to the specification of the abstraction — remain unresolved. However, CLU does provide limited support for multiple implementations without the need for any guidelines to be followed by cluster designers, and in this respect it supports the idea better than Ada does.

Alphard has exactly the same problems in supporting multiple implementations as does CLU [Shaw 81].

One of the differences between CLU and Alphard, and languages like RESOLVE and Ada, is that a reusable component in CLU and Alphard provides an abstract object rather than a type. In particular, a cluster itself is considered a type. This approach precludes a component from providing more than one type. Designers of Alphard, which is similar to CLU in this respect, note that this was a mistake [Shaw 81]. Because a CLU cluster provides an object, CLU is sometimes called an object-oriented language. However, CLU
and Alphard do not support inheritance (a mechanism that permits a component to share its specification and/or implementation with another) which is commonly considered to be one of the characteristics of an object-oriented language [Meyer 88, Stroustrup 88].

2.8.3 Eiffel

Eiffel is an object-oriented language designed with the goal of supporting construction of reusable software components [Meyer 88]. An abstract data object and operations on it are encapsulated by a class in Eiffel. A class in Eiffel is like a type. Variables can be declared to be of a class. The only operations that can be used to manipulate these variables are the operations listed in their class. Every variable of a class is represented indirectly using a pointer just as in RESOLVE, CLU, and Alphard. This pointer is, however, visible to Eiffel programmers.

An interesting feature in Eiffel is that a class may or may not include actual code for some or all of the operations provided by that class. A class that specifies only interface syntax for all its operations, but does not include code for any of them, is termed a deferred class. A deferred class serves as a specification for an abstraction in the same (syntactic interface) sense as in Ada or CLU. Meyer, the designer of Eiffel, considers formal specification and certification of correctness of software components to be important for software reuse, but dismisses them based on their current impracticality [Meyer 88].

One of the important programming mechanisms in Eiffel is inheritance. A class in Eiffel can inherit from another class its data and/or operations. The class that inherits is termed a heir and that from which it inherits is called its parent. There are two distinct uses of inheritance in Eiffel — inheritance of specification from a deferred class, and inheritance of
implementation details such as data and code for operations from a regular class. Specification inheritance can be effectively used to support multiple implementations of an abstraction, and we discuss this use of inheritance in more detail. Use of implementation inheritance either to add additional operations to an existing class or to modify the code for some of its operations has to little to offer in the context of reusable software components, and we have previously warned against its usage in [Muralidharan 90]. Others have noted a similar distinction and difficulties with implementation inheritance [LaLonde 89, Raj 89, Snyder 86].

Let A1 be a deferred class and let C1 and C2 be two heirs of this class, each representing the instance data of class A1 differently and providing different implementations for the deferred operations of that class. Here A1 is analogous to a RESOLVE conceptualization. Heirs C1 and C2 are like two realizations of A1. Each heir has an independent name and can have its own parameter lists. A client can use both C1 and C2 simultaneously. Thus, Eiffel provides much better support for multiple implementations of an abstraction than Ada or CLU, though Eiffel class designers have to follow some conventions.

There are, however, other problems that seem inherent to this approach. In Eiffel, a variable from a descendant class may be assigned to a variable of its parent class. For example, let x, y, and z be variables of classes A1, C1, and C2 respectively. Now y or z can be assigned to x. If y is assigned to x, when an operation P is invoked on x the code for P in the class C1 will be executed. However, if z is assigned to x, the code for P in the class C2 will be executed. In general, given an operation invocation, the code that should be executed can be determined only at run-time. This dynamic binding results in a run-time table look up for every operation call, resulting in some inefficiency.
Another problem is that sometimes a binary (or higher-arity) operation from a deferred class can be invoked with arguments in different representations. Each heir of a deferred class can handle only variables in its particular representation, and no heir has the code to handle the case where one of the arguments to an operation is not in its representation. We discuss this issue further in a different context in Chapter IV.

C++ can support multiple implementations as well as Eiffel, though a few design guidelines such as the ones discussed in the context of Ada have to be followed by class designers in C++ [Stroustrup 86]. C++ also has all the problems that plague Eiffel.

2.9 Summary

In this chapter, we have discussed how the RESOLVE framework provides the technical machinery needed for realizing a software component industry. In particular, we have seen how implementation-independent formal specifications of abstract components can be written, and how such abstract components can be realized in several different ways permitting interesting performance trade-offs. A client of an abstract component can easily switch realizations for that component and thus alter its performance without affecting its functionality. Other programming languages ostensibly designed for reuse do not provide all the machinery needed for effectively supporting multiple implementations of an abstraction.
CHAPTER III
PARAMETRIZING PERFORMANCE OF SOFTWARE COMPONENTS

In a successful reusable component industry, a new reusable component typically will be constructed using components already available in the market. The builder of a new component will decide on what abstractions should be used in constructing the component and how they should be used. To provide each of these abstractions, several components with different performance will be available. The builder of a new component may choose fixed components for constituent abstractions, and then market a version of the component. However, this component will have one specific performance behavior and clients will not be able to tune it.

The builder of a new component may also choose to market a tunable version of the component whereby clients are told what components should be plugged in and how the performance of these components will affect that of the component under consideration. Given this information, clients will have to decide which components should be plugged in to suit their performance requirements. This dissertation does not address the question of how clients can make this decision, but it seems likely that there will be performance tools to assist them. The issue here is to identify and suggest programming language mechanisms needed for implementing and using performance-tunable, general reusable components. These mechanisms are not part of the current version of RESOLVE (July 1990) and are proposed here as additions to the language.
The chapter is organized as follows. Section 3.1 explains the basic idea of performance tuning that is based on allowing clients to plug in constituents of a component. Section 3.2 discusses a set of possible language mechanisms for constructing maximally tunable reusable components. Section 3.3 discusses how a tunable component can be used to construct various specially tailored components. Section 3.4 provides a summary.

3.1 The Basic Idea and Mechanisms for Performance Tuning

In this section, we discuss an example to illustrate how realizations for constituent abstractions of a realization affect its performance. We conceive a basic mechanism for permitting parameterless realizations to be passed as parameters to realizations. The more general mechanism is discussed in the next section.

3.1.1 An Example

A RESOLVE realization typically uses facilities from other conceptualizations. In the realization Linear_Search shown in Figure 2.8, facilities RF of Record_2_Template and LF of One_Way_List_Template have been used. The performance of Linear_Search depends on the realizations associated with each of these facilities. The realization Standard used for RF provides constant-time implementations for all its operations. Let us analyze the impact of the realization used for LF on the performance of the operations provided by Linear_Search.
realization Linear_Search for Partial_Map_Template
realization parameters

... end realization parameters

realization auxiliary
facilities
RF is Record_2_Template(Domain, Range)
realized by Standard renaming

... end renaming

LF is One_Way_List_Template(RF.Record)
realized by Pointer_Based renaming

... end renaming
end facilities

local operations

... end local operations
end realization auxiliary

interface
type Partial_Map is represented by LF.List

... operation Is_Undefined_Everywhere returns control parameters
preserves m: Partial_Map
end parameters
ensures Is_Undefined_Everywhere iff "forall d: Domain, not Defined(m(d))"
begin
LF.Reset(m)
if LF.At_Right_End(m) then
return yea
else
return no
endif
end Is_Undefined_Everywhere
... end interface
end Linear_Search

Figure 3.1 — A Fragment of Linear Search for Partial_Map_Template

Figure 3.1 reproduces the facility declarations of the realization Linear_Search from Figure 2.8 for ready reference. Also shown is an implementation for the operation
Is_Undefined_Everywhere. Following the notations discussed in the last chapter, the execution time of this operation can be expressed precisely as:

\[ T_{Is\_Undefined\_Everywhere}(m) = T_{LF\_Reset}(m) + T_{LF\_At\_Right\_End}(m) \] (3.1)

The realization Pointer_Based used for LF provides a constant time implementation of each of the One_Way_List_Template operations. The execution time of Is_Undefined_Everywhere is therefore a constant.

There is another parameterless realization Pair_Of_Stacks for One_Way_List_Template that has performance characteristics identical to Pointer_Based except that the complexity of the operation Reset(m) is \(\Theta(n)\), where \(n\) is the number of elements in the list \(m\). The precise details of this implementation are not essential for the discussion here. Briefly, it uses two stacks to represent a one-way list: one represents the part of the list to the left of its conceptual cursor and the other represents the part to the right. The Reset operation is implemented by popping off all the elements of the stack representing the left part of the list and pushing them on the other stack. This operation therefore has average and worst-case time complexity of \(\Theta(n)\).

If Pair_Of_Stacks is used for the facility LF in Figure 3.1 (instead of Pointer_Based), the complexity of the operation Is_Undefined_Everywhere will no longer be \(\Theta(1)\). Instead, it will be \(\Theta(n)\), where \(n\) is the number of domain values on which the partial map \(m\) (the only parameter to Is_Undefined_Everywhere) is defined. Clearly, the realizations used for the facilities in Linear_Search affect the performance of the operations provided by it.
It is possible to imagine several versions of the realization Linear_Search, all of them identical except that each uses a different realization of One_Way_List_Template for the facility LF. The performance of the operations from Linear_Search may be different in each case. Typically, a reusable component is constructed using facilities from several abstractions, and the realizations chosen for each of these facilities influence the performance of the component. If a component uses m abstractions and each of these abstractions has n different implementations, as many as \( n^m \) performance behaviors are possible for the component without any modifications to the algorithms used in the component. This observation naturally raises the question of whether this aspect of performance can be parametrized. Parametrization will allow a client of Linear_Search to choose the realizations to be associated with each of the facilities used in Linear_Search, and therefore tune the performance of the operations provided by Linear_Search to suit his or her needs.

In the component industry view, only object code versions of realizations will be sold, and not source code. This means that a client cannot do the above-proposed performance tuning without parametrization mechanisms, because the facility declarations (where realizations of constituent abstractions are set) are embedded within the source code of the realization Linear_Search. Even if source code were available, though, performance tuning could not be done with the generality permitted by the mechanisms proposed in this chapter.

### 3.1.2 Passing Parameter-less Realizations as Parameters to Realizations

Figure 3.2 shows a sketch of the realization Tunable_Linear_Search for Partial_Map_Template. This realization is similar to the realization Linear_Search shown in
Figure 2.8 except for the following two differences. Tunable_LINEAR_SEARCH has an additional formal parameter LR, a realization for One_Way_List_Template. The facility LF is realized by the formal realization LR rather than an actual realization such as Pointer_Based.

```
realization Tunable_LINEAR_SEARCH for Partial_Map_Template
  realization parameters
    operation Are_Equal_Domain_Values returns control
      parameters
        preserves x: Domain
        preserves y: Domain
      end parameters
      ensures Are_Equal_Domain_Values iff "x = y"
    end Are_Equal_Domain_Values
  end realization parameters

realization auxiliary
  facilities
    RF is Record_2_Template(Domain, Range)
      realized by Standard
  end facilities
  LF is One_Way_List_Template(RF.Record)
    realized by LR
  end facilities
  ...
end realization auxiliary

interface
  ...
end interface
end Tunable_LINEAR_SEARCH
```

Figure 3.2 — A Performance Tunable Realization for Partial_Map_Template

A client of Tunable_LINEAR_SEARCH is informed how performance of the operations provided by it are affected by the performance behavior of the realization of One_Way_List_Template that is actually plugged in. Figure 3.3 shows a client of Tunable_LINEAR_SEARCH similar to the one in Figure 2.9. Observe that in the facility declaration for PMF, a realization Pointer_Based has been passed as a parameter to
Tunable_Linear_Search. To have a different performance for PMF, one of the things the client can do is to pass Pair_Of_Stacks as a parameter. Of course, the client can also choose to use one of the other realizations of Partial_Map_Template as discussed in the last chapter.

```
... realization auxiliary
facilities
  RNF is Real_Number_Template realized by Standard
  IF is Integer_Template realized by Standard
  PMF is Partial_Map_Template(IF.Int, RNF.Real)
    realized by Tunable_Linear_Search(Are_Equal_Int, Pointer_Based)
end facilities
local operations
... end local operations
end realization auxiliary
...
```

Figure 3.3— A Client of Tunable_Linear_Search

The solution suggested here is certainly elegant and simple. However, this solution needs to be modified when realizations can have parameters. In the example, both the realizations of One_Way_List_Template are parameterless. If realizations have parameters such as the ones discussed in the last chapter for Partial_Map_Template, passing them as parameters to realizations results in the need for a modified parameter-passing mechanism as explained in the next section.
3.2 General Mechanisms for Constituent Performance Tuning

3.2.1 An Example

To explain the issues involved in passing a realization that has parameters as a parameter to another realization, we consider another example here. In particular, we discuss a realization that has as one of its parameters a realization of Partial_Map_Template. Because there are several realizations for Partial_Map_Template each with a different set of parameters, this example helps illustrate the ideas clearly.

Figure 3.4 shows a generic conceptualization AC_2D_Array_Template. This conceptualization abstracts an unbounded, two-dimensional array of Item in which all but a few entries have a constant item value. Hence the name almost-constant (AC), two-dimensional array. The constant elements have the value Item_Const, also brought in as a parameter to the conceptualization. Notice that what is brought in as a parameter is only a mathematical definition of Item_Const, and not a constant used in programs. As noted earlier, RESOLVE conceptualizations cannot bring in program operations (and hence constants, which are really zero-ary operations) as parameters, but can use only their mathematical counterparts. The integers used for indexing the array come from the facility parameter IF of Integer_Template.

```plaintext
conceptualization AC_2D_Array_Template
parameters
  facility IF is Integer_Template
  renaming
    IF.Int as Int
    IF.Integer as Integer
end renaming
  type Item
  definition Item_Const: math[Item]
end parameters
```
auxiliary
  math facilities
    Tuple_2_Theory is Tuple_2_Theory_Template
      (Integer, Integer)
      renaming
        Tuple_2_Theory.Tuple as RC_Pair
        Tuple_2_Theory.Projection_1 as Row
        Tuple_2_Theory.Projection_2 as Column
      end renaming

    Function_Theory is Function_Theory_Template
      (RC_Pair, math[Item])
      renaming
        Function_Theory.Function as AC_2D_Model
      end renaming
  end math facilities
end auxiliary

interface
type AC_2D_Array is modeled by AC_2D_Model
  exemplar a
    initially "forall p: RC_Pair, a(p) = Item_Const"
  end AC_2D_Array

  operation Swap_Entry
    parameters
      alters a: AC_2D_Array
      preserves i: Int
      preserves j: Int
      alters x: Item
    end parameters
    ensures "exists p: RC_Pair, 
      Row(p) = i and Column(p) = j and Delta(a, p) and 
      a(p) = #x and x = #a(p)"
  end Swap_Entry

  operation Replica returns b: AC_2D_Array
    parameters
      preserves a: AC_2D_Array
    end parameters
    ensures "b = a"
  end Replica

  operation Are_Equal returns control
    parameters
      preserves a: AC_2D_Array
      preserves b: AC_2D_Array
    end parameters
    ensures Are_Equal iff "a = b"
  end Are_Equal
end interface
AC_2D_Array_Template provides a type AC_2D_Array. This type has been modeled as a total function whose domain is the set of ordered pairs of integers and whose range is math[Item]. Initially, the function is constant everywhere and maps all domain values to Item_Const. The Swap_Entry operation can be used to change the value of the function at a given domain value. This conceptualization also provides the operations Replica and Are_Equal on the type provided. It is possible to provide efficient implementations of these operations only if they are made primary, i.e., only if the representation of the type AC_2D_Array is available. If these operations are not made primary operations, unless clients of this conceptualization keep track of each call to Swap_Entry, they cannot implement Replica and Are_Equal operations efficiently. Therefore, following the design guidelines suggested earlier, these operations have been made primary.

Figure 3.5 shows one possible realization for AC_2D_Array_Template. It uses the abstractions Record_2_Template and Partial_Map_Template. The “interesting” elements of the unbounded array — those not equal to Item_Const — are the ones stored in the partial map representing the type AC_2D_Array. Integer_Additional_Capability provides the secondary operations Replica and Are_Equal for Integer_Template and Record_2_Additional_Capability provides these operations for Record_2_Template. (The standard implementation of Record_2_Additional_Capability needs equality-testing and replica operations on each of the two types of items contained in the record to be passed as parameters.)
realization AC_2D_Array_Real for AC_2D_Array_Template
realization parameters
  operation Item_Const_Opn returns c: Item
  ensures "c = Item_Const"
end Item_Const_Opn

operation Are_Equal_Item_Values returns control parameters
  preserves x: Item
  preserves y: Item
end parameters
  ensures Are_Equal_Item_Values iff "x = y"
end Are_Equal_Item_Values

operation Item_Replica returns y: Item
  preserves x: Item
end parameters
  ensures "y = x"
end Item_Replica
end realization parameters

realization auxiliary
facilities
  ICF is Integer_Additional_Capability(IF)
  realized by Standard
  RF is Record 2_Template(Int, Int) realized by Standard
  RCF is Record 2_Additional_Capability(RF)
  realized by Standard(ICF.Are_Equal, ICF.Replica, ICF.Are_Equal, ICF.Replica)
  PMF is Partial_Map_Template(RF.Record, Item)
  realized by Linear_Search(RCF.Are_Equal)
  renaming
    PMF.Defined as Defined
    PMF.Range_Value as Range_Value
  end renaming
end facilities
end realization auxiliary

interface
  type AC_2D_Array is represented by PMF.Partial_Map
  realization exemplar real_a for a
    correspondence "forall p: RC_Pair,
    forall x: math[Item],
    (a(p) = x and x /= Item_Const) iff
    (Defined(real_a(p)) and
    x = Range_Value(real_a(p)))"
end AC_2D_Array
operation Swap_Entry
parameters
  alters a: AC_2D_Array
  preserves i: Int
  preserves j: Int
  alters x: Item
end parameters
ensures "exists p: RC_Pair,
  Row(p) = i and Column(p) = j and Delta(a, p) and
  a(p) = x and x = a(p)"

begin
local variables
  i_copy, j_copy: IF.Int
  p: RF.Record
  temp: Item
end local variables

i_copy := ICF.Replica(i)
j_copy := ICF.Replica(j)
RF.Swap_Entry_1(p, i_copy)
RF.Swap_Entry_2(p, j_copy)

temp := Item_Const_Opn
if PMF.Is_Defined(a, p) then
  if Are_Equal_Item_Values(x, temp) then
    PMF.Make_Undef
  else
    PMF.Swap_Entry(a, p, x)
  endif
else
  if not Are_Equal_Item_Values(x, temp) then
    PMF.Make_Def
    x := temp
  endif
endif
end Swap_Entry

operation Replica returns b: AC_2D_Array
parameters
  preserves a: AC_2D_Array
end parameters
ensures "b = a"

begin
local variables
  temp: PMF.Partial_Map
  p, p_copy: RF.Record
  x, x_copy: Item
end local variables
maintaining \( \text{"temp} = b \text{ and} \)

\[
(\exists p : \text{PF.Pair}, \text{Defined}(\#a(p)) \implies \\
(\exists r : \text{math[Item]}, \text{Range Value}(\#a(p)) = r \text{ and} \\
(\text{Defined(temp(p)) and Range Value(temp(p)) = r) iff} \\
\text{not Defined}(a(p))))\]

while not \( \text{PMF.Is_Undef ined_Everywhere}(a) \) do

\( \text{Make Any One Undefined}(a, p, x) \)

\( p\_copy := \text{RCF.Replica}(p) \)

\( x\_copy := \text{Item Replica}(x) \)

\( \text{PMF.Make Defined}(\text{temp}, p, x) \)

\( \text{PMF.Make Defined}(b, p\_copy, x\_copy) \)

endwhile

\( a := \text{temp} \)

end Replica

end interface

end realization AC_2D_Array_Real

Figure 3.5 — A Realization of AC_2D_Array_Template Using Partial_Map_Template

For the facility PMF, the realization Linear_Search has been used. How can we construct a tunable version of the realization AC_2D_Array_Real whereby a client can pick an appropriate realization for Partial_Map_Template? The mechanism suggested in the previous section for parametrization raises the following problem.

Different realizations of Partial_Map_Template have different parameter lists. If a client is allowed to choose a realization for Partial_Map_Template, who can pass the appropriate parameters to that realization — the client or the implementer of AC_2D_Array_Real? For the realization Linear_Search, an operation Are_Equal_Domain_Values has to be passed as a parameter. The domain in Figure 3.5 is RF.Record which comes from a local facility of the realization AC_2D_Array_Real, and a client of AC_2D_Array_Real does not know this type. Therefore, a client cannot pass appropriate realization parameters.
One method to solve this problem is to provide clients with all the information contained in the local facility declarations in Figure 3.5, by making all these facilities as parameters to AC_2D_Array_Real. This solution violates basic principles of information hiding and has several associated disadvantages, including making performance tuning expensive for clients. Therefore, we do not consider this possibility any further.

If clients are allowed to choose a realization for Partial_Map_Template and they cannot pass parameters to it, only the implementer of AC_2D_Array_Real must pass these parameters. However, if the realization used for the facility PMF is made a parameter, how can the implementer pass appropriate realization parameters because the parameters to be passed depend on the realization chosen for Partial_Map_Template (by a client)? An answer to this question involves addition of a few other language mechanisms to RESOLVE as explained in the next few sections.

3.2.2 Realization Forms

In the last chapter, we discussed four different realizations of Partial_Map_Template. Each of them has a parameter list of its own. It is also possible that two different realizations have (syntactically and semantically) identical parameter lists. Ordered_Search and Binary_Search_Tree in the example have this property. A realization form captures this commonality, though this is not the only reason for the introduction of this new construct as will become clear later. A realization form is associated with a conceptualization, and it provides one possible realization parameter list for that conceptualization. Figure 3.6 shows a realization form for Partial_Map_Template. Figure 3.8 shows another.
realization form Ordering_Form for Partial_Map_Template

definition $R(x, y: \text{math}[\text{Domain}]): \text{Boolean}$

operation Are_Ordered_Domain_Values returns control
  parameters
  preserves $x$: Domain
  preserves $y$: Domain
  end parameters
  ensures Are_Ordered_Domain_Values iff "$R(x, y)$"
  end Are_Ordered_Domain_Values

constraints
  "forall $x, y, z: \text{math}[\text{Domain}], R(x, x) \text{ and } R(x, y) \text{ and } R(y, x) \implies x = y \text{ and } R(x, y) \text{ and } R(y, z) \implies R(x, z) \text{ and } (R(x, y) \text{ or } R(y, x))""
  end constraints
end Ordering_Form

Figure 3.6 — An Example Realization Form

A realization for a conceptualization no longer needs to list its parameters explicitly. Instead it refers to a form as shown in Figure 3.7. Figure 3.7 shows how the realization Ordered_Search can be written if realization forms are added to RESOLVE. If a realization is parameterless, then the realization heading need not include a form. Of course, different realizations may refer to the same form.

realization Ordered_Search for Partial_Map_Template
  with form Ordering_Form

  realization auxiliary
  ...
  end realization auxiliary

  interface
  ...
  end interface
end Ordered_Search

Figure 3.7 — Usage of a Realization Form
In the components industry, a client of a component is supplied its conceptualization, object code version of a realization, the form of the realization, and also a performance statement of how the parameters in the form affect the performance of the component. A realization form is similar in spirit to the structural interface of a component in traditional engineering disciplines. It can be used as a programming construct to inform a client about the structural interface details of a realization, though structure in this case is both syntactic and semantic.

```
realization form Hashing_Form for Partial_Map_Template

facility IF is Integer_Template
    renaming
      IF.Integer as Integer
      IF.Int as Int
    end renaming

definition Math_Number_Of_Buckets: Integer

operation Number_Of_Buckets returns size: Int
    ensures "size = Math_Number_Of_Buckets"
end Number_Of_Buckets

definition Math_Hash_Fn (x: math[Domain]): Integer

operation Hash_Opn returns h: Int
    parameters
      preserves x: Domain
    end parameters
    ensures "h = Math_Hash_Fn(x)"
end Hash_Opn

operation Are_Equal_Domain_Values returns control
    parameters
      preserves x: Domain
      preserves y: Domain
    end parameters
    ensures Are_Equal_Domain_Values iff "x = y"
end Are_Equal_Domain_Values

constraints
  Math_Number_Of_Buckets > 0
end constraints
end Hashing_Form
```

Figure 3.8 — Another Realization Form for Partial_Map_Template
A realization may either use a realization form or list its parameters explicitly. First, we assume that every realization of Partial_Map_Template discussed here uses a form and propose mechanisms based on this assumption. For example, the realization Hashed_Search is assumed to use the form Hashing_Form (shown in Figure 3.8), Tunable_Linear_Search is assumed to use a form, say, Tunable_Linear_Form that lists the parameters shown in Figure 3.2, and so on. Later, we discuss the case when realizations are allowed to list parameters explicitly without using a form.

3.2.3 Mechanisms for Passing Realizations with Selected Forms As Parameters

With the introduction of forms, we are ready to discuss how they can be used in performance parametrization. Figure 3.10 shows a tunable version of the realization AC_2D_Array_Real for AC_2D_Array_Template.

The realization form used by Tunable_AC_2D_Array_Real is shown in Figure 3.9. Observe that along with the formal realization parameter PMR, the names of some of the realization forms for Partial_Map_Template are also listed. Any actual realization that is passed by a client must have one of these forms or must be parameterless. If not, that will result in a compiler-detectable, syntax error.

The realization Tunable_AC_2D_Array_Real can handle only two realization forms (Ordering_Form and Hashing_Form) for Partial_Map_Template. Refer to the declaration of the facility PMF in Tunable_AC_2D_Array_Real. This facility is shown realized by the formal realization PMR. The parameters passed to this realization are conditional. If the form of the actual realization passed for PMR happens to be of the form Ordering_Form,
all and only the actuals for the parameters listed in Ordering_Form (Math_Pair_Leq and Are_Ordered_Pair_Values) are passed to PMR. If the form happens to be Hashing_Form, a different set of parameters are passed as shown. If the actual realization is parameterless, no parameters are passed. This is implicit. However, if the actual realization has none of these forms, Tunable_AC_2D_Array_Real is not equipped to handle it.

The realization auxiliary section shows the actual mathematical definitions and operations that are passed to the realization PMR. Some of the operations come directly from facilities and the rest have been shown implemented as local operations. Integer_Additional_Capability, for example, provides secondary operations such as Mod based on Integer_Template. Only the definitions and operations needed by the two forms Ordering_Form and Hashing_Form have been coded, and this is the reason the formal parameter list for Tunable_AC_2D_Array_Real in Tunable_AC_2D_Array_Form in Figure 3.9 restricts explicitly the forms of PMR that can be handled. Notice that construction of these definitions and operations relies on knowing the actual conceptualization parameters passed in instantiating the facility PMF, and these are known only to the developer of Tunable_AC_2D_Array_Real, and not to any client. For example, the domain used for PMF, RF.Record, comes from a local facility declaration RF.

This interface section of this realization is identical to that of AC_2D_Array_Real in Figure 3.5, and hence has not been shown here.
realization form Tunable_AC_2D_Array_Form for AC_2D_Array_Template

operation Item(Const)Opn returns c: Item
  ensures "c = Item(Const)"
end Item(Const)Opn

operation Are_Equal_Item_Values returns control
  parameters
    preserves x: Item
    preserves y: Item
  end parameters
  ensures Are_Equal_Item_Values iff "x = y"
end Are_Equal_Item_Values

operation Item_Replica returns y: Item
  parameters
    preserves x: Item
  end parameters
  ensures "y = x"
end Item_Replica

realization PMR for Partial_Map_Template
  with form alternatives
    Ordering_Form
  or Hashing_Form
  end form alternatives
end PMR

end Tunable_AC_2D_Array_Form

Figure 3.9 — A Realization Form for AC_2D_Array_Template

realization Tunable_AC_2D_Array_Real for AC_2D_Array_Template
  with form Tunable_AC_2D_Array_Form

realization auxiliary
  renaming
    IF.Math_Leq as Math_Leq
    IF.Math_Mult as Math_Mult
    IF.Number_Theory.Mod as Math_Mod
    IF.Number_Theory.Constant_97 as Math_Constant_97
  end renaming

definition Math_Number_Of_Buckets: Integer
  "Math_Constant_97"

definition Math_Pair_Hash_Fn(p: math[Pair]): Integer
  "Math_Mod(Math_Mult(|Projection_1(p)|,
  |Projection_2(p)|), Math_Constant_97)"
definition Math_Pair_Leq(p1, p2: math[Pair]): Boolean
"(Math_Leq(Projection_1(p1), Projection_1(p2)) and
not Math_Leq(Projection_1(p2), Projection_1(p1))) or
(Math_Leq(Projection_1(p1), Projection_1(p2)) and
Math_Leq(Projection_1(p2), Projection_1(p1)) and
Math_Leq(Projection_2(p1), Projection_2(p2)))"
operation Are_Ordered_Pair_Values returns control parameters
preserves p1, p2: RF.Record
end parameters
ensures Are_Ordered_Pair_Values iff "Math_Pair_Leq(p1, p2)"
begin
local variables
x1, y1, x2, y2: IF.Int
result: BF.Bool
end local variables
RF.Swap_Entry_1(p1, x1)
RF.Swap_Entry_2(p1, y1)
RF.Swap_Entry_1(p2, x2)
RF.Swap_Entry_2(p2, y2)
if IF.Less_Than_Or_Equal(x1, x2) and
not IF.Less_Than_Or_Equal(x2, x1) then
BF.Set_True(result)
else
if IF.Less_Than_Or_Equal(x1, x2) and
IF.Less_Than_Or_Equal(x2, x1) and
IF.Less_Than_Or_Equal(y1, y2) then
BF.Set_True(result)
else
BF.Set.False(result)
endif
endif
RF.Swap_Entry_1(p1, x1)
RF.Swap_Entry_2(p1, y1)
RF.Swap_Entry_1(p2, x2)
RF.Swap_Entry_2(p2, y2)
if Is_True(result) then
return yes
else
return no
endif
end Are_Ordered_Pair_Values
end local operations
end realization auxiliary
interface
...
end interface
end realization Tunable_AC_2D_Array_Real

Figure 3.10 — A Tunable Realization of AC_2D_Array_Template Using Partial_Map
Figure 3.11 shows a client of Tunable_AC_2D_Array_Real. The client needs to pass four parameters to this realization as listed in the form Tunable_AC_2D_Array_FORM. The fourth parameter is a realization for Partial_Map_Template. The client has chosen to pass Hashed_Search. The client need not worry about the parameters to be passed for Hashed_Search. If the client wants to pass Ordered_Search or Binary_Search_Tree that can be done by just substituting these realizations in place of Hashed_Search. Their parameters are passed by Tunable_AC_2D_Array_Real. We have discussed two other realizations for Partial_Map_Template—Linear_Search and Tunable_Linear_Search. The parameter lists of these realizations match neither of the forms Ordering_Form or Hashing_Form, and hence they cannot be passed as parameters to Tunable_AC_2D_Array_Real. The next section discusses the mechanisms needed to allow these realizations to be passed.

```plaintext
realization auxiliary
facilities
  RNF is Real_Number_Template realized by Standard
  renaming
    RNF.Real_Number_Theory.Constant_0 as Constant_0
  end renaming
  RCF is Real_Number_Additional_Capability(RNF)
  realized by Standard
  IF is Integer_Template realized by Standard
  AF is AC_2D_Array_Template
    (IF, RNF.Real, Constant_0)
  realized by Tunable_AC_2D_Array_Real
    (RCF.Generate_0, RCF.Are_Equal,
    RCF.Replica, Hashed_Search)
end facilities
end realization auxiliary
...

Figure 3.11— A Client of AC_2D_Array_Template
3.2.4 Enhancing The Generality of a Tunable Realization

Assume that a form Tunable_Linear_Form with parameters exactly as shown in Figure 3.2 exists and the realization Tunable_Linear_Search has this form. To be able to handle this new form of Partial_Map_Template, the developer of Tunable_AC_2D_Array_Real has to make two changes to his implementation. Tunable_Linear_Form has to be added to the alternatives in the form Tunable_AC_2D_Array_Form shown in Figure 3.9, and the facility declaration for PMF has to be modified to take care of this new realization form as shown in Figure 3.12.

```
facilities

PMF is Partial_Map_Template(RF.Record, Item)
    realized by PMR(Ordering_Form:
        Math_Pair_Leq,
        Are_Ordered_Pair_Values
    Hashing_Form:
        IF,
        Math_Number_Of_Buckets,
        Number_Of_Buckets,
        Math_Pair_Hash_Fn,
        Pair_Hash_Opn,
        RCF.Are_Equal
    Tunable_Linear_Form:
        RCF.Are_Equal,
        Pointer_Based)

renaming
    PMF.Partial_Map as AC_2D_Array_Rep
end renaming

end facilities
```

Figure 3.12— Modification of Tunable_AC_2D_Array_Real

Two interesting observations can be made in this context. The first is that existing clients of Tunable_AC_2D_Array_Real are unaffected by the change. For example, note that the
client in Figure 3.11 is not affected by the change to Tunable_AC_2D_Array_Real discussed here.

The second observation relates to the practicability of this approach. If a realization R1 has as one of its parameters a realization R2 of some conceptualization C2, then addition of a new realization form for C2 will result in the above modification of R1. This is certainly a problem, but this situation is at least as good as it is in traditional engineering disciplines. In the electronics industry, a component is typically built with certain requirements on the structural interfaces of the components that are plugged into that component. If a component that is to be plugged in does not match the requirements, it just cannot be used. The proposed solution for the software components industry is certainly better than this situation in the electronics industry. In a reusable components industry typically new realizations can be built with some standard forms that are already in the market, and rarely will new forms have to be constructed. Also, a developer of a tunable realization will usually consider all possible realization forms that already exist in the market for the realization needed to be made a parameter.

The realization Tunable_AC_2D_Array_Real allows a client to choose a realization for a facility of Partial_Map_Template used in it. The next question is whether this generality can be extended to deeper layers. Can a client also be allowed to choose the realizations that are used in the chosen realization for Partial_Map_Template? The realization form Tunable_Linear_Form, for example, lists a realization LR of One_Way_List_Template as one of its parameters. In Figure 3.12, the developer of Tunable_AC_2D_Array_Real has chosen to pass the realization Pointer_Based (for One_Way_List_Template) for the facility PMF for the alternative Tunable_Linear_Form. How can a client of Tunable_AC_2D_Array_Real be allowed to choose this realization as well? Figure 3.13
shows how the parameter list of Tunable_AC_2D_Array_Real (in Tunable_AC_2D_Array_Form) should read to make this possible. To the facility declaration PMF, the developer now passes the formal realization LR for One_Way_List_Template rather than Pointer_Based as shown in Figure 3.14.

\begin{verbatim}
realization form Tunable_AC_2D_Array_Form for AC_2D_Array_Template

operation Item_Const_Opn returns c: Item
  ensures "c = Item_Const"
end Item_Const_Opn

operation Are_Equal_Item_Values returns control parameters
  preserves x: Item
  preserves y: Item
end parameters
  ensures Are_Equal_Item_Values iff "x = y"
end Are_Equal_Item_Values

operation Item_Replica returns y: Item parameters
  preserves x: Item
end parameters
  ensures "y = x"
end Item_Replica

realization PMR for Partial_Map_Template with form alternatives
  Ordering_Form
  or Hashing_Form
  or Tunable_Linear_Form
realization parameters
  realization LR for One_Way_List_Template
end realization parameters
end form alternatives

end Tunable_AC_2D_Array_Form
\end{verbatim}

Figure 3.13 — A Realization Form for AC_2D_Array_Template
facilities

PMF is Partial_Map_Template(RF.Record, Item)
realized by PMR(Ordering_Form:
    Math_Pair_Leq,
    Are_Ordered_Pair_Values
Hashing_Form:
    IF,
    Math_Number_Of_Buckets,
    Number_Of_Buckets,
    Math_Pair_Hash_Fn,
    Pair_Hash_Opn,
    RCF.Are_Equal
Tunable_Linear_Form:
    RCF.Are_Equal,
    LR)

renaming
    PMF.Partial_Map as AC_2D_Array_Rep
end renaming
end facilities

Figure 3.14— Modification of Tunable_AC_2D_Array_Real

The parameter list in Figure 3.13 tells a client that whenever a realization with form Tunable_Linear_Form is passed as a parameter, then the client must also pass a realization of One_Way_List_Template as a parameter as shown in Figure 3.15. If, however, a client uses a realization with one of the other forms of Partial_Map_Template as does the client in Figure 3.12, that client is not affected. If a realization of One_Way_List_Template is made an additional parameter to Tunable_AC_2D_Array_Real, every client will be forced to pass a realization of One_Way_List_Template as a parameter even though some realizations forms of Partial_Map_Template may not use that realization.
realization auxiliary
facilities
  RNF is Real_Number_Template realized by Standard renaming
      RNF.Real_Number_Theory.Constant_0 as Constant_0
  end renaming
  RNCF is Real_Number_Additional_Capability(RNF)
      realized by Standard
  IF is Integer_Template realized by Standard
  AF is AC_2D_Array_Template
      (IF, RNF.Real, Constant_0)
      realized by Tunable_AC_2D_Array_Real
      (RNCF.Generate_0, RNCF.Are_Equal,
       RNCF.Replica, Tunable_Linear_Search(Pointer_Based))
  end facilities
end realization auxiliary

Figure 3.15— A Client of AC_2D_Array_Template Using Tunable_Linear_Search

The parameter nesting discussed here can be arbitrarily deep. It is possible that alternative forms may be nested as well. This would indeed be the case if the realizations for One_Way_List_Template had alternative forms rather the ones discussed here. A facility declaration may also include nested choices for these alternatives, as illustrated in section 3.2.6. The mechanisms discussed here are therefore general, and allow developers and clients the ability to implement and tune reusable software components as finely as desired. In practice, developers may choose to parametrize and thus provide control for clients over only some constituents of a component, as seen in the examples here.
3.2.5 On Passing Realizations Without Forms As Parameters

In the discussions so far, we have assumed that every realization refers to a realization form. One more mechanism has to be added to permit realizations which do not refer to any named form, but explicitly list their parameters. This case is explained in this section.

Suppose that the realization form Ordering_Form does not exist and that the realizations Ordered_Search and Binary_Search_Tree list their parameters explicitly. How can the builder of Tunable_AC_2D_Array_Real allow these realizations to be passed as parameters? Figure 3.16 shows the modified Tunable_AC_2D_Array_Form to handle this case. The only difference between this form and the form shown in Figure 3.13 is that the definition of a new form called Local_Ordering_Form has been explained locally. Inside the realization Tunable_AC_2D_Array_Real, the facility declaration for PMF now refers to the local name Local_Ordering_Form rather than a globally known name Ordering_Form. A client can pass Ordered_Search as a realization for Partial_Map_Template because the parameter list of of that realization matches that listed under Local_Ordering_Form.

Notice the important but subtle distinction between realization parameters that constitute a form and parameters that must be supplied by a client for the case of a particular form. The parameters listed under Local_Ordering_Form (and the ones implicitly listed by Hashing_Form or Tunable_Linear_Form) state what parameters are supplied by the developer of Tunable_AC_2D_Array_Real, whereas the parameter realization LR for One_Way_List_Template must be supplied by a client. The purpose of realization forms in this context should now be clear. They make the presentation of the formal parameter list of Tunable_AC_2D_Array_Real more comprehensible.
realization form Tunable_AC_2D_Array_Form for AC_2D_Array_Template

operation Item_Const_Opn returns c: Item
    ensures "c = Item_Const"
end Item_Const_Opn

operation Are_Equal_Item_Values returns control
    preserves x: Item
    preserves y: Item
end parameters
    ensures Are_Equal_Item_Values iff "x = y"
end Are_Equal_Item_Values

operation Item_Replica returns y: Item
    preserves x: Item
end parameters
    ensures "y = x"
end Item_Replica

realization PMR for Partial_Map_Template with form alternatives
Local_Ordering_Form
    definition R(x, y: math[Domain]): Boolean

operation Are_Ordered_Domain_Values returns control
    preserves x: Domain
    preserves y: Domain
end parameters
    ensures Are_Ordered_Domain_Values iff "R(x, y)"
end Are_Ordered_Domain_Values

constraints
    "forall x, y, z: math[Domain], 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))"
end constraints
or Hashing_Form
or Tunable_Linear_Form
    realization parameters
    realization LR for One_Way_List_Template
end realization parameters
end form alternatives
end Tunable_AC_2D_Array_Form

Figure 3.16 — A Modified Realization Form for AC_2D_Array_Template
3.2.6 A Comprehensive Example Showing Performance Parametrization

Before concluding our discussion on constructing tunable realizations, it is useful to see what a generic, tunable realization looks like. Let us assume that we are building a tunable realization TR1_A1 with form F1_A1 for A1. F1_A1 and a form F1_A2 used by F1 are shown below.

```
realization form F1_A1 for A1

  realization R_A2 for A2
  with form alternatives
    F1_A2:
      realization parameters
        realization R_A3 for A3
        with form alternatives
          F1_A3:
          or F2_A3:
        end form alternatives
      end realization parameters
    or F2_A2:
    end form alternatives

end F1_A1

realization form F1_A2 for A2

  realization R_A3 for A3
  with form alternatives
    F1_A3:
    or F2_A3:
  end form alternatives

end F1_A2
```

In this figure, A2 and A3 are conceptualizations. F1_A2 and F2_A2 are realizations forms for A2, and F1_A3 and F2_A3 are forms for A3. The facility declaration of A2 within the tunable realization TR1_A1 is shown below.
Finally, a client facility declaration using TR1_A1 for A1 appears as below. Here, TR1_A2 is a realization of A2 with form F1_A2 and R1_A3 is a realization of A3 with form F1_A3.

facilities
...  
A2_F is A2(...)
   realized by R_A2(F1_A2:
      R_A3(F1_A3:
        F2_A3:
        ...
      )
    )
  )
F2_A2:
...
)

end facilities
...

3.3 Reshaping of Realizations

We have discussed methods that permit construction of highly performance-tunable reusable software components in the last few sections. Sometimes, a developer of a component may decide to market a customized version of a component to suit a specific client for whatever reasons. In this case, we suggest that a tunable component be built first and then a specific version of that component be derived from it as shown in this section. For example, a specific version of the realization Tunable_AC_2D_Array_Real can be created whereby a realization of Partial_Map_Template no longer needs to be passed as a parameter. Figure 3.17 shows how this realization named
Hashed_Search_AC_2D_Array_Real can be constructed. The form of this realization
Non_Tunable_AC_2D_Array_Form has only the first three parameters listed in the
Tunable_AC_2D_Array_Form in Figure 3.9.

```
 realization  Hashed_Search_AC_2D_Array_Real  for
             AC_2D_Array_Template
             with  form  Non_Tunable_AC_2D_Array_Form

 realization auxiliary
 facilities
 ACF  is  AC_2D_Array_Template(IF, Item, Item_Const)
 realized  by
 Tunable_AC_2D_Array_Real(Item_Const_Opn,
 Are_Equal_Item_Values, Item_Replica,
 Hashed_Search)
 end facilities
 end realization auxiliary

 interface
 renaming
 ACF.AC_2D_Array  as  AC_2D_Array
 ACF.Swap_Entry  as  Swap_Entry
 ACF.Are_Equal  as  Are_Equal
 ACF.Replica  as  Replica
 end renaming
 end interface
 end Hashed_Search_AC_2D_Array_Real
```

Figure 3.17 — Creating a Specific Version from a Tunable Version

The realization in Figure 3.17 has declared a facility ACF of AC_2D_Array_Template
using the tunable realization Tunable_AC_2D_Array_Real, passing Hashed_Search as a
parameter realization. The other parameters passed to Tunable_AC_2D_Array_Real are
exactly the ones passed to Non_Tunable_AC_2D_Array_Real by a client.

The interface section does not directly provide the type or the code for each of the
operations of AC_2D_Array_Template. Instead, the type AC_2D_Array is supplied by
renaming the type ACF.AC_2D_Array. The operations are provided by renaming the

appropriate operations from the facility ACF. Renaming of an operation in the interface section has the following semantics. If an operation P on the right hand side of renaming is called, the operation Q on the left hand side of that renaming is executed, and the results returned by Q are returned by P. For each renaming, the syntax as well as specifications of the operations on the left and right hand sides must be identical. The operations Initialize and Finalize on the type AC_2D_Array are implicitly provided by the renaming of the type ACF.AC_2D_Array as AC_2D_Array. The other operations are provided by explicit renaming.

There is another use for the reshaping mechanism as well. The realization form Tunable_AC_2D_Array_Form shown in Figure 3.13 apparently does not allow a client to pass Linear_Search as a realization of Partial_Map_Template. Linear_Search has the form Linear_Form shown in Figure 3.18, and this is not exactly the same as any of the three forms recognized by Tunable_AC_2D_Array_Real.

```
realization form Linear_Form for Partial_Map_Template
operation Are_Equal_Domain_Values returns control
parameters
  preserves x: Domain
  preserves y: Domain
end parameters
ensures Are_Equal_Domain_Values iff "x - y"
end Are_Equal_Domain_Values
end Linear_Form
```

**Figure 3.18— Another Realization Form for Partial_Map_Template**

One approach to solving this problem is to request the developer of realization Tunable_AC_2D_Array_Real to modify its form to include Linear_Form as one of the possibilities for Partial_Map_Template realization. A comparison of the forms
Linear_Form and Tunable_Linear_Form suggests approach based on reshaping, discussed below.

Figure 3.19 shows another realization of Partial_Map_Template using Tunable_Linear_Form. This realization creates a facility of Partial_Map_Template using the realization Linear_Search. Tunable_Linear_Form lists two parameters of which only Are_Equal_Domain_Values is needed by Linear_Search. This is the only parameter that is used by Reshaped_Linear_Search. All its other parameters are ignored. It is interesting to note that the performance statement of Reshaped_Linear_Search is independent of the realization of One_Way_List_Template passed as a parameter to it.

```plaintext
realization Reshaped_Linear_Search for Partial_Map_Template
    with form Tunable_Linear_Form

    realization auxiliary
        facilities
            PMF is Partial_Map_Template(Domain, Range)
            realized by Linear_Search(Are_Equal_Domain_Values)
        end facilities
    end realization auxiliary

interface
    renaming
        PMF.Partial_Map as Partial_Map
        PMF.Make_DEFINED as Make_DEFINED
    ...
    end renaming
end interface
end Reshaped_Linear_Search
```

Figure 3.19 — Reshaping a Realization to Suit a Specific Form
3.4 Summary

In this chapter, we have discussed a set of programming language mechanisms for constructing and using highly tunable reusable software components. Even when specific versions are needed with particular performance behaviors, it is easy to construct them from tunable versions. How effectively these mechanisms can be used depends partly on the availability of decision-making tools for clients that will help them pick appropriate components to plug into a component to tune its performance to their needs. The complexity of this decision-making problem depends on the number of operations provided by an abstraction and the number of components available to realize a given abstraction.

Finally, it is obvious that fully general tuning results in very complicated realizations. The good news is that reshaping of such a complex realization is straightforward and concise and that clients — even of the fully tunable version — are no more complex than those of non-tunable versions.
CHAPTER IV
DISTRIBUTED EXECUTION OF SOFTWARE COMPONENTS

The performance of a software component is influenced by the manner in which a hardware architecture is used in executing the component. Some components marketed in the software industry will be tailored to execute efficiently on some specific hardware architectures. Some others will be more general, built making few or no assumptions about the hardware architecture on which they may be executed, and can therefore be used across a wide spectrum of architectures. This chapter discusses the issues involved in tuning these general components for efficient execution on an arbitrary distributed hardware architecture.

In the software components industry view proposed in this dissertation, a component is marketed in object-code form with its specification. For a tunable component, clients are also supplied information about its constituents and the influence of the performance of constituents over the performance of the component. The performance statement provides a measure of interaction among the constituents and the component. A client can use this information along with the knowledge about his or her specific hardware architecture, such as the execution efficiency of each machine in the architecture and the communication delays between each pair of these machines, to make an appropriate decision on mapping the component and its constituents to the machines in the hardware environment for efficient execution. In making this decision, the client may be assisted by automated
distribution tools. Given that components have been appropriately distributed among the processors of an architecture, this dissertation proposes a suitable method of execution.

The proposed method of execution allows representations of data values to be distributed maximally, and thus permits components to manipulate them in parallel, effectively exploiting the potential for performance resulting from distribution. A distinguishing characteristic of the method is that it permits communication of values among participating components without requiring access to the actual representations of these values. This is critical because representations are hidden in object code versions of components and, therefore, are not universally known. Even if representational details are available, movement of representations is undesirable for several reasons as explained later in this chapter. Most approaches to distributed execution of software rely on knowing representational details of data values [Liskov 83, Black 87], and move them among machines during execution.

The proposed method considers only execution of components written in a sequential programming language like RESOLVE. In the component industry, some components may be sequential and others may be “parallel.” However, a parallel component may be built using already available sequential components. In general, even when developers explicitly identify the parallelism in their components using some parallel programming language, there is considerable room for parallel execution of sequential parts of these components, since the grain level specified for parallelism in a parallel program may not be “right” for a given hardware environment [Kruatrachue 88]. Though the proposed method considers only execution of sequential components, there is no reason why it cannot be extended for components written in a well-designed parallel programming language as well. This question is not further considered here, though.
The chapter is organized as follows. Section 4.1 presents an example that is used in the rest of this chapter. Section 4.2 discusses several elements of the proposed execution method and shows execution snapshots for the example. It includes a discussion of methods for extracting parallelism automatically. Section 4.3 reviews the communication mechanisms in other distributed systems, discusses various issues related to parameter passing to remote operations, and shows why data representation movement used in other systems is both undesirable and unnecessary for performance tuning in the component industry view. Section 4.4 provides a summary.

4.1 An Example

We will use an example RESOLVE component built from other components to illustrate the proposed execution method. Figure 4.1 shows the conceptualization for a component that provides a generic type Queue and the operations Enqueue, Dequeue, and Is_Empty for manipulating variables of type Queue. Figure 4.2 shows the specification for a secondary operation Queue_Reverse on the type Queue and Figure 4.3 shows the source code for a realization of Queue_Reverse_Capability using Stack_Template. This implementation of Queue_Reverse takes linear time in the number of elements in the queue to be reversed, and is not efficient as it could be if implemented directly, as noted in section 2.4. However, it helps illustrate several aspects of the proposed distributed execution method.
conceptualization Queue_Template
parameters
type Item
end parameters

auxiliary
math facilities
String_Theory is String_Theory_Template (math[Item])
renaming
String_Theory.String as String
String_Theory.Lambda as Lambda
String_Theory.Post as Post
String_Theory.Pre as Pre
end renaming
end math facilities
end auxiliary

interface
type Queue is modeled by String
exemplar q
initially "q = Lambda"
end Queue

operation Enqueue
parameters
alters q : Queue
consumes x : Item
end parameters
ensures "q = Post(#q, #x)"
end Enqueue

operation Dequeue
parameters
alters q : Queue
produces x : Item
end parameters
requires "q ≠ Lambda"
enforces "#q = Pre(q, x)"
end Dequeue

operation Is_Empty returns control
parameters
preserves q : Queue
end parameters
ensures Is_Empty iff "q = Lambda"
end Is_Empty
end interface

description
Queue_Template provides the type family "Queue of Item," where Item is any type. In the formal specifications above, an abstract queue is a string of Items, with the front of the queue at the left end of the string. Initially, every queue is empty.
“Enqueue(q, x)” adds x to the end of queue q. Since x is a consumes parameter, it has an initial value for its type upon return.

“Dequeue(q, x)” dequeues the front element off queue q and returns it in x.

“Is_Empty(q)” returns yes if and only if q is an empty queue.

Figure 4.1 — Specification of Queue_Template

collection Queue_Reverse_Capability
  parameters
    facility QF1 is Queue_Template
    facility QF2 is Queue_Template
  restrictions
    QF1.Item = QF2.Item
  end restrictions
end parameters

auxiliary
  renaming
    QF1.String_Theory.Reverse as Reverse
  end renaming
end auxiliary

interface
  operation Queue_Reverse
  parameters
    consumes q1: QF1.Queue
    produces q2: QF2.Queue
  end parameters
  ensures "q2 = Reverse(#q1)"
end Queue_Reverse
end interface

description

Figure 4.2 — Specification of Queue_Reverse Operation
realization Linear_Time_Using_Stack for
Queue_Reverse_Capability
realization parameters
realization SR for Stack_Template
end realization parameters

realization auxiliary
renaming
QFI.Item as Item
QFI.String_Theory.Cat as Cat
end renaming
facilities
SF is Stack_Template(Item) realized by SR
end facilities
end realization auxiliary

interface
operation Queue_Reverse
parameters
consumes ql: QFI.Queue
produces q2: QF2.Queue
end parameters
ensures "q2 = Reverse(ql)"
begin
local variables
x: Item
s: SF.Stack
q: QF2.Queue
end local variables
q2 := q
maintaining "Cat(Reverse(s), ql) = #ql"
while not QFI.Is_Empty(ql) do
    QFI.Dequeue(ql, x)
    SF.Push(s, x)
endwhile

maintaining "Cat(s, Reverse(q2)) = #s1"
while not SF.Is_Empty(s) do
    SF.Pop(s, x)
    QF2.Enqueue(q2, x)
endwhile
end Queue_Reverse
end interface
end Linear_Time_Using_Stack

Figure 4.3 — One Possible Realization of Queue_Reverse

Figure 4.4 shows a tunable component R that uses Queue_Template and Queue_Reverse_Capability. To make the example a bit simpler, the realization SR of
Stack_Template has not been made a conditional parameter as suggested in the last chapter. (It will be needed when the realization Linear_Time_Using_Stack is passed as a parameter for Queue_Reverse_Capability and may not be needed if other realizations are used.)

realization of C_Template by R
realization parameters
  realization IR for Integer_Template
  realization SR for Stack_Template
  realization QR for Queue_Template
  realization RQR for Queue_Reverse_Capability
end realization parameters
realization auxiliary
...
facilities
  IF is Integer_Template realized by IR
  QF is Queue_Template(IF.Int) realized by QR
  RQF is Queue_Reverse_Capability(QF, QF)
    realized by RQR(SR)
end facilities
end realization auxiliary

interface
...
operation P
  parameters
...
end parameters
ensures "..."
begin
  local variables
...
    q1, q2 : QF.Queue
  end local variables
...
  RQF.Queue_Reverse(q1, q2)
...
end P
...
end interface
end R

Figure 4.4 — A Tunable Component That Uses Queue_Reverse_Capability
The source code of the realization Linear_Time_Using_Stack and a code fragment of R are shown here only to illustrate how the execution method works. They will not be available to a client in the component industry, of course.

4.2 Elements of the Proposed Execution Method

This section discusses several aspects of the proposed execution method. Execution snapshots of the example discussed in the last section are used to illustrate the ideas.

4.2.1 Distribution on Physical Machines

Figure 4.5 shows a client of C_Template. This is a realization built by an "end-user" for one-time use. Based on the performance statement supplied with components, this client has decided to use the realization R for C_Template and has also chosen appropriate realizations for each of the constituents of R.

```
realisation of Client_Concept by Client
realisation auxiliary
facilities
  CF is C_Template realised by
  R(Standard, Pointer_Based,
    Pointer_Based, Linear_Time_Using_Stack)
end facilities
initialisation
  CF.P
end initialisation
end realisation auxiliary
end Client
```

Figure 4.5 — An End User
The client shown in Figure 4.5 also illustrates how a "main program" looks in RESOLVE. The code for the main program is the module initialization code of the outermost realization, which in this case is a single statement in which operation P from Figure 4.4 is invoked.

Having decided on the components to use, the client then has to assign them to the physical machines in a hardware architecture so that its potential for parallel execution can be effectively utilized. It is possible that more than one component may be assigned to the same machine and copies of the same component may be assigned to more than one machine. While distributing components makes it possible in principle to extract parallelism, a compromise usually has to be made when taking into account the efficiency of communication among the machines in a physical system. If communication is fast compared to executing a local instruction on a physical processor, then the ideal physical distribution will place every component on a separate machine and if possible, copies of components on several machines. However, if communication is very slow then it may pay to map all components to a single physical machine, avoiding communication totally. This choice depends on the exact nature of the physical system as well as the interaction among the components.

4.2.2 Distribution on Logical Machines

The physical distribution of components makes some operation calls local and some remote. To permit clients complete freedom in distribution, the proposed execution method assumes a general distribution, termed a logical distribution, where every call is remote. Any method for distributed execution centers around the details involved in remote communications between components residing on different machines [Birrel 84]. If a method works satisfactorily for the case when no two components are assumed to be on the
same machine, i.e., every operation call is remote, then the method should also work when the components are distributed differently.

In the proposed method, components are assumed distributed on an unlimited number of logical machines. Every logical machine is assumed to have one processor and it executes one component. More than one logical machine may execute the same component. Figure 4.6 shows a logical distribution of the components used by the client program in Figure 4.5. In this figure, a square with rounded corners denotes a logical machine. A sequence of such squares denotes duplicates. During execution, of course, only a finite number of these machines are used. In the discussion that follows in this chapter, we refer to a logical machine simply as a "machine." When a physical machine is meant, it is made explicit.

4.2.3 The Proposed Execution Method

We are now ready to trace the execution of the example discussed in the last section. In the current implementation of RESOLVE, all the facilities that will ever be used by a program are created even before the first statement of the program is executed. The execution method, however, is independent of this notion.

Creation of Facilities. The logical component distribution guides creation of the facilities used by components. Given the component distribution in Figure 4.6, Figure 4.7 shows where the facilities are created for the example program after it begins execution. A facility is created on one of the machines where its realization is available. (Because every realization has been used to create only one facility in the example, only one of the machines with each realization has been used.)
Figure 4.6 — Distribution of Components on Logical Machines
Figure 4.7 — Distribution of Facilities on Logical Machines
Creation of a facility results in appropriate binding of conceptualization parameters and realization and its parameters. After this binding, the facilities declared within this realization are created. Finally, if the facility needs any module initialization (i.e., its realization has module initialization code like Client) that is done.

In the example, the facility CF is created first. This results in the creation of IF, QF, RQF, and SF in that order. None of these facilities have any initialization code to be executed. Finally, the initialization code found in Client is executed. For an understanding of the execution method, it is not necessary to know how exactly the facility details are stored or used. It suffices to know that given an operation call, it can be successfully located.

Creation of Data Representations. Figure 4.8 shows an execution snapshot after P from the facility CF has been called. First, the operation Initialize is executed from the facility QF for the local variable q1 (and similarly for q2) in P. This operation is invoked by a compiler-generated call that does not appear in the code of Figure 4.4.

It is on the remote machine Queue Machine and does not have any parameters. The representation for an empty queue is created and a pointer to this representation is returned to the caller. Addresses (or pointers) have a system-wide meaning. Each address has two parts: an address of a machine and a local address in the memory on that machine. In the figures, a square icon stands for the representation of a variable's value, and a triangle icon stands for a pointer to a representation.
Figure 4.8 — Remote Call to an Initialize Operation
In our discussion above, we have been referring to pointers to representations. To RESOLVE programmers these pointers are invisible and irrelevant. To them, every variable has a value from the domain of the type of that variable. We are using pointers here only to explain the proposed automated distributed execution method, i.e., an aspect of RESOLVE implementation.

The representations of q1 and q2 are shown to be on Queue_Machine in the figures. This need not be the case, in general. Initialize for type Queue may call operations from other facilities in order to create the representation of a queue. These operations may in turn call other Initialize operations. Therefore, the representation for a queue may be on the Queue_Machine, or on some other machine, or may be distributed on more than one machine. Wherever the representation is created, an address of it is returned to the caller. It turns out that the assumption that the representations of q1 and q2 are on Queue_Machine has no impact on the execution method, as explained later.

Assume that P calls several other operations and builds an interesting value of q1 before calling RQF.Queue_Reverse. The representations of the elements of q1, which are integers, are on the machine that has the facility IF from Integer_Template. (If there are duplicates of IF, then these integers may all be on different machines.) The representation of the value of q1 thus may be distributed on a number of machines. This is typical of most values, and it is this distribution that can permit parallel manipulation of the elements of large data structures.

*Parameter Passing to Procedure Calls.* RQF.Queue_Reverse(q1, q2) is executed next. We use this call to show how the “call by swapping” parameter passing is implemented in a distributed environment. By our implementation conventions, only pointers to the
representations of q1 and q2 are available to the caller operation P. These pointers are now moved (not copied) as parameters to the operation Queue_Reverse. Figure 4.9 shows this call. Note that the pointers are no longer present on C_Machine. Conceptually, the values of q1 and q2 have been swapped with the called operation, and are no longer available with the caller. (The values swapped back to P before Queue_Reverse begins execution are irrelevant, since it cannot proceed to compute with these values anyway.) The parameters will be returned (moved, not copied) to the caller after the completion of execution of the called operation. Exactly what is returned after the call depends on the parameter mode and the effect of the operation.

The operation Queue_Reverse is a secondary operation and it cannot directly access the representations of its parameter queues. So passing the representations of q1 and q2 to Queue_Reverse would be of no use even if they were available with the caller, since to manipulate these representations, operations from the facility QF of Queue_Template have to be invoked. Hence, moving pointers is the only reasonable way to pass parameters to secondary operations. Because pointers are never copied, every representation of a data value has only one pointer pointing to it at any time during the execution of a program. This feature of the execution method significantly influences how parallelism can be extracted automatically from sequential components, as discussed in section 4.4.

In the snapshot shown in Figure 4.9, all the local variables in Queue_Reverse have been initialized. The local variables of an operation are in plain text, whereas parameters are shown in boldface. The variables x, s, and q are initialized here just as q1 and q2 were initialized earlier.
The swap statement "q2 := q" in Queue_Reverse can be executed locally on the Reverse_Machine, because only the pointers of q2 and q have to be exchanged and they are available on that machine. (Conceptually, q2 now has an initial value for the type Queue).
Queue_Reverse then calls the primary operations on stacks, queues, and integers. All these operations are remote. Figure 4.10, for instance, shows the remote operation call SF.PUSH (s, x).

Figure 4.10 — Remote Call to a Primary Operation
The operation Push is a primary operation on stacks, and it directly accesses only the representation of $s$, which is present on the same machine as Push. Push does not need to access the representation of $x$. A similar statement holds for all primary operations. Creating the representation of a variable's value on one of the machines that has all the primary operations on the type of the variable (i.e., the machine that has the facility from which its type comes from) ensures that when a primary operation needs to access the representation, it will be present on the same machine as the operation. Thus, we need to move only references, not representations, as parameters to primary operations.

In summary, it is possible to pass only references to data representations between machines. This is true for primary as well as secondary operations.

4.2.4 Parallel Execution

The trace of execution of the example discussed in the last section is sequential. Some techniques for extracting parallelism from sequential components are discussed in this section.

A Control-Flow Directed Data-Oriented Approach to Execution. The proposed method creates the representations of data values that come into existence so that they are distributed maximally. It is possible for operations manipulating different representations to execute in parallel. This observation leads to a "control-flow directed, data-dependent" approach for detecting parallelism during execution, similar to the ones discussed in [Baldwin 87, Bensley 88, Liskov 88b]. The idea is that when a variable is passed as a parameter in a operation call (i.e., the pointer to the representation of the variable's value is
moved to the site of the operation and therefore is no longer available to the caller), it is defined to be "in use" at the caller machine. Sequential execution can proceed in the caller as long as a variable that is in use need not be passed as a parameter to another operation call. In fact, operation calls can be made even when some of the arguments are in use elsewhere or have not yet been computed, as suggested in [Liskov 88b].

The fact that there exists only one pointer to a representation at any time during execution is an important aspect of the proposed parallel execution method. For example, two operations such as Dequeue(q1, x) and Push(s, y) can always be executed in parallel because there is no possibility of aliasing. This is also a limitation because even when an argument is only read but not changed by two operations, they cannot be executed in parallel. If it is permissible to copy a pointer, a pointer to a read-only representation can be passed simultaneously to two operations allowing them to execute in parallel. This cannot be done in RESOLVE because of the absence of the conventional value parameter passing mechanism. The preserves mode guarantees that the value of a parameter passed in that mode will be the same after the operation finishes execution, but permits it to change during execution.

The argument against copying pointers discussed in section 2.5 does not arise here because of the read-only nature of the representation. For more efficiency in the context of distributed execution, a parameter mode that does not allow the value of a parameter to change at all may be useful. An important objection to having this mode is that it reveals an aspect of implementation of an operation. Preserves parameter mode tells a client of an operation all that it needs to be know about a parameter's abstract value — that it does not change from beginning to end. Whether it changes during the execution of the operation
should not be of concern to a client. It is, however, of some concern for automatically parallelizing the operation.

**Compiler Detectable Parallelism.** If RESOLVE programs are specially compiled for the proposed distributed execution method, then additional potential for parallelism can be detected by compilers. Observe that in Queue_Reverse, in its first while statement, reproduced below, when the previous element is being pushed on stack s, the next element can be dequeued from q1.

```plaintext
while not QF1.Is_Empty(q1) do
    QF1.Dequeue(q1, x)
    SF.Push(s, y)
endwhile
```

This can be detected by a compiler noticing that the parameter mode is consumes for x pushed on the stack s and it is produces for x dequeued from q. Push is at the end of one iteration and Enqueue is at the beginning of the next. This implies that there is no abstract connection between the x pushed and the next x dequeued. The compiler can therefore replace the above loop with the following code:

```plaintext
while not QF1.Is_Empty(q1) do
    QF1.Dequeue(q1, x)
    x := y
    SF.Push(s, y)
endwhile
```

where y is a new variable of type Item. With this modified while-loop, the execution time will be almost halved. The same modification can be done for the other while-loop. In this example, the information contained in RESOLVE parameter modes was useful in detecting parallelism. In general, formal specifications can provide important information needed for such compiler optimizations.
Though the techniques discussed here do allow parallelism to be extracted, it must be emphasized that this potential need be utilized only if remote execution is efficient on the client hardware architecture. That there are several remote operation calls in this example, as noted earlier, is because of the particular implementation of Queue_Reverse. The client always has the option to map all components to a single machine and avoid remote communications totally.

*Extraction of Fine-Grain Parallelism.* In this sub-section, we briefly discuss the intuition behind the claim that the execution method allows parallelism to be extracted at any grain-level. So far, we have considered parallel execution of operations such as Enqueue and Push. In principle there is no reason why operations at the level of, say, record and array accesses cannot be carried out in parallel. The representations of the types Queue and Stack are constructed on top of more rudimentary representations such as records and pointers. The representation of a queue, if it is a record, can be on a machine by itself with a "Record" component. Therefore the operations on type Record called by the implementations of the primary operations of Queue_Template can be executed in parallel following the same ideas discussed earlier. It is possible in principle to continue this way down to the bit level, but arrays and records are going to be primitively implemented types in most practical systems. Architectures for extracting fine-grain parallelism at about this level are discussed in the literature [Athas 88, Dally 87, Welch 90a].

*Creating Duplicates of a Facility to Enhance Parallelism in Execution.* We noted earlier that the integers that are elements of a queue may all be on different machines if duplicates of the facility IF are available. Likewise, the queues q1 and q2 have no reason to be on the same machine. In general, every representation that comes into existence during the
execution of a program may be placed on a different machine. Such distribution improves the potential for parallel execution. The two initialize operations for q1 and q2 in the operation P in the example can be executed in parallel if q1 and q2 are on different machines. If there are copies of QF on two machines, the operation to initialize q1 may be called from one and the operation to initialize q2 from the other. But having copies of the same facility raises some interesting issues. If there are duplicates of a facility, when an operation (other than an initialize) is called, which facility should be used for executing the call? The answer depends on the nature of the facility as described here.

If there are duplicates of a facility that provides primary operations such as QF, then the choice is determined by the parameters of the operation. For example, when QF.Enqueue(q1, y) is called, the operation from the machine on which q1 resides will be called, and this decision is made by referring to the address of the argument q1. Because the first part of the address of q1 denotes the machine on which the representation of q1 resides, the Enqueue operation from the copy of QF on that machine must be called. Why should the address of q1 be used in making this decision and not that of y, the other parameter to the Enqueue operation? This is because Enqueue is a primary operation on the type Queue, and therefore the argument that is of type Queue must be used. This solution works as long as a primary operation has one parameter of the type for which it is primary.

What happens if a primary operation has more than one parameter of the type for which it is primary? For example, consider the operation IF.Add (from Integer_Template). If the representations of the two arguments passed as parameters to Add are on two different machines (because IF has been duplicated) which Add must be called? Whichever Add is called, it must have access to the representations both arguments. There are two ways this can be accomplished. The first solution is not to allow duplicates of facilities such as IF
which have primary operations with more than one parameter of the type provided by the facility. In this case, there will exist exactly one copy of each facility and all integers from a facility will be located on the same logical machine (and hence the same physical machine). The other solution is to allow representations of these variables to be moved. It is conceivable that types such as Integers will have standard representations even though RESOLVE permits them, like any other, to have multiple representations. In addition, these representations are usually small, comparable to the size of a pointer, and hence moving them may not be expensive. Most conceptualizations we have conceived in the Reusable Software Research Group do not have primary operations with more than one parameter of the provided type [Weide 86b], and hence movement of representations for variables of most types can be conveniently avoided.

When an operation has no parameters, a different problem surfaces. An operation without parameters will be provided only by a sub-standard facility. This operation typically affects the shared information of that facility. We preclude duplication of facilities having shared information because the shared data cannot itself be duplicated without adding much complexity. Therefore, if an operation has no parameters, there is exactly one facility that provides it and the operation will be executed from that facility. As noted in Chapter II, few conceptualizations need to be sub-standard. Hence, this issue is not very significant either.

Allowing duplicates of facilities, such as RQF, that provide only secondary operations poses no problems. Because secondary operations never need direct access to any representations, when there are multiple copies available, any one can be picked and executed. The operation that is actually chosen to execute in the physical system may depend on load balancing techniques used by the system, for example.
Duplication of facilities discussed here should not be confused with facilities from identical declarations. For example, if IF1 and IF2 are two identically declared facilities of Integer_Template, the integers from these two facilities need not be co-located and they do not have to be moved, because no one primary operation can have arguments of types IF1.Int and IF2.Int.

4.2.5 Discussion

The last few sections argued how data distribution permits extraction of parallelism at the desired grain level. Such distribution, however, need not result in representation movement. In this approach to distributed execution, representations of parameter values are never passed; only pointers to the representations are passed to all operation calls (local or remote) by the call-by-swapping mechanism. If an operation is primary for a data type, it can directly access the representation only for parameters of that type. Typically, there is one such parameter, and the representation of this parameter is available on the same machine as the operation. If an operation is secondary, then it cannot directly access the representations of any of its parameters, and in this case only addresses to the representations need to be passed.

This method therefore has made it unnecessary to move the data structures that represent parameter values across logical (hence physical) machines. This observation, that representation values usually need not be moved, is important because in the proposed component industry view representations will be hidden within the object code versions of components and hence will not be available. On the other hand, most approaches to
distributed execution encourage representation movement. This difference is the topic of the next section.

4.3 Why Representations of Data Values Should Not Be Moved

Even when source code of components is available, why should representations of data values ever be shipped from one physical machine to another? This section discusses the reasons most often given for moving representations of data values between machines, and the mechanisms for such movement in other distributed systems, and explains why such movement is not only unnecessary but undesirable. For purposes of discussion, we assume that each of the logical machines in the previous figures is mapped to a single-processor computer. Hence, the terms distribution and data movement apply both logically and physically.

4.3.1 Review of Representation Movement Methods in Other Distributed Systems

Most systems propose different parameter passing mechanisms for local and remote operation calls. In particular, most systems do not pass pointers (to variable representations or objects) as parameters to remote operations even though they would pass them to local operations. (In the discussion here, an object includes a data representation plus the code for manipulating that data.) The reason for data or object movement is a belief that passing references might result in many more remote invocations to the parameter variable representations (or objects) by the called operation if they are not at the operation site. Two of the most ambitious distributed system projects, Emerald [Black 87] and Argus [Liskov 83], take this view, though they propose different mechanisms. Though reuse appears to be a concern of both these systems, they do not distinguish between component developers
and clients, but instead simply refer to "programmers." The view that a single programmer is omniscient is what has led these systems to completely different solutions and mechanisms for performance tuning.

It should be noted, however, that these systems discussed were designed with very different goals from ours. Argus, for example, is also concerned with handling of long-lived data (e.g., files, which exist over long periods of time) and short-lived data (e.g., integers, which exist only during the execution of a program) [Black 87, Liskov 83, Liskov 88a]. We have restricted our attention to the distribution of only the short-lived components of a program, by this notion.

**Emerald.** Developed at The University of Washington, Emerald is an object-oriented language designed for programming distributed sub-systems and applications. Goals of Emerald include providing a uniform object model and supporting abstract data types [Black 87]. (An abstract data type as used here is a misnomer since there is no notion of formal specification, abstraction, or information hiding in Emerald.)

In Emerald, a programmer has the option to decide whether an object reference or the object itself should be passed as a parameter to a remote operation call. Black et al. propose a new parameter passing mode, named "call by move," for remote operation calls [Black 87]. A call-by-move parameter object is passed by reference just like any other parameter. However, at call time, the call-by-move object is physically moved to the destination site, allowing it to be accessed efficiently by the called operation. Object mobility is one of the important features of Emerald [Levy 88]. The designers point out that the decision of whether to move a parameter object depends on the size of the object, the number of active invocations of the object methods, and the number of invocations of the object methods
initiated by the called operation [Black 87]. In addition, Emerald has special language constructs for specifying which sub-objects of an object are to be moved, and for moving and locating objects, but it is possible for programmers to ignore the locations of the objects, and let the system locate them.

Argus. The M.I.T. Argus system is a distributed language project that extends CLU to support atomic transactions in a distributed environment. This system is intended primarily for programs that maintain on-line data for long periods of time [Liskov 88a]. The goals include data consistency in the presence of concurrent accesses, fault tolerance, and dynamic reconfiguration of programs.

In Argus, parameter passing is handled differently for local and remote operations [Liskov 83]. Argus supports two kinds of objects, Guardians, which are network-wide objects, and CLU objects, which are local objects of a Guardian. A Guardian is an abstraction for a physical machine. While the CLU objects communicate through shared memory, Guardians communicate through a network. Performance is the reason for these two different kinds of objects in Argus. Programmers, however, must decide when a Guardian or a CLU object should be used at the time of software development. When a Guardian communicates with another Guardian, data are passed by value. Thus, all parameters to remote operations are passed by value, and the Argus system does this automatically. This involves having standards for representations, representation conversions, and moving representations across machines [Herlihy 82]. CLU objects are local objects within a Guardian, and they communicate by passing references to data values.

The RPC System. While value parameter passing may be needed for whatever reasons in some systems, it is not obvious that it does not degrade the performance of the system, if
the values to be moved are represented by large data structures. In the RPC system developed at Rice University [Almes 86], a new parameter passing mode is added to Modula-2: “result var.” The commonly used parameter mode “var” is an “in out” parameter, whereas result var is a pure “out” parameter; no information flows to the called operation from the caller. In the RPC system, all parameters including var and result var parameters are passed by value. However, in the case of result var parameter mode, large data values are not transmitted to the called operation, even though they may be transmitted back after the execution of the called operation. Thus the essential difference between these two modes is that some data values are transmitted one way instead of both ways when possible. Almes reports considerable performance improvements due to the introduction of this new parameter passing mode [Almes 86]. These results suggest that the sizes of messages transmitted in a distributed system play an important role in the overall performance.

Distributed Ada. Distributed Ada is an on-going research project at Honeywell Systems and Research Center [Cornhill 83, Kamrad 87], and it aims at executing Ada programs in distributed systems, without introducing new constructs in Ada. One of the important goals of this system is similar to the one discussed here, to separate the concerns of software development from concerns for distribution. The Distributed Ada project is an attempt to find solutions to the problems involved in distributed computing without adding any special-purpose constructs to Ada to handle distribution. This project proposes that distributed computing be handled in two separate phases. The first phase is writing a program in Ada, totally ignoring distribution, to solve the given problem. This program is portable to any environment, distributed or single-computer. The next phase is to develop a distribution specification for this program for a given target architecture to maximize execution efficiency. This specification can be written in a separate configuration language,
called Ada Program Partitioning Language (APPL). Kamrad et al. argue for the need for this separation of concerns [Kamrad 87]. However, multiple representations for a type and hiding representations are not considered key issues in Ada, and hence, they are not addressed by the Distributed Ada approach or by other approaches to distributing Ada programs, such as in [Volz 89].

The execution method proposed in this chapter may, however, be applicable to well-designed Ada components [Muralidharan 90], because Ada does not place any restrictions on how parameters must be passed in local or distributed systems.

4.3.2 Why Representations of Data Values Should Not Be Moved

One of the objectives of most distributed systems is to make the semantics of local and remote operation calls uniform [Birrel 84, Black 87]. Such uniformity should also extend to parameter passing mechanisms, and we claim it is possible to achieve this uniformity without sacrificing performance. Providing different language mechanisms for passing parameters to local and remote operations makes component developers focus on the characteristics and distributed nature of the target system, and this in turn considerably complicates the development of the component itself. While it can be argued that component developers must have the choice to use these mechanisms, we know of no convincing evidence that permitting movement of representation data structures will consistently result in better performance, even if developers with this flexibility are omniscient. This section discusses many reasons why data representations of the values of parameters to the site of remote operation calls. Even when it is possible to do so, it is not desirable.
**Parameter Movement Reduces the Potential for Parallelism.** In object-oriented languages, when a parameter object is moved, its representation data structure is moved. The primary operations on that object must also be moved, if not available on the destination machine, since these are the only operations that can directly access the representation. If all parameter objects are moved to the site of the called operation, then the parameter objects and the local objects of the called operation cannot be manipulated in parallel. The potential for parallelism in a distributed system can be exploited only by distributing objects, and moving parameter objects to a common site will limit such parallelism. To suggest but one example of how distribution helps in parallel computing, assume that the value of each element of an array (which is a real numbers) is to be doubled. This can be done in parallel only if each element of the array is on a different machine.

**Decisions on When or What to Move Are Not Easy.** Assuming that the parallelism gained from distribution is not appreciable in some cases (e.g., when communication overheads override the benefits of parallelism), it is non-trivial to decide when a parameter's representation should be moved. Even when source code is accessible, this decision will not be easy in large application programs composed of modules built independently by different programmers.

Assuming that a decision to move a parameter representation has been made, the next question to be resolved is what parts of it should be moved. This is a difficult decision if the representation is distributed on many machines. It is conceivable that parts of a large structure may be distributed. (In our approach, this is usually the case.) If the entire representation were moved to the site of the called operation (e.g., the representation of q1 in the remote operation call Queue_Revers), then this would result in communication traffic to collect the distributed components. Because the apparent advantage of moving the
representation is to avoid many remote operation calls by the called operation to access this representation, this tends to defeat the original intention. On the other hand, moving only a part of the structure may not be sufficient to avoid all remote invocations by the called operation since it may have to access the parts left behind. Thus the decision on what parts of a representation should be moved is dependent on the size and structure of the representation, on the parts of the object that are accessed by the operation, and on the particular context of execution (e.g., the components to be moved for one parameter representation may depend on the value of another parameter).

In general, it is not possible for even an omniscient person to decide at software development time when a representation should be moved, or what parts of it should be moved, and therefore it is unclear how mechanisms for movement of these representations can be effectively used in large software projects, even if a language provides them.

**Parameter Movement is Non-Trivial to Implement.** In our approach, data movement might be easier than for most other languages, because every value representation has just one pointer to it. Hence, moving a representation would not result in an arbitrary number of pointer updates as it can in other systems. Even when there are not many updates to be made, though, parameter movement is not easy to accomplish. Cycles in representations can make automatic movement impossible, for instance; even when cycles are absent, the task is non-trivial [Herlihy 82]. While mechanisms to forward messages to moved objects have been suggested in the literature [Black 87], the overhead involved in setting up a new process to handle a moved object may be considerable. These methods therefore do not generalize nicely to various kinds of distributed systems suited for different grain-sizes of parallelism.
Moving Large Representations Can Affect the Overall System Performance. Most distributed systems are based on the assumption that message communication overheads are very large, and that the actual transmission time for a message, which is dependent on the message size, is almost negligible compared to the per message overhead. Systems whose hardware base is workstations on an Ethernet, for example, may suggest such an assumption. They therefore attempt to reduce the number of messages even if some of the resulting messages are very large. However, there is reason to believe (from examining simple results for M/G/1 queues [Kleinrock 75]) that large variance in message sizes can contribute significantly to performance degradation due to queuing delays. Programs written in languages supporting reusable software (generic data types or objects) are likely to manipulate very large data structures (since it is easy to define large structures in these languages), so advantages gained by reducing the number of messages by moving representations are considerably offset if the representations to be moved are very large. Packetizing large messages into smaller ones does not solve this problem because of the effects of bulk arrivals [Kleinrock 75].

Furthermore, large message overhead is not inherent to all distributed systems. [Dally 87], discusses a message-driven, non-shared memory architecture for object-oriented languages. The architecture is geared for handling short messages efficiently, and message passing times are comparable to the execution times of other instructions. One of the important goals of this architecture is to facilitate fine-grain parallelism by reducing message overheads. [Welch 90a] discusses an architecture specially tailored for efficiently executing remote operation calls. Design and development of architectures such as these support our claim that performance in distributed systems may not be based merely on message count. [Stamos 90] also notes problems with large data movement.
Object Movement is Even More Complex in Heterogeneous Environments. A heterogeneous system where different machines can execute different components more efficiently can guide component distribution (on the physical machines). For example, we can readily imagine a machine specially suited for executing a component providing real numbers. Machines particularly designed for executing specific abstract components have been discussed in [Bastani 87].

Object movement has additional problems in such heterogeneous distributed systems. Moving an object may involve moving the executable code for the methods associated with the object. However, machine-dependent executable code cannot be moved in a heterogeneous environment. The solution is either that a copy of the executable code for the operations on each object should be available on each machine, or that the executable code should be machine-independent. While the excessive replication may be undesirable in the former case, machine-independent executable code will normally make execution rather slow.

Moving the representation of an object may also involve appreciable overhead because of the translation processes needed at the source and destination machines [Herlihy 82]. Bershad et al., the designers of the remote operation call facility for the Heterogeneous Computer Systems Project [Bershad 87], suggest that automatically moving representations may not be possible for complex data types or data types involving pointers, and that programmers should specify how to handle the movement of these representations. This is undesirable in a reusable software approach to system design, as noted earlier.
4.4. Summary

We discussed a method for distributed execution that separates the concerns of development of reusable software components and distribution of these components on specific hardware architecture for efficient execution. This is important in the component industry view where performance-tunable components are marketed in object-code form and clients need to distribute them on arbitrary architectures to utilize the potential for parallelism in these architectures. A distinguishing characteristic of the method is that it avoids representation movement and thus does not need access to source code of components. In addition it does not suffer from several problems associated with movement of representations. The method applies uniformly to machines that support large-grain parallelism as well as those supporting fine-grain parallelism.
The potential of software reuse has been widely acknowledged as a solution to escalating software development costs. However, there are several technical impediments to constructing software largely from reusable software components. Though many programming languages have been designed and developed to facilitate the construction of reusable software components, none of them provide the support needed for realizing a software component industry similar to those of other mature engineering disciplines. There is in principle no reason to believe that a reusable component industry similar in nature to traditional engineering disciplines cannot emerge.

When it does, a component will be marketed in a sealed form without revealing source code, but with a formal, implementation-independent specification of its functionality. Several components with different performance behaviors will be available from different manufacturers in the industry for providing the same abstract component, and customers will be able to choose among them based on their performance requirements. Some of the marketed components will have a specific performance and will be tailored to particular hardware architectures. Others will be more general. Clients will be able to tune these components to suit their performance needs.
It is the development and control of performance-tunable reusable software components that has motivated this work. The dissertation discussed a layered model for software construction based on abstractions and their concrete implementations. This model shows how the client of an abstract component can change its performance without affecting its functionality. Three important aspects of performance of a component that must be controllable by clients were identified based on this model, and programming language mechanisms and other methods for providing clients such control were presented. These techniques permit a fine control of performance at little cost to clients. Performance control methods that provide source code for components to clients and then allow them to modify the source code have tremendous software costs associated with them, and hence are undesirable. These costs occur not only during development time, but more importantly during maintenance.

The view of a component industry where only object code versions of components are marketed raises the challenging question of how performance can be tuned by clients without having access to the internal details of a component, and also suggests a solution as demonstrated in this dissertation. The solutions presented in this dissertation in the context of software components are far more powerful in nature than their counterparts in engineering disciplines where parametrization techniques do not generalize as well.

5.1 Contributions

The dissertation for the first time demonstrates that all important aspects of performance of a reusable software component can be controlled without altering its functionality. It shows that this is possible to do under an apparently restrictive (but highly desirable)
framework of a software component industry. In particular, its contributions include the following.

*Characterization of a Software Component Industry.* Though the basic idea of a software component industry itself is not new, the characterization proposed in this dissertation based on reusable software components is new. A clear distinction between a developer and a client of a component is made, and a case for what the developer and client need and need not know is made. Based on this distinction, the dissertation provides justification for why a reusable software component must have a formal, implementation-independent, human-understandable yet machine-processable specification of its functionality and must be marketed in an object-code form with performance information.

*A Model for Controlling Performance of a Component Independent of Its Functionality.* A layered model of a software component constructed using abstract components (that can have several concrete implementations with varying performance characteristics) is presented. This model is implicit in some other reuse attempts, but has neither been spelled out nor has its impact been analyzed carefully. It shows how the performance of a software component can be changed without affecting its functionality, a feature that has important implications for the cost-effectiveness of performance tunability. The ability of clients to alter performance independently, without needing access (and hence modification) to source code of a component, eliminates costs of re-compilation and re-verification.

Based on this model, three major factors that affect the performance of a client of an abstract component are identified: the concrete component chosen for realizing the abstraction, the implementations used for constituents of that component, and the particular way the component is distributed and executed in a hardware architecture. The dissertation
makes the following other contributions, presented within the framework of RESOLVE, in demonstrating that it is possible to control these factors effectively within the framework of the software component industry view.

**Parametrization and Performance Specification of Implementations.** While most programming languages for reuse seem to have recognized the need for separating the concerns of specification and implementation of a component, few have realized the need for language support for having multiple implementations of a specification. The dissertation demonstrates the need for multiple implementations with varying performance behaviors (for a specification) and illustrates that operations, (mathematical) definitions, facilities, and realizations are needed as parameters to realizations. It shows that in principle the performance of each operation from a realization can be precisely specified in terms of the parameters to the operation and the realization, and that this can be done without revealing source code.

A specific problem of incompleteness is shown to exist in passing facilities as parameters to realizations, and it is suggested that a distinction be made between standard and substandard conceptualizations in order to overcome it.

**Parametrization of Performance of a Component.** In the component industry, the dissertation argues, some components will be tunable, i.e., clients can plug in appropriate constituents to suit their performance needs. Language mechanisms needed for construction and use of such components are proposed as additions to RESOLVE. The mechanisms are general. They make it possible to tune the performance of a component by tuning its constituents at arbitrarily deep levels in a uniform manner. This uniformity permits a realization to change its parameters to a great extent without affecting its clients.
A programming construct comprising parameters of a realization, termed a realization form, is introduced. A client of a realization can see its form and a statement explaining how its constituents affect its performance, which are sold along with its specification and object-code.

The need for the ability to build special-purpose components from their generalized versions is explained, and mechanisms to permit this possibility are suggested.

A Distributed Execution Method. The mechanisms for building performance-tunable components also provide (at least some of) the information needed for a client to be able to distribute a component and its constituents for effective utilization of the potential in his or her hardware architecture. It is not clear that any additional information is needed to do such distribution.

Given that a client has chosen some architecture and some distribution, a method for the execution of reusable software components that permits parallelism to be extracted at any granularity and that does not need access to representational details of components (contained in the source code) is proposed. A notation for explaining distributed execution is presented. Suggestions for extracting parallelism automatically are given, including a control-flow directed, data-oriented approach for execution and some compiler optimizations based on formal specification. The execution method does not require movement of representations of data values to remote operation calls, and shows that it is possible to avoid the need for canonical representations and representation movement (some things that are not readily possible in the component industry view) without
sacrificing the potential for parallel execution. The dissertation argues why it is both unnecessary and undesirable to move representations.

In summary, the dissertation makes a case for what a reusable software component and its associated industry must look like, and provides a suitable model for developing software based on reusable components. It adds significantly to the understanding of mechanisms required of programming languages to support the construction and use of performance-tunable reusable software components. It contributes to the field of distributed execution by showing how reusable components can be executed efficiently without access to their source code either for static distribution or at execution time.

5.2 Directions for Future Research

The most important extensions to this research are related to the decisions to be made by clients, given the mechanisms and methods proposed in this dissertation, to effectively use them for controlling performance. In this section, we discuss these and other topics for further research.

A Formal Treatment of Performance. The performance statements presented in this dissertation are informal. RESOLVE presently includes a syntactic slot for performance specification, but a formal notation must be developed for it. Performance should include both time and space behaviors, and if at all possible they should be kept distinct from specification of functionality. A basic problem with specifying space behavior has to do with the fact that every operation provided by every module should in some sense make sure there is "sufficient space" to execute the operation. The same problem appears in specifying time behavior as well, if there are real-time bounds.
Performance Correctness. Once methods for formal performance specification are in place, it should be possible to prove that a given software component correctly meets its performance specification. Such proofs will need machinery similar to the ones used for proving functional correctness of software components; for example, software developers may have to provide certain performance assertions (similar to loop invariants) for loops. Performance and functional correctness proofs of software components can be kept separate to a great extent. (There are certain basic abstractions where the distinction between the two may not be apparent, as in the case of the specification of an abstraction of "bounded space." ) If this is done, performance tuning will result only in re-verifying performance.

Tools for Decision Making. Given formal methods for performance specification, it should be possible to construct automatic tools for producing components meeting a client’s specific performance needs from the general versions of these components. These tools will take performance behaviors of components and clients’ performance requirements as input, and output appropriate tuning to be done (i.e., the constituents to be plugged into a general version of a component). For the case of a distributed hardware architecture, such tools will take hardware performance specifications as additional input, and produce a distribution of components as an additional output. The claim made in this dissertation that the performance statements supplied with a component provide a good measure of interaction of the component with its constituents, and no additional information may be needed, has to be critically analyzed.

"Parallel" Components. RESOLVE does not include parallel programming constructs. These constructs (if added to RESOLVE) may, for example, specify which constituents of
a component must be co-located. Even assuming that some mechanisms to specify parallelism are added to RESOLVE, the functional specifications of a RESOLVE component built using these constructs will still remain similar to the ones discussed here, because they are an aspect of a particular implementation. Their usage will only affect the performance of components.

The impact of parallel constructs on the proposed distributed execution method must be studied. As we note in the dissertation, even if it is possible to construct parallel components, the need for executing sequential components in parallel will still exist. Therefore, both architectural support for extraction of parallelism from sequential components and compiler-detectable optimizations based on formal specifications should be explored further.

5.3 Conclusions

The dissertation shows that it is possible to effectively control important aspects of performance of reusable software components without changing their functionality. It demonstrates this thesis by providing programming language mechanisms and methods within the context of a software component industry that will evolve when impediments to constructing software based largely on reusable components are overcome.
APPENDIX

classification Integer_Template

auxiliary

math facilities

Number Theory is Number Theory Template
renaming
Number Theory.Integer as Integer
Number Theory.Less as Math_Less
Number Theory.Leq as Math_Leq
Number Theory.Add as Math_Add
Number Theory.Sub as Math_Sub
Number Theory.Mult as Math_Mult
Number Theory.Abs as Math_Abs

end renaming
end math facilities

math constants

max_int : Integer
min_int : Integer

end math constants

constraints "Math_Leq(min_int, 0) and
Math_Less(0, max_int)"

end auxiliary

interface

type Int is modeled by Integer
exemplar i
constraint "Math_Leq(min_int, i) and
Math_Leq(i, max_int)"
initially "i = 0"
end Int

operation Get_Min_Int returns min : Int
ensures "min = min_int"
end Get_Min_Int

operation Get_Max_Int returns max : Int
ensures "max = max_int"
end Get_Max_Int

operation Increment parameters
alters \textit{i} : \textit{Int}
end parameters
requires "Math\_Less(\textit{i}, \text{max\_int})"
ensures "\textit{i} = Math\_Add(\#\textit{i}, 1)"
end Increment

operation Decrement
parameters
alters \textit{i} : \textit{Int}
end parameters
requires "Math\_Less(\text{min\_int}, \textit{i})"
ensures "Math\_Sub(\#\textit{i}, 1)"
end Decrement

operation Add
parameters
alters \textit{i} : \textit{Int}
preserves \textit{j} : \textit{Int}
end parameters
requires "Math\_Leq(Math\_Add(\textit{i}, \textit{j}), \text{max\_int})"
ensures "\textit{i} = Math\_Add(\#\textit{i}, \textit{j})"
end Add

operation Subtract
parameters
alters \textit{i} : \textit{Int}
preserves \textit{j} : \textit{Int}
end parameters
requires "Math\_Leq(Math\_Sub(\textit{i}, \textit{j})), \text{min\_int})"
ensures "\textit{i} = Math\_Sub(\#\textit{i}, \textit{j})"
end Subtract

operation Multiply
parameters
alters \textit{i} : \textit{Int}
preserves \textit{j} : \textit{Int}
end parameters
requires "Math\_Leq(Math\_Mult(\textit{i}, \textit{j})), \text{min\_int})" and "Math\_Leq(Math\_Mult(\textit{i}, \textit{j}), \text{max\_int})"
ensures "\textit{i} = Math\_Mult(\#\textit{i}, \textit{j})"
end Multiply

operation Divide
parameters
alters \textit{i} : \textit{Int}
preserves \textit{j} : \textit{Int}
end parameters
requires "Math\_Leq(\textit{j}, 0) implies (Math\_Less(Math\_Mult(\textit{j}, Math\_Add(\text{max\_int}, 1)), \textit{i}) and (Math\_Less(\textit{i}, Math\_Mult(\textit{j}, Math\_Sub(\text{min\_int}, 1))))"
ensures "exists quo: Integer, Math\_Leq(Math\_Abs(Math\_Mult(\textit{j}, quo)), Math\_Abs(\#\textit{j})) and Math\_Leq(Math\_Abs(Math\_Sub(\textit{i}, Math\_Mult(\textit{j}, quo))), Math\_Abs(\textit{j}))"
end Divide
operation Less_Than_Or_Equal returns control
parameters
  preserves i : Int
  preserves j : Int
end parameters
ensures Less_Than_Or_Equal iff "Math_Leq(i, j)"
end Less_Than_Or_Equal
end interface
description
This template provides a type Int, whose mathematical model is just an integer. However, Int is a computational integer in the sense that it has a lower and an upper bound given respectively by min_int and max_int, which are constants defined by the realization. Initially, a variable of type Int has the value zero.

- "Get_Min_Int" returns min_int.
- "Get_Max_Int" returns max_int.
- "Increment (i)" adds one to i.
- "Decrement (i)" subtracts one from i.
- "Add (i, j)" adds j to i.
- "Subtract (i, j)" subtracts j from i.
- "Multiply (i, j)" multiplies i by j.
- "Divide (i, j)" returns in i the integer part of the result of dividing the previous i by j.
- "Less_Than_Or_Equal (i, j)" returns yes iff i ≤ j
end description
end Integer_Template

Figure 6.1 — A Conceptualization for Integer_Template
auxiliary
  math facilities
  Pair_Theory is
  Tuple_2_Theory_Template(math[Item1], math[Item2])
  renaming
    Pair_Theory.Tuple as Pair
    Pair_Theory.Projection_1 as Projection_1
    Pair_Theory.Projection_2 as Projection_2
  end renaming
end math facilities
end auxiliary

interface
type Record is modeled by Pair
  exemplar r
  initially "Item1.Init(Projection_1(r)) and
    Item2.Init(Projection_2(r))"
end Record

operation Swap_Entry_1
  parameters
    alters r: Record
    alters x: Item1
  end parameters
  ensures "x = Projection_1(#r) and Projection_1(r) = #x
    and Projection_2(r) = Projection_2(#r)"
end Swap_Entry_1

operation Swap_Entry_2
  parameters
    alters r: Record
    alters x: Item2
  end parameters
  ensures "x = Projection_2(#r) and Projection_2(r) = #x
    and Projection_1(r) = Projection_1(#r)"
end Swap_Entry_2
derm interface
description
  This template provides a type Record (of 2 Items). The
  conceptual model is a pair of Item1 and Item2.

  "Swap_Entry_1 (r, x)" exchanges the previous value of the
  first element of r with the previous value of x.

  "Swap_Entry_2 (r, x)" exchanges the previous value of
  the second element of r with the previous value of x.
end description
end Record_2_Template

Figure 6.2 — A Conceptualization for Record_2_Template
conceptualization Array_Template
parameters
type Item

facility IF is Integer_Template
renaming
IF.Int as Int
IF.max_int as max_int
end renaming
end parameters
end auxiliary
renaming
IF.Integer as Integer
IF.Math_Leq as Math_Leq
IF.Math_Less as Math_Less
end renaming
facilities
Function_Theory is Function_Theory_Template (Integer,
Item)
renaming
Function_Theory.Function as Integer_To_Item
end renaming
Pair_Theory is Tuple_2_Theory_Template (Integer,
Integer_To_Item)
renaming
Pair_Theory.Tuple as Array_Model
Pair_Theory.Projection_1 as size
Pair_Theory.Projection_2 as map
end renaming
end facilities
end auxiliary
interface
type Array is modeled by Array_Model
exemplar a
initially "size (a) = 0 and forall i : Integer,
Item.Init ((map (a)) (i))"
lemma "forall i : Integer, (Math_Less(i, 0) or
Math_Leq(size (a), i) implies Item.Init ((map
(a)) (i)))"
end Array

operation Set_Size
parameters
alters a: Array
consumes n: Int
end parameters
requires "Math_Leq(0, n)"
ensures "size (a) = #n and
forall i : Integer, Item.Init ((map (a)) (i))"
end Set_Size

operation Get_Size returns this_size: Int
parameters
preserves a: Array
end parameters
ensures "this_size = size (a)"
end Get_Size

operation Swap_Entry
parameters
alters a: Array
preserves i: Int
alters x: Item
end parameters
requires "Math_Leq(0, i) and Math_Leq(i, size(a))"
ensures "size (a) = size (#a) and Delta (map (a), {i})
and (map (a)) (i) = #x and x = (map (#a)) (i)"
end Swap_Entry

end interface

description
This template encapsulates the type Array, a function from
the first size non-negative integers to Item. The
conceptual model is a pair: the current size, and a function
from the integers to Item. In the abstract, the function is
total, but the only operation that can evaluate or change it
requires that the index into an Array (the point at which
the function is to be evaluated and changed) must be non­
negative but less than the Array's current size. Therefore,
the function value is meaningful only on this interval of
its domain.

Initially, an Array's size is zero and its function maps
every integer to an initial value of type Item.

- "Set_Size (a, n)" sets the size of Array a to n. Notice
  that it also resets a's function so it maps every integer
to an initial value of type Item, and that it consumes n.

- "Get_Size (a)" returns the current size of the Array a.

- "Swap_Entry (a, i, x)" exchanges the previous value of
  a(i) with the previous value of x.

If "fetch" and "store" operations are desired (for most
purposes they are not actually needed), a copy operation on
type Item must be available. These two operations can then be built on top of "Swap_Entry" as secondary operations.

end description

deck Array_Template

Figure 6.3 — A Conceptualization for Array_Template

case conceptualization One_Way_List_Template
parameters
type Item
d type parameters
end auxiliary

math facilities
String_Theory is String_Theory_Template (math[Item])
renaming
String_Theory.String as String
String_Theory.Lambda as Lambda
String_Theory.Pre as Pre
String_Theory.Cat as Cat
end renaming

tuple 2_Theory is
Tuple_2_Theory_Template(String, String)
renaming
Tuple_2_Theory.Tuple as Pair_Of_Strings
Tuple_2_Theory.Projection_1 as Left
Tuple_2_Theory.Projection_2 as Right
end renaming
end math facilities
end auxiliary

deck interface
type List is modeled by Pair_Of_Strings
exemplar 1
initially "Left(1) = Lambda and Right(1) = Lambda"
end List

operation Reset
parameters
alters 1: List
d type parameters
ensures "Left(1) = Lambda and
Right(1) = Cat(Left(1), Right(1))"
end Reset

operation Advance
parameters
alters 1: List
d type parameters
requires "Right(1) /= Lambda"
ensures "Cat(Left(1), Right(1)) =
   Cat(Left(#1), Right(#1)) and
   exists x: math[Item],
   Right(#1) = Pre(Right(1), x)"
end Advance

operation Add_Right
parameters
   alters l: List
   consumes x: Item
end parameters
ensures "Left(l) = Left(#l) and
   Right(1) = Pre(#x, Right(1))"
end Add_Right

operation Remove_Right
parameters
   alters l: List
   produces x: Item
end parameters
requires "Right(1) /= Lambda"
ensures "Left(l) = Left(#l) and
   Right(#l) = Pre(x, Right(1))"
end Remove_Right

operation Swap_Rights
parameters
   alters l1: List
   alters l2: List
end parameters
ensures "Left(l1) = Left(#l1) and
   Right(l1) = Right(#l2)
   Left(l2) = Left(#l2) and
   Right(l2) = Right(#l1)"
end Swap_Rights

operation At_Right_End returns control
parameters
   preserves l: List
end parameters
ensures At_Right_End iff "Right(1) = Lambda"
end At_Right_End

end interface

description
This template encapsulates the type List. The conceptual model is a pair of strings: one string holding the items to
the left of the conceptual cursor and other the items to the right. This is a one way list because the cursor can be
moved only to the right of the list.

Initially, a list is empty.

• "Reset" places the cursor of the list l at the beginning.
• "Advance (1)" moves the cursor to the right, past the next element in the list 1.

• "Add_Right (1, x)" inserts x in the list 1 immediately to the right of its cursor.

• "Remove_Right (1, x)" removes the item in the list 1 immediately to the right of its cursor, and returns it in x.

• "Swap_Rights (11, 12)" exchanges the string of items to the right of the cursor in the two lists 11 and 12.

• "At_Right_End (1)" returns true if the cursor of the list 1 is at the end of the list.

end description

end One_Way_List_Template

Figure 6.4 — A Conceptualization for One_Way_List_Template
BIBLIOGRAPHY


[Baldwin 87] Baldwin, D., Why We Can't Program Multiprocessors the Way We're Trying to Do It Now, Department of Computer Science, The University of Rochester, Rochester, NY, August 1987, TR 224.


[Harms 89a] Harms, D.E., and Weide, B.W., Efficient Initialization and Finalization of Data Structures: Why and How, Department of Computer and
Information Science, The Ohio State University, Columbus, OH, March 1989, OSU-CISRC-3/89-TR11.


[Weide 86a] Weide, B.W., Design and Specification of Abstract Data Types Using OWL, Department of Computer and Information Science, The Ohio
State University, Columbus, OH, January 1986, OSU-CISRC-TR-86-1.


