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.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9105123

The influence of software reuse on programming language design

Harms, Douglas Eugene, Ph.D.

The Ohio State University, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

The Influence of Software Reuse on Programming Language Design

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the Graduate

School of the Ohio State University

By

Douglas Eugene Harms, B.S., M.S.

* * * * *

The Ohio State University

1990

Dissertation Committee:

Dr. Bruce W. Weide

Dr. Neelam Soundararajan

Dr. Timothy J. Long

Approved by

Bruce W. Weide

Adviser

Department of
Computer and Information Science

To my parents, Clarence and Mary Ann Harms,

my wife, Mary Beth,

and daughters, Rebecca Elizabeth, Gretchen Marie,

and Alisha Anne

ACKNOWLEDGEMENTS

I express sincere appreciation to my adviser, Bruce Weide, for his patience and guidance during this research, and for his confidence. I also thank the other members of my committee, Timothy Long and Neelam Soundararajan, for finding time in their busy schedules to make constructive suggestions for both the research and dissertation organization.

I am grateful to the members of the Reusable Software Research Group at Ohio State, especially Bill Ogden, Stu Zweben, Mike Stovsky, Murali, Joan Krone, and Joe Hollingsworth. This research has been positively influenced and directed by the ideas and suggestions expressed during our many discussions.

I am indebted to the Muskingum College community for supporting me in both financial and moral senses during the years it has taken me to achieve this goal. I especially thank Dan Van Tassel, Sam Speck, Jim Smith, Ralph Hollingsworth, Ray Rataiczak, and Art DeJong.

I am also thankful for the encouragement and support provided by the members of the United Methodist Church in New Concord and St. Andrew Presbyterian Church in Columbus.

Finally, and most importantly, I am grateful to my wife, Mary Beth, and children, Rebecca, Gretchen, and Alisha, who made this adventure possible with their understanding and constant encouragement.

VITA

June 10, 1957	Born — Hillsboro, Kansas
1979	Bachelor of Science Muskingum College New Concord, Ohio
1979-1981	System Programmer NCR Corporation Cambridge, Ohio
1981-1983	Lecturer Muskingum College New Concord, Ohio
1983	Master of Science The Ohio State University Columbus, Ohio
1983-present	Assistant Professor Muskingum College New Concord, Ohio

PUBLICATIONS

Types, Copying, and Swapping: Their Influences on the Design of Reusable Software Components, with B.W. Weide, Department of Computer and Information Science, The Ohio State University, Columbus, OH, March 1989, OSU-CISRC-3/89-TR13.

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

"Deadlock-Avoidance Mechanisms in Distributed Systems", with A. Datta, S. Ghosh, and A. Elmagarmid, *Computer Systems Science and Engineering*, Vol. 3, No. 2 (April 1988), pp. 67-82.

Swapping — A Desirable Alternative to Copying, with B.W. Weide, Department of Computer and Information Science, The Ohio State University, Columbus, OH, January 1988, OSU-CISRC-1/88-TR2.

"Deadlock Avoidance in Real-Time Resource Sharing Distributed Systems: An Approach Using Petri Nets", with A. Datta and S. Ghosh, *Proceedings of the Real-Time Systems Symposium*, December 1984, pp. 49-61.

FIELDS OF STUDY

Major Field: Computer and Information Science — Software Engineering
Minor Fields: Programming Language Design, Compiler Construction Techniques, Computer Architecture, Formal Program Specification and Verification, and Theoretical Computer Science

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

Introduction

Computers and the software that control them have a profound impact on our everyday lives. They control our microwave ovens when we cook, allow us access to our checking accounts at automatic tellers, assist doctors in diagnosing our maladies, and play an integral role in our nation's defense, to name a few. In fact, our current standard of living and lifestyle (and for some, literally their lives) would not be possible without computers and computer software.

The reliability of these computer systems is by no means perfect. Stories of computer failure abound in the popular press and trade publications, and "computer error" is a contemporary phrase everyone knows. For example, the first scheduled launch of the space shuttle *Columbia* was postponed because of a synchronization problem among the five on-board computers, caused by a software bug in which a delay factor was reset from 50 to 80 milliseconds [Joyce 85]. After five years and twenty space flights, the *Columbia* astronauts were still being supplied with a book, *Program Notes and Waivers*, containing a list of known software problems, ranging from interleaved messages on the display, to a bug where the contents of a ground communication buffer was lost if an astronaut happened to be typing while a program was being uploaded [Joyce 85]. In 1986 a software bug in a linear accelerator exposed a cancer patient to a massive radiation dose of 16,500 rads in less than a second, orders of magnitude over the normal 2-8 rads per second, resulting in the patient's death [Fitzgerald 87]. More recently, a bug in switching software resulted in a nine-hour failure of the AT&T long-distance network in January 1990 [Fitzgerald 90]. The list goes on and on.

A direct consequence of these sorts of software quality problems is the fact that software engineers are beginning to be held legally liable for their software products. Computer litigation is among the fastest growing areas of law, and computer "lemon laws" are a real possibility in the near future [Carver 88]. This provides the software engineering community even more incentive for developing the tools and techniques for building error-free software.

Why is it so difficult to design software that works? It seems that many software engineers regard design of error-free software as software engineering's holy grail. Articles such as [Poston 87] treat zero-defect software as an unrealistic goal, while [Currit 86] and [Lipow 82] assume that software *will* fail, and define estimates for mean time to failure (MTTF) of software and number of faults per line of code, respectively. It is interesting to note that other engineering disciplines don't seem to need these measures. For example, civil engineers don't calculate the mean time to failure of the *design* for a bridge (although they may calculate the MTTF of the actual bridge structure due to deterioration of materials or negligent maintenance).

Although the software engineering community realizes that error-free software is very difficult to design, not all software engineers are as pessimistic as those just mentioned. For example, Hamilton [Hamilton 86, Hamilton 76] believes that developing error-free software is possible, though very difficult. An *a priori* assumption of the work presented here is that zero-defect software is a viable goal in any software project.

Another problem facing software engineers is the productivity of the people building software. In recent years the productivity of computer hardware engineers has increased dramatically, while software productivity seems to be holding its own at best, with code being produced at the same old rate of one to two delivered lines of code per person-hour [Boehm 87]. Even relatively small increases in software productivity would be economically significant. For example, it is estimated that a mere 20% improvement in software productivity would be worth $45 billion in 1995 for the US and $90 billion worldwide [Boehm 87].

Designing reusable software components is an approach to software engineering that attacks both of these problems (i.e., software reliability and productivity). Reusability is a mainstay of other engineering disciplines [Stovsky 90], and its applicability to

software engineering is promoted by many software engineers (e.g., [Meyer 88], [Brooks 87], [Biggerstaff 87], and [Hamilton 86]). Surprisingly, the state-of-the-practice suggests that designing reusable software parts is presently not encouraged [Meyer 88]. This is not to say that software is not reused, but rather that software components in general are not presently *designed* specifically for reuse within larger systems.

The Reusable Software Research Group at Ohio State has been investigating issues relevant to the design of reusable software components. The issues addressed by our group include programming language design, program development methodology, programming environment design, program specification and verification, program implementation, and program testing. These issues are not being addressed in isolation, but in the larger context of design of reusable software parts.

The work presented here is the identification of the salient features of a programming language and environment that encourage the development of reusable software components. The programming language and system developed from this research is called RESOLVE, for REusable SOftware Language with Verifiability and Efficiency.

Programming languages have a profound impact on software reuse, since they strongly influence the way a programmer designs and reasons about a program. In this context the following three points comprise the thesis of this work:

- One of the biggest technical impediments to the widespread design and (re)use of software is that no modern programming language has an adequate set of constructs to support the design of reusable software parts. In fact, existing languages are based on some constructs that actually thwart the design of reusable software.

- It is possible to design a practical programming language based entirely upon constructs that support and even encourage the design of reusable software parts.

- When a programming language and editing environment are designed at the same time, each can influence the other in positive ways.

The approach taken in supporting this thesis is to define the characteristics of "good" reusable parts, and to identify the programming language features necessary to support the design and development of such parts. Several important contemporary programming languages are then evaluated with respect to these language features, demonstrating that none has all of the features necessary to support the development of reusable parts. Next, a programming language is defined that does have an adequate collection of features; this language is called RESOLVE, and is the primary contribution of this research. Finally, a prototype RESOLVE editing environment is discussed that points out some ways the environment influenced the design of RESOLVE, and also compares this environment to several popular programming environments.

The dissertation is organized as follows. Chapter 2 contains background definitions and discussion of the issues surrounding reusable software components and the influence of programming languages on their design. Also included in this chapter is a presentation of features of several programming languages and a demonstration that none has all of the constructs necessary to encourage and facilitate the development of reusable software parts. Chapter 3 contains the definition of RESOLVE and argues why it is appropriate for the design and implementation of reusable software components. Chapter 4 contains a description of the prototype RESOLVE program development environment, and Chapter 5 discusses possible future work and conclusions drawn from the work so far.

CHAPTER II

Definitions and Framework

The context of the work presented here is established in this chapter by defining several terms used throughout this dissertation and discussing some relevant issues. The chapter begins with definitions for several common software engineering buzzwords in Section 2.1. The remaining sections address issues relevant to the design of reusable software, especially with respect to language design. Section 2.2 contains a discussion of the issues inherent to the design and implementation of reusable software components. Section 2.3 presents some characteristics of well-designed parts, and discusses their implications on programming language design. Section 2.4 contains a discussion of some issues relevant to the design of reusable components. Section 2.5 contains a discussion of several other issues relating to design of programming languages that support and encourage the design of reusable components. Section 2.6 concludes the chapter with a survey of existing languages evaluated with respect to the issues discussed in earlier sections.

2.1 Definitions

Several important terms have long been associated with software engineering. Essentially all software engineers use them, but unfortunately their definitions have not been standardized. The following are the definitions of these terms as used here.

Requirements are informal natural language statements of what a software component should do once it is designed and built. The design of a software component usually begins with requirements, which are often developed by (or by consultation with) a customer who needs the software but may or may not know precisely what he or she

expects it to do. The job of a software engineer in doing "requirements analysis" (or more recently, "domain analysis") is to pin down exactly what the software needs to do.

Specifications are formal statements of what a software component does. A specification is a complete and unambiguous description of the component, and constitutes a client's view of that component. Specifications are developed by the software engineer from the requirements and are the only official description of the component. In the event that requirements and specifications are inconsistent, specifications *always* indicate what happens. Specifications are formal statements, and in practice are mathematical assertions of some sort. For reasons discussed in Section 2.3.1, natural language statements, no matter how well-written, are not adequate for specifications.

Information hiding is a technique for keeping the irrelevant (e.g., implementation) details about a software component secret from all client programmers of that component[1]. There are two major reasons for doing this. First, if these details are revealed a client programmer may be confused by irrelevant information, making the component more difficult to understand. Second, once implementation details are revealed it is very tempting for a client programmer to take advantage of this information when using the component. This may make it difficult or impossible to upgrade the component later on with a new and improved version that has, say, better performance.

Abstraction is a technique for making a clear, comprehensible presentation of the information provided by a software component, explained in terms of "higher level" concepts than those used to implement the component. For example, abstraction is utilized when the operations provided by a character string manipulation package are defined using mathematical concepts such as "string" rather than implementation concepts such as bits, bytes, and null-terminated arrays of ASCII characters. Abstraction is important for the sole reason that without it, it would be all but impossible to reason about the behavior of any non-trivial software component.

---

1 Note that this definition hides information from client *programmers*, not just from client code. To many software engineers (e.g., [Meyer 88]), information hiding applies only to code. The client programmer may *see* internal structures and code, even though he or she may not write code that references them.

It is important to realize that information hiding and abstraction are different and it is possible to have the former without the latter (but not vice versa). For example, consider a LIFO stack whose behavior is explained in terms of an array with a "top" index. If the client programmer is prohibited from directly accessing the representation of a stack, information hiding is enforced even though abstraction was not used to describe the component. Alternatively, a stack could be explained to the client in terms of a mathematical string (see Sections 2.4.2.3 and 3.1.1), and implemented using an array with a top index. If the implementation is not mentioned in the explanation of the behavior, both information hiding and abstraction are utilized.

2.2 Reusability Issues

The primary motivation for this research is the development of reusable software components. The concept of reusable software raises several legitimate questions. For example, what exactly is meant by the term "reusable software component?" What advantages are there to designing reusable components? If reusable components are so wonderful, why aren't they developed more often? What would characterize a good reusable component? Answers to these questions are presented in this section.

2.2.1 What Is A "Reusable Software Component?"

In this dissertation "reusable software" refers to software components that can be incorporated into a variety of programs *without modification* (except possibly parameterization). Furthermore, reusing these software parts should not compromise other software engineering principles such as information hiding and data abstraction. Like many popular buzzwords, reusable software has many conflicting definitions.

Some researchers (e.g., [Lanergan 84]) consider source code skeletons or templates to be examples of reusable software. In this approach, templates contain the structure of commonly used sections of code and/or declarations, which are essentially pasted into new applications and customized using an editor. This approach does not meet our definition of reusable software because the templates must be modified when they are reused, and information hiding and data abstraction principles are violated because the client has complete access to the source code.

A similar approach to software reuse is the development of new programs by modifying existing ones. This methodology is popular in the UNIX™ community because of the availability of source code for much of the system. This reuse of software does not meet the definition here because the component is modified, and information hiding and data abstraction principles are violated.

Transforming source code from one language to another (e.g., [Boyle 84]) is sometimes considered to be an application of reusable software because it permits a routine coded in one programming language (e.g., FORTRAN) to be "reused" within a program coded in another language (e.g., Ada). This approach fails our reusability test because the component is modified, and information hiding and data abstraction principles are violated.

Some researchers (e.g., [Jones 84]) consider every program that has been executed more than once to be reusable software, because every time the program is executed it is reused. This definition of reuse does not meet our definition because it is not concerned with reusable software components incorporated into client programs.

A simple example of a reusable software part is a routine from a run-time library where the object code for the routine is linked with a client's object code to form an executable image. It should, however, be possible to define reusable components that are more powerful and versatile than those found in traditional run-time libraries. So although the traditional run-time library does not completely demonstrate the power of reusable software, it does convey the gist of it.

An observation is in order at this point. It seems highly unlikely that a software component not intentionally designed to be reused would qualify as a reusable component. Thus, a corollary of this definition of reusable software is that reusable components are not written by accident or "scavenged" from existing code by "software archeology," but must instead be consciously and deliberately designed for reuse.

2.2.2 Advantages of Software Reuse

There are several reasons why designing and building reusable software components potentially improves both software quality and programmer productivity. First, because

the cost of designing and building a component can be amortized over many uses, it is economically feasible to commit the time, intellectual energy, and money to do things right the first time. Committing necessary resources during the appropriate stages of the component's lifecycle improves quality in an obvious way.

Second, the designer of a reusable part knows that it will be used in applications unimaginable at the time of design, and will likely take the job seriously and design a quality part. He or she will probably take the time to look at the larger picture, imagine and anticipate uses and variations of the part, and make the design general by factoring out idiosyncrasies of specific applications. In addition, the psychological effect of knowing one's design will be scrutinized by future programmers can have a positive impact on the quality of the part's design.

Third and perhaps most important, programmer productivity will increase because it is usually easier to reuse a well-designed software component than to design and implement one on the fly. This proposition is considered dubious by some who envision only very simple components as reusable, but for components with complex behavior (or especially those with simple behavior but complicated implementations) it is quite obvious. However, this claim does not seem to have been established by experimental evidence yet, due partly to the near-absence of truly reusable software components.

2.2.3 Non-Technical Impediments to Software Reuse

Why is reuse not more common? The answer involves a combination of technical and non-technical issues. The non-technical issues are discussed here, while the technical issues are presented in the next section.

One non-technical obstacle is an economic one. Software companies are in the business of producing and selling software. If a company sells a truly general and reusable component to a customer, that customer may no longer need the services of the software company. In order for reusable software to be economically profitable, a pricing structure and incentive system must be found that rewards manufacturers of reusable components without eliminating the economic market for the components.

Another obstacle is an organizational one. Manufacturers of reusable components must provide potential customers with detailed catalogs describing the available components, and customers must be able to efficiently search through these catalogs and easily determine whether a particular reusable component is appropriate for a particular application[2]. The customer should also be able to electronically order and immediately download selected components for incorporation into new projects.

Negative psychological effects are another non-technical impediment to software reuse. The "Not Invented Here" syndrome is alive and well in the workplace, and programmers need encouragement to reuse someone else's component rather than develop their own home-brew version. Another form of this phenomenon might be called the "Macho Programmer Syndrome" where programmers (and indeed entire companies) regard software reuse as an admission that they are not as good as the other programmer, and therefore see reuse as a weakness. Of course, availability of reusable components that are significantly more reliable, efficient, and cost effective than custom-designed ones could provide the necessary incentives for programmers and companies to reuse software components.

Efforts are underway to address some of these obstacles (see, for example, [Biggerstaff 89a], [Biggerstaff 89b], [IEEE 87], and [IEEE 84]). However, the discussion in this dissertation does not address them.

2.2.4 Technical Impediments to Software Reuse

In addition to the non-technical obstacles just mentioned, there are several technical issues that currently keep reusable software from becoming a reality. In a very real sense solutions to these technical issues are more important than solutions to the non-technical ones, for until it is technically possible to design and build truly reusable components, management and organization cannot achieve widespread reuse of software.

The first technical obstacle is the lack of formal specifications for components. A programmer cannot be expected to reuse an existing part unless its functionality is

---

2 Of course, these catalogs would not be the kind one reads in the library or uses to elevate toddlers at the dinner table, but instead would be an electronic database.

crystal-clear. All too often programmers "reinvent the software wheel" because the functionality of existing parts is unclear or vague, and the alternatives — deciphering source code and trial-and-error testing — are often more painful than simply starting from scratch. A component will only be reused if its behavior is completely and unambiguously specified in a form understandable by potential programmers. As discussed in Section 2.3.1, these specifications should be mathematically rigorous. Specifically, informal natural language descriptions are not sufficient. Also, the principles of information hiding and abstraction should be followed, so providing the client with source code of the component (as suggested in [Berard 89]) is not acceptable.

A second technical impediment is the inability to certify the correctness of a component. Of course, attempting to certify the correctness of a component whose specification is incomplete or ambiguous is an exercise in futility — how can one say whether a part behaves correctly when it is unclear exactly how that part is supposed to behave? Obviously the problem of formal specification must be solved before this issue can be meaningfully addressed. However, even with formal specification the problem of certification (i.e., formal verification) is difficult, in part because the techniques are still being developed and have generally not been applied to large programs with complex data structures. Also, many programming languages have constructs (such as aliasing) that significantly complicate program verification [Hoare 83, Krone 88]. Testing, a weaker certification method, has received much attention recently (e.g., [Mills 87]), yet there are subtle barriers facing the would-be tester of a reusable software component [Hegazy 89].

Another technical obstacle is the relatively poor performance of reusable parts. Part of the problem here is the assumed trade-off between generality and performance that most programmers believe exists. In fact, there is no theoretical basis for this belief, although empirical evidence seems to support it. The problem is that most parts classified as reusable were designed and implemented using classical data structures and algorithms as taught in introductory computer science classes. These components, however, were not designed to be reusable, and performance suffers as a result. New evidence suggests that reusable parts can be designed that exhibit no significant performance degradation relative to non-reusable custom parts [Harms 89a].

2.3 Characteristics of Reusable Parts

Discussion in previous sections dealt with important reusability issues such as a definition for reusable software, why reusable software is a good thing, and some of the reasons why reusable software components are currently not designed and built very often. This discussion is concluded by presenting some characteristics that well-designed reusable components would exhibit. The discussion is on a general level and language-independent. However, the intent is to define some organizational structures and constructs that should be included in any language supporting the development of reusable software.

2.3.1 Formal Specification

Every part must have a specification. A specification is essential for several reasons. First, it tells a potential client programmer exactly what the part does [Parnas 72]. This is important because a programmer cannot be expected to reuse a component whose function is ambiguous or vague. In addition, if information hiding is enforced (the desirability of which is assumed throughout this dissertation), a specification is the *only* way for a client programmer to know what a particular piece of software does.

Second, a specification tells the implementer of a part what the code must accomplish — abstractly, completely, precisely, and unambiguously [Parnas 72]. The implementer should not need any more information than what is presented in the specification. In particular, he or she should not need to know anything about the client, for if he or she did, the part would not be so reusable. Thus, a specification is the dividing line between a client and an implementer, constituting the only contract (and contact) between them.

Third, formal specifications are a necessity if formal verification is to succeed [Liskov 75]. An implementation of a part cannot be proved correct unless it is known precisely what the code is supposed to do. Of course, some researchers (e.g., [DeMillo 79]) consider formal verification as a hopelessly futile endeavor. However, others are much more optimistic about the prospects, making statements such as "Software engineering without mathematical verification is no more than a buzzword"

[Mills 87]. One thing is for certain though — formal verification is not possible without formal specification.

A fourth and final advantage of formal specification is that the process of writing a formal specification often helps reveal ambiguities and inconsistencies in the part's design. This is because "formal specifications naturally lead the specifier to raise some questions that might have remained unasked, and thus unanswered, in an informal approach" [Meyer 85]. Thus, a formally specified part probably has a better design than one not subjected to the rigors of formal specification.

Obviously, the language in which specifications are written must permit (and preferably encourage) the expression of precise and unambiguous statements. Historically the languages best suited for this have been the languages of mathematics, such as first-order predicate calculus. In this view a specification consists of mathematical assertions of some sort (see Section 2.4.2).

But must specifications necessarily be mathematical in nature? Wouldn't it be just as good (or even better) for specifications to be well-written natural language statements rather than some mathematical mumbo-jumbo? It appears that the answers to these questions are "yes" and "no," respectively. A natural language is an inappropriate specification language for precisely the same reason it is such a wonderful medium for poetry and literature — it is easy to say one thing and mean another. [Meyer 85] provides a detailed example and analysis of the inherent dangers of English as a specification language.

Although natural language statements are not specifications, they can be instrumental in helping someone understand the function of a part [Parnas 72]. For this reason the designer of a part is encouraged to include an informal natural language description of the part with the part's definition. This description is included solely for use by programmers (both client and implementer) and is not intended for any kind of automatic processing (e.g., compilers and verifiers). Because a description is written in natural language it is by nature imprecise, and formally checking the consistency between it and the formal specification is impossible. In the event of a contradiction between the description and specification, the specification is *always* the final word.

Formal specification impacts the design of a programming language in several ways. First, mathematical assertions must be an integral part of the language, not simply considered as comments optionally entered by the programmer. Assertions should be required at strategic points in the code, such as pre- and post-conditions of operations and loop invariants. The language in which assertions are written must be powerful enough to completely specify a part's function, such as first-order predicate calculus. Specifically, propositional logic expressions are not sufficiently powerful. Of course, assertions may be ignored by the compiler, but it should be possible to mechanically process them by some other tool, such as a verifier.

Second, for verification to be possible, the semantics of control and data structuring mechanisms of the language must be be formally defined. Several methods are available to express the semantics of a language, including denotational semantics, axiomatic semantics, and operational semantics [Gordon 79, Marcotty 76]. Formal semantics influence language design because some programming language features are difficult to define formally and therefore should not be included in the language[3].

A consequence of this last point is that a language encouraging the development of formally-specified programs must be explicitly *designed* with this as a goal. It is very difficult to "retrofit" formal specification into an existing language, since it most likely has constructs not at all amenable to formal semantic description (and thus formal verification).

This is perhaps the primary flaw with specification languages that are defined independently of a programming language (e.g., Z [Spivey 89]). In these systems the formal specification of a software component is written in the specification language, and the actual implementation is coded in some programming language. This code can then be verified, assuming it has been decorated with appropriate assertions and the semantics of the implementation language are formally defined. This is unlikely unless the programming language was designed with formal specifications in mind, in which case the specification language would have been defined as part of the programming language, and would not be language-independent!

---

3 Some language designers (e.g., [Wirth 83]) argue that features that are difficult to formally define would also be difficult for a programmer to understand, and thus should not be included in a language.

Very few programming languages offer a programmer the ability to formally specify a program. In fact, it is not possible to formally specify a program in any of the "popular" languages[4] (e.g., COBOL, FORTRAN, Pascal, C, and Ada). Also, some languages that appear to offer this capability don't actually have the power to write formal specifications (e.g., Eiffel, discussed in Section 2.6.7). Some languages that are exceptions to this are Alphard, Gypsy, and Euclid, which are discussed in Section 2.6.

2.3.2 Separation of Specification from Implementation

A second characteristic of a well-designed reusable component is the separation of the part's specification and implementation into separately-compilable units. This is beneficial for several reasons. First, it is perhaps the most effective method of realizing and enforcing information hiding. A client programmer is told what a part does by providing him or her with the part's specification. Placing the implementation someplace else and not allowing the programmer access to it prevents him or her from knowing anything about how that part is implemented. What better way to hide information!

Second, separating specification from implementation permits designers and client programmers alike to consider a part's abstract function and performance (e.g., space and time) as separate issues. This also opens up the possibility for a particular specification to have multiple implementations, each with different performance characteristics. Additional discussion about this is contained in Section 2.3.4.

Third, separating specification from implementation allows a client program using a part to be compiled and verified even before an implementation is written. (Of course an implementation must be written before the client can be linked and executed.) This makes it possible to have an implementer and client programmer coding independently, which is crucial in large programming projects. It also means that a client will not need to be recompiled (or reverified) when the implementation is selected (or changed).

---

4 It is important to realize that what many programmers and languages call specifications (e.g., Modula-2's "definition module" and Ada's "package specification") are merely signatures of exported items and not specifications in the sense used here.

This also implies that a client can be verified independently of any implementation. The verifier assumes that all parts referenced in the code are implemented correctly, which significantly reduces the amount of work the verifier needs to do [Krone 88].

Most modern languages supporting "modular" design (e.g., Ada and Modula-2) separate implementation from "headers" containing signatures of exported items. Since these headers do not contain formal specifications, not all advantages discussed here can be realized (e.g., verification). However, such languages have many of these advantages.

2.3.3 Generics

A well-designed reusable part should be generic if at all possible. A generic part is one whose definition is parameterized somehow (e.g., by a component type). For example, rather than defining separate parts for "stack of ints" and "stack of chars," it would be better to define a generic part for "stack of Item," where Item is a type parameter to the part. A generic part is not a usable part on its own, but is instead a *template* for a family of parts. When a template is instantiated a usable part is created that has arguments bound to all parameters. For example, instantiating "stack of Item" with type int creates a "stack of ints," and instantiating it with type char creates a "stack of chars." Generic parts achieve a degree of reusability in an obvious way.

Implementations of generic parts should not depend upon specific actual type parameters. For example, an implementation of stacks must implement stack of *any* component type, including stacks of ints and stacks of queues of chars. In other words, implementations of generic specifications must themselves be generic.

The ability to define generic parts is provided by many modern programming languages, including Ada, CLU, and Eiffel, which are discussed in Section 2.6.

2.3.4 Multiple Implementations

A particular specification has infinitely many possible implementations. Each one can be characterized by any number of performance characteristics, such as the amount of space required to store the structure, the amount of time necessary to access the structure, and

the amount of space required for the implementation code. Many of these potential implementations exhibit such poor performance that no sane programmer would ever code them. However, most parts seem to have several implementations that exhibit "reasonable," yet different, performance. For example, one implementation of a part may require O(1) and O(n) time to insert and remove items from the structure, respectively, whereas another implementation may require O(log n) for both operations. Which of these two implementations is better? The answer is very dependent on the particular client, so it is not possible to say that one is universally better than the other.

A client programmer should be able to select an implementation from a list of available ones. This selection need not be made when the reusable part (i.e., the specification) is chosen, but can instead be postponed until later, thereby factoring decisions regarding a part's function from its performance.

For flexibility, a client programmer should be allowed to choose an implementation for one part, and another implementation for a different instance of that same part (e.g., one stack implemented as a linked list, and another stack within the same client implemented as an array with top index[5]). Thus, implementation of a part is selected on an instance-by-instance basis, rather than on a client-by-client one.

Also, it should be possible for a client programmer to select a different implementation of a part without having to recompile or reverify the client. This allows a client's performance to be changed without materially altering its code, and allows a client programmer to take advantage of new implementations developed at a later time.

Multiple implementations affect the design of a programming language in several ways. First, multiple implementations of a part must be permitted, and the specification of a component must not be bound to any particular implementation or set of implementations. In other words, a specification is completely independent of its implementation(s), and should not be affected if and when a new implementation is developed[6].

---

5 Because of information hiding, the implementations should be described by abstract performance characteristics, not in terms of implementation structures such as linked lists and arrays.

6 The converse is not true — i.e., it is perfectly reasonable to bind an implementation to exactly one specification. This is because it is very unlikely that an implementation realizes more than one specification.

Second, the mechanism used to bind a specification of a part to an instance of it within a client must be separate from the mechanism used to bind an implementation to that instance. In fact, it is conceivable at first glance that only the former binding (i.e., specification of an instance) be done in the client code, leaving the implementation binding to be specified at "link" time.

Third, the language must have constructs through which the performance characteristics of an implementation can be expressed in abstract terms (e.g., "big-O" notation). This is necessary because the client programmer must be able to understand the performance of an implementation without studying its code. (Besides, the client programmer doesn't even have access to the implementation code, since we're assuming information hiding.) One implication of this is that an implementation will not be described in terms of implementation structures (e.g., linked lists or arrays), but rather in abstract terms.

No modern programming language supports this criterion well. For example, Modula-2 binds every DEFINITION MODULE to exactly one IMPLEMENTATION MODULE. Ada effectively does the same, except that it's possible to write a package specification in such a way that multiple package bodies (i.e., implementations) can be defined. However, it is not possible to bind different package bodies to different instances of the same package within a single client.

Inheritance in object-oriented languages such as Eiffel is a mechanism whereby multiple implementations can be written for a class or operation. However, inheritance allows (and encourages) multiple implementations of individual operations within a class, rather than multiple implementations of entire classes. Section 2.4.1.1 contains a more detailed discussion of inheritance.

2.3.5 Efficient Implementations Possible

A part that has no efficient implementation will be neither used nor reused. This statement seems obvious, yet its impact is subtle, especially when considered in conjunction with the other characteristics discussed in this section. For example, many parts will be generic (e.g., stack of Item), and must be efficient for any component type. In other words, an implementation of stacks must be efficient for both "stack of ints" and "stack of queue of chars."

Perhaps the biggest implication of this is that unnecessary copies of data should not be made because copying is inefficient (usually taking time linear in the size of the representation data structure). This significantly impacts programming language design, especially with respect to parameter passing mechanisms. For example, Figure 1 shows an informal definition of a module providing a generic stack and operations Push, Pop, and IsEmpty[7]. This definition is in a hypothetical "Pascal-like" language, and is quite common in data structure texts, such as [Sedgewick 88].

```
definition module Stack_Template (type Item);
  exports Stack, Push, Pop, IsEmpty;

  type Stack;

  procedure Push (var S:Stack; x:Item);
    -- places x onto stack S, without changing x

  procedure Pop (var S:Stack; var x:Item);
    -- requires stack S is not empty
    -- removes the topmost item from stack S, returning it in x

  function IsEmpty (S:Stack) : boolean;
    -- returns true iff stack S is empty

end Stack_Template;
```

Figure 1

Informal Description of a Template Providing Type Stack

One problem with this definition centers around the specification of procedure Push. Note that Push cannot change the actual parameter for x, effectively forcing a copy to be made of the item being pushed onto the stack. This is tolerable if type Item is a simple type, such as int or char, but it must be considered unacceptable in general, especially if there is a more efficient alternative.

The alternative suggested by most programming languages (and programmers) is to pass a pointer to the item being pushed, rather than passing a copy of the item. This could be accomplished either explicitly (e.g., make the type of parameter x "pointer to Item") or implicitly (e.g., make x a "call-by-reference" parameter). This solves the problem at hand. However, introducing pointers at this level makes formal specification and

---

7 Note this is *not* the recommended definition of stacks. The definition that solves the problems inherent to this example is presented in Figure 2 in Section 3.1.1.

verification difficult (see Section 2.3.1) and has other significant problems discussed in Section 2.5.1. Pointers are not the answer.

If copying data structures is bad, and introducing pointers into the language is bad, what other options are there? To date, no language has offered any other alternative. However, in Section 3.3.1, swapping the values of two variables is introduced as a data movement primitive that solves the problem of moving large data structures efficiently and does not introduce pointers into the language definition.

As just mentioned, no modern programming language offers anything other than pointers as an alternative to copying large structures. Consequently, no modern language permits implementations of generic parts to be both formally specified and efficient. This is strong evidence supporting the thesis that no modern programming language has all of the constructs necessary to encourage and facilitate the design of reusable software parts. In fact, existing languages have constructs, such as implicit aliasing, that actually thwart design of and with reusable software components.

2.4 Reusable Part Design Issues

The discussion in the previous section dealt with characteristics of reusable parts. However, there are at least two open issues regarding the design of reusable components:

- What functionality should be incorporated into a reusable part?
- How does one go about formally specifying a reusable part?

These issues are addressed in the following subsections.

2.4.1 Encapsulation

A reusable part is usually defined by a module encapsulating a number of different kinds of items — e.g., types, operations (i.e., procedures and/or functions), state variables, and constants — which are provided to a client of the part. One of the tasks of a designer of a part is to decide which items are provided by the reusable part. These

decisions strongly influence the "flavor" of the part as well as the extent to which the part may actually be reused.

In this section two approaches to module design are discussed — the Abstract Data Object approach and the Abstract Data Type approach. Throughout this discussion it is important to understand that these approaches are simply two reference points on the "modularization approach continuum." The design of most modules actually falls at some intermediate point on this continuum. However, the classification is useful, since programming languages generally support and encourage module design skewed toward one of these reference points.

Section 2.4.1.3 concludes with a brief discussion of an important classification of operations — primary vs. secondary.

2.4.1.1 Abstract Data Objects

In the Abstract Data Object (ADO) approach to encapsulation, data and operations to manipulate that data are encapsulated in something called an object. A module defines a template for a class of objects, and a client uses templates to declare objects. The data portion of an object is not directly available to the client. Rather, the only way for a client to do anything with the data is by invoking its attached operations.

For example, the following is an informal definition in a hypothetical language of a module defining a template for stacks of integers, where the available operations on a stack are Push, Pop, and IsEmpty[8].

---

8 ADO modules do not need to define data because a client cannot access it. The only reason for having data declarations would be to specify the effect of the operations. Since this is just an informal description data declarations are not included.

```
class Stack is

  procedure Push (x:integer);
    -- places x onto the stack without changing x

  function Pop () returns integer;
    -- requires stack is not empty
    -- removes the topmost item from the stack, returning it

  function IsEmpty () returns boolean;
    -- returns true iff the stack is empty.

end Stack;
```

A client would use the variable declaration facility to declare Stack objects. For example:

```
var s1 : Stack;
var s2 : Stack;
```

declares two Stack objects, called s1 and s2. Conceptually, each Stack object (i.e., variable) contains the representation for a stack as well as code for operations Push, Pop, and IsEmpty. Pushing the value of integer variable y onto stack s1 would be accomplished by the statement:

```
s1.Push (y);
```

Note the use of the "." in this statement. The object being manipulated is s1 and the operation being performed is the Push operation — not just any Push, but the one belonging to object s1.

In the ADO approach, an object is treated as a black box with one or more buttons on it. Each button (i.e., operation) may have slots around it (i.e., the operation's parameters) into which items may be placed, and from which items may appear. The meaning of each button is described in some sort of owner's manual (i.e., specification). To use the box, necessary items are placed in the appropriate slots, a button is pressed, the internal mechanisms of the box churn, and any items produced by the operation are removed from their slots.

This analogy suggests other important characteristics of the ADO approach. For instance, the only purpose of an operation (i.e., button) is to do something to its associated object (i.e., black box). In other words, *every* operation must be explicitly attached to exactly one object. An operation affecting more than one object (e.g.,

procedure Push in the above example) is attached to one of the objects (e.g., a Stack), and the rest are passed as parameters (e.g., integer x). This asymmetry is quite natural for many operations, such as those for Stacks, but makes the definition of others seem awkward. For example, a function that returns the sum of two integers is conceptually symmetric, yet in the ADO approach it must be attached to one of the integers. The function invocations "`x.add(y)`" or "`y.add(x)`" are awkward ways of calculating the sum of x and y, which is expressed as "`x + y`" or "`add(x,y)`" in traditional languages.

As a second example of ADO characteristics, suppose a client wished to define an operation that reverses the items on a Stack[9]. This operation must be attached to an object, and it seems reasonable to attach it to Stacks. Because of information hiding, however, the client cannot modify an existing class. Instead, the client might define a new class that *inherits* Stacks and defines an additional operation. The new class is often called an heir or descendant of the inherited (or ancestor) class. For example, the following might be the syntax (again in the hypothetical language):

```
class Rev_Stack inherits Stack is

  procedure Reverse ();
    -- Reverses the stack

end Rev_Stack;
```

Rev_Stack objects have operations Reverse, Push, Pop, and IsEmpty. Also, since Rev_Stack objects inherit all operations of Stacks, they can be used in any context where a Stack object is legitimate. For example, if a parameter to an operation needed to be a Stack, a Rev_Stack would work just as well.

Inheritance can also be used to "redefine" an inherited operation, allowing (presumably minor) modification of the behavior of the inherited class. Assuming that information hiding is observed, these modifications must be coded in terms of operations defined in the inherited class[10]. The specification of the operation should also change to reflect the modification, which may have repercussions on other operations. For example, class

---

9 This can be accomplished by invoking operations Push, Pop, and IsEmpty, and using a temporary FIFO queue. In other words, access to the internal representation of Stacks is not necessary.

10 Allowing a descendant to have direct access to its ancestor's representation increases the usefulness of inheritance, and many "object-oriented" languages allow it, even though it directly violates the principle of information hiding.

Rev_Stack could redefine procedure Push. If an operation that needed a Stack object was given a Rev_Stack instead, all invocations of this parameter's Push invoke the procedure redefined in Rev_Stack rather than the one defined in Stack. Thus, when the operation invokes Push, it has no way of knowing which Push will actually be invoked. The ability to modify an operation increases the power and flexibility of a class, but complicates formal specification and verification, for obvious reasons.

The ADO approach underlies what is commonly called "object-oriented programming," which is encouraged and facilitated by programming languages such as Simula-67, C++, CLU, Smalltalk-80, and Eiffel. Note, however, that our definition of the ADO approach factors out several characteristics sometimes associated with object-oriented languages, such as polymorphism, dynamic typing, "typeless" variables, and variables as object references. These are considered to be characteristics of specific languages rather than characteristics of an encapsulation approach. Consequently, they are discussed in the context of specific languages in Section 2.6.

2.4.1.2 Abstract Data Types

In the Abstract Data Type (ADT) approach to encapsulation, a module (sometimes called a package) defines one or more types and operations involving parameters of those types. A client declares variables of the provided type, and a variable is considered to contain a value from the domain of its type. A client cannot directly access the internal representation of the variables, but must pass them as actual parameters to operations.

For example, the following is an informal definition in a hypothetical language of a module providing a type for stacks of integers and operations Push, Pop, and IsEmpty:

```
module Stack_Module is

  type Stack;
    -- type for stacks of integers.

  procedure Push (s:Stack; x:integer);
    -- places integer x onto stack s, without changing x.

  function Pop (s:Stack) returns integer;
    -- requires stack s is not empty
    -- removes the topmost item from stack s, returning it

  function IsEmpty (s:Stack) returns boolean;
    -- returns true iff stack s is empty.

end Stack_Module;
```

A client would use the variable declaration facility to declare stacks. For example,

```
var s1 : Stack;
var s2 : Stack;
```

declares two stacks variables, called s1 and s2. Pushing the value of integer variable y onto stack s1 would be accomplished by the statement:

```
Push(s1,y);
```

and pushing integer z onto stack s2 would be accomplished by:

```
Push(s2,z);
```

Note that conceptually there is now only one Push operation, which pushes an item onto any stack it is given.

Operations can be defined independently of a type, in the sense that they can be declared outside of any module providing a type. For example, suppose a client wanted to declare a procedure to reverse the items on a Stack (which does not need access to the representation of a Stack). He or she could easily write a procedure called Reverse that has one parameter of type Stack. This procedure could be invoked with any Stack variable.

In a "black-box" analogy similar to that presented for the ADO approach, an ADT variable is treated as a black box containing some data but no operations. Each operation is treated as a machine. Each machine has exactly one button, and may have one or more slots (i.e., the operation's parameters) into which data may be placed, and

from which data may appear. The meaning of each machine is described in an owner's manual (i.e., the operation's specification). To use a machine, necessary data items are placed in the appropriate slots, the button is pushed, the machine munges with the data items, and any data items produced are removed from their slots.

Some popular languages that encourage and facilitate this style of programming are Modula-2 and Ada. Specific features of these languages are discussed in Section 2.6.

2.4.1.3 Primary and Secondary Operations

When designing a module, there are potentially many operations that could be included in it. For example, reversing a stack is an operation that is occasionally useful. Should it be included as an integral part of stacks, or relegated to the status of an operation that can be added later? A classification of operations that is useful in this context distinguishes between *primary* operations and *secondary* ones.

Primary operations are those that are necessary to define the "essence" of the structure, in the sense that all "useful" operations can be defined in terms of them. All other operations are secondary ones, and can be implemented by invoking combinations of primary operations[11]. For example, operations Push, Pop, and IsEmpty are primary operations for a stack, since they are essential in defining a LIFO structure. On the other hand, Reverse and Copy are secondary operations because they can be coded by invoking the primary operations.

2.4.2 Specification

Section 2.3.1 argues that formal specification is a necessary characteristic of reusable software parts, though it does not address how one goes about formally specifying a part. This section presents some background for formal specification, and discusses two popular approaches to constructing formal specifications — the algebraic approach, and the model-based approach[12].

---

11 This distinction is similar to that made in [Parnas 72] between "normal" functions and "mapping" functions.

12 Other specification techniques, such as state machines and axiomatic descriptions, are discussed in [Liskov 75].

2.4.2.1 Mathematical Theories

Formal specification, as argued in Section 2.3.1, must be mathematical. Therefore, let's take a closer look at what mathematical theories are, and how they are developed.

A typical mathematical theory is characterized by a *signature* and a set of *axioms*. The signature is a set of symbols that are defined by the theory. These symbols stand for mathematical types, mathematical constants, or mathematical functions. The axioms are statements that define the mathematical type by defining its domain and a notation for expressing functions and relations among the values in this domain.

Proofs in a theory construct statements (or formulas) from the axioms using certain rules of inference, such as modus ponens and substitution. These formulas are the *theorems* of the theory. Conversely, a formula is said to be provable (i.e., a theorem) if and only if there exists a proof for constructing it. In addition, *definitions* are often made, which are simply short-hand notations for use in writing formulas.

For example, one characterization of mathematical string theory over an alphabet Σ has the signature (Σ^*, Λ, C) and three axioms. Here, Σ^* is the mathematical type of strings over Σ, $\Lambda:\Sigma^*$ is a particular (constant) string, and $C:\Sigma^* \times \Sigma \rightarrow \Sigma^*$ is a function. These symbols are implicitly defined by the following axioms:

I. $\forall \alpha:\Sigma^*, \forall z:\Sigma,$
$C(\alpha, z) \neq \Lambda$

II. $\forall \alpha,\beta:\Sigma^*, \forall y, z:\Sigma,$
$(C(\alpha, y) = C(\beta, z)) \Rightarrow (\alpha = \beta \wedge y = z)$

III. $\forall S \subseteq \Sigma^*, \forall \alpha:\Sigma^*, \forall z:\Sigma,$
$((\Lambda \in S) \wedge (\alpha \in S \Rightarrow C(\alpha, z) \in S)) \Rightarrow S = \Sigma^*$

Some useful definitions might be:

$Cat : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$
(i) $Cat(\alpha,\Lambda) = \alpha$
(ii) $Cat(\alpha, C(\beta, z)) = C(Cat(\alpha,\beta), z)$

$\texttt{Len} : \Sigma^* \rightarrow \mathbf{N}$

(i) $\texttt{Len}(\Lambda) = 0$

(ii) $\texttt{Len}(\texttt{C}(\beta, x)) = \texttt{Len}(\beta) + 1$

Theorems of string theory are constructed from these axioms and definitions using the rules of inference. For example, the following is a theorem:

$\forall \alpha,\beta:\Sigma^*, \texttt{Len}(\texttt{Cat}(\alpha,\beta)) = \texttt{Len}(\alpha) + \texttt{Len}(\beta)$

For a theory to be useful, several properties must be proved. One of these is *consistency*, which says that it is impossible to prove both a formula and its negation (e.g., it is not possible to prove both "$\forall \alpha,\beta:\Sigma^*, \texttt{Len}(\texttt{Cat}(\alpha,\beta)) = \texttt{Len}(\alpha) + \texttt{Len}(\beta)$" and "$\forall \alpha,\beta:\Sigma^*, \texttt{Len}(\texttt{Cat}(\alpha,\beta)) \neq \texttt{Len}(\alpha) + \texttt{Len}(\beta)$").

Theorems and axioms are just sequences of symbols with no inherent meaning or notion of truth. In other words, theorems have syntactic structure but no semantic meaning. Of course, the most important function of mathematics is its ability to say things *about* something, so a theory is practically useless unless its theorems and axioms have meaning.

Meaning is associated with a theory by modeling (or interpreting) each symbol by some meaningful entity. For example, one meaningful model of string theory interprets the constant Λ as the empty string, and the value of the function $\texttt{C}(\alpha, z)$ as the string obtained by attaching the symbol z to the right-hand end of the string α. The definition Cat corresponds to concatenating two strings together, and definition Len is the length of a string. This interpretation associates semantic meaning to every axiom and theorem of string theory. For example, the theorem mentioned above means that the length of the concatenation of two strings is the sum of the lengths of the two strings. Although this is the intended and most straightforward model for string theory, there are many other ones.

There are many statements in the vocabulary of string theory whose meaning under a particular interpretation is either invalid or nonsensical. For example, under the interpretation for string theory given in the previous paragraph, the statement "$\Lambda\Lambda\texttt{Cat}(\Lambda,\Lambda)\texttt{Len}(x)$" is nonsensical, while the statement

"$\forall\alpha,\beta{:}\Sigma^*$, `Len` (α) = `Len` (β)" is invalid. If either of these statements happened to be theorems in string theory, we'd be in big trouble!

In fact, for an interpretation to be a useful model of a theory, two important properties must hold between the theory and the interpretation — soundness and completeness. A theory is *sound* with respect to an interpretation if and only if the meaning of every theorem under the interpretation is valid. For example, if string theory is sound with respect to our interpretation, the meaning of every theorem of string theory is valid. In other words, no theorem would have an invalid or nonsensical meaning. Soundness is crucial for obvious reasons.

A theory is *complete* with respect to an interpretation if and only if every valid statement under the interpretation is a theorem of the theory. Completeness says that it is possible to construct a theorem for every statement that is valid. For example, in our interpretation of string theory the statement "`Cat(Cat(`α,β`),`γ`)` = `Cat(`α`, Cat(`β,γ`))`" is valid. If string theory is complete with respect to this interpretation, this statement is also a theorem.

Showing that a theory is both complete and sound with respect to an interpretation is a non-trivial task, and requires a formal semantic definition of the interpretation. Without these results, however, a theory is practically useless.

2.4.2.2 Algebraic Specification

In the algebraic approach to formal specification, a mathematical theory is developed for each data structure. For example, Guttag [Guttag 77] develops the following theory for "Stacks of T" (denoted S(T)):

```
Signature is (S(T), Γ, Push, Pop, Top, IsEmpty), where
        Γ : S(T)
        Push : S(T) × T → S(T)
        Pop : S(T) → S(T) ∪ {⊥}
        Top : S(T) → T ∪ {⊥}
        IsEmpty : S(T) → B
```

```
Axiom Schemas are
      I.    IsEmpty(Γ) = true
      II.   IsEmpty(Push(s,x)) = false
      III.  Pop(Γ) = ⊥
      IV.   Pop(Push(s,x)) = s
      V.    Top(Γ) = ⊥
      VI.   Top(Push(s,x)) = x
```

In the intended interpretation of this theory, constant Γ is the empty stack, function IsEmpty(s) is true iff s is the empty stack, Pop(s) returns the stack obtained by removing the top element from s, and Top(s) returns the top element of s without removing it. Also, it is an error to attempt to Top or Pop from the empty stack.

Several important questions must be answered if we are to have any confidence in a theory developed in this manner — is the theory consistent, is the theory sound with respect to our intended interpretation, and is the theory complete with respect to this interpretation? In other words, does the theory (e.g., Stack Theory) *exactly* capture our intuitive notion of the data structure (e.g., LIFO stacks)? As mentioned in the previous section, proving soundness and completeness are non-trivial exercises. However, they must be done for each theory developed.

Also, since these theories are "brand new" there are no theorems available to help a verifier prove the correctness of a program. Developing a set of useful theorems is also non-trivial.

Another potential problem with this approach is visualization — specifically, axioms are not easy to visualize. For example, a stack is usually visualized by drawing pictures or having in mind some physical model (e.g., a "stack of cafeteria trays"). In these visualizations notions like "top" and "empty" make a lot of sense. On the other hand, no images pop into mind (pun intended!) with the axiom "`Pop(Push(s,x)) = s`". However, the lack of visualization may simply be a problem with our current thinking, and given enough practice with these theories, visualizations may appear quite naturally.

Assuming that a theory is sound and complete, how does one use it to specify a reusable part? There are two approaches to this. In the first approach, all operations provided by

the reusable component are defined as mathematical functions in the theory. In a sense, the theory is not merely *used* to specify the part, it *is* the specification. One problem with this is that adding a new operation to a part involves rewriting the theory and consequently, reestablishing soundness and completeness. It also means that theories may have more functions defined than absolutely necessary. For example, a theory for stacks can be defined without function IsEmpty or axioms I and II [Liskov 75]. However, if the reusable part provides an IsEmpty operation, IsEmpty and the corresponding axioms must appear in the theory. This approach is promoted in [Guttag 78] and [Musser 80].

A second approach to using these theories is to model the data provided by the reusable part as a value from the domain of the mathematical type. Operations are defined with assertions (i.e., pre- and post-conditions) involving functions and relations defined in the theory. This is actually a variant of the model-based approach discussed in the next section, and more will be said there. This approach is encouraged by languages such as Larch [Wing 87, Guttag 85].

2.4.2.3 Model-Based Specification

Unlike the algebraic approach to specification, the model-based approach specifies reusable parts in terms of existing mathematical theories. The data provided by a part is modeled as a value from some mathematical domain. For purposes of reasoning about program data, a corresponding mathematical value is considered to be its mathematical model. Operations are defined with pre- and post-conditions, which are assertions about the values of the operation's parameters before and after the operation invocation. Within these assertions parameters are considered to be mathematical values of the appropriate type, and the assertions involve functions and relations in the theories defining those types.

For example, a "stack of T" can be modeled as a "mathematical string of T," with the top of the stack at the right-hand end of the string. For purposes of reasoning, stacks are considered to be mathematical strings. The effect of operations Push, Pop, and IsEmpty can be expressed formally as assertions in string theory:

```
procedure Push(s:Stack,x:T)
  ensures "s = C(#s,x) and x = #x"

procedure Pop(s:Stack,x:T)
  requires "s ≠ Λ"
  ensures "#s = C(s,x)"

function IsEmpty(s:Stack) : boolean
  ensures IsEmpty iff "s = Λ"
```

In this notation, a requires clause specifies the pre-condition that is expected to be true when the operation is invoked, and an ensures clause specifies the post-condition that is guaranteed to be true when the operation returns. Within an ensures clause, a '#' in front of a parameter denotes the value of that parameter at the beginning of the operation. For instance, the effect of Push(s,x) is that upon completion, the value of s (treated as a string over alphabet T) equals the old value of s with x appended to the right-hand end, and x is unchanged.

A major advantage to the model-based approach is that existing mathematical theories are "reused." This means that the specifier can take advantage of the knowledge gained over the years by mathematicians — for example, completeness and soundness properties of the theories, and a wealth of theorems.

Some might argue, though, that many interesting data types cannot be appropriately modeled by "common" mathematical theories such as number theory and string theory. Recent research, however, has demonstrated that a large number of useful reusable parts can be appropriately (and intuitively) specified in terms of surprisingly few mathematical theories. These theories include function and relation theory, set theory, number theory, string theory, and graph theory [Weide 86a].

Another advantage is that these theories are easy to visualize, and drawing pictures is a good way to understand a part's formal specification.

2.5 Programming Language Design Issues

Section 2.3 presents characteristics of well-defined reusable parts, and discusses the impact these characteristics have on programming language design. This section "inverts" the presentation by discussing programming language issues and their impact

on the design of reusable parts. Included in the discussion of each issue is the approach taken in RESOLVE.

The issues discussed in this section relate to pointers (Section 2.5.1), data movement and parameter passing (Section 2.5.2), and types and variables (Section 2.5.3).

2.5.1 Pointers

Pointers (a.k.a. references and indirect addresses) are one of the most fundamental and powerful concepts in modern computer architectures, which allows a value to be interpreted either as a value or as the address of a value. Fundamental machine structures such as program counters, stack pointers, base registers, and return addresses are based on this idea. In fact, it is practically impossible to imagine a machine language program that does not rely on pointers. However, pointers are also one of the most dangerous aspects of machine code programming because of the potential for confusion between an address and its contents. Program bugs resulting from misuse of pointers are notoriously disastrous and difficult to find and correct.

A good high level language protects the programmer from the ugliness and dangerousness of the actual machine while enhancing the programmer's ability to solve problems. Unfortunately, high level languages have not been successful at hiding all details of the machine from the programmer. For example, pointers in one form or another appear in just about every language. In some languages, such as FORTRAN, pointers are implicitly part of the language definition because they are necessary to explain parameter passing to subroutines (which is discussed in more detail in the next section). Pointers are also implicit in many "object-oriented" languages, such as CLU [Liskov 81] and Eiffel [Meyer 88], because all class variables are defined as references to the actual class structure. Knowing these implicit uses of pointers is essential to understanding the meanings of many constructs in these languages.

Not only are these implicit pointers dangerous, the potential for implicit aliasing tends to thwart formal specification and verification efforts. Consequently, languages that include them may be unsuitable for creating reusable parts. The problem essentially boils down to the following. If a variable is not mentioned in a statement, it is usually assumed that its value does not change during execution of that statement. However, if

uncontrolled aliasing is allowed, the semantic definition of every statement must take into account the possibility that any variable — even one not mentioned in the statement — may have its value changed as a side-effect of executing that statement. Incorporating this into the formal semantic definition of a language greatly increases the complexity of the definition, and therefore the complexity of reasoning about program behavior. For example, much of the complexity of the proof rules for Pascal found in [Hoare 73] and [Luckham 79] is a consequence of implicit aliasing.

In Pascal and Pascal-like languages, such as Modula-2 [Wirth 82] and Ada [DoD 83], pointers are also apparent since the programmer is allowed to explicitly declare pointer variables. Programmers in these languages not only have the power of machine addresses available to them, they also have the potential problems that come with the territory, such as unintended aliasing and dangling references.

The appropriateness of including pointers in programming languages has received considerable attention within the computer science community. C.A.R. Hoare, for example, feels "their introduction into high-level languages has been a step backward from which we may never recover" [Hoare 83]. The problem is not so much with formally specifying pointers, since their specification is very similar to the specification for arrays [Luckham 79]. Rather, the problem is the temptation to use pointers inappropriately, unnecessarily increasing the complexity of the specification and proof. For example, consider the following similar Pascal segments (which are provable using [Hoare 73] and [Luckham 79]):

```
var x,y : integer;
...
{assert true}
x := 3;
y := 5;
{assert x=3}
```

```
var x,y : ^integer;
...
{assert x≠nil and y≠nil and x≠y}
x^ := 3;
y^ := 5;
{assert x^=3}
```

Note that the segment using pointers is more complicated to specify and verify than the one without pointers because a precondition concerning the values of the pointers must be specified and proved. In fact, note that the precondition involves variables not even mentioned in the postcondition (i.e., y^). In general, the precondition involves all variables of pointers to types mentioned in the postcondition (e.g., ^integer in the above example).

Some language designers (e.g., [Meyer 88]) argue that pointers should be defined in programming languages because they are necessary to efficiently implement traditionally "linked" structures, such as lists. Our research, however, has shown this argument to be invalid — indeed, it is possible to formally define modules that safely encapsulate the notion of "pointer" for use in implementing linked structures. One of these modules is discussed in Section 3.6.2, and others can be found in [Pittel 90] and [Weide 86b]. If these modules are definable by and available to the programmer, there is no need to include pointers as a part of a programming language, which is the approach taken in RESOLVE.

2.5.2 Data Movement Primitives

An essential feature of any programming language is the ability to move data from one place to another. Modern languages provide only one mechanism to do this — copying the data. Examples of the (implicit) use of copying are the assignment statement and call-by-value and call-by-value-result parameter passing mechanisms.

For example, consider the assignment statement "x := y" where x and y are variables of the same type. The usual semantics of this statement is that, after it is executed, the values of variables x and y are the same, and furthermore, that this value is equal to the value of y immediately before the assignment statement. In implementation, this means making a copy of the value in y and placing it in variable x. Parameter passing by value and value-result imply copying in the same sense.

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

- Physically copying a large data structure is usually inefficient.
- Copying a pointer is not tantamount to copying the data structure it points to, and often produces programs that are incorrect and difficult to debug.

- "Copy thinking" leads to design of generic procedures having *no* efficient implementation.

As mentioned in Section 2.3.5, the time required to physically copy a data structure is usually linear in the size of the structure. Copying simple values, such as integers and characters, is relatively cheap, whereas copying large structures, such as records or linked structures, is potentially quite expensive, resulting in intolerable performance.

Most current languages attempt to overcome this inefficiency by copying pointers to data structures rather than physically copying the structures themselves. These pointers may appear in the language explicitly through a pointer type, or implicitly through call-by-reference parameter passing. As mentioned in Section 2.5.1, the introduction of pointers into a language causes potential problems with reasoning about programs and interferes with program verification.

In addition, pointers conflict with "hidden" types definable in many languages (e.g., private types in Ada and opaque types in Modula-2). For example, consider the assignment statement "x := y" where the representations of variables x and y are hidden. This statement either makes a copy of the data structure contained in y (if y is the actual data structure) or it makes a copy of a pointer to the data structure (if y is a pointer to the data structure). Unfortunately, the client programmer has no way of knowing which of these two semantically inequivalent actions is taken, since the representation is hidden[13].

There is at least one more problem with "copy thinking." Because copying is the only built-in data movement primitive defined in current programming languages, an entire generation of reusable software components has been designed in such a way that they cannot possibly be implemented efficiently. An example of such a component was presented in Figure 1 and discussed in Section 2.3.5.

Two key notions regarding parameter passing are apparent from the above discussion. First, passing parameters is a primary means of moving data in a program. Second, every parameter passing mechanism in existing languages (e.g., call-by-value, call-by-

---

13 Some (e.g., [Meyer 88]) would argue that this situation justifies allowing the client programmer access to the implementation. However, the problem appears to lie not with information hiding or hidden types, but with pointers.

value-result, call-by-reference[14]) either copies data structures or is defined in terms of pointers to them (with the potential for implicit aliasing). Either way, none of these parameter passing mechanisms successfully addresses the issues raised in this discussion.

An alternative to copying is swapping the values of two variables. Swapping as a data movement primitive successfully addresses all three problems inherent to copying — it can be implemented as a constant time operation (i.e., it is efficient for all types), it is defined without the need to mention pointers (and it does not have problems of implicit aliasing), and it suggests a "swapping-style" of generic procedure design that can be efficiently implemented. In RESOLVE, swapping is the only built-in data movement primitive. One implication of this is that all parameters are passed "by-swapping." A complete discussion of the advantages and consequences of this is presented in Section 3.3.1, and also in [Harms 88] and [Harms 89b].

2.5.3 Types and Variables

The concept of data type is incorporated in some form or another into every modern imperative programming language, and is usually coupled with the notion of variables. This section explores these two ideas — types and variables — and their relationship to programming language design.

We'll start by examining several common views of types and their role in programs and programming languages (Section 2.5.3.1 and 2.5.3.2). Sections 2.5.3.3 through 2.5.3.6 address the issues of type equivalence, type coercion, dynamic vs. static typing, and type initialization/finalization, respectively. This section concludes with a discussion of built-in data types, presented in Section 2.5.3.7.

2.5.3.1 Meaning of Types and Variables

There is very little concensus in the computer science community about the meaning of "data type" and "variable," although practically every programming language is defined in these terms. About the only thing all definitions of data type have in common is the

---

[14] Definitions of these and other parameter passing mechanisms can be found in most programming languages texts (e.g., [Pratt 84]).

notion that a type is, at least partially, a set of values. There is little agreement on what these values are, or on what else is encapsulated with this set.

In most languages designed prior to the mid-1970's (e.g., Pascal [Wirth 74]) a data type is simply a set of values a variable may assume. These languages provide a number of primitive built-in types (e.g., integers, characters, and booleans), and constructors for creating aggregate types out of simpler ones (e.g., arrays and records). All types are defined in terms of these primitive types or constructors. This scheme has the advantages of simplicity and intuition, and seems to follow nicely from the notion of types from mathematics.

However, it also has two significant drawbacks — it incorporates neither abstraction nor information hiding. Abstraction cannot be used in defining a type because all types are defined in terms of implementation-level building blocks. Information hiding is not achieved because a programmer knows about the implementation of every type and can write code that takes advantage of this information. For example, type stack must be defined in implementation terms such as record, array, and integer. The fact that there are operations such as push and pop has nothing to do with the definition of type stack. Worse yet, a programmer can write code that directly accesses the fields of the stack record.

A popular approach that addresses information hiding defines a data type (sometimes called an abstract data type) to be a set of values together with the set of allowable operations on those values, similar to the *class* concept in Simula-67 [Dahl 70]. Ada , CLU, and Eiffel are some of the languages that adhere to this definition, and in fact, this is the definition presented most often in the literature (e.g., [Wing 87], [Guttag 78]) and in recent texts on programming languages (e.g., [Pratt 84], [Horowitz 84]).

With this approach, it is possible to define a module (variously called a package, class, or cluster) that encapsulates a set of values and a set of operations on those values. The actual implementations of the set of values and the operations are hidden from a client of the module, so the only means of accessing the values are by invoking the associated operations. For example, accessing a stack must be done by invoking operations such as push and pop. The obvious advantage to this is that information hiding is enforced.

There are, however, several problems with this approach. First, associating operations directly with a type's definition does not match the intuitive notion of type[15]. Types are usually discussed within the context of data, which is intuitively a passive quantity that is acted upon by operations, which are active agents. Defining a data type (intuitively passive) as a set of operations (intuitively active) seems quite strange[16]. Even proponents of this approach seem confused about this. For example, the Ada reference manual talks about a "value of type integer" [DoD 83], which doesn't make any sense if type integer is considered to be a set of values together with a set of operations. On the other hand, this phrase makes a lot of sense if type integer is considered to simply be a set of values. Also, in many of these languages (e.g., Ada) an identifier defined as a "type" essentially represents only a set of values, and does not have any operations associated with it, which is not consistent with other uses of the term type.

A second problem with this approach deals with operation extensions. Taken literally (as is done in languages such as CLU), the only operations available for a type are those that define the type. For example, if type stack were defined with operations Push, Pop, and IsEmpty, it would not be possible to later define an operation Top that returns the top item from a stack without popping it. This is somewhat anti-convenient, especially since this particular operation can be implemented independently of any realization of stacks by invoking operations Pop and Push. Thus, the designer of a module must anticipate all possible operations that would ever be useful for that type, which is neither possible nor desirable. This limits the "reusefulness" of the type.

A third and final problem here is that the issue of abstraction is not addressed by this approach. As defined in Section 2.1, abstraction is a technique for making a clear, comprehensible presentation of the information provided by a software component, explained in terms of "higher level" concepts than those used to implement the

---

15 Some would argue that this intuition is wrong. However, a definition of types that is both intuitive and addresses the concerns raised in this discussion would be better than one that addresses the concerns but is unintuitive. RESOLVE's definition of types (discussed shortly) falls into the former category.

16 An interesting approach is defined in Russell [Donahue 85]. In Russell, a data type is a set of operations that provide meaning for untyped values from a universal value space. (An analogy is drawn to computer hardware, where the untyped universal value space is the set of all possible bit strings storable in a computer's memory. Each of these strings is inherently untyped. Typing is provided by instructions (i.e. operations) operating on the value.) In other words, a data type does not consist of a set of values, but only a set of operations.

component. A data type in this approach is obviously not defined in terms of implementation-level concepts, which is good. Unfortunately, it is not defined in terms of *any* concepts, either high-level or low-level! In fact, data types defined this way are no more abstract than comparable ones defined in earlier languages such as Pascal, and the term "abstract data type" is actually inappropriate in this context[17].

What is needed are mechanisms that hide information from clients (i.e., implementation detail) while at the same time provide necessary information to clients (i.e., abstract specifications). RESOLVE accomplishes these goals by separating types and variables into two groups — mathematical and program.

A mathematical type is the name given to a mathematical domain (i.e., a set of abstract values defined by a set of axioms). For example, mathematical type "integer" might be the name given to the domain defined by the axioms of mathematical number theory. Similarly, mathematical type "string of integer" might be the name given to the domain defined by mathematical string theory, which is a generic theory in the sense that it is parameterized by another mathematical type. Note that a type is not a set of values, but rather the name of a set of values. The distinction is subtle, yet is important for future discussions.

A mathematical variable is a symbol that stands for some value from the domain of the variable's type. For example, if "x" is a mathematical variable of mathematical type integer, then x stands for some value from the domain of integer.

---

17 Some might consider the operations of the type definition as the "axioms" defining the type, and argue that this is therefore an "abstract" definition. This might be true, provided the operations were formally specified, which is not done in the examples and discussion presented in the literature. Also, formally defining a type this way suffers from the same problems inherent to algebraic specifications, discussed in Section 2.4.2.2.

Mathematical types are defined and discussed completely in terms of mathematics with no mention of programs. The values which form their domains are entirely abstract. We say things about such values by writing assertions in some formal language, such as predicate calculus. For instance:

```
∀ x : integer, x + 1 > x
```

says that the integer obtained by adding one to any integer is larger than the original value.

A program type is a name given to a program domain, which is also a set of values[18]. Each element in a program domain has a concrete representation (which may be as low-level as a configuration of bits in memory) and is modeled by an element from a corresponding mathematical domain. In other words, there is a total (mathematical) function from a program domain to a corresponding mathematical domain. The values in a program domain are defined by an associated set of operations, which is discussed in Section 3.4.

As with mathematical variables, a program variable is a symbol that stands for some value from the domain of the variable's type. A program variable is declared with the notation "x : T" to say x is a variable of type T, and consequently x stands for some value from the domain of T. However, because every value in a program type's domain has a mathematical model, it is possible to *reason* about the value of a program variable as a mathematical value, rather than as a concrete representation. In other words, the value of a program variable in a client program is considered to be the abstract value of the mathematical model of that variable's concrete representation.

For example, program type "int" might be modeled as a mathematical integer, and a client of int reasons about ints as if they were mathematical integers. Similarly, program type "stack" might be modeled as a mathematical string, and a client of stack reasons about stacks as if they were mathematical strings[19]. If a client declares "i" as a variable

---

18 Actually, a RESOLVE program type is the name of a *marker*, which in turn maps to a program domain. The reasons for introducing markers is introduced in Section 3.4. However, for the current discussion, it is sufficient to consider a program type to be the name of a program domain.

19 Strings are used only to reason about stacks. Program stacks probably would not be implemented as strings.

of type int, i should be treated as a symbol that stands for a mathematical integer. For reasoning about the program, the value of i must not be thought of as a sequence of 32 bits (or whatever the concrete representation of an int), and it certainly must not be thought to be the address that contains the concrete representation. Defining program types in terms of mathematical models allows RESOLVE's types to be truly "abstract data types."

2.5.3.2 Role of Types

Now that we know what types are, let's see how they are used in a program. Types play two potential roles in a programming language — syntactic type checking, and overloaded operator resolution. Syntactic type checking validates the legality of an operation invocation. In modern programming languages all parameters to operations have associated types. When an operation is invoked, the types of actual parameters are checked against the types of formal parameters. If they are not equivalent (discussed in Section 2.5.3.3), an error is generated (or in some languages the type is automatically "coerced" to another type, as discussed in Section 2.5.3.4). It is not necessary for the language to define the semantics of an operation invocation where the actual parameters are not of the correct types — this is simply illegal, and therefore has no semantic meaning. This greatly simplifies the semantic definition of a language.

Note that nothing is mentioned about when the checking occurs. If it is done by the compiler, the language is said to be statically-typed. If it is not done until execution the language is said to be dynamically-typed. This is discussed in more detail in Section 2.5.3.5.

The second role of types, overloaded operation resolution, is used to select the appropriate operation from a set of possible ones. Each operation defined in modern programming languages has an associated name and type signature, which consists of the number and types of all parameters. In many languages it is possible to declare several operations with the same name, as long as they all have unique type signatures. This is called overloading. When one of these operations is invoked, the compiler (in the case of static typing) or run-time system (in the case of dynamic typing) compares the signature of the invocation with the signatures of all definitions. If one matches, that operation is invoked, otherwise an error is flagged. For example, "+" is overloaded in

many languages, standing for both integer addition and floating point addition. Overloading is a convenient way of naming related groups of operations, yet it can easily lead to complexity both for humans trying to understand a program and compilers parsing a program [MacLennan 83, Aho 86].

In RESOLVE, the only use of types (both mathematical and program) is so the structure of program statements can be checked against expected usage at compile time. Types are not used to resolve overloaded operations for the sole reason that operations cannot be overloaded!

2.5.3.3 Type Equivalence

The previous discussion indicated that a compiler or run-time system must be able to determine whether two types are equivalent, for example when checking the type of an actual parameter against the type of the formal parameter. There are two common definitions of type equivalence, usually called name equivalence and structural equivalence. With name equivalence, two types are equivalent if and only if their names are the same. With structural equivalence, two types are equivalent if their structures are the same.

For example, consider the following type declarations in a Pascal-like language:

```
type T1 = array [1..10] of integer;
type T2 = array [1..10] of integer;
type year = integer;
type temperature = integer;
```

With name equivalence, all four of these types are inequivalent, whereas with structural equivalence T1 is equivalent to T2, and year is equivalent to temperature[20].

Both approaches have advantages and disadvantages. Name equivalence is simple to understand and implement, and provides extra protection against type errors. However, it places reliance on globally-declared type identifiers, and can be a nuisance to programmers since they often have to make up extra type names [Welsh 77]. Structural equivalence is more flexible and at first glance more intuitive. However, unambiguous definitions of it are complex and difficult to understand and implement [Welsh 77].

---

20 Good discussions of type equivalence are presented in [Welsh 77], [Pratt 84], and [Horowitz 84].

Also, with structural equivalence two types that are not logically equivalent (e.g., year and temperature in the above example) but happen to be declared identically are considered equivalent, and consequently some programming errors (e.g., copying a variable of type year into a temperature variable) are not detected.

Most languages use name equivalence (or some variation of it) for reasons of safety, understandability, and ease of implementation [MacLennan 83].

As discussed in Section 3.4, types in RESOLVE are formally defined in terms of sets and mathematical functions on these sets, and type equivalence is defined in terms of mathematical functions mapping to the same element. Essentially, though, program type equivalence is name equivalence, and mathematical type equivalence is structural equivalence.

2.5.3.4 Type Coercion

In some languages it is possible to violate the normal type equivalence rules and consider a value of one type to be a value of an inequivalent type. This ability is called type coercion. There are two common ways for types to be coerced — implicit conversion, and representation interpretation. With implicit conversion, an operation is implicitly invoked that converts the value from one type to another. A common example of this is the implicit conversion of numeric data from one representation to another (e.g., "integer" to "real"), permitting a programmer to write "mixed expressions." Although this might reduce the number of keystrokes required to construct a program, there are several inherent dangers. First, mistakes made by the programmer may go unreported, or may be reported at some other location. For example, consider a hypothetical language where booleans are implicitly coerced to integers if necessary. Suppose a programmer needs to place the sum of integers j and k into integer i, but instead of entering the statement "`i := j + k`" he or she accidentally enters the statement "`i := j = k`" (an easy mistake on keyboards where "+" is a shifted "="). In the hypothetical language, this error will not be detected, because the language will implicitly coerce the boolean expression "`j = k`" to an integer.

A second problem with implicit coercion is that the programmer may not be aware of costly conversions that are taking place as a result of coercion, making programs very

inefficient. COBOL [COBOL 74] is a language which supports many forms of implicit coercion and encourages their use. For instance, consider the statement "ADD A TO B" where A is an integer represented as a BCD character string and B is an integer represented as a 2's-complement binary value. This statement implicitly converts A to a 2's-complement integer before performing the addition. The conversion is much more costly than the addition. There may be no way around this conversion. However, it may be possible for the programmer to reduce the number of necessary conversions, but he or she may not realize that conversions are taking place because they are implicit.

The other method of type coercion — representation interpretation — allows the representation of a value (i.e., the bit sequence) to be interpreted as a value of another type. For example, the statement "`i = (int)j + (int)k`" in C [Kernighan 78] or C++ [Stroustrup 86] causes the representation of variables j and k to be treated as if they were ints for this statement, regardless of the types of j and k. Conversion does not take place, and thus there is no inherent inefficiency. However, this relies on the programmer having complete knowledge of the representations of the types being coerced, which is a blatant violation of information hiding. If, for example, the representation of a type is changed, coercing the value may be completely bogus.

Languages that do not include coercion mechanisms are often said to be strongly-typed languages because the only values allowed in a particular context are those whose type is equivalent to what is expected.

From the standpoint of reusability, coercion is bad. The first form encourages programs that are inefficient, and the second form violates information hiding and complicates formal specification and verification. For these reasons, RESOLVE does not include any form of coercion.

2.5.3.5 Dynamic vs. Static Typing

In some languages (e.g., APL [Iverson 62], SNOBOL4 [Griswold 71], and Smalltalk-80 [Goldberg 83]) the types of variables mentioned in a statement are not known until that statement is actually executed. For example, the types of variables x and y in the expression "x + y" may change every time the expression is evaluated. Under these circumstances the responsibility for type-checking is forced on to the run-time

environment because the types of variables cannot be determined by the compiler. Languages exhibiting these characteristics are said to be dynamically-typed languages, and variables are often said to be typeless since they have no fixed type.

The advantages of dynamic typing are that the programmer is freed from the nuisance of declaring variables, and there is much flexibility in that the data object associated with a variable name may change as needed during program execution. There are also several disadvantages. First, dynamic typing is expensive in terms of space and time, since it requires overhead to store type information at run-time (i.e., "tags"), and requires code to check the types of parameters prior to every operation invocation. Second, formal specification and verification is difficult because nothing is known about the value of any variable prior to execution. Program debugging is difficult because many errors are not detectable until run-time. Also, it is quite likely that some statements were not adequately tested, and type errors associated with them are not discovered until much later when the program is in production.

In statically-typed languages (e.g., Pascal and its descendants), the programmer meticulously associates type information with every variable and operation parameter. The compiler checks each actual parameter to an operation against the expected type, and flags an error if the types are inequivalent. There are several advantages to this scheme. First, many programming errors are detectable at compile-time, which is preferable to waiting until run-time [Hoare 83]. Second, there is no need to keep type information around at run-time and no need to dynamically check the types of parameters. Thus, statically-typed language have no run-time inefficiencies associated with type-checking. Third, it is possible to formally specify and verify programs written in statically-typed languages.

Dynamically-typed languages are not conducive to designing reusable parts, because programs in them are inefficient and very difficult to formally specify and verify. RESOLVE is therefore a statically-typed language.

2.5.3.6 Type Initialization and Finalization

In many programming languages, a variable's value is unknown (or worse, possibly unreferenceable) when the block of code containing its declaration begins execution.

These so-called uninitialized variables cause headaches for both programmers and verifiers. Errors resulting from referencing a variable before a known value is placed in it are often difficult to find, especially when the behavior of the program is unpredictable as a result of the initial "garbage bits" in the variable. Particularly insidious errors are likely to occur if the variable's representation involves pointers.

The possibility of uninitialized values also significantly increases the complexity of the formal semantic definition of a language, thus working against formal specification and verification. For example, one useful component of many formal systems is the specification of an invariant for a type, which is an assertion about the value of a variable of that type that is true at all times. An initialization assertion becomes the base case for an inductive proof that the invariant holds at all times.

The problem is not with the notion of unknown values. In fact, saying that the value in a variable is unknown is a perfectly reasonable thing to say, as long as the variable is known to contain *some* legitimate value from the domain of its type. (Of course, the unknown value must satisfy the invariant, if any, and the program's behavior should not depend upon this unknown value.) Rather, the problem crops up when the bits in a variable may not represent *any* value from the domain of the its type (e.g., before memory is allocated for a representation that involves pointers).

Most languages require the programmer to explicitly initialize all variables. Other languages (e.g., C++ and Euclid) allow types to have associated initialization routines that are automatically invoked when a variable of that type is declared. Although this latter approach is a step in the right direction, it does not prevent problems resulting from uninitialized variables because initialization is "voluntary" rather than required of all types. Indeed, what is needed to solve these problems is to have initial values specified for all types, and initialization routines that are automatically invoked when variables come into existence. This approach lends itself to formal specification and verification (and hence to design and implementation of more reusable components) because every variable is guaranteed to have a value meeting the initial specification for its type before any statement is executed.

Some might argue that requiring every variable to be initialized is too costly, and thus not conducive to software reuse. There are three reasons this is not generally true.

First, a type's initial value is chosen by the part's designer, and should be one that is easy to construct (e.g., an empty stack). In fact, experience has shown that it is possible to define initial values that require a constant amount of time to construct for a wide variety of reusable components [Harms 89a]. Second, it is possible to implement "lazy initialization" where a variable is not actually initialized until it is referenced for the first time, which is discussed in Section 3.8.2. Finally, of course, even if initialization is not automatic it is nonetheless mandatory for most complex types because "garbage bits" cannot always be interpreted as a legitimate value of the type in question. Manual initialization saves nothing in these cases.

Variables whose representations involve dynamically-allocated memory should have that memory released when it is no longer needed. One way to reclaim memory is to rely on automatic garbage collection by the run-time system. Another way is to explicitly finalize every variable at the end of the block in which it is declared. Garbage collection has the advantage that it is transparent to the programmer. However, it has the disadvantage that the user may notice significant performance degradation at seemingly random (and possibly inopportune) times while the garbage collector examines and/or compacts and/or defragments memory. Using garbage collection, it is not at all easy to characterize the performance of operations because the garbage collector may execute almost any time. The problem is that the garbage collector is part of the system, and has no knowledge about the specific application or memory utilization patterns for specific reusable parts. Writing a general garbage collector that does not occasionally cause the computer to "go out to lunch" is not possible.

Another approach is to define a finalization routine for each type where dynamic memory must be reclaimed, and which is automatically invoked. It would seem that the performance of finalization routines would not be very good, since they must be able to finalize whatever structure is left in a variable (e.g., empty stacks as well as stacks containing one million items). Surprisingly, our research has shown that it is possible to design representations such that initialization and finalization require a constant amount of time, even for complex types such as arrays and lists [Harms 89a]. This is due in part to the fact that the finalization routine has knowledge about how the various parts of the representation interact, something a garbage collector would have no way of knowing.

So, a language supporting the design of reusable software should require every type to have an initial value and a corresponding initialization routine. In addition, it should allow an implementer to write a finalization routine if dynamically allocated memory is to be released. Both of these routines should be automatically invoked. Initialization is necessary largely to support formal specification and verification, and finalization is required for efficiency and performance characterization. As explained in Chapter 3, this is the approach taken in RESOLVE.

2.5.3.7 Built-In Types

Almost every programming language has a number of types and type constructors defined in the language (e.g., integer, character, boolean, array, and record). There are several reasons why these types are built-in. First, simple types and constructors have been defined in languages since the early days of FORTRAN, lending historical credibility to the notion of built-in types. It is interesting to note that in languages such as FORTRAN, these simple types were the *only* types available to the programmer, and these languages were consistent in the sense that *all* types were built-in. This of course is not true of modern languages where programmers can define their own types.

A second rationale for defining built-in types is that it makes the task of constructing a program somewhat easier. For instance, many programs need these types, so why force the programmer to explicitly declare them? Also, accessing built-in values is often more convenient than accessing user-defined structures because languages provide special syntax for built-in types, such as infix notation (e.g., x + y) and array subscripting (e.g., a[i]).

Third, there is the perception that it is more efficient to implement simple types within the language, as opposed to letting the programmer implement them. The confusion here is between specifying a type and implementing it. Obviously some types (e.g., integer in the standard representations) have very efficient implementations that cannot be expressed in the high-level language. However, this does not justify building the specification of the type into the language.

Finally, most languages have control structures, such as if and while statements, whose execution depends on some value of a particular type (e.g., integer or boolean), which

therefore must be defined in the language. It is possible, however, to define all control structures in such as way that they do not rely directly on data values. This is the approach taken in RESOLVE, discussed in Section 3.3.2.

The conclusion from this discussion is that there are no overwhelming technical reasons for including built-in types in a language. But what harm can they do — specifically, do they discourage reusable software? The answer is that built-in types do have several disadvantages, and they do actually discourage reusable software design. For instance, built-in types are usually treated differently than other types, as discussed previously (i.e., the programmer doesn't declare them, and special syntax is used to access their values). In fact, built-in types in some languages are conceptually very different from other types. For example, all types in Eiffel are objects in the typical objected-oriented sense *except* the built-in types which are considered as types in traditional nonobject-oriented languages. The inconsistency between built-in types and user-defined types increases the complexity for a programmer learning the language and for its formal semantics. A better alternative is to have all types uniformly declared and referenced via identical mechanisms.

A second disadvantage of built-in types is that multiple implementations are not available, and the programmer is required to use whatever representation is defined in the language implementation. Specifically, the programmer cannot choose the most appropriate implementation from an implementation library[21], which violates one of the characteristics of reusable parts — multiple implementations — discussed in Section 2.3.4.

A third disadvantage is that built-in types must be explicitly incorporated into the proof rules and formal semantic description of the language, thus complicating the formal language description. An approach that simplifies the language description uses the same language mechanism to define *all* types.

---

21 Some languages (e.g., C, C++, and Ada) define several flavors of integers (e.g., short, int, and long). The differences between these are not defined or characterized in the language, but are instead defined in each language implementation. Also, their distinctions are usually described in implementation-level terms such as "bits" and "2's-complement," which violates the principle of information hiding. Thus, this approach does not adequately address the concerns raised in the current discussion.

RESOLVE has no built-in types, and is similar to Alphard [Shaw 81] in this respect. The specific implications of this are discussed in Chapter 3.

2.6 Programming Language Survey

In Section 2.3 well-defined reusable parts were characterized, along with a discussion of how these characteristics affect programming language design, and brief evaluation of features of existing languages with respect to these characteristics. The conclusion was that no modern language has all of the necessary constructs that encourage and facilitate the design and implementation of reusable software components.

Despite this conclusion, it is worthwhile examining several important programming languages to see exactly why they are not as appropriate as they could be for development of reusable software. Sections 2.6.1 through 2.6.4 examine several Pascal-like languages, namely Ada and Anna, Modula-2, Euclid and Gypsy, and Alphard. In the following three sections several "object-oriented" languages are examined — C++, Eiffel, and Larch/CLU. The last section examines Z, which is an implementation language-independent specification language.

2.6.1 Ada and Anna

Ada [DoD 83] is the quintessential product of "design by committee." The U.S. Department of Defense developed the specification for the language in several stages (called Strawman, Woodenman, Tinman, Ironman, and finally Steelman) over several years. The actual language was selected from a group of sixteen submitted in an international competition. Reviews were held at each stage, allowing many people the opportunity to influence the design and selection of the language.

As a result, Ada is a "jack of all trades." It is a general-purpose language based on Pascal, it is a real-time language for such applications as guided missiles, and it is a systems programming language. It has facilities to model parallel tasks and handle exceptions, and allows access to system dependent parameters and precise control over the representation of data. In addition it encourages programs to be designed using several software engineering principles such as modularization and information hiding.

Unfortunately, Ada does not include the constructs to formally specify the intended behavior of a program. Anna [Luckham 87] is a language extension to Ada that partially addresses this void. It is designed to "meet a perceived need to augment Ada with precise machine-processable annotation so that well established formal methods of specification and documentation can be applied to Ada programs" [Luckham 87]. Annotations can supposedly be used by a verifier to formally verify a program, and many of them can also be used to generate run-time consistency checks. The extensions to Ada are in the form of formal comments (sometimes called virtual Ada text), so Ada is a proper subset of Anna in the sense that any valid Anna program is also a syntactically-correct Ada program[22].

The basic encapsulation mechanism in Ada (Anna) is the *package* in which types, constants, and operations are defined for access by a client. Packages may be generic in the sense that they are parameterized (see Section 2.3.3). There is no notion of an inheritance hierarchy, so package design generally follows the abstract data type approach rather than the abstract data object approach (see Sections 2.4.1.1 and 2.4.1.2).

A package "specification" in Ada is no more than operation signatures. However, Anna packages can be formally specified by defining axioms and other assertions, thus encouraging formal specification using the algebraic approach (see Section 2.4.2.2). The specification of a package can be separated from its implementation (called the *package body*), allowing the implementation of routines and the representation of types (called *private* types) to be hidden from clients. However, to simplify compilation of clients, the representation of a private type must be defined within the package specification, even though a client cannot reference this information. Thus, all clients must be recompiled every time the representation of a private type changes. Also, Ada is designed to have one package body per package, so multiple implementations of a package are not encouraged, though it is possible through some sequence of awkward contortion in an Ada programming environment. Even in this case, though, a client program must have one package body for every instance of that package, so only part of the flexibility of multiple implementations can be achieved.

---

22 Because of this relationship, any reference to Ada in this discussion applies to Anna as well.

As discussed in Section 2.5.2, the meaning of assignment and equality operations on private types depend upon whether a type uses pointers in its representation. Ada's solution to this ambiguity involves the notion of a *limited private* type on which the use of the predefined operators for assignment and equality testing is not permitted. In other words, a client cannot assign a value to a variable whose type is limited private.

The built-in operators available on private types are assignment and equality check. These operators are not available on limited private types, providing a solution to the problem of interference between hidden types and pointers discussed in Section 2.5.2.

With these features, Anna appears to be a language encouraging the design and implementation of reusable parts — a part can be formally specified, the specification can be separated from its implementation, and generic parts can be defined. Unfortunately, the keyword here is "appears." In fact, Anna has some serious flaws making it unsuitable for designing reusable parts. For example, it is not at all easy to define multiple implementations of a specification, and many problems of efficiency discussed in Sections 2.3.5 and 2.5.2 are not adequately addressed.

However, the biggest problems inherent to Anna are correctness and specification problems. For example, because formal assertions can be incorporated into an Anna program, they appear to be formally specified and verifiable. However, the definition of Anna does not include the proof rules, axioms, and semantic definitions that would be necessary to formally verify Anna programs. The authors allude to this omission, claiming that these would be similar to those developed for Pascal (i.e., [Hoare 73] and [Luckham 79]), and therefore almost trivial to develop. However, Ada is significantly more complex than Pascal, with many of the same features that complicate Pascal's proof system (e.g., implicit aliasing). Its proof system would not be a trivial extension to Pascal's, and from all indications has not been developed. It appears that Anna is actually oriented toward run-time checking rather than formal verification, despite early claims to the contrary.

Another problem with Anna is that many potential errors are not easily checkable. For example, the order in which actual parameters are evaluated is not defined, and it is an error for a program to take advantage of any particular evaluation order. However, this

error is not detectable by the compiler, nor is it possible to write Anna assertions that detect it.

Finally, there are no restrictions placed on actual parameters, causing a problem when the same identifier appears more than once in an argument list. For example, consider the following Ada procedure:

```
procedure pathology (x,y : in out INTEGER) is
begin
  x := x + 1;
  y := y + 2;
end pathology;
```

If this procedure is invoked with "`pathology(b,b)`" the value in b upon return has either been incremented by 1, 2, or 3, depending upon the order of parameter evaluation and whether **in out** is implemented as value/result or reference. This is clearly not a good idea[23].

To summarize, Anna supports and encourages the development of formally specified generic packages. Facilities are provided to separate the implementation of a part from its specification. However, Anna does not easily support multiple implementations for a specification. Also, Anna includes features such as implicit aliasing which make it extremely difficult to develop a proof system. In fact, there are no proof systems for Anna, and until one is developed it is not possible to formally verify Anna programs. Also, Anna does not provide alternatives to the problems inherent to copying. The conclusion is that Anna does not provide all of the features necessary to support and encourage the design and implementation of reusable software components.

2.6.2 Modula-2

Modula-2 [Wirth 82] is a programming language that extends Pascal with concepts for modular program organization, multiprogramming, and access to low-level machine facilities. It is intended to be a general-purpose language encouraging structured programming and permitting the development of "system" programs (e.g., to control hardware devices such as printers).

---

23 It is interesting to note that using the same variable more than once in an actual argument list was prohibited in the preliminary definition of Ada ([Ada 79] and [Ichbiah 79]), yet is not prohibited in the final definition.

Modula-2 programs are organized into units called *modules* that encapsulate data type definitions and operations with parameters of those types. This suggests the development of modules designed around the abstract data type approach (see Section 2.4.1). Modules control which identifiers are available to clients using explicit export lists, and clients indicate which identifiers they need using explicit import lists. Modules cannot be parameterized, so it is not possible to define generic modules.

It is possible to divide a module declaration into two parts — a *definition module* containing the signatures of exported types and operations, and an *implementation module* containing data structures and code implementing those types and operations. Each definition module has exactly one implementation module, so it is not possible to have multiple implementations of a definition. A client program can access only those items defined in the definition module. Similarly, a client programmer should not need access to anything in the implementation module. Thus information hiding is enforced for both client programs and client programmers.

A type defined in a definition module (i.e., exported) may be either transparent or opaque. A transparent type is completely defined in the definition module and clients can directly access its representation. On the other hand, the representation of an opaque type is defined only in the implementation module, and clients cannot access its representation. To simplify compilation of clients, the representations of all opaque types must occupy a fixed amount of storage — specifically, the amount of memory necessary to store a machine address (i.e., a pointer). This implies that most opaque types are represented as pointers to representation structures. Unfortunately, a client has no way of knowing this for sure, leading to the ambiguities discussed in Section 2.5.2.

It is important to note that a definition module does *not* contain a formal specification of the module, but merely signatures of operations and (hopefully) informal descriptions within comments. In fact, formal specification was not a design goal of Modula-2, and no facilities are provided for writing specifications.

In summary, Modula-2 provides the mechanisms to separate (informal) specification from implementation, which is a significant improvement over Pascal. However, it does not address the issues of formal specifications and formal verification, it does not permit the declaration of generic modules, it does not permit multiple implementations of

a specification, and it does not provide any alternatives to the problems inherent to copying (discussed in Sections 2.3.5 and 2.5.2). Under the criteria for reusable parts developed in Section 2.3, the conclusion is that Modula-2 does not support the design and implementation of reusable software parts.

2.6.3 Euclid

Euclid [Lampson 77, Lampson 81] is a programming language "evolved from Pascal by a series of changes intended to make it more suitable for verification and for systems programming" [Popek 77]. These goals unfortunately conflict with each other, and the language is therefore a compromise between them.

The encapsulation mechanism is the *module*, which is similar to a record (i.e., it is defined as a "type"), with the possibility of type and operation fields. Modules (actually, any type declaration) may have parameters, thus allowing the definition of generic parts. The identifiers (i.e., fields) that are available to a client are explicitly listed in an export list, allowing the compiler to enforce information hiding from the client program. If language-defined assignment and/or equality checking are to be allowed on variables of an exported type, these operators must be explicitly exported with the type.

Initialization and finalization routines can be defined for a module, which are automatically invoked for every variable of the *module* type. Unfortunately, it is not possible to define initialization or finalization routines for types provided by the module. Thus, module initialization/finalization is supported, but not data type initialization/ finalization.

A module is formally specified using assertions that specify invariants and pre- and post-conditions[24]. There are neither mechanisms for defining axioms for a type, nor for modeling a type as a complex mathematical type (e.g., string). Rather, types are defined in terms of mathematical models of the built-in types (i.e., Integer[25], Boolean, and Char) and type constructors (e.g., record, array, and pointer). In other words, types are

---

24 Although formal specification and verification are primary goals of the language, none of the sample programs in [Lampson 77] or [Lampson 81] are formally specified!

25 Type Integer is strictly a mathematical type, and it is not possible to declare variables of type Integer. All "integer" variables must be declared as a subrange of Integer. Two Integer subranges — SignedInt and UnsignedInt — are predefined for convenience.

defined in terms of implementation structures, and therefore should not be considered "abstract" data types.

Formal specification of the items defined in a module are placed with the definition of each item, so there is no attempt to separate specification from implementation (i.e., the module is not divided into separate "specification" and "implementation" parts). The implications of this are that 1) it is not possible to have multiple implementations of a specification, and 2) it is not possible to enforce information hiding from a client programmer.

Both explicit and implicit pointers are defined in Euclid. The formal specification of explicit pointers (and explicit aliasing) is similar to the proof system presented in [Hoare 73] and [Luckham 79]. To make verification somewhat easier, Euclid introduces the notion of a collection. Each collection has an associated type, and pointers are declared to be elements of a collection rather than pointers to a type. A pointer from one collection cannot be assigned to a pointer from another one, even if the collection types are the same. Thus, pointers from different collections cannot alias the same location, and this knowledge can simplify verification. For example, consider the following code segment:

```
...
{assert ...}
x^ := 3;
y^ := 5;
{assert x^=3 and y^=5}
```

If x and y are pointers from the same collection, they may alias the same location. Therefore the condition "$x \neq y$" must be proved for the code to be valid. On the other hand, if x and y are from different collections, they cannot alias each other, so nothing additional has to be proved. Thus, collections allow a programmer to partition pointer variables to indicate some of the knowledge about how they will be used.

A more significant simplification for the verifier is that implicit aliasing cannot occur. This is enforced by not allowing a location to be passed as an actual call-by-reference parameter if it could be accessed by another name within the routine. For example, the invocations "`proc1(a,a)`" and "`proc2(b,b[2])`" are illegal if both parameters are call-by-reference. The invocation "`proc1(b[i],b[j])`" requires the assertion "$i \neq j$" to be proved by the verifier. Also, the invocation "`proc1(a,c)`" is not allowed if either a or c

is directly accessed within the routine. This last example demonstrates that the compiler must have access to the implementation of all routines, because it must determine which non-local variables each routine references.

In spite of this, Euclid includes some strange constructs that come tantalizing close to aliasing. For example, it is possible to declare several variables as explicit aliases for one location, similar to FORTRAN's EQUIVALENCE statement. Also, it is possible to explicitly indicate the fixed memory address for a variable, which could conceivably be anywhere in memory. This latter construct makes it impossible to guarantee that implicit aliasing cannot occur, and in fact makes it impossible to develop a verification system that is independent of a language implementation.

Also, Euclid defines type equivalence to be essentially "structural" equivalence, which significantly complicates the typing system.

In summary, Euclid is an extension to Pascal that has formal specification and verification as primary goals. Formally specified types and operations can be encapsulated into modules, which may be generic. However, specification is not separated from implementation, and multiple implementations for a specification are not supported. Also, there is no attempt to provide alternatives to the problems inherent to copying. Therefore Euclid does not provide the features necessary to encourage the design and implementation of reusable software parts.

2.6.4 Gypsy

Gypsy [Ambler 77] is a Pascal derivative whose design was driven by the development of a comprehensive methodology for constructing verified programs oriented toward communications processing. The goals set by the language designers were ambitious — formal specification and verification of programs, constructs to enable development of systems programs (e.g., concurrency and synchronization structures), and the ability to support execution in imperfect execution environments (e.g., exception handlers).

Gypsy does not provide any mechanism for defining generic types nor for encapsulating types and operations into a syntactic unit. Instead of encapsulation, an access list can be associated with a type, indicating the set of operations that have access to its internal representation. For example, one could define a type stack which permits only

operations Push, Pop, and IsEmpty access to its internal representation. This permits information hiding from client code to be enforced, but does not permit information hiding from a client programmer. The claim is that this scheme permits the development of abstract data types. However, with all programs coded as monolithic compilation units, this approach surely does not *encourage* the development of abstract data types.

Types and operations are formally specified by assertions such as pre- and post-conditions and invariants. These assertions are written in terms of program structures (e.g., integers, characters, arrays, and records) as well as a handful of mathematical types (e.g., sequence). The specification of an item is placed with its implementation, so a specification is not separated from its implementation. Another implication of this is that it is not possible to define multiple implementations of a specification.

Gypsy has several features that make verification easier. For instance, explicit pointers are not defined in the language, procedures cannot access non-local data, and functions cannot produce side-effects. However, it is not clear whether implicit aliasing (resulting from call-by-reference parameter passing) is possible.

In summary, Gypsy includes constructs for formal specification and verification. However, it does not have mechanisms for declaring generic components, for separating specification from implementation, for declaring multiple implementations for a specification, nor does it address the problems inherent to copying. Also, there is no encapsulation construct, so parts as separately-compilable units cannot be developed. The conclusion is that Gypsy cannot be used to design and implement reusable software parts.

2.6.5 Alphard

Alphard [Shaw 81] is a programming language designed to support data abstraction and program verification, and evolved between 1974 and 1980 by a research group at Carnegie Mellon University. Development of Alphard, at least as a focused research project, stopped around 1980. However, many ideas developed in Alphard are germane to the current discussion of reusable parts, and worth examining in some detail.

The encapsulation mechanism in Alphard is the *form*, which defines a data type along with operations having parameters of that type. A form may be generic in the sense that

it may have other forms (i.e., types) as parameters. A data type characterizes the possible values a variable may have, and determines the contexts where variables can be used (i.e., type checking of actual against formal parameters). Because types are not considered to be values along with operations, module designs following the abstract data type approach are suggested (see Section 2.4.1).

The meaning of a form identifier is context-sensitive. When used in a variable declaration it stands for only the data type defined in the form. When used as an actual parameter to an instantiation of a generic form it stands for the encapsulation of its type and operations. For example, "`var x:integer`" declares x to be a variable of the type defined by form integer. (Note that variables are considered to be values, *not* encapsulations of values with operations.) On the other hand, "`var s:stack(integer)`" instantiates generic form stack, passing it form integer as a parameter. In this example, all operations defined by form integer are passed to the instantiation of form stack. Overloading the meaning of a form identifier is unfortunate because it confuses the distinction between an encapsulation and a data type.

Language mechanisms are provided to formally specify types and operations. Types are specified using mathematical models, and operations are specified with pre- and post-conditions written in terms of functions and relations defined in mathematical theories. For example, type stack could be modeled as a mathematical string, and the effect of operation push would be specified as assertions using functions defined in string theory. This is the model-based approach to formal specification, discussed in Section 2.4.2.3. Alphard does not, however, provide mechanisms to formally define mathematical theories, and a verifier would have to obtain these definitions from someplace other than an Alphard program.

Every type has an initially assertion associated with it that is guaranteed to be met before a variable of that type is accessed for the first time — in other words, every variable has an initial value. There is an initialization routine for each type that is invoked for every variable declared of that type, and a corresponding finalization routine that is invoked when variables are destroyed. The advantages of this with respect to reusability are discussed in Section 2.5.3.6.

In addition to formal specification of the type and operations, a form contains declarations of the representation of the data type as well as implementations of all operations (including initialization and finalization). A client program can only access the identifiers defined in the specification portion of the form, and therefore does not have access to the representation of the data. However, a client programmer has access to the entire form. Thus, Alphard supports information hiding from the client program but not the client programmer.

This organization has other consequences. For example, it is not possible to define multiple implementations of a specification because specification and implementation are coupled into one compilation unit. Also, it is not possible to parameterize specification separately from implementation. Thus, for example, the parameter list of a form defining an associative memory structure may mention things like "hash function" which have nothing to do with understanding the specification. Thus, placing specification and implementation into the same module can lead to confusion, and parts developed would not be as reusable as they might be if multiple implementations were permitted.

Another interesting characteristic of Alphard is that only two forms are built-in to the language — boolean and rawstorage. Boolean is built-in because of its relationship with some control structures (e.g., the conditional statement), and rawstorage because it is needed to bootstrap all other forms. There is, however, a "standard prelude" of forms that is automatically appended to the beginning of every compilation, which includes forms such as integer and vector. Some advantages to this approach are discussed in Section 2.5.3.7.

Despite the emphasis on formal specification, there are a number of features of Alphard that work against verification. For instance, restricted use of aliasing is part of the language definition. An example of restricted aliasing is that actual parameters cannot alias a location (see the example in Section 2.6.1), except if the formal parameter is specified with the qualifier 'alias.' It was felt that placing appropriate restrictions on aliasing was safe enough to warrant its inclusion, and that the impact on the proof system would be minimal. Unfortunately, this was not the case, and the inclusion of

aliasing contributed to the inability to develop a formal proof system for Alphard [Shaw 81].[26]

Another feature that complicates the language (and development of a proof system) is the horrendously complex type matching scheme. This complexity is partly the result of the capability to place restrictions on forms that are passed as actual parameters to instantiations of generic forms. For example, it is possible to require the actual form to have an assignment operation defined, or for that matter *any* set of operations meeting arbitrary specifications.

Finally, note that Alphard makes no attempt to provide alternatives to copying large structures, except the usual reliance on pointers.

In conclusion, Alphard was designed to encourage and promote the development of formally specified and verifiable modules defining abstract data types. Unfortunately, the language became so cumbersome and complex that this goal was not achieved, although significant progress was made toward it.

2.6.6 C++

C++ [Stroustrup 86] is an extension to C [Kernighan 78] that incorporates encapsulation of data with operations, and other constructs often associated with "object-oriented" programming languages. C++ is a superset of C in the sense that most C programs are legal C++ programs, and consequently C++ inherits (pun intended!) many of C's problems.

The encapsulation mechanism provided by C++ is the *class*, which is essentially a C `struct` extended in two ways — 1) fields (called *members*) are categorized as either *public* or *private*, and 2) members may be data or functions. A private member is accessible only within member functions, whereas a public member is accessible in both member and non-member (i.e., client) functions. In typical classes, data members are private and function members are public. This organization prevents a client program from directly accessing the representation of a class, thereby enforcing information

---

26 It is interesting that no one realized that explicitly allowing aliasing of actual parameters wouldn't complicate the proof system, and equally surprising that after the Alphard experience was reported, later specification/verification systems persisted in embracing aliasing.

hiding from the client program. However, the client programmer has access to the private members of the class, so information hiding is not enforced from the client programmer.

Note that the members of a class are data and functions. Specifically, a class cannot have a type member. The encapsulation approach suggested is therefore the abstract data object approach, discussed in Section 2.4.1.1.

The definition of a member function in a class definition includes the function's signature and optionally the function's code. Notably lacking from the definition is any kind of formal specification. Indeed, neither formal specification nor formal verification are mentioned in [Stroustrup 86], and it is safe to conclude that formal methods played no role in the design of C++.

It is possible to separate "specification" (i.e., function signatures) from implementation using the header file mechanism of C++. In this approach, the definition of a class is contained in a header file, and does not include the code for member functions. A client of the class only needs access to this header file. The code for the member functions is contained in a separately compiled file. For example, a header file (e.g., IntStk.h) for class IntStk might contain:

```
class IntStk {
  int arr[100];          // representation of integer stacks
  int top;
public:
  IntStk ();             // constructor
  void Push(int x);      // add integer onto the stack
  int Pop ();            // remove the top integer
  int IsEmpty ();        // return true iff stack is empty
};
```

A separately compiled file would contain the implementation of the member functions:

```
#include "IntStk.h"

IntStk::IntStk ()
{ top = 0; }

void IntStk::Push (int x)
{ arr[++top] = x; }

int IntStk::Pop ()
{ return arr[top--]; }

int IntStk::IsEmpty()
{ return top==0; }
```

A sample client of IntStk is:

```
#include "IntStk.h"
...
IntStk s1,s2;
int x,y;
...
s1.Push (y);        // push y onto stack s1
x = s2.Pop ();      // pop the top item from s2 into x
```

This example suggests and demonstrates two important characteristics of C++ classes. First, each class has exactly one representation (declared as private members of the class), and each member function, like every C++ function, has exactly one implementation. In other words, C++ does not directly support multiple implementations for a specification, although derived classes can be used to effect some aspects of this. Another implication is that if the representation (i.e., a data member) of a class changes, *all* clients of that class must be recompiled, even though this information is contained in the private portion of the class and is not accessible by the clients.

Second, it is possible to define constructor functions (and a destructor function) for a class. The constructor function has the same name as the class (e.g., IntStk) and is automatically invoked on each variable of the class at the beginning of a block containing the variable's declaration. A constructor may have parameters, in which case actual parameters are placed in parentheses following the variable's name. A destructor function (not demonstrated in the example) has the name of the class preceded by a tilde

(e.g. ~IntStk) and is automatically invoked on each variable of the class at the end of a block where it is declared.

C++ allows a class to be *derived* from another class (called the *base class*) — a mechanism called *inheritance* in most object-oriented languages. The derived class inherits all members of the base class (though it only has access to the public members), and it optionally adds members of its own. Because of this relationship, every variable of the derived class is also considered a variable of the base class. The converse is not true, however (i.e., a variable of the base class is not considered to be a variable of its derived classes). A derived class extends only one base class, so C++ supports "single inheritance."

A derived class can redefine (i.e., overload) functions defined in the base class. Redefined functions are not required to have the same signatures as the functions in the base class. If a derived class redefines all functions from its base class, it has essentially reimplemented the base class. This is a very restricted form of multiple implementation of a specification, but is the only one available to C++ programmers.

A base class may define some of its member functions as *virtual* functions, which derived classes may redefine. A virtual function redefined in a derived class must have the same signature as the virtual function in the base class. There is a difference between overloading a function defined in the base class and redefining a virtual function — binding of overloaded functions is done statically at compile-time, whereas binding of virtual functions is done dynamically at run-time.

C++ does not directly support the specification of generic classes. However, some characteristics of generic classes can be effected by defining a macro (i.e, `#define`) that, when invoked with actual type(s), defines a new "non-generic" class. This is only practical if the implementation is not separate from the specification, and the allowable actual parameters to the macro are restricted to all be the same size (e.g., pointers). This may appear to be somewhat convoluted, which it is.

A feature of C++ inherited from C is its extreme reliance on explicit pointers. Not only can a pointer variable contain the address of dynamically-allocated memory, it can also contain the address of a statically allocated variable. In addition, "pointer arithmetic" can be performed on pointer variables. These features (along with many others) make a

modular proof system for C++ virtually impossible to define. Consequently there is little hope that large C++ programs could ever be formally specified or verified.

In summary, C++ is a language that does not adequately meet any of the criteria for reusable software development. It is not possible to formally specify or verify C++ programs. The "specification" of a part can be separated from its implementation, but this separation is not encouraged. Through a complex system of macro definitions it is possible to define a very restricted generic class, which is cumbersome and not the suggested approach. Likewise, it is possible to use inheritance to permit multiple implementations of a specification, but this is not encouraged. Finally, C++ offers no alternative to the problems inherent to copying, except reliance on explicit pointers. In conclusion, C++ cannot be used to design and implement reusable software components, as defined in Section 2.3.

2.6.7 Eiffel

Eiffel [Meyer 88] is an "object-oriented" programming language designed to encourage the development of reusable software components that are both correct and efficient. With these goals, it is not surprising that many of the issues raised in this dissertation are addressed by Eiffel. What is interesting, though, are the significant differences between the approach taken by Meyer in designing Eiffel, and the approach taken in RESOLVE (discussed in Chapter 3).

In Eiffel, items (called *features*) are encapsulated in a structure called a *class*, which may be generic. There are two kinds of features — data (called attributes) and operations. Features (both attributes and operations) that can be directly accessed by a client are explicitly exported from the class. All non-exported features are inaccessible by the client, enforcing information hiding from client programs. Because data (rather than types) are encapsulated in classes, Eiffel modules are designed using the abstract data object approach, discussed in Section 2.4.1.1.

A type in Eiffel is either simple (i.e., integer, real, character, or boolean) or a class. Structured types, such as array, are predefined classes from a system library. All simple types are built-in to the language, and are defined in the abstract data type style (see Section 2.4.1.2) — the language defines a type and a set of operations with parameters

of that type, and a variable of a simple type is considered to contain a value from its type's domain. Classes, on the other hand, are defined in the program (or the system library) using the abstract data object style — they encapsulate data with the operations on that data. Furthermore, a variable of a class type is considered to contain a pointer to the class' representation (i.e., data and operations). An Eiffel programmer must consequently alternate between the abstract data type and abstract data object styles of programming.

Every variable contains a known value at the beginning of the block in which it is declared. The value is determined by the variable's type — integers contain zero, booleans contain false, characters contain the null character, reals contain 0.0, and classes contain a void reference (recall that class variables are considered to be pointers). An actual object is created by executing the Create operation on the class variable, which allocates memory for the object and initializes its attributes. A class may explicitly define a Create operation to initialize its attributes. Otherwise, all attributes are initialized to the appropriate known value, as discussed above. Memory occupied by unreferenceable objects is reclaimed by automatic garbage collection. Thus, Eiffel addresses some of the problems discussed in Section 2.5.3.6 concerning uninitialized variables, but automatic user-defined finalization is not supported.

It is possible to have more than one class variable reference a given object. This potential for (explicit) aliasing complicates the formal specification and verification of a program, for reasons discussed in Section 2.5.1. Of more significance, though, is the fact that implicit aliasing can occur as a result of parameter passing, significantly increasing the complexity of the formal proof system, as discussed in Section 2.6.1.

Eiffel includes syntactic slots for placing assertions defining such things as pre- and post-conditions, module-level invariants, and loop invariants. However, formal specification was not the primary motivation in the design of Eiffel, and Meyer admits that Eiffel is not powerful enough to formally define reusable parts. For example, assertions must be boolean expressions that can be evaluated at run-time. The assertion language does not include the means to express universal or existential quantification, making it so weak that even the simplest reusable component — the LIFO stack — cannot be formally defined [Meyer 88]. Thus, assertions are used more as a debugging aid than for formal specification.

Also, Eiffel assertions are written in terms of the component's state, which consists of the values of all attributes defined within the class. Abstraction cannot be utilized in defining a component, since the definition must be made in terms of implementation-level structures (i.e., attributes). This restriction limits the usefulness of assertions for specifying the behavior of a component.

The specification and implementation of a class are defined together, and information hiding from a client programmer is not enforced. In fact, Meyer argues that the implementation of a class should not necessarily be kept secret from a client programmer. However it is possible to have the environment produce a "short" version of a class containing only the public information.

Eiffel supports and encourages development of classes that inherit one or more other classes. A class that inherits the features of another class is called an "heir" to that class, and the class that is inherited is said to be its "ancestor." A class is compatible with all of its ancestors (e.g., a variable of class A that is an heir of class B may be assigned to a variable of class B). Eiffel supports multiple inheritance because a class can have multiple ancestors.

Eiffel uses inheritance in three ways — class extension, class modification, and multiple implementations of a class. In class extension an heir adds attributes and/or operations to those defined in the ancestor. An heir may also modify an ancestor by reimplementing some of its features. However, the modifications must meet the specifications defined in the ancestor (e.g., a redefined operation must have the same signature and meet the pre- and post-conditions specified in the ancestor).

Finally, a class can declare one or more features as *deferred*, which must be implemented by heirs of the class. The specification of the deferred feature is defined in the ancestor, and all implementations of that feature must meet this specification. A class that has one or more deferred features is called a deferred class. Variables of a deferred class can be declared, but it is not possible to Create an object of a deferred class. Deferred classes provide a mechanism for realizing a limited form of multiple implementations for a specification — the ancestor class contains the specification, and heirs contain the implementations. The problem with this approach is that a client programmer changes implementation of an object by changing that object's class. The

fact that the new class is a different implementation is only apparent from examining the class hierarchy.

Eiffel takes an interesting approach to information hiding between a class and its heirs — a class has access to *all* features of its ancestors. In other words, there is no information kept secret between a class and its heirs. The argument for this is the flexibility permitted by allowing an heir to "reuse" and extend portions of an ancestor's implementation in ways not necessarily foreseen by its original designer. Of course, an obvious implication is that when an implementation of a class changes, *all* heirs must be examined and possibly modified — a formidable task at best.

In summary, Eiffel is a language that encourages the development of software modules (called classes) using the abstract data object approach to encapsulation. Classes may be generic, and it is possible to specify a class. However, the specification language is not powerful enough to allow complete formal specification, which is rather unfortunate because many of the right ideas are addressed — for example, an operation that is redefined in an heir must still meet the specification given in the ancestor.

It is not possible to declare explicit pointer types, but class variables are implemented as pointers to an object, which is necessary to know in order to understand the language. Also, both explicit and implicit aliasing are permitted, which would complicate a proof system if Eiffel had one.

Information hiding is enforced only between a class and a client program. By design, information hiding is not enforced between a class and a client programmer, nor between a class and its heirs. The inheritance mechanism can be used to separate specification from implementation and to have multiple implementations of a specification. However, this is a somewhat cumbersome way to accomplish these objectives, and in fact, neither objective is seen as a particularly interesting or worthwhile goal in and of itself. Finally, Eiffel does not provide any alternative to the problems inherent to copying, except for pointers. The conclusion is that Eiffel does not have the necessary mechanisms to encourage the design of reusable software components, even though this is one of its primary design goals.

2.6.8 CLU and Larch/CLU

CLU [Liskov 81] is a programming language designed to support program development by defining *clusters* that encapsulate data with operations. Though reusability was not a design goal of CLU, it provides many of the constructs necessary to design and implement reusable parts, which are discussed in this section. CLU does not provide mechanisms for formal specification and verification.

Larch [Guttag 85, Wing 87] is a family of specification languages. The Larch Shared Language is used to define mathematical theories (called *traits*) by formally defining mathematical types (called *sorts*), constants, and functions, similar to the presentation in Section 2.4.2.1. A Larch Interface Language is defined for a programming language for which programs are to be formally specified, and essentially extends the programming language with constructs necessary for formal specification. The design of each Larch Interface Language is heavily influenced by the syntax and semantics of the particular language. Specification of a program in a Larch Interface Language is written in terms of mathematical functions and relations for a trait developed in the Larch Shared Language. Larch/CLU is one such interface language, and programs written in it are formally specified CLU programs.

In order for a Larch Interface Language to be useful for formal verification, proof rules and formal semantics must be defined for it. It is not clear from the literature whether Larch Interface Languages include constructs for annotating actual code, or whether they are used solely for specifying the behavior of a program (or module). All discussion and examples presented in [Wing 87] and [Guttag 85] concentrate on specification rather than formal semantics or proof rules.

A cluster (which may be generic) encapsulates data with all operations that manipulate that data. Because data rather than types are encapsulated, CLU (and Larch/CLU) support the abstract data object approach to encapsulation, discussed in Section 2.4.1.1. However, CLU does not support cluster inheritance, and so is not considered a true "object-oriented" language.

The Larch/CLU specification of a cluster models the data as a value from a sort (i.e., mathematical type) whose trait (i.e., mathematical theory) was developed in the Larch

Shared Language, and operations are specified using pre- and post-conditions that involve functions and relations from that trait. In this respect, Larch/CLU specifications follow the model-based approach discussed in Section 2.4.2.3. However, the development approach described in [Wing 87] suggests a new trait be developed for each cluster, rather than defining a cluster in terms of an existing trait. This approach has all of the problems inherent to the development of new mathematical theories, discussed in Section 2.4.2. Larch, though, does not prevent a designer from using a trait from a library (such as [Guttag 86]), and in fact this possibility is briefly mentioned in [Guttag 85].

The specification of a Larch/CLU cluster is defined independently of any implementation, and the CLU library mechanism allows several implementations to exist for a specification. Larch/CLU therefore supports both separation of specification from implementation, and multiple implementations for a specification. However, a single client cannot choose different implementations for different instances of the same specification.

CLU was not designed for formal verification, and includes some features that complicate formal specification and verification. For example, all variables are considered to be pointers to objects, and it is possible (and sometimes necessary) to alias an object by having several variables reference it. As discussed in Section 2.5.1, this aliasing complicates specification and verification.

In conclusion, Larch/CLU provides mechanisms for formally specifying a generic cluster and separating this specification from its implementation(s). A specification is developed using a two-tiered approach, with a mathematical theory specified in the Larch Shared Language, and the specification of a cluster in Larch/CLU. Unfortunately, formal semantics and proof rules are not mentioned for Larch/CLU, which are necessary for formal verification. CLU contains some structures, such as pointers and aliasing, that complicate formal specification and verification. Finally, CLU does not offer any solutions to the inefficiency inherent to copying. In summary, Larch/CLU has some, but not all, of the constructs necessary to encourage the design and implementation of reusable software components.

2.6.9 Z

Z [Spivey 89] is a specification language defined independently of any programming language, and is useful for formally specifying mathematical theories and operations. Mathematical theories are specified by defining a signature and a set of axioms, and it is possible to make mathematical definitions (e.g., concatenation of strings). Theories and definitions can be generic. Z is similar to the Larch Shared Language, although the syntax is quite different. A library of mathematical theories, called the mathematical toolkit, is provided, and contains definitions of useful theories such as sets, relations, functions, numbers, sequences, and multisets.

An operation is formally specified by defining its effect on a state space (i.e., one or more mathematical variables) using pre- and post-conditions. Thus, a reusable part can be formally specified by defining a state space (i.e., set of mathematical variables) that mathematically models the data, and operations that affect that state space. Unfortunately, there are no encapsulation mechanisms in Z, so the the specification of the part is simply a set of independent specifications.

Z is not associated with any programming language, but is strictly a specification language. For this reason it does not include mechanisms necessary for formal verification of programs, such as code annotations and proof rules for code. These structures need to be added to Z to make it useful for development and verification of real programs.

As discussed in Section 2.3.1, programs in a programming language not specifically designed for formal verification will most likely be difficult to specify and verify because the language probably has constructs, such as pointers and implicit aliasing, that frustrate formal specification and verification. For this reason the value of specification-only languages, such as Z, for formal verification is probably minimal.

In summary, Z is a specification language that is not associated with any programming language. It is used to formally specify mathematical theories and operations, and is powerful enough to formally specify a reusable part. Obviously, the specification of the part (written in Z) is separate from its implementation (coded in some programming language). However, because it does not include the capacity for actual code, it does

not address the remainder of the reusable software part issues, such as multiple and efficient implementations. Therefore, Z alone is not appropriate for designing and implementing reusable software components, nor is Z in combination with any implementation language we know.

2.7 Summary

Reusable software refers to software components that can be incorporated into a variety of programs *without modification* (except possibly parameterization). Furthermore, reusing these software parts should not compromise other software engineering principles such as information hiding and data abstraction. Designing and building reusable software components potentially improves both software quality and programmer productivity for three reasons — it is cost effective to commit the necessary resources to design the components properly, the designer will likely take the job seriously and design a higher quality part, and it is usually easier to reuse a well-designed component than to design and implement one on the fly. Unfortunately, there are several technical and non-technical impediments to widespread software reuse.

A reusable component has several characteristics. First, its functionality is formally specified, which serves as an unambiguous contract between a client and implementer of a part, and is essential for formal verification. Second, the specification and implementation of a part exist in separately-compilable units, which enforces the principle of information hiding, opens up the possibility of multiple implementations of a specification, and allows a client to be compiled and verified before an implementation is coded. Third, a component is generic (i.e., parameterized) if at all possible, which allows one specification to describe an infinite collection of related software components. Fourth, a component potentially has multiple implementations, each with possibly different performance, allowing a client programmer to select the implementation that best suits the application. Finally, a reusable component has efficient implementations even when it is a generic part.

There are two general approaches to encapsulation. The abstract data object approach encapsulates data with the operations that manipulate that data, and is the basic paradigm associated with "object-oriented" programming and languages. The abstract data type

approach encapsulates a type with operations that have one or more parameters of that type.

There are two popular approaches to formally specifying software. In the algebraic approach, a mathematical theory is developed for each reusable component, which, in order for it to be useful, must be shown to have several properties, such as consistency, and soundness and completeness with respect to the intended interpretation. In the model-based approach, the data and operations of a part are specified in terms of existing mathematical theories, for which important properties and theorems have already been proved.

The design of a programming language has a profound impact on the ability to design and implement reusable components. For example, uncontrolled pointers and aliasing complicate the formal definition of a language as well as specification and verification of programs written in that language. Defining copying as the only data movement primitive in a language leads to designs of generic procedures that have no efficient implementation. Also, issues relating to types and variables — what are they, when are two types equivalent, etc. — affect the complexity of the language's formal language definition and the ability to write readable specifications and to formally verify programs.

A survey of several important programming languages validates the first point of the thesis — namely, that no modern programming language has all of the constructs necessary to encourage and facilitate the design of reusable software components, as defined in this chapter.

CHAPTER III

RESOLVE

RESOLVE (an acronym for REusable SOftware Language with Verifiability and Efficiency) is a programming language and environment specifically created to encourage the design and implementation of reusable software components. RESOLVE is an imperative language, with control structures similar to those found in most structured languages such as Pascal and Ada. A program is organized as a collection of separately-compiled modules. The behavior of each module is formally specified in a *conceptualization*, with the structures and code implementing each conceptualization contained in a *realization*. RESOLVE permits (and even encourages) several realizations for any conceptualization, allowing multiple implementations, each with potentially different performance characteristics.

An interesting feature of RESOLVE is that no types are built-in to the language. Instead, every type is provided by some conceptualization, including those normally built-in such as integer, character, and boolean. Likewise, structured types such as arrays and records are not defined in RESOLVE, but are defined by conceptualizations. A similar approach is taken with respect to pointer variables. This design makes RESOLVE very regular if a bit primitive.

Another feature of RESOLVE is that data is moved by swapping the contents of two variables, rather than copying the contents of one variable to another. The ability to make a copy of a data value is not a built-in operation in RESOLVE. This unique approach offers several significant advantages.

Finally, every type has an initial value defined for it, and every variable is automatically initialized to an initial value of its type before it is referenced in any executable statement.

This facilitates verification and eliminates the possibility of program bugs caused by uninitialized variables.

This chapter informally defines RESOLVE and argues why it encourages the design and implementation of reusable software components, as defined in Section 2.3. RESOLVE is defined by presenting examples of RESOLVE modules, with an accompanying discussion of the motivation and implications with respect to reusability.

3.1 Conceptualizations and Type Parameters

The functionality of a RESOLVE component is defined within a *conceptualization.* This section presents some basic characteristics of conceptualizations. Advanced features of conceptualizations are presented in Sections 3.5 and 3.6. The example used throughout this section is the LIFO stack.

3.1.1 LIFO Stacks and Conceptualization Stack_Template

The Last In First Out stack is perhaps the most studied abstract data type in computer science. It seems that everyone uses it as a benchmark in describing approaches to data specification and abstraction.

There are several reasons why the stack is a logical choice for describing an approach to data abstraction. First, it is a good reference point, since the general intuitive notion of a stack is understood by the vast majority of practitioners of computer science. Even though the precise definition of stacks varies somewhat among camps, the typical person reading the literature will most likely understand the problem being addressed. Second, stacks are simple enough to describe in a relatively small amount of space, yet are surprisingly non-trivial. And finally, stacks have two simple implementations — an array with a top index, and a list.

A RESOLVE conceptualization for stacks is presented in Figure 2.

```
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

    procedure Push
      parameters
        alters s : Stack
        consumes x : Item
      end parameters
      ensures "s = Post(#s, #x)"
    end Push

    procedure Pop
      parameters
        alters s : Stack
        produces x : Item
      end parameters
      requires "s ≠ Lambda"
      ensures "#s = Post(s, x)"
    end Pop

    control Is_Empty
      parameters
        preserves s : Stack
      end parameters
      ensures Is_Empty iff "s = Lambda"
    end Is_Empty
  end interface
```

Figure 2

Specification for a Module Providing the Generic Type Stack

Figure 2 (continued)

```
    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
```

Perhaps one of the most striking characteristics of a RESOLVE module is its verboseness, especially with respect to keywords. This was a conscious design decision, made without apologies. RESOLVE modules are written primarily to be read and understood by programmers, and it is felt that cryptic abbreviations for keywords thwart this goal. If an appropriate editing environment is used for module construction — such as the one presented in Chapter 5 — a programmer never types keywords. Thus verbosity need not imply a penalty on the time necessary to construct a program.

A RESOLVE conceptualization provides a formal specification as well as an informal description of a software component. The majority of the text of the conceptualization is devoted to formal specification, which provides the "official" definition of the component. Informal prose descriptions of a component are also useful, since they can provide the human reader with a "feel" for what the component does. The informal description is contained within the **description** section of a conceptualization. It must be understood that descriptions are "unofficial," and in the event that the informal

description contradicts the formal specification of a component, the specification is *always* used as the component definition.

A type is formally defined in RESOLVE by modeling it as a mathematical type from some mathematical theory, and operations are formally defined by pre- and post-conditions written as assertions in one or more mathematical theories. The reader should recognize this as the model-based approach to formal specification, discussed in Section 2.4.2.3.

Let's take a closer look at what exactly is defined by Stack_Template. The interface section contains definitions of all items provided by the conceptualization to a client. It should be apparent that Stack_Template is exporting (i.e., providing) a type (called Stack) and operations to add an item to a stack (procedure Push), remove an item from a stack (procedure Pop), and determine if a stack is the empty stack (control Is_Empty).

In this example, stacks are modeled as mathematical strings from string theory, with the right end of a string containing the top item on the stack. Operations Push, Pop, and Is_Empty are defined by assertions in string theory. The empty stack is modeled as the empty string, operation Push concatenates an item onto the right end of a string, Pop removes and returns the rightmost item from a string, and Is_Empty determines if a string is the empty string.

3.1.2 Generic Conceptualizations and Type Parameters

The conceptualization in Figure 2 is for homogeneous stacks, where all items contained in a stack are the same type. However, the actual type of items to be contained in the stack is not specified in the conceptualization, but is indicated as conceptualization parameter Item. For this reason Stack_Template is a *generic* conceptualization. There are no restrictions on what type Item may be, so this one conceptualization actually defines an infinite number of stack types, including, for example, stacks of integers, stacks of characters, and even stacks of stacks of integers. Stack_Template defines a *type family* called Stack and *operation families* called Push, Pop, and Is_Empty. An actual type (e.g., stack of integers) and actual operations (e.g., Push, Pop, and Is_Empty for stacks of integers) are created only when the conceptualization is *instantiated*, as discussed in Section 3.2.3.

As demonstrated by this example, formal parameters to a conceptualization are declared in the parameters section. Two kinds of parameters can be passed to create a conceptualization instance — types and facilities. The role that type parameters play is discussed in this example, while facility parameters are discussed in Section 3.5.

3.1.3 Mathematical Theory Modules

Conceptualization Stack_Template is defined in terms of mathematical string theory. But what exactly is mathematical string theory? Anyone who has taken a course in discrete mathematics probably has encountered string theory at least enough to have an intuitive understanding of it. String theory defines a type for strings over some alphabet, along with a constant representing the empty string, and definitions such as concatenation of two strings. It should be clear that string theory is parameterized (i.e., generic) because it is defined with respect to some alphabet.

Relying upon a person's intuition of string theory (or any mathematical theory for that matter) is not precise enough for formal specification, and in fact would lead to ambiguities — the very problem formal specification is supposed to solve! For this reason, mathematical theories must themselves be formally specified. In RESOLVE this is accomplished within a theory module, which is a module that specifies mathematical types and mathematical functions. Theory modules are similar to conceptualizations, except that theory modules provide *mathematical* types and functions, whereas conceptualizations provide *program* types and operations. The precise contents and syntax of theory modules have not been finalized, and in fact are not a part of the research presented here.

Nonetheless, the syntactic slot for specifying mathematical theories is included in conceptualizations. Specifically, theories are instantiated within the **auxiliary** section of a conceptualization, in a manner similar to conceptualization instantiation discussed in Section 3.2.3. In the Stack_Template example, theory String_Theory_Template is instantiated to create a math *facility* called String_Theory. This provides a mathematical type for strings over (the math model of) type Item, called String_Theory.String

(renamed String[27]). A function that concatenates an item onto the right end of a string called String_Theory.Post (renamed Post), and the empty string constant String_Theory.Lambda (renamed Lambda), are also provided. This mathematical type is used as the model for stacks, as indicated in the declaration for type Stack.

3.1.4 Specification of Types

The exemplar clause in a type specification indicates an identifier that represents a prototypical variable of that type. The exemplar is referenced in the assertions of the type specification, and is also treated as a program variable in initialization and finalization routines, as discussed in Section 3.7.2.

Every program type in RESOLVE has an initial value specification defined in the initially clause of the type definition. Every variable is guaranteed to have a value that meets the initial value specification for its type before the variable's first reference. The motivation for initial values was discussed in Section 2.5.3.6. In this example every variable s of type Stack initially satisfies the assertion "`s = Lambda`"; in other words, all stacks are initially modeled by the empty string, and hence are empty stacks.

3.1.5 RESOLVE Operations

Three kinds of operations can be defined within a RESOLVE program — procedures, functions, and controls. The differences among these operations are the manner in which information is exchanged between the invoker and the operation. With procedures, all information is exchanged via parameters. This is similar to procedures in most structured languages such as Pascal and Ada, except that RESOLVE has no global variables, so indeed parameters are generally the *only* means of exchanging information between an invoker and procedure. (There are, however, shared module variables available to operations provided by a particular module instance.)

RESOLVE functions are similar to procedures, except that a function returns exactly one value to the invoker, and parameters are used exclusively to provide the function with information from the invoker. Because functions cannot access global variables and

---

27 In essence, renaming an item provides a shorthand identifier for it. Here, for example, the type provided by this instance can be called either String_Theory.String or String.

parameters cannot be used to return information to the invoker, functions in RESOLVE do not have side effects. A function can be invoked only within an assignment statement, which is discussed in Section 3.3.1.4.

Conceptualization Stack_Template does not define any functions. However, Figure 3 contains the definition for a function Top that returns a copy of the top item of a stack without effectively removing it from the stack[28]. In this example, it is assumed that Item, Stack, String, Lambda, and Post are defined as in Stack_Template in Figure 2.

```
function Top returns topitem : Item
  parameters
    preserves s : Stack
  end parameters
  requires "s ≠ Lambda"
  ensures "∃ r : String, s = Post(r,topitem)"
end Top
```

Figure 3

Specification of Function Top

Controls are similar to functions in that exactly one piece of information is returned to the invoker, and parameters are used solely to provide the control with information from the invoker. The difference between functions and controls is that a control does not return a data value, but a state value used exclusively to determine the action of an if or while statement. The implications are obvious — controls can be invoked only within an if or while statement, and controls return one of two possible state values. As presented in Sections 3.3.2 and 3.3.3 execution of a control terminates when one of two return statements is executed within the control — return yes or return no. The state value returned by the control is determined by which return statement is executed.

The motivation for including control operations in RESOLVE centers around the design goal of not having any types built-in to the language, per the discussion in Section 2.5.3.7. In most languages, the alternation and iteration constructs (e.g., if and while statements) require a type be built-in to the language (e.g., boolean). Controls permit RESOLVE to be defined with no built-in types. (It is crucial to understand that "yes"

---

28 The reason Top is not included as an operation in Stack_Template is because it is a secondary operation, as defined in Section 2.4.1.3.

and "no" are *not* data values of some type, but rather state values used exclusively to control the action taken by if and while statements. For example, the result of a control invocation cannot be assigned to a variable.)

3.1.6 Parameter Modes

An operation's formal parameters are declared within the parameters section of the operation. In addition to an identifier and type, each formal parameter has an associated *mode* — consumes, produces, alters, or preserves. The parameter mode describes how a parameter is used in the exchange of information between invoker and operation, and also helps streamline pre- and post-conditions for the operation. Parameter modes do *not*, however, describe the *parameter passing mechanism* used to exchange information between invoker and operation. As discussed in Section 3.3.1.3, all parameters are passed by swapping.

Information flows strictly from the invoker to the operation via a consumes parameter. As the name implies, the information provided by the invoker (in the actual parameter) is "consumed" by the operation, and in fact the actual parameter contains an initial value for its type when the operation returns. For example, the item to be pushed onto a stack is supplied to procedure Push by the invoker, and information is flowing strictly from the invoker to procedure Push via parameter x; thus, x is a consumes parameter. The motivation for specifying this unusual behavior has to do with efficient implementation of generic modules, discussed in Section 3.8.3.

A produces parameter is in a sense just the opposite of consumes, since information flows strictly from the operation to the invoker. Information originally in the actual parameter is discarded (i.e., finalized) by the operation. For example, the item removed from a stack by procedure Pop is returned to the invoker through parameter x. Since x is not used to supply Pop with information from the invoker, it is a produces parameter.

An alters parameter indicates that useful information is passed to the operation by the invoker, and also returned to the invoker by the operation. The information returned might not be the same information originally sent to the operation. In other words, the invoker gives information to the operation, the operation possibly alters that information, and then gives it back to the invoker. For example, procedure Push is

given a stack via parameter s, which it modifies (by concatenating a new item to it), and then gives back to the invoker. Thus, parameter s is an alters parameter.

A preserves parameter is similar to alters except that the value returned by the operation is guaranteed to be the same as the value sent to it. From the invoker's point of view, it lets the operation use a value, and the operation agrees not to change it. For example, control Is_Empty is given a stack via parameter s, determines if s is the empty stack, and then returns it to the invoker unchanged. Thus, s is a preserves parameter.

It is important to note that an operation is allowed to change a preserves parameter during its execution, *as long as it restores the parameter to its original value before returning*. For example, an operation is permitted to pop items from a stack that is passed to it as a preserves parameter, provided that all items are pushed back onto the stack before the operation returns. This is a subtle yet important characteristic of preserves parameters.

It should also be noted that preserves is the only mode that does not potentially alter the value of the actual parameter. RESOLVE functions and controls are not allowed to alter their parameters. Therefore, all parameters to functions and controls must be preserves parameters.

3.1.7 Operation Specification

The effect of an operation is formally defined using pre- and post-conditions. A requires clause, if present, specifies the pre-condition of the operation. If a requires clause is not present, the pre-condition is assumed to be true (indicating the operation does not have a pre-condition). Similarly, the post-condition of an operation is specified in an ensures clause. Since each operation is assumed to have some effect, every operation must have an ensures clause.

The requires clause is an assertion that the operation assumes is true at the time it is invoked. Normally the requires clause specifies restrictions placed upon the values passed to the operation by the invoker. For example, it is not meaningful to pop from an empty stack, so the requires clause of procedure Pop specifies that parameter s must not be the empty stack (which is modeled as the empty string).

The ensures clause is an assertion the operation guarantees to be true when it returns, *provided the requires clause was true when the operation was invoked.* The implications of this last part are discussed shortly. For now, let's assume the requires clause was met when the operation was invoked. The ensures clause usually relates the values of parameters at the end of the operation to the original values of parameters when the operation was invoked. In other words, it is necessary to reference the values of parameters at two points in time — the value at the beginning of the operation and the value at the end. Within an ensures clause, a "#" preceding a parameter identifier (e.g., #s) denotes the value of that parameter when the operation is invoked. A parameter identifier without a "#" (e.g., s) denotes the value of that parameter when the operation returns.

For example, the ensures clause of procedure Pop is "`#s = Post(s,x)`", meaning the string created by concatenating the value of stack s (modeled as a string) with item x when Pop returns equals the string contained in s when Pop was invoked. This is a somewhat roundabout way of saying that Pop has the effect of removing the rightmost element of the string modeling stack s. This example also demonstrates that ensures clauses are indeed assertions and not assignment statements.

Each parameter mode can affect requires and ensures clauses in two ways — it may implicitly add a clause to the ensures clause, and it may place restrictions upon how a parameter may be used in assertions. For example, a produces parameter is not used to pass information to an operation, so it may not be mentioned in a requires clause.

A consumes parameter always has an initial value when the operation returns. In effect, the conjunction "`and init(x)`" is implicitly part of the ensures clause for any consumes parameter x. It is never necessary (and is not valid) to mention the new value of a consumes parameter within an ensures clause.

Similarly, the value of a preserves parameter at the end of the operation is the same as it was at the beginning, so the conjunction "`and x = #x`" is implicitly part of the ensures clause for any preserves parameter x. This means there is no difference between x and #x in the ensures clause. By convention only x may appear there.

The value of a produces parameter at the beginning of an operation cannot have an effect upon the outcome of that operation. Thus, "x" cannot appear in the requires clause and "#x" cannot be mentioned in an ensures clause for any produces parameter x. A similar restriction holds for the identifier representing the value returned by a function (e.g., topitem in Figure 3).

Finally, let's return to an issue raised in the previous discussion — what happens if an operation is invoked and its requires clause is not met? In this situation nothing is assumed about the operation's effect, so the operation can do anything, including crash the system, return bogus results, or commence World War Three. The designer of an operation does not need to specify what happens if the requires clause is violated. Likewise, the programmer implementing the operation does not have to worry about what to do in this situation — anything is considered valid! Put another way, the requires clause specifies under what conditions it is meaningful to call an operation. Invoking an operation when the requires clause is violated is meaningless, and therefore the results of that invocation are meaningless. The problem is not in the operation definition, but rather in the client invoking the operation.

3.1.8 Another Example: Conceptualization One_Way_List_Template

For a second example of a RESOLVE conceptualization, let's examine a *one-way list*, which is a structure useful for storing elements that are accessed sequentially in one direction only (e.g., left to right). A one-way list can be described abstractly as a sequence of items with a marker of the "current position," which is called the *fence*, located between two of the items in the sequence. For example, <3 9 4‡8 1> represents a list of integers consisting of 3, 9, 4, 8, and 1, with the fence (denoted by "‡") between 4 and 8. <3 9 4 8 1‡> represents a list containing the same elements as before, but with the fence at the right end. <‡3 9 4 8 1> represents a list with the fence at the left end. The operations on a one-way list allow the contents of the sequence to be altered, the fence to move a step at a time in one direction (hence, the name one-way), and tests concerning the position of the fence.

This structure is often called a "linked list" in data structures texts. This name is inappropriate — the structure described is a list, and "linked" is simply one of several

possible *representations* of lists. The abstract descriptions of a one-way list and its operations most assuredly should *not* talk about nodes and pointers, which are implementation details.

A RESOLVE conceptualization for one-way lists is presented in Figure 4. The informal description section is not included in this figure, since it is essentially presented in the accompanying text.

```
conceptualization One_Way_List_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.Pre as Pre
          String_Theory.Post as Post
          String_Theory.Cat as Cat
        end renaming

      Tuple_2_Theory is Tuple_2_Theory_Template(String,String)
        renaming
          Tuple_2_Theory.Tuple as List_Model
          Tuple_2_Theory.Projection_1 as Left
          Tuple_2_Theory.Projection_2 as Right
        end renaming
    end math facilities
  end auxiliary

  interface
    type List is modeled by List_Model
      exemplar L
      initially "Left(L) = Lambda and Right(L) = Lambda"
    end List
```

Figure 4

Specification for a Module Providing the Generic Type List

Figure 4 (continued)

```
procedure Reset
  parameters
    alters L : List
  end parameters
  ensures "Left(L) = Lambda and
             Right(L) = Cat(Left(#L),Right(#L))"
end Reset

procedure Advance
  parameters
    alters L : List
  end parameters
  ensures "Cat(Left(L),Right(L)) =
           Cat(Left(#L),Right(#L))
           and ∃x:Item, Left(L) = Post(Left(#L),x)"
end Advance

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

procedure Remove_Right
  parameters
    alters L : List
    produces x : Item
  end parameters
  requires "Right(L) ≠ Lambda"
  ensures "Left(L) = Left(#L) and
           Pre(x,Right(L)) = Right(#L)"
end Remove_Right

procedure Swap_Rights
  parameters
    alters L1 : List
    alters L2 : List
  end parameters
  ensures "Left(L1) = Left(#L1) and
           Right(L1) = Right(#L2) and
           Left(L2) = Left(#L2) and
           Right(L2) = Right(#L1)"
end Swap_Rights
```

Figure 4 (continued)

```
      control At_Left_End
        parameters
          preserves L : List
        end parameters
        ensures At_Left_End iff "Left(L) = Lambda"
      end At_Left_End

      control At_Right_End
        parameters
          preserves L : List
        end parameters
        ensures At_Right_End iff "Right(L) = Lambda"
      end Is_Empty
    end interface

    description
      ...
    end description

end One_Way_List_Template
```

In this conceptualization, type family List is modeled as a cartesian product of two strings. Strings are specified in theory String_Theory_Template, which provides math type String, math constant Lambda for the empty string, and math functions Pre, Post, and Cat for constructing strings. (Pre defines concatenation of an item onto the left of a string, Post defines concatenation of an item onto the right end of a string, and Cat defines concatenation of two strings.)

Cartesian products of two (parameter) types are formally defined in theory Tuple_2_Theory_Template, which provides a math type called Tuple, and math functions for projecting either part of a Tuple, called Projection_1 and Projection_2. Here, the instance of Tuple_2_Theory_Template is called Tuple_2_Theory, and the type defined is renamed List_Model, with the projection functions renamed Left and Right.

One of the strings of a List holds the items to the left of the fence, and the other string holds the items to the right of the fence. These two strings are projected from a list with math functions Left and Right, respectively. A list is initially empty, represented by both Left and Right of the List being the empty string.

Procedure Reset places the fence at the left end of a List. Advance moves the fence one item to the right. Add_Right inserts an item into a List immediately to the right of the fence, and Remove_Right removes and returns the item immediately to the right of the fence. Procedure Swap_Rights exchanges the right parts of two lists. Controls At_Left_End and At_Right_End determine if the fence is at the left or right end of a List, respectively.

3.1.9 Summary

In this section RESOLVE conceptualizations for two common data structures were presented — the LIFO stack and the one-way list. These conceptualizations demonstrate some fundamental characteristics of RESOLVE conceptualizations, including the role type parameters play in the specification of generic conceptualizations, the motivation for math facilities in formal specification, the method used to formally specify types and operations, the three kinds of RESOLVE operations (procedures, functions, and controls), and the role of parameter modes.

3.2 Simple Realizations

A RESOLVE realization contains the data structures and algorithms that implement the functionality specified in a conceptualization. This section discusses realizations by presenting a realization of Stack_Template using a one-way list. This discussion actually addresses two issues — how to implement a conceptualization, and how to make use of the types and operations specified in a conceptualization by instantiating it.

3.2.1 Realization Stack_Real_1 of Stack_Template

A realization of Stack_Template using One_Way_List_Template is presented in Figure 5.

```
realization of Stack_Template by Stack_Real_1

  conceptualization  auxiliary
    renaming
      String_Theory.Reverse as Reverse
    end  renaming
  end  conceptualization  auxiliary

  realization  auxiliary
    facilities
      List_Facility is One_Way_List_Template (Item)
        realized by List_Real_1
        renaming
          List_Facility.List as List
          List_Facility.Add_Right as Add_Right
          List_Facility.Remove_Right as Remove_Right
          List_Facility.At_Right_End as At_Right_End
          List_Facility.Right as Right
          List_Facility.Left as Left
        end  renaming
    end  facilities
  end  realization  auxiliary

  interface
    type Stack is represented by List
      exemplar s_rep
      conventions "Left(s_rep) = Lambda"
      correspondence "Right(s_rep) = Reverse(s)"
    end Stack

    procedure Push
      parameters
        alters s : Stack
        consumes x : Item
      end  parameters
      performance "O(1)"
      begin
        Add_Right (s,x)
    end Push

    procedure Pop
      parameters
        alters s : Stack
        produces x : Item
      end  parameters
      performance "O(1)"
      begin
        Remove_Right (s,x)
    end Pop
```

Figure 5

Realization Stack_Real_1 of Stack_Template Using One_Way_List_Template

Figure 5 (continued)

```
        control Is_Empty
          parameters
            preserves s : Stack
          end parameters
          performance "O(1)"
          begin
            if At_Right_End(s) then
              return yes
            else
              return no
            end if
        end Is_Empty
    end interface

    description
      ...
    end description

end Stack_Real_1
```

A realization does not exist in isolation, but as a realization of a particular conceptualization. The realization heading indicates the conceptualization that is being realized as well as the name of the realization. For example, the realization presented in Figure 5 is for conceptualization Stack_Template, and is called Stack_Real_1.

Some of the text in a realization is identical to corresponding text in the conceptualization, for example operation names and formal parameters. These portions of a realization could easily be included by the editing environment as unmodifiable text. In the realizations presented in this dissertation, text included from the conceptualization is italicized.

Every name defined in a conceptualization is available for use within a realization of it. For example, Item is defined as a formal type parameter to conceptualization Stack_Template, and is therefore implicitly defined as a type within realization Stack_Real_1. Similarly, String_Theory is defined as a mathematical theory within Stack_Template, and is known within Stack_Real_1. Every name in the scope of the conceptualization is in the scope of its realization.

In addition to definitions made in the conceptualization, realizations also define many things themselves. In the next sections realization Stack_Real_1 is discussed in detail, with the intent of demonstrating the flavor of RESOLVE realizations by presenting a rather simple example first. Advanced features of realizations are presented in Section 3.7.

3.2.2 Conceptualization Auxiliary Section

It is possible that the mathematical functions provided to a realization from the conceptualization are not sufficient, or some of the definitions provided by the conceptualization may not be convenient. For example, a realization may need mathematical theories in addition to those defined in the conceptualization, or a realization may want to rename a function from the conceptualization for convenience.

Conceptual declarations are made in the **conceptualization auxiliary** section of a realization. Declarations in this section do not replace or override declarations made in the conceptualization. Rather, they add to those in the conceptualization.

For example, the correspondence assertion for type Stack in realization Stack_Real_1 uses the definition of string reversal, defined in math facility String_Theory in conceptualization Stack_Template. This assertion could have been written as "`Right(s_rep) = String_Theory.Reverse(s)`." However, it is somewhat more convenient to rename the definition of string reversal from String_Theory as Reverse, which is accomplished in the conceptualization auxiliary section of Stack_Real_1. Note this renaming is not done in conceptualization Stack_Template since the specification does not use string reversal, and renaming it there is unnecessary (although it would have been legal).

3.2.3 Realization Auxiliary Section

The **realization auxiliary** section contains declarations necessary to explain the implementation of types and operations defined by the conceptualization. (Note the parallel between this and the auxiliary section of a conceptualization, which contained declarations of items necessary for the *specification* of types and operations defined by the conceptualization.) This section contains instantiations of conceptualizations needed

for the realization, as well as declarations of variables and operations local to the realization.

Before we go any further with this discussion, let's review the definitions of some terms. A realization that makes use of a conceptualization must *instantiate* that conceptualization. The instance of a conceptualization is called a *facility*, and the realization is said to be a *client* of that conceptualization. The phrases "declare a facility" and "instantiate a conceptualization" are synonymous.

All facilities are declared within the **facilities** subsection of the realization auxiliary section of a realization. A facility declaration states the name of the facility along with the name of the conceptualization being instantiated and the name of a realization of that conceptualization. Actual parameters must also be provided for all formal parameters to the conceptualization and realization.

For example, realization Stack_Real_1 represents stacks using one-way lists, so it instantiates conceptualization One_Way_List_Template. Facility List_Facility is declared as an instance of One_Way_List_Template using List_Real_1 as the realization. (It is assumed here that List_Real_1 is a realization of One_Way_List_Template.) Type Item, which is the formal parameter to conceptualization Stack_Template representing the type of item contained in the stack, is passed as the actual parameter to One_Way_List_Template. Thus List_Facility is a facility exporting the type one-way List of Items.

An instance of a conceptualization makes available to the client all types and operations specified in the interface section of the conceptualization, as well as all math names declared in the auxiliary section of the conceptualization[29]. Referencing any of these is accomplished by prefixing the local name by the facility name followed by a dot. For example, List_Facility.List is the name of the type provided by the instance of

---

29 The instance also provides names for all formal parameters. For example, List_Facility.Item is a name for the parameter to One_Way_List_Template in the instantiation that creates List_Facility, and is also the name of the type of the second parameter to operations List_Facility.Add_Right and List_Facility.Remove_Right. Normally, these formal parameter identifiers provided by the instance are not referenced in the client, since it is much easier to simply reference the actual parameter (e.g., Item). However, this is important for the discussion of type equivalence in Section 3.4.

One_Way_List_Template in Figure 5. This naming scheme is necessary to disambiguate identically named items provided by two different instances[30].

For convenience, it is possible to alias any "dotted name" to a more descriptive identifier. This aliasing is described in a renaming section of the facility declaration. For example, List_Facility.List is renamed as List, so identifiers List and List_Facility.List both stand for the type provided by facility List_Facility. Similarly, the definition List_Facility.Right is renamed Right.

It is important to understand that facilities are static entities defined at compile-time, as opposed to dynamic entities created at run-time. All facilities exist for the entire execution of the program. It is not possible to declare a facility within an operation, have that facility come into existence only when that operation is invoked, and disappear when the operation returns.

3.2.4 Interface Section

The **interface** section of a realization contains the data structures and code that implement the types and operations defined in the interface section of the conceptualization. The types and operations defined in the interface section of a realization are exactly those types and operations specified in the interface section of the conceptualization[31]. For example, the interface section of Stack_Real_1 contains the representation for type Stack, and implementations of operations Push, Pop, and Is_Empty.

3.2.4.1 Type Representations

In realization Stack_Real_1, a Stack is represented by a List of Items, which is simply type List_Facility.List (renamed List). This is indicated in the first line of the type declaration for Stack.

The remainder of the declaration for type Stack specifies exactly how a one-way list is used to represent a stack. In this realization, the items on a stack are contained in the

---

30 For consistency, this naming scheme is enforced even if all names provided by the facilities are unambiguous without being qualified with a facility name.

31 For this reason, much of the text appearing in the interface section can be automatically included in a realization by the editing environment.

right portion of a one-way list, with the top item on the stack immediately to the right of the fence in the one-way list. For example, a stack of integers whose model is the string <4 2 7 5> (with 5 as the topmost item) would be represented by the one-way list <‡5 7 2 4>. In other words, the right string of a one-way list is the reverse of the string that models the stack. This correspondence between the representation of a type and its abstract model is stated formally in the correspondence clause of the type declaration. In this clause, the identifier s_rep is an exemplar denoting a list used to represent a stack, and s is an exemplar (defined in the conceptualization) denoting the abstract stack.

The conventions clause in a type declaration describes an invariant that is guaranteed to be true before and after any operation invocation. In this example, the conventions clause states that the left portion of any one-way list representing a stack will always be the empty string. Conventions and correspondence clauses play a crucial role in formal verification of the realization, as discussed in [Krone 88].

Recall from the discussion in Section 3.1.4 that every type has an associated initial value described as part of the type specification in the conceptualization. In the case of stacks, every variable of type Stack is initially an empty stack. What one-way list represents an empty stack? The conventions and correspondence clauses from the declaration of type Stack indicate that an empty list represents an empty stack. Since the initial value of a one-way list (as defined in One_Way_List_Template in Figure 4) is the empty list, nothing need be done to an initial one-way list for it to represent an empty stack. However, this is not the case in general, and as discussed in Section 3.7.2, it is generally necessary to have code within the type declaration that creates the representation for an initial value.

Similarly, as discussed in Section 3.7.2, it may be necessary to have code that finalizes a variable, releasing memory occupied by its representation. In the case of stacks, finalizing the one-way list representing the stack is sufficient, and no code is explicitly required within the type declaration to accomplish this.

3.2.4.2 Operation Implementations

Implementation of the stack operations is straightforward. The operation name and formal parameter list is simply duplicated from the conceptualization. The performance clause is an assertion indicating performance characteristics of the operation's implementation, and is typically in "big-O" notation. Here, all operations take a constant amount of time to execute, assuming the one-way list operations realized by List_Real_1 execute in constant time.

The code implementing an operation is introduced by the keyword begin. Procedure Push simply inserts the item being pushed into the one-way list immediately to the right of the fence. Pop removes and returns the item immediately to the right of the fence. Is_Empty returns yes if the fence is at the right end of the one-way list, and returns no otherwise.

It is important to note that every variable (and formal parameter) of type Stack is considered a variable of type List within the code portions of realization Stack_Real_1. This is the reason there isn't a type compatibility problem when a Stack is passed as an actual parameter to a one-way list operation. Type equivalence is discussed in Section 3.4.

3.2.5 Summary

In this section a simple implementation of conceptualization Stack_Template was presented — one that represents stacks using one-way lists. Although the realization seems trivial, it demonstrates two important features of RESOLVE — the relationship between a conceptualization and a realization of it, and the relationship between a conceptualization and a client of it. This example also demonstrates that a facility is an instance of a conceptualization with actual values bound to formal conceptualization parameters and a specific realization chosen. A conceptualization must be instantiated in order for a client to reference the types and operations specified by the conceptualization.

3.3 Data Movement and Control Structures

The code implementing the operations in realization Stack_Real_1 in Figure 5 is very simple, consisting solely of operation invocations and one if statement. Although RESOLVE has a minimal set of control and data movement primitives, this example obviously does not demonstrate its entire repertoire! This section discusses control structures and data movement in RESOLVE.

The control structures defined in RESOLVE are those found in most modern block-structured languages, such as Ada and Pascal. Specifically, RESOLVE's control structures include an if statement for alternation, a while statement for iteration, a procedure invocation statement, and a return statement to return to the invoker from an operation. In addition, the function assignment and swap statements each effect the movement of data.

Data movement in RESOLVE is discussed in Section 3.3.1, along with operation invocation, assignment, and swap statements. The if and while statements are discussed in Section 3.3.2, and the return statements are discussed in Section 3.3.3.

3.3.1 Swapping — RESOLVE's Data Movement Primitive

One of the fundamental actions performed during execution of a program is the movement of data within the execution environment. In modern imperative languages, this movement occurs as a direct result of assignment statements and procedure invocations. As discussed in Section 2.5.2, these generally involve making *copies* of data, which is inherently inefficient.

The problems inherent to copying data are addressed quite simply by RESOLVE — namely, copying is not defined as a built-in operation. Instead, *swapping* the values of two variables is the *only* data movement primitive. This is a rather radical approach, and is one of the most unique and interesting features of RESOLVE.

It may not be intuitively obvious that swapping is indeed powerful enough to warrant its inclusion as the sole data movement primitive in the language. The justification for this decision is presented in the following subsections.

3.3.1.1 The Swap Statement

The first embodiment of the swap primitive is the swap statement. The BNF description of this statement is:

```
<SWAP-STMT>:   <VAR-NAME> :=: <VAR-NAME>
```

When a swap statement is executed, the obvious happens — the values of the two variables are exchanged. For example, assume variables x and y are integer variables, and that x = 5 and y = 10. After executing the statement "x :=: y" we have x = 10 and y = 5.

No restrictions are placed upon the types of variables that can be swapped, except that the two variables must be the same type. For example, it is perfectly legal to swap two stacks of integers, but it is not legal to swap an integer with a character.

3.3.1.2 Swapping Is Efficient

One justification for including swapping as the only data movement primitive is that it is very efficient to implement. In fact, it can always be executed in constant time. In other words, swapping two stacks each containing a million elements takes no more time than swapping two integers.

At first glance this doesn't seem possible. However, all that is needed is a well-known implementation trick. The important thing to keep in mind is that each variable *appears* to always have a value from the domain of its type. From the programmer's viewpoint, the swap statement exchanges the values of two variables, as shown in Figure 6.

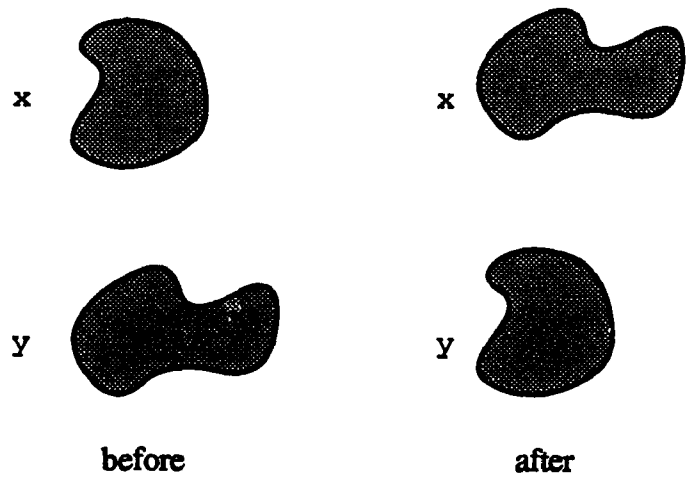

Figure 6

Abstract Effect of Swap Statement "x :=: y"

It is possible to represent each variable as a pointer to a data structure that represents its value[32]. This pointer is part of the implementation of RESOLVE, and is completely invisible to the programmer. Figure 7 describes the action of the swap statement "`x :=: y`" from the implementation (i.e., run-time environment) viewpoint. It is apparent from this figure that swapping the values of two variables simply involves swapping two pointers. The time required to swap these pointers is independent of the sizes of structures they point to. Therefore swapping two variables is a constant-time operation. Representing variables in this way has other advantages, which are discussed in Section 3.8.3.

---

32 Actually, if the data structure representing a value fits into the memory required for a pointer, the pointer is not necessary. For example, integers might be implemented without pointers.

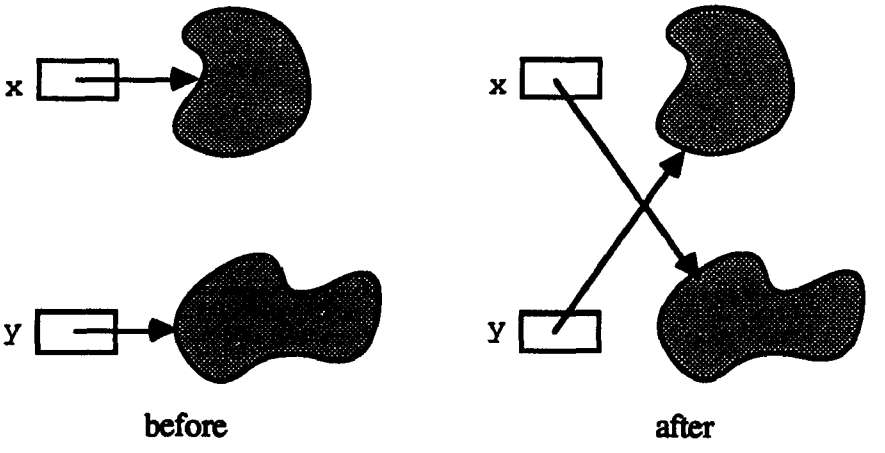

Figure 7

Implementation of Swap Statement "x :=: y"

3.3.1.3 Operation Invocation and Parameter Passing

Passing information between an invoker and an operation via parameters also involves the movement of data. In RESOLVE this is also accomplished by swapping. To see how this works, it is important to understand the role of formal and actual parameters, and the action taken by the RESOLVE run-time environment when an operation is invoked.

It is also imperative that parameter modes (i.e., consumes, produces, alters, and preserves) are not confused with parameter passing mechanisms. Recall from Section 3.1.6 that parameter modes describe how each parameter is used in the exchange of information between an invoker and an operation. They also help streamline pre- and post-conditions. Parameter modes do *not* describe a mechanism for this exchange of information. In RESOLVE, swapping is the mechanism used in the exchange of information between invoker and operation, regardless of parameter mode. Thus, RESOLVE's parameter passing mechanism is "call-by-swapping."

For this discussion, let's assume that all actual parameters are variables (i.e., not function invocations). In fact, this is the restriction placed upon actual parameters in the current version of RESOLVE, although it would not be difficult to relax this somewhat.

When an operation is invoked, the formals may be assumed to have unknown values of their types. Upon invocation, the actual parameters are swapped with corresponding formal parameters. This swapping occurs sequentially, but the order is not specified (i.e., one may *not* assume that parameters are swapped left to right). See Figure 9b. After the parameters have been exchanged the code for the operation is executed. When the operation returns, actual parameters are again swapped with their corresponding formal parameters. See Figure 9d.

Figures 8 and 9 demonstrate the effect of call by swapping by showing the contents of actual and formal parameters at critical times during the operation invocation sequence. In these figures it is assumed that zero is the initial value for type Int.

```
procedure proc
  parameters
    consumes a : Int
    produces b : Int
    alters c : Int
    preserves d : Int
  end parameters
```

(a) Definition of Procedure Proc

```
proc (w,x,y,z)
```

(b) Client Invocation of Proc

Figure 8

Definition and Invocation of Sample Procedure

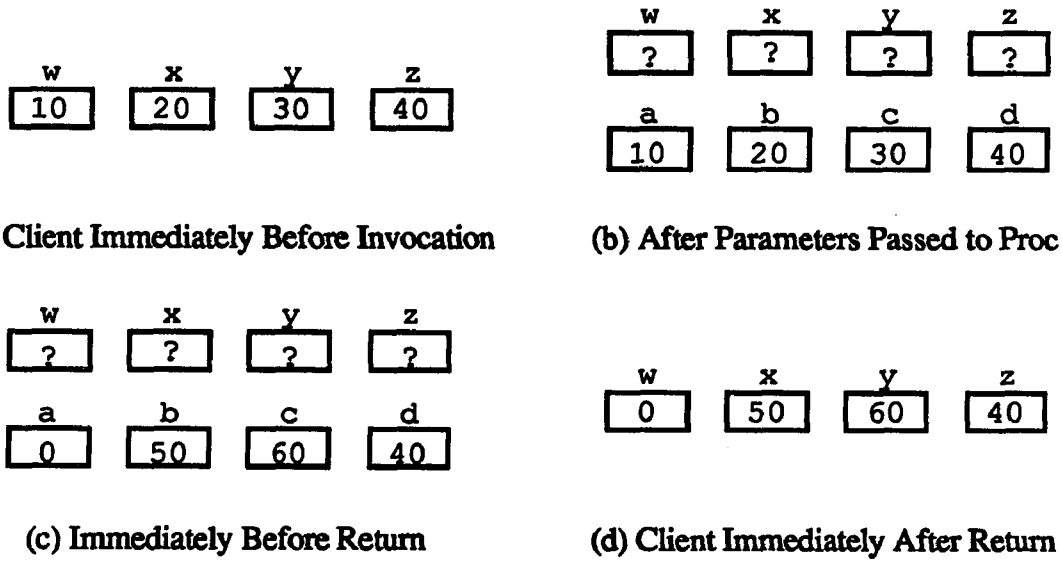

Figure 9

Effect of Sample Procedure Invocation

Note that operation call/return overhead in RESOLVE is constant, since all parameters are passed by swapping, and swapping is a constant-time operation. This is a significant performance advantage over some traditional parameter passing mechanisms such as call by value and call by value-result.

A restriction is placed upon actual parameters in RESOLVE — namely, a variable cannot appear more than once in an argument list. This restriction is necessary because actual parameters are swapped sequentially with formal parameters when an operation is invoked. After an actual parameter is swapped with the corresponding formal parameter, the value of the actual parameter is unknown. If that variable appeared elsewhere in the actual parameter list, its current value (i.e., an unknown value) would be swapped with the corresponding formal parameter, which is most likely not the intention. Note that this problem is not unique to swapping, as discussed in Section 2.6.1.

3.3.1.4 The Function Assignment Statement

The assignment statement in RESOLVE places the result of a function invocation into a variable. The BNF description of this statement is:

```
<ASSIGN-STMT>: <VAR-NAME> := <FUNC-CALL>
```

Note that the right-hand side of an assignment statement *must* be a function invocation — specifically, it cannot be a variable. In other words, the assignment statement cannot be used to implicitly copy a variable. (Making copies of variables is discussed in Section 3.3.1.5.) Also, the assignment statement is the only context where a function invocation is allowed, and the assignment target variable can be an actual parameter to the function call (e.g., the assignment "`x := f(x)`" is legal).

When an assignment statement is executed, RESOLVE's run-time environment invokes the function in the manner described in the previous section. The function's return value identifier (e.g., topitem in Figure 3) is initialized like all other local variables of the function. When the function returns, the value in the function's return value identifier is swapped with the value of the assignment target variable, and the return value identifier is finalized. The effect of this is that the original value in the target variable is finalized, and its new value is the return value of the function.

3.3.1.5 Copying a Variable

RESOLVE does not provide the programmer with a built-in feature to make copies of variables. There are two primary reasons for this, as discussed in Section 2.5.2. First, copying is usually expensive. Including it as a language primitive permits (and possibly encourages) programmers to unwittingly make copies of variables, thereby paying an implicit performance penalty. Second, copying a variable demands type-specific code. Since no types are built-in to RESOLVE, the compiler cannot even generate the code to make a copy of something as simple as an integer. The actual representation of integers is known only within some realization! Of course, there are times when it is necessary to make a copy of a variable, and RESOLVE would be quite useless if there were no way to accomplish this.

If it should be possible for a client to make a copy of a variable of a particular type, it is the responsibility of the writer of the conceptualization providing that type (or one that uses it) to explicitly specify an operation that accomplishes this. For example, Figure 10 contains the specification of a function Replica that creates and returns a copy of a variable of type T. The assignment statement "`a := Replica(b)`" would have the effect of the traditional assignment statement "`a := b`."

```
function Replica returns clone : T
  parameters
    preserves x : T
  end  parameters
  ensures "clone = x"
end Replica
```

Figure 10

Specification of Function Replica for Type T

If a conceptualization does not provide an operation that copies a variable of a provided type, there is no direct way to make a copy for that type[33]. In other words, RESOLVE does not require every type have a copy operation.

3.3.2 Ifs, Whiles, and Control Invocations

This section discusses RESOLVE's definition of the two most elementary control structures in any block-structured language — the if and while statements — and the special relationship these statements have with control operations.

---

33 This does not necessarily mean there is no way at all to make a copy for that type. For example, it is possible to make a copy of a stack by popping all items from the stack into a temporary stack, then popping each item from the temporary stack, making a copy of that item, pushing the copy onto the duplicate stack, and pushing the original item onto the original stack. This algorithm relies on the existence of an operation to copy an item, but is independent of the actual representation of a stack. See Sections 3.5.2 and 3.7.1.

The syntax of the if statement is relatively straightforward:

```
<IF-STMT>:      if [ not ] <CTRL_CALL> then
                   <CODE>
                [ else
                   <CODE> ]
                end if
```

When an if statement is encountered during execution, the control operation called within the if statement is invoked, as discussed in Section 3.3.1.3. Recall from the discussion in Section 3.1.5 that a control operation returns a control state to the invoker by executing either a return yes statement or a return no statement. If the indicated control returns via a return yes statement, the statements following **then** are executed. Otherwise the statements following **else** are executed if present, or execution continues at the statement following **end if** if **else** is not present. If **not** is placed in the if statement, the control state returned by the control is inverted, so that return yes acts as if it were return no, and vice versa.

The BNF description of the while statement is:

```
<WHILE-STMT>:   <LOOP-ASRT>
                while [ not ] <CTRL-CALL> do
                   <CODE>
                end while

<LOOP-ASRT>:    maintaining <ASSERTION>      |
                ensuring <ASSERTION>
```

The operational semantics of the while statement are what you'd expect — the control operation is invoked (as discussed in Section 3.3.1.3), and the statements within the body of the loop are executed if and only if the control returns via a return yes statement (or return no for a while not statement). This continues until the control returns via a return no statement.

The loop assertion must be present, and formally specifies the effect of the loop. The assertion is either part of a maintaining clause or part of an ensuring clause. If the loop assertion is contained in a maintaining clause, the assertion is a loop invariant. If the assertion is contained in an ensuring clause, it is a post-condition for the loop, and relates the values variables have before loop execution (denoted by a '#' before the variable name, e.g., #x) to the values they have after loop execution (denoted by the

variable name, e.g., x). The loop assertion has no effect upon the execution of the loop, but is used for verification, as described in [Krone 88].

The syntax and semantics of both statements are similar to most if and while statements. However, a control invocation determines the action taken. Indeed, controls can only be invoked within these two contexts. The motivation for this scheme is simple — the complete separation of data (and data types) from control structures, discussed in Section 2.5.3.7. RESOLVE control structures are defined independently of all data types (even type "boolean"), meaning RESOLVE can be completely defined without any types built-in to the language.

3.3.3 Return Statements

The return statement does the obvious, namely passes control from an operation back to its invoker. Actual and formal parameters are also swapped when an operation returns, as discussed in Section 3.3.1.3. There are actually three forms of the return statement in RESOLVE:

```
<RETURN-STMT>: return        |
               return  yes   |
               return  no
```

Use of the simple return statement is permitted only within procedure and function operations (i.e., it is not permitted within control operations). When return is executed, parameters are swapped as discussed in Section 3.3.1.3, and control passes back to the operation's invoker. Note that the return value of a function is contained in the function's return value identifier, and is *not* specified as part of the return statement.

Use of return yes and return no statements is permitted only within control operations (i.e., they are not permitted within procedure or function operations). When executed, parameters are swapped with the invoker as discussed in Section 3.3.1.3, and control passes back to the control's invoker (which must be in an if statement or while statement). As discussed in Section 3.3.2, the action taken by the if or while statement is determined by which return statement the control executed.

Every procedure and function has an implicit return statement at the end of the code, so an explicit return statement is not necessary. For example, see Figure 5. On the other

hand, control operations do not have an implicit return statement. Reaching the end of a control without executing either a return yes or return no statement is illegal.

3.3.4 Summary

In this section data movement within RESOLVE programs was discussed, as well as the control structures defined by RESOLVE. Data movement is accomplished by swapping the values of two variables. The swap and assignment statements explicitly involve data movement (i.e., swapping); in addition, all parameters are passed between an invoker and an operation by swapping actual with formal parameters.

Copying the value of a variable is not a built-in operation in RESOLVE, but is an operation specified in a conceptualization, just like all other operations except swapping. RESOLVE does not require that every type be copyable (i.e., a conceptualization does not have to specify a copy operation); thus, it may not be possible to copy some variables.

The control structures defined by RESOLVE are the if statement, while statement, operation invocation, and return statement. The definitions of these control structures are straightforward. This is a minimal set of control structures compared to RESOLVE's ancestors such as Pascal and Ada. However, these control structures are sufficiently powerful and easy to understand.

3.4 Types and Type Equivalence

RESOLVE is a statically-typed language, meaning that every variable has a type that is fixed and known at compile time. As discussed in Section 2.5.3, the purpose of types is to check that variables are used in legal contexts, e.g., that the two variables in a swap statement have the same type. If a variable appears in a context that is illegal for its type, that statement is simply invalid, and no meaning is defined for it.

But what precisely are types in RESOLVE, what is the relationship between program types and mathematical types, and what does it mean for two types to be equivalent? This section addresses these questions and discusses the relationships among types, domains, and module instances.

3.4.1 Domains

In RESOLVE a *math domain* is an anonymous set of anonymous values defined by an instance of a *theory*. It is defined implicitly by the axioms of the theory. For example, instance Number_Theory of Number_Theory_Template in Figure 15 provides the axioms defining the domain of integers, and instance String_Theory of String_Theory_Template in the same figure provides the axioms defining the domain of strings over integers.

Similarly, a *program domain* is defined by an instance of a *conceptualization*. Each element in a program domain is modeled by some element in a corresponding math domain. The elements in a program domain are those values that are "reachable" by executing the operations defined in the conceptualization. For example, instance List_Facility in Figure 5 defines a program domain whose elements are modeled by 2-tuples of strings. The elements in this domain are exactly those elements whose models are reachable by executing all possible combinations of one-way list operations provided by List_Facility.

When a facility is declared, the instantiated theory or conceptualization is chosen from a library of theories and conceptualizations. Even though these libraries contain a finite number of items, there are an infinite number of possible instances that can be created from them, due to the fact that many of the theories and conceptualizations are generic. For example, it is possible to declare a math facility providing a domain for integer, one providing a domain for strings of integers, another providing a domain for strings of strings of integers, still another providing a domain for strings of strings of strings of integers, etc.

Given a library of available theories, there exists an infinite collection of math domains provided by all possible instantiations of these theories, shown as set $\mathfrak{D}_{\mathfrak{m}}$ in Figure 11. Similarly, given a library of available conceptualizations and realizations for them, there exists an infinite collection of program domains defined by all possible instantiations, shown as set $\mathfrak{D}_{\mathfrak{p}}$ in Figure 11. Every math domain defined by a math facility is an element of set $\mathfrak{D}_{\mathfrak{m}}$, and every program domain defined by a program facility is an element of set $\mathfrak{D}_{\mathfrak{p}}$.

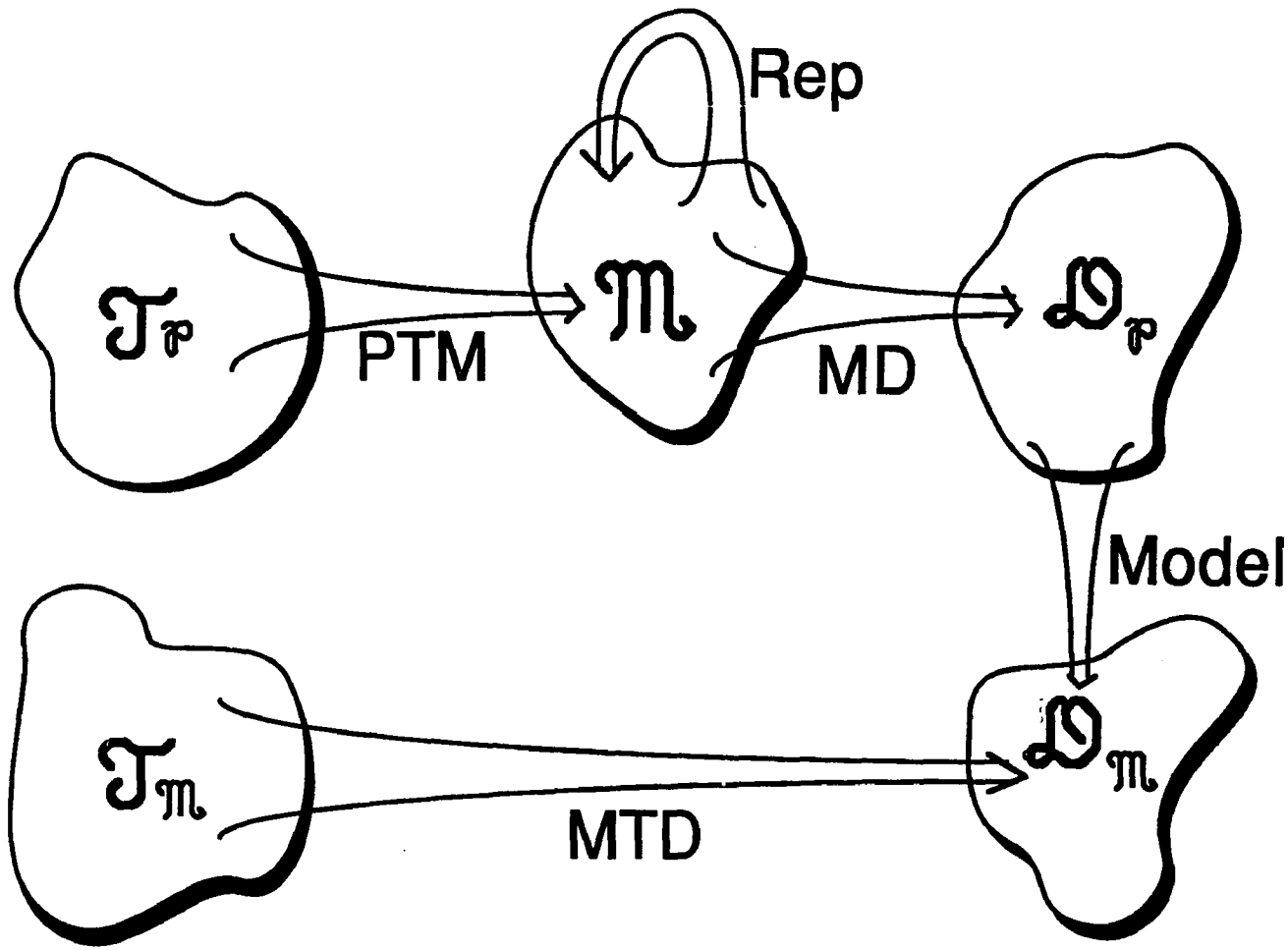

Figure 11
Relationship Between Types, Markers, and Domains

Total function Model:$\mathcal{D}_{\mathcal{P}} \rightarrow \mathcal{D}_{\mathcal{M}}$ indicates the math domain that models each program domain, and is shown as a hollow arrow in Figure 11. This mapping for a particular program type is specified in the "is modeled by" clause of the type declaration within a conceptualization. This function is neither one-to-one nor onto, because more than one program domain may be modeled by a particular math domain, and there exist math domains that do not model any program domain.

3.4.2 Math Types

When a client module instantiates a theory, every math domain defined by the resulting math facility is given a name that is unique within the client. A *math type* is simply the name given to a math domain by a client. Each client has an associated set containing all math types defined within it, shown as set $\mathcal{T}_{\mathcal{M}}$ in Figure 11.

Total function MTD:$\mathcal{T}_{\mathcal{M}} \rightarrow \mathcal{D}_{\mathcal{M}}$ (which stands for Math Types-to-Domain) indicates the math domain associated with each math type. This function is not one-to-one since several math types may name the same math domain.

As an example of how the MTD mapping is created for a particular client, consider the facilities section of a realization shown in Figure 12. The mappings between the various sets is shown in Figure 13. (Conceptualizations for Stack_Template and Bounded_Integer_Template are presented in Figures 2 and 21, respectively.) Here, two math domains are referenced — integers and strings of integers. However, there are seven math types declared that name these math domains.

```
facilities
  Int_Facility1 is Bounded_Integer_Template
    realized by Standard_Int_Real
    renaming
      Int_Facility1.Int as Int
    end  renaming

  Int_Facility2 is Bounded_Integer_Template
    realized by Standard_Int_Real

  IntStack_Facility is Stack_Template (Int)
    realized by Stack_Real_1
    renaming
      IntStack_Facility.Stack as Stack
    end  renaming
end  facilities
```

Figure 12

Example Type Declarations

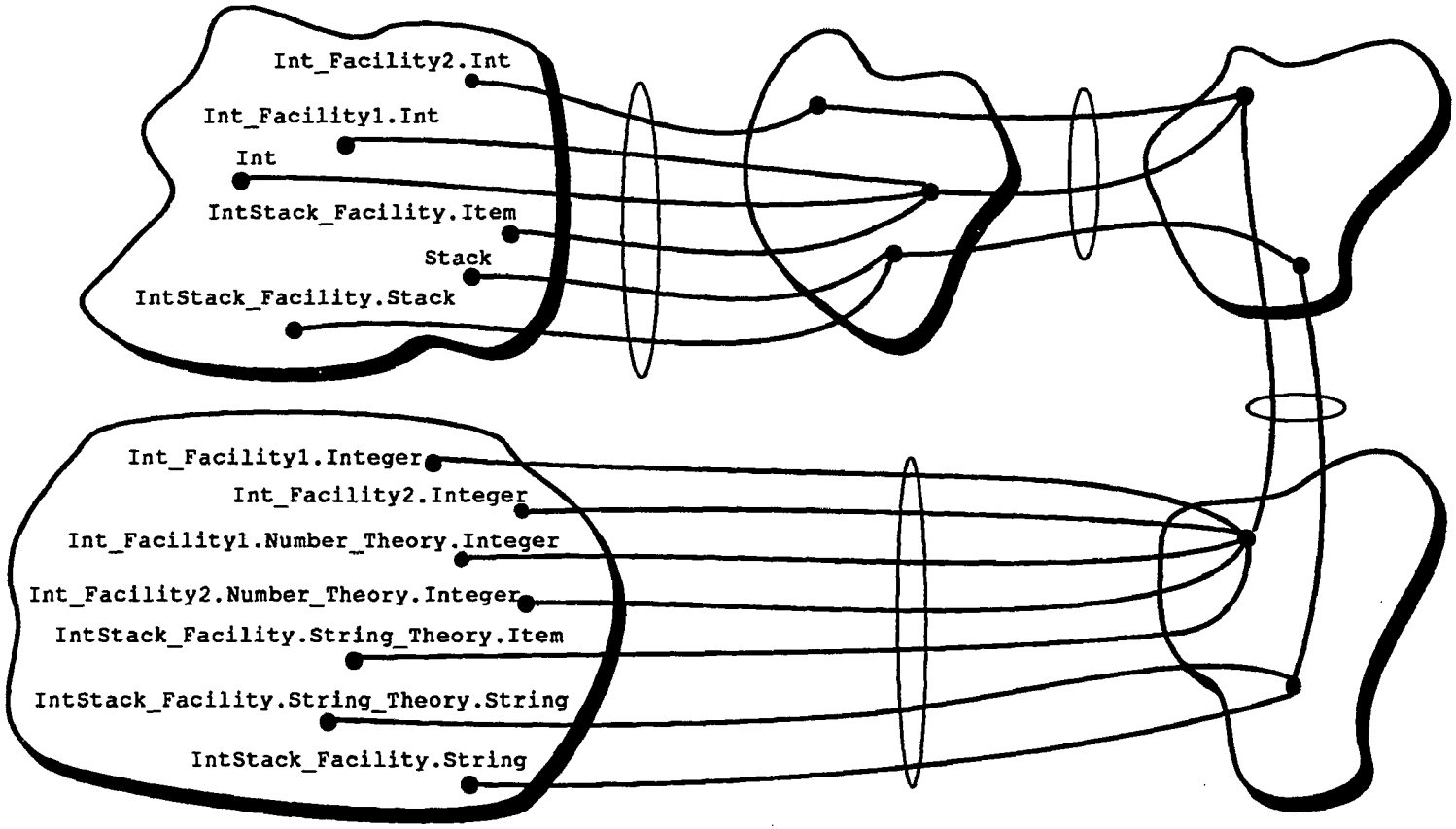

Figure 13

Type Mappings for Example Type Declarations

3.4.3 Program Types and Markers

As with math types, *program types* are simply names of things. However, because of type equivalence issues discussed shortly, program types do not directly name program domains, but instead name *markers*. When a conceptualization is instantiated by a client, a marker is created for each program domain defined. A marker is also created for each type parameter to the client and for each type provided by the client. Each client has an associated set containing all its markers, shown as set $\mathfrak{M}$ in Figure 11.

A program type is the name for a marker, which in turn designates a program domain. Each client has an associated set containing all program types declared in it, shown as set $\mathfrak{T}_{\mathfrak{P}}$ in Figure 11.

As an example, consider the facilities section of the client shown in Figure 12, with the corresponding mappings shown in Figure 13. Three markers are declared — for Int provided by Int_Facility1, Int provided by Int_Facility2, and Stack (of Int) provided by IntStack_Facility — with six names for them, which are the program types known in this client.

Total function $\mathsf{PTM}: \mathfrak{T}_{\mathfrak{P}} \rightarrow \mathfrak{M}$ (which stands for Program Types-to-Markers) indicates the marker associated with each program type. This function is onto, because each marker has at least one name. This function is not one-to-one, because a particular marker may have more than one name associated with it, as demonstrated in Figures 12 and 13.

Total function $\mathsf{MD}: \mathfrak{M} \rightarrow \mathfrak{D}_{\mathfrak{P}}$ (which stands for Markers-to-Domains) indicates the program domain associated with each marker. Markers created for identically-declared facilities map to the same program domain, as demonstrated by Int_Facility1 and Int_Facility2 in Figures 12 and 13. Because of this MD is not one-to-one. Of course, it is not onto because $\mathfrak{M}$ is the finite set of markers for a particular program module, while $\mathfrak{D}_{\mathfrak{P}}$ is the infinite set of all possible program domains creatable from the entire library of conceptualizations.

Partial function $\mathsf{Rep}: \mathfrak{M} \rightarrow \mathfrak{M}$ is defined only on the markers whose program types are provided by the client, and indicates the marker used to represent the provided type.

This function is defined by the "is represented by" clause of a type declaration within a realization. For example, Figure 14 shows the sets and mappings for realization Stack_Real_1 from Figure 5. Here, function Rep shows that provided type Stack is represented by type List.

Figure 14 also demonstrates two other important characteristics of type mappings. First, it demonstrates the mapping of type parameters (e.g., Item) and generic types (e.g., Stack and List). In the case of a type parameter the formal parameter maps to a point in $\mathfrak{M}$, but the mapping of this point to $\mathfrak{D}_{\mathfrak{P}}$ (and $\mathfrak{D}_{\mathfrak{M}}$) is unknown, as indicated by question marks. Similarly, the name of a generic program type maps to a point in $\mathfrak{M}$, but this point maps to an unknown point in $\mathfrak{D}_{\mathfrak{P}}$ and $\mathfrak{D}_{\mathfrak{M}}$. Although the mathematical model of a generic program type is unknown, a math type can be assigned to it, indicated by function MTD mapping $\mathfrak{T}_{\mathfrak{M}}$ to the same unknown point in $\mathfrak{D}_{\mathfrak{M}}$. For example, Figure 14 indicates that program type Stack is modeled by math type String (among others, all of which are names for the same math domain).

Second, Figure 14 demonstrates that all math types defined by identical instances of a theory map to the same point in $\mathfrak{D}_{\mathfrak{M}}$. For example, String (defined by the instance of String_Theory_Template in conceptualization Stack_Template) and List_Facility.String_Theory.String (defined in the instance of One_Way_List_Template) map to the same point in $\mathfrak{D}_{\mathfrak{M}}$ (as do many other names).

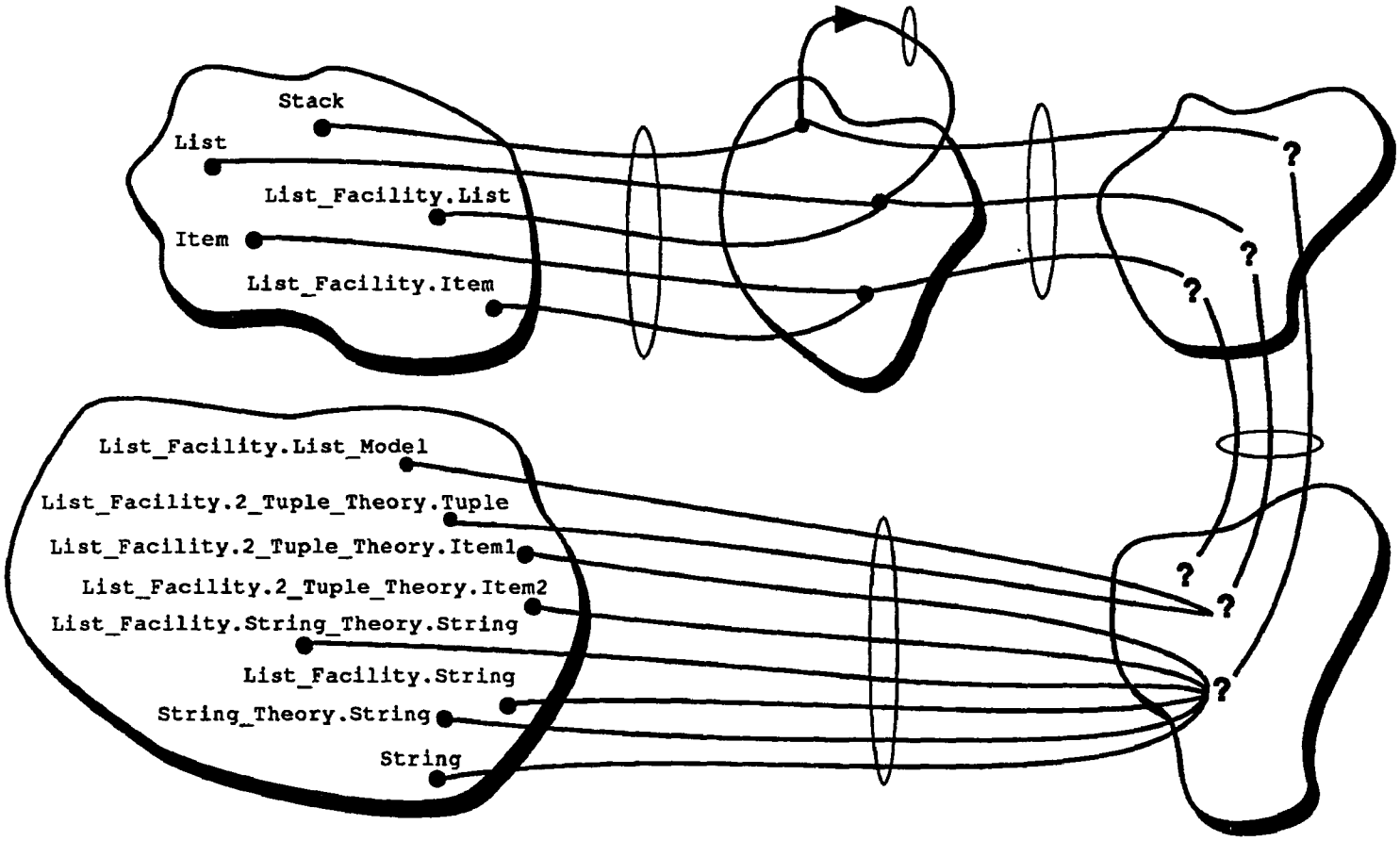

Figure 14

Type Mappings for Realization Stack_Real_1

3.4.4 Type Equivalence

Using the definition of types presented in the previous section, type equivalence is a matter of determining if two types map to the same point in some set. This may seem rather straightforward (and it is), yet some interesting issues are raised, which are discussed here.

The obvious question arising from this approach to type equivalence is which set to use to determine type equivalence. For math types there is only one choice — set $\mathfrak{M}$. Thus, type equivalence for math types t1 and t2 is defined as:

```
t1 ≡ t2 ⇔ MTD(t1) = MTD(t2)
```

For example, types List_Facility.String and String_Theory.String in Figure 14 are equivalent, since they both map to the same point in DM. However, neither of these types is equivalent to List_Facility.List_Model, since it maps to a different point.

The mathematical model of a program type t, written **math**[t], is defined implicitly by the following:

```
MTD(math[t]) = Model(MD(PTM(t)))
```

It is thus legitimate to talk about type equivalence between the mathematical model of a program type and other mathematical types. For example, the mathematical model of program type Int_Facility2.Int in Figure 13 is equivalent to the mathematical model of program type Int. Similarly, the mathematical model of program type Stack in Figure 14 is equivalent to mathematical type List_Facility.String. However, the mathematical models of program types Stack and List in Figure 14 are not equivalent.

Type equivalence of program types t1 and t2 is defined as:

```
t1 ≡ t2 ⇔
     PTM(t1) = PTM(t2) ∨
     Rep(PTM(t1)) = PTM(t2) ∨
     PTM(t1) = Rep(PTM(t2))
```

In other words, two program types are equivalent iff they both map to the same marker, or if one of the types is represented by the other. For example, program type Int in

Figure 13 is equivalent to type Int_Facility1.Int, but is not equivalent to type Int_Facility2.Int. Similarly, program type List in Figure 14 is equivalent to type List_Facility.List, and is also equivalent to type Stack.

The justification for defining program type equivalence with markers rather than program domains is based partly upon our design philosophy that realizations determine performance characteristics of the operations and do not affect the contexts where those operations can be legally invoked. In other words, changing the realization of a facility should not alter the syntactic correctness of the module. Note that changing the realization of a facility does not alter function PTM, yet it does alter function MD because realizations are part of the definition of elements of $\mathfrak{D}\,\mathfrak{p}$. Thus, if program type equivalence were defined in terms of $\mathfrak{D}\,\mathfrak{p}$, changing a realization of a facility could potentially alter the syntactic acceptability of the module, which is not a problem when defining program type equivalence in terms of markers.

3.4.5 Influence on Compiler Implementation

The sets and functions defined in the previous sections are entirely abstract, and their use was solely for the purpose of defining types and type equivalence. However, this framework is also very useful in implementing the symbol tables for a RESOLVE compiler, which is discussed in this section.

Sets $\mathfrak{D}\,\mathfrak{m}$ and $\mathfrak{D}\,\mathfrak{p}$ are infinite, so it is not possible to actually represent them within the symbol table. Because $\mathfrak{D}\,\mathfrak{p}$ is not used in determining type equivalence though, the symbol table need not represent any elements from it. However, mathematical type equivalence is defined in terms of elements of $\mathfrak{D}\,\mathfrak{m}$, so a subset of it must be maintained during compilation.

The compiler assigns a tag to each element of DM that is referenced in the client it is compiling. When a mathematical type is defined in that client by instantiation of a theory, the compiler determines the type's signature, which includes the type identifier and all actual parameters of the math facility. If the symbol table already contains a math type with that signature, the new math type is assigned the same tag as the math type already in the symbol table. Otherwise the new math type is assigned a new tag and is

added to the symbol table. The tag of a type is also assigned to all types that rename it. Two mathematical types are equivalent iff their tags are the same.

In this scheme, the "identifier" fields within the symbol table represent set $\mathfrak{T}_{\mathfrak{M}}$, the tags are elements of $\mathfrak{D}_{\mathfrak{M}}$, and the "tag" fields within the symbol table represent function MTD.

A similar approach can be used for program types, with the compiler basically constructing set $\mathfrak{M}$ for the client it is compiling. An unique marker is assigned to each formal type parameter of the client (e.g., Item in Figure 14), to each type provided by the client (e.g., Stack in Figure 14), and to each program type provided by instances of conceptualizations within the client (e.g., List in Figure 14). A type that is a formal parameter to an instantiated conceptualization (e.g., List_Facility.Item in Figure 14) is assigned the marker of the actual parameter. The marker of a type is also assigned to all types that rename it.

The symbol table includes a "represented by" field for all types provided by the client, which is assigned the marker of the type that represents the provided type. The symbol table also includes a "math model" field which is meaningful for all program types except formal type parameters. This field is assigned the tag of the mathematical type that models the program type.

Two program types are equivalent iff they have the same marker, or the marker of one is the "represented by" marker of the other.

In this scheme, the "identifier" fields within the symbol table represent set $\mathfrak{T}_{\mathfrak{P}}$, the markers are elements of $\mathfrak{M}$, the "marker" fields within the symbol table represent function PTM, the "represented by" fields represent function Rep, and the "math model" fields represent the composition of function MD with Model.

3.4.6 Summary

In this section a domain was defined as an unnamed set of values, and a type as a name given to a domain. A mathematical type was defined by an instance of a theory, while a program type was defined by an instance of a conceptualization. Given a finite library of theories, there is an infinite set $\mathfrak{D}_{\mathfrak{M}}$ that contains all possible math domains

definable by these theories. Similarly, given a finite library of conceptualizations, there is a set called $\mathfrak{D}\,\mathfrak{p}$ that contains all possible program domains definable by these conceptualizations. Given a module there are sets $\mathfrak{T}\,\mathfrak{p}$ and $\mathfrak{T}\,\mathfrak{m}$ that contain respectively all program types and math types defined within that module.

Every program domain (i.e., every element of $\mathfrak{D}\,\mathfrak{p}$) has a corresponding math domain that models it. This is formally defined as total function Model that maps elements of $\mathfrak{D}\,\mathfrak{p}$ to $\mathfrak{D}\,\mathfrak{m}$.

Every module has associated with it a finite set $\mathfrak{M}$ of markers that contains an element for each program domain declared in that module. A total function PTM maps elements of $\mathfrak{T}\,\mathfrak{p}$ to $\mathfrak{M}$. Partial function Rep maps $\mathfrak{M}$ to $\mathfrak{M}$, and indicates the representation of all types provided by a realization module.

Two mathematical types are equivalent if and only if they both map to the same point in $\mathfrak{D}\,\mathfrak{m}$. Similarly, two program types are equivalent if and only if both map to the same point in $\mathfrak{M}$, or one of the types is represented by the other.

These definitions for type and type equivalence are formal, simple to understand and to implement, and maintain the advantages of static type checking. By contrast, definitions surveyed by [Danforth 88] are complicated and generally do not apply for static typing.

3.5 Conceptual Facility Parameters

Ordinarily, every program type referenced within a conceptualization must be equivalent to some program type in the client. If this were not the case it might be impossible to invoke some operations because the client could not pass an actual parameter of an equivalent type. Thus, all types referenced in a conceptualization are either defined by the conceptualization (e.g., Stack in conceptualization Stack_Template) or somehow passed to the conceptualization via its parameters (e.g., Item in conceptualization Stack_Template).

Type parameters are used to pass types from the client to the conceptualization. The conceptualization cannot place any restrictions upon the type that is passed to it, and likewise it cannot make any assumptions about the type. This mechanism is used to define generic types where the component type is provide by the client.

There are situations, however, where a conceptualization needs the client to pass it a particular type, such as Int provided by an instance of Bounded_Integer_Template. *Facility parameters* must be used in these situations. This section presents three examples that demonstrate the need for and use of facility parameters — bounded stacks, copying stacks, and arrays.

3.5.1 Conceptualization Bounded_Stack_Template

Conceptualization Stack_Template in Figure 2 contains a specification for unbounded stacks where no *a priori* limit is placed on the number of items that can be contained in a stack. A different conceptualization of stacks places a finite limit (or bound) on the maximum number of items that a stack can hold. A conceptualization that captures this notion, called Bounded_Stack_Template, is presented in Figure 15.

```
conceptualization Bounded_Stack_Template
  parameters
    type Item
    facility Int_Facility is Bounded_Integer_Template
      renaming
        Int_Facility.Int as Int
      end renaming
  end parameters

  auxiliary
    math facilities
      Number_Theory is Number_Theory_Template
        renaming
          Number_Theory.Integer as Integer
        end renaming

      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.Length as Length
        end renaming

      Tuple_2_Theory is Tuple_2_Theory_Template
          (String,Integer)
        renaming
          Tuple_2_Theory.Tuple as Bounded_Stack_Model
          Tuple_2_Theory.Projection_1 as Stack_Items
          Tuple_2_Theory.Projection_2 as Stack_Max_Size
        end renaming
    end math facilities
  end auxiliary

  interface
    type Bounded_Stack is modeled by Bounded_Stack_Model
      exemplar s
      initially "Stack_Items(s) = Lambda and
                 Stack_Max_Size(s) = 0"
    end Stack

    procedure Set_Max_Size
      parameters
        alters s : Bounded_Stack
        consumes Max_Size : Int
      end parameters
      requires "Max_Size > 0"
      ensures "Stack_Items(s) = Lambda and
               Stack_Max_Size(s) = #Max_Size"
    end Set_Max_Size
```

Figure 15

Specification for a Module Providing Generic Type Bounded_Stack

Figure 15 (continued)

```
    function Get_Max_Size returns Max_Size : Int
      parameters
        preserves s : Bounded_Stack
      end parameters
      ensures "Max_Size = Stack_Max_Size(s)"
    end Get_Max_Size

    function Get_Size returns Size : Int
      parameters
        preserves s : Bounded_Stack
      end parameters
      ensures "Size = Length(Stack_Items(s))"
    end Get_Size

    procedure Push
      parameters
        alters s : Bounded_Stack
        consumes x : Item
      end parameters
      requires "Length(Stack_Items(s)) < Stack_Max_Size(s)"
      ensures "Stack_Items(s) = Post(Stack_Items(#s),#x) and
               Stack_Max_Size(s) = Stack_Max_Size(#s)"
    end Push

    procedure Pop
      parameters
        alters s : Bounded_Stack
        produces x : Item
      end parameters
      requires "Stack_Items(s) ≠ Lambda"
      ensures "Stack_Items(#s) = Post(Stack_Items(s),x) and
               Stack_Max_Size(s) = Stack_Max_Size(#s)"
    end Pop

  description
    ...
  end description

end Bounded_Stack_Template
```

Each bounded stack is modeled as a 2-tuple consisting of a string (referenced with projection function Stack_Items) that contains the items currently in the bounded stack, and an integer (referenced with projection function Stack_Max_Size) that contains the maximum allowable size of the bounded stack. The number of items in a bounded stack at any time is simply the length of the string.

Functions Get_Max_Size and Get_Size return the maximum size and current size of a bounded stack, respectively. Operations Push and Pop accomplish the obvious. Note that a requires clause is specified for procedure Push, whereas one is not specified for procedure Push in conceptualization Stack_Template.

A client of Bounded_Stack_Template is presented in Figure 16, and the corresponding program type mappings are shown in Figure 17. In this example type BStack is a bounded stack of characters, and the size and maximum size of a bounded stack are integers of type Int.

```
  ...
facilities
  Int_Facility1 is Bounded_Integer_Template
    realized by Standard_Int_Real
    renaming
      Int_Facility1.Int as Int
    end  renaming

  Int_Facility2 is Bounded_Integer_Template
    realized by Standard_Int_Real

  Char_Facility is Char_Template
    realized by Standard_Char_Real
    renaming
      Char_Facility.Char as Char
    end  renaming

  Char_Stack_Facility is Bounded_Stack_Template
    (Char,Int_Facility)
    realized by BStack_Real_1
    renaming
      Char_Stack_Facility.Bounded_Stack as BStack
      Char_Stack_Facility.Get_Size as Get_Size
    end  renaming
end  facilities
  ...
local  variables
  i : Int
  j : Int_Facility2.Int
  s : BStack
end  local  variables
  ...
```

Figure 16

Bounded_Stack_Template Client

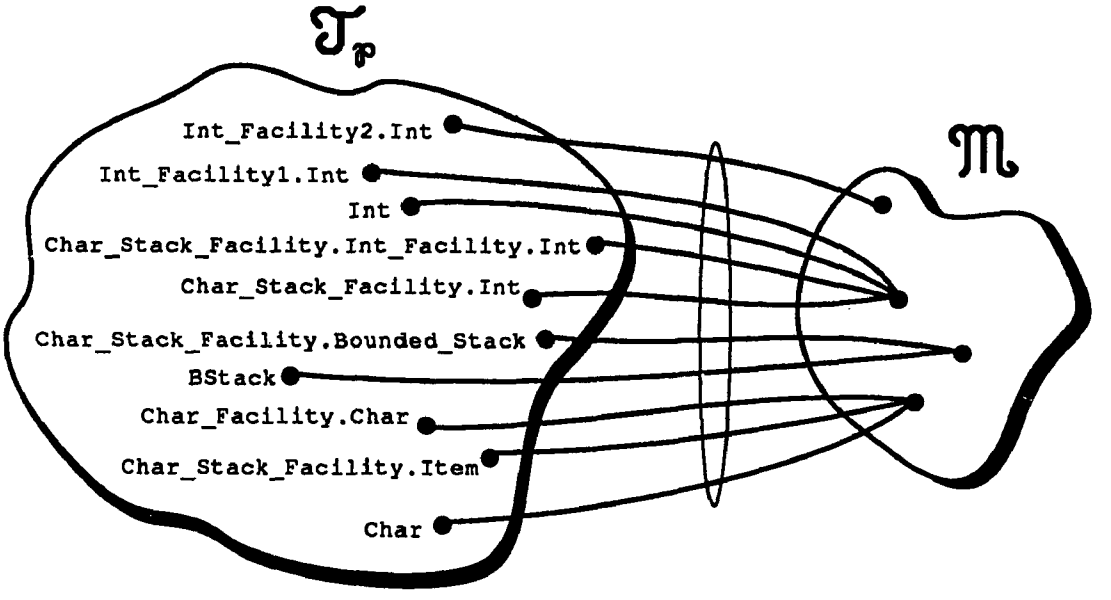

Figure 17

Program Type Mappings for Bounded_Stack_Template Client

In the above client, the type of variable i (i.e., Int) is equivalent to the type of function Get_Size (i.e., Char_Stack_Facility.Int) because both types have the same marker. Therefore, the assignment statement "`i := Get_Size(s)`" is legal. However, the statement "`j := Get_Size(s)`" is illegal, because types Int_Facility2.Int and Char_Stack_Facility.Int are not equivalent.

3.5.2 Conceptualization Copy_Stack_Template

Conceptualization Copy_Stack_Template in Figure 18 defines procedure Copy_Stack that copies one stack into another.

```
conceptualization Copy_Stack_Template
  parameters
    facility Stack_Facility1 is Stack_Template
    facility Stack_Facility2 is Stack_Template

    restrictions
      Stack_Facility1.Item = Stack_Facility2.Item
    end restrictions
  end parameters

  interface
    procedure Copy_Stack
      parameters
        preserves s1 : Stack_Facility1.Stack
        produces s2 : Stack_Facility2.Stack
      end parameters
      ensures "s2 = #s1"
    end Copy_Stack
  end interface

  description
    . . .
  end description

end Copy_Stack_Template
```

Figure 18

Specification for a Module Providing Procedure Copy_Stack

The interesting characteristic of procedure Copy_Stack (and in fact the motivation for creating this conceptualization) is that the stack types need not be equivalent. In other words, it is possible to copy a stack into a stack with a different realization. Two facilities are passed to Copy_Stack_Template — the first provides the type of the "source" stack, and the second provides the type of the "destination" stack.

The only restriction placed on the stacks is that the types of items contained in them must be equivalent. This restriction is specified in the restrictions section of the conceptualization parameters. Restriction clauses must be of the form "`type1 = type2`" where type1 and type2 are passed to the conceptualization from the client. Restrictions are enforced by the compiler when the conceptualization is instantiated.

An example client of Copy_Stack_Template is presented in Figure 19.[34]

---

[34] The reason for the realization parameter in this example (i.e., Char_Copy) is discussed in Section 3.7.1.

```
   .
   .
   .
facilities
  Char_Facility is Char_Template
    realized by Standard_Char_Real
    renaming
      Char_Facility.Char as Char
      Char_Facility.Copy as Char_Copy
    end  renaming

  Stack_Facility1 is Stack_Template (Char)
    realized by Stack_Real_1

  Stack_Facility2 is Stack_Template (Char)
    realized by Stack_Real_2

  Copy_Stack_Facility is Copy_Stack_Template
    (Stack_Facility1,Stack_Facility2)
    realized by Copy_Stack_Real_1 (Char_Copy)
    renaming
      Copy_Stack_Facility.Copy_Stack as Copy_Stack
    end  renaming
end  facilities
   .
   .
   .
local  variables
  s1 : Stack_Facility1.Stack
  s2 : Stack_Facility2.Stack
end  local  variables
   .
   .
   .
  Copy_Stack(s1,s2)
   .
   .
   .
```

Figure 19

Copy_Stack_Template Client

In this example, procedure Copy_Stack copies a stack realized by realization Stack_Real_1 into a stack realized by realization Stack_Real_2. If the client also needed to copy stacks in the other direction (i.e., from Stack_Real_2 to Stack_Real_1) it would be necessary to declare another instance of Copy_Stack_Template, passing it facilities Stack_Facility2 and Stack_Facility1 in that order.

3.5.3 Conceptualization Array_Template

As a final demonstration of facility parameters, consider conceptualization Array_Template in Figure 20.

```
conceptualization Array_Template
  parameters
    type Item

    facility Integer_Facility is Bounded_Integer_Template
      renaming
        Integer_Facility.Int as Int
      end renaming
  end parameters

  auxiliary
    math facilities
      Number_Theory is Number_Theory_Template
        renaming
          Number_Theory.Integer as Integer
        end renaming

      Function_Theory is Function_Theory_Template
          (Integer, math[Item])
        renaming
          Function_Theory.Function as Integer_To_Item
          Function_Theory.Delta as Delta
        end renaming

      Tuple_2_Theory is Tuple_2_Theory_Template
          (Integer,Integer_To_Item)
        renaming
          Tuple_2_Theory.Tuple as Array_Model
          Tuple_2_Theory.Projection_1 as size
          Tuple_2_Theory.Projection_2 as map
        end renaming
    end math facilities
  end auxiliary
```

Figure 20

Specification for a Module Providing Generic Type Array

Figure 20 (continued)

```
    interface
      type Array is modeled by Array_Model
        exemplar a
        initially "size(a) = 0 and
                   ∀i:Integer, Item.init ((map(a)) (i))"
        lemma "∀i:Integer, (i < 0 or i ≥ size(a)) ⇒
               Item.init ((map(a)) (i))"
      end Array

      procedure Set_Size
        parameters
          alters a : Array
          consumes n : Int
        end parameters
        requires "n ≥ 0"
        ensures "size(a) = #n and
                 ∀i:Integer, Item.init ((map(a)) (i))"
      end Set_Size

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

      procedure Access
        parameters
          alters a : Array
          preserves i : Int
          alters x : Item
        end parameters
        requires "0 ≤ i < size(a)"
        ensures "size(a) = size(#a) and Delta(map(a),{i}) and
                 (map(a)) (i) = #x and x = (map(#a)) (i)"
      end Access
    end interface

    description
      . . .
    end description

  end Array_Template
```

Array_Template defines type Array which is a structure encapsulating the notion of quasi-static arrays. A quasi-static array is one where the size is set at execution time. However, once the size is set, it effectively cannot be changed. The indices of the array are the first size non-negative integers (i.e., 0 to size-1).

An Array is formally defined as a pair: the size (called size), and a function (called map) from integers to type Item, the component type of the array. In the abstract, map is a total function, but the requires clause of procedure Access (which is the operation that alters the contents of an Array) restricts the index to be non-negative and less than the array's size. Therefore, function map is meaningful only on this interval.

An Array initially has a size of zero, and its function maps all integers to an initial value of type Item. Procedure Set_Size sets the size of an Array and also resets the Array's function so it maps all integers to an initial value of type Item. In other words, the old contents of the Array are lost when the size is changed. Procedure Get_Size returns the current size of an Array. Procedure Access swaps the previous value of an element of Array with the previous value of parameter x.

Array_Template also introduces two notations — lemmas and deltas. A lemma, such as specified in the specification for type Array, is an invariant (i.e., an assertion that is true at all times) that can be proved using other assertions defined in the conceptualization. For example, the lemma defined in Array states that an Array's function maps all integers less than zero or greater than or equal to size to an initial value of type Item. This is easy to prove, given that an Array's function maps all integers to an initial value of type Item when the Array is created and after invoking Set_Size (by Array's initially clause and Set_Size's ensures clause), and that the only other operation that changes this mapping is Access, which cannot change the mapping of integers less than zero or greater than or equal to size (by Access's requires and ensures clauses).

Math function Delta, defined by theory Function_Theory, is a convenient notation indicating that a mathematical function's mapping potentially changes on only a subset of its domain. Specifically, the following is the definition for Delta, where f is a mathematical function and s is a set of values from f's domain:

```
Delta(f,s) = ∀ i:Domain(f), i ∉ s ⇒ f(i) = #f(i)
```

For example, Access's ensures clause states (among other things) that function map potentially changes only on the element indexed.

3.5.4 Summary

As a consequence of RESOLVE's definition of type equivalence, it is often necessary for a client to pass an actual type to an instance of a conceptualization. Facility parameters accomplish this in RESOLVE, where an actual instance of a specific conceptualization is passed to the conceptualization being instantiated. Three examples of conceptualizations that have facility parameters were presented in this section — bounded stacks, a conceptualization providing an operation to copy stacks, and arrays.

3.6 Conceptual Constants and Variables

Conceptualizations are defined in terms of formal conceptualization parameters and declarations made within the auxiliary section. In all conceptualizations presented so far, the auxiliary sections contained only instances of theories. However, some conceptualizations are defined in terms of constant values defined by the realization or in terms of conceptual variables that contain state information.

This section demonstrates auxiliary constants and variables with three examples — conceptualization Bounded_Integer_Template, conceptualization Single_Link_Ref_Template, and conceptualization ADO_Stack_Template.

3.6.1 Conceptualization Bounded_Integer_Template

Conceptualization Bounded_Integer_Template, presented in Figure 21, defines "computational integers" which are mathematical integers with lower and upper bounds defined.

```
conceptualization Bounded_Integer_Template

  auxiliary
    math facilities
      Number_Theory is Number_Theory_Template
        renaming
          Number_Theory.Integer as Integer
          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
      min_int : Integer
      max_int : Integer
    end math constants

    constraint "min_int ≤ 0 < max_int"
  end auxiliary

  interface
    type Int is modeled by Integer
      exemplar i
      initially "i = 0"
      lemma "min_int ≤ i ≤ max_int"
    end Int

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

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

    procedure Increment
      parameters
        alters i : Int
      end parameters
      requires "i < max_int"
      ensures "i = Math_Add(#i,1)"
    end Increment
```

Figure 21

Specification for a Module Providing Type Int

Figure 21 (continued)

```
    function Add returns Sum : Int
      parameters
        preserves i : Int
        preserves j : Int
      end parameters
      requires "min_int ≤ Math_Add(i,j) ≤ max_int"
      ensures "Sum = Math_Add(i,j)"
    end Add

    function Subtract returns Diff : Int
      parameters
        preserves i : Int
        preserves j : Int
      end parameters
      requires "min_int ≤ Math_Sub(i,j) ≤ min_int"
      ensures "Diff = Math_Sub(i,j)"
    end Subtract

    function Multiply returns Prod : Int
      parameters
        preserves i : Int
        preserves j : Int
      end parameters
      requires "min_int ≤ Math_Mult(i,j) ≤ max_int"
      ensures "Prod = Math_Mult(i,j)"
    end Multiply

    function Divide returns Quo : Int
      parameters
        preserves i : Int
        preserves j : Int
      end parameters
      requires "(j ≤ 0) ⇒
                (Math_Mult(j,Math_Add(max_int,1)) < i and
                i < Math_Mult(j,Math_Sub(min_int,1))"
      ensures "Math_Abs(Math_Mult(j,Quo)) ≤ Math_Abs(i) and
                Math_Abs(Math_Sub(i,Math_Mult(j,Quo))) ≤
                Math_Abs(j)"
    end Divide

    control Less_Than_Or_Equal
      parameters
        preserves i : Int
        preserves j : Int
      end parameters
      ensures Less_Than_Or_Equal iff "i ≤ j"
    end Less_Than_Or_Equal
  end interface

end Bounded_Integer_Template
```

Constants min_int and max_int, which represent the minimum and maximum representable value, respectively, are defined by a realization of Bounded_Integer_Template. These constants are part of the auxiliary section of the conceptualization, and are used only to define operations and types in the conceptualization. Specifically, they are not directly available to a client. However, Bounded_Integer_Template defines functions Get_Min_Int and Get_Max_Int that return the minimum and maximum integer values.

The auxiliary section of Bounded_Integer_Template contains a constraint clause, which is an invariant (i.e., an assertion that is true at all times). In this respect constraints are similar to lemmas, discussed in Section 3.5.3. Unlike lemmas, constraints cannot be proved using other assertions defined in the conceptualization. Rather, the realization guarantees that constraints are met at all times.

Finally, a brief explanation of the definition of procedure Divide is in order. First, it is useful to rewrite the requires and ensures clauses using standard infix notation:

```
requires "(j ≤ 0) ⇒ (j·(max_int + 1) < i < j·(min_int - 1))"
ensures "|j·Q| ≤ |i| and |i - j·Q| < |j|"
```

The requires clause places two restrictions on the parameters — the divisor cannot be zero, and the quotient must be between min_int and max_int, inclusive. The first restriction (i.e., division by zero) is expressed by the fact that when j is zero, no value of i satisfies the requires clause. The second restriction (i.e., representable quotient) is interesting only when j is negative, for when j is positive the quotient is representable since it is between the dividend and zero, inclusive. Thus, the requires clause restricts the parameters only when j is less than or equal to 0.

3.6.2 Conceptualization Single_Link_Ref_Template

Conceptualization Single_Link_Ref_Template, presented in Figure 22, provides type Reference that is useful for implementing traditional "singly-linked" data structures involving nodes consisting of information and a next field. The motivation for presenting this example is twofold. First, it declares conceptualization auxiliary variables that hold module state information, and second it demonstrates that it is

possible to formally define a module providing the functionality traditionally associated with "pointers" even though RESOLVE does not define pointers as a built-in type.

```
conceptualization Single_Link_Ref_Template
  parameters
    type Item
  end parameters

  auxiliary
    math facilities
      Number_Theory is Number_Theory_Template
        renaming
          Number_Theory.Integer as Integer
        end renaming

      Function_Theory_1 is Function_Theory_Template
        (Integer,math[Item])
        renaming
          Function_Theory_1.Function as Integer_To_Item
          Function_Theory.Delta as Delta
        end renaming

      Function_Theory_2 is Function_Theory_Template
        (Integer,Integer)
        renaming
          Function_Theory_2.Function as Integer_To_Integer
        end renaming
    end math facilities

    math variables
      Unused : Integer
      Info : Integer_To_Item
      Next : Integer_To_Integer
    end math variables

    initially "Unused = 1 and
               ∀i:Integer, Item.init(Info(i)) and Next(i) = 0"

    lemma "Unused ≥ 0 and ∀i:Integer, (i ≤ 0 or i ≥ Unused) ⇒
           (Next(i) = 0 and Item.init(Info(i)))"
  end auxiliary
```

Figure 22

Specification for a Module Providing Generic Type Reference

Figure 22 (continued)

```
interface
  type Reference is modeled by Integer
    exemplar r
    initially "r = 0"
    initialization
      ensures "Unused = #Unused and
               Info = #Info and Next = #Next"
    finalization
      ensures "Unused = #Unused and
               Info = #Info and Next = #Next"
    lemma "r ≥ 0"
  end Reference

  procedure New_Ref
    parameters
      alters r : Reference
      consumes x : Item
    end parameters
    ensures "r = #Unused and Unused = #Unused + 1 and
             Delta(Info,{r}) and Info(r) = #x and
             Next = #Next"
  end New_Ref

  procedure Swap_Info
    parameters
      preserves r : Reference
      alters x : Item
    end parameters
    requires "r ≠ 0"
    ensures "Delta(Info,{r}) and Info(r) = #x
             and x = #Info(r) and Unused = #Unused
             and Next = #Next"
  end Swap_Info

  procedure Advance_Next
    parameters
      alters r : Reference
    end parameters
    ensures "r = Next(#r) and Unused = #Unused and
             Info = #Info and Next = #Next"
  end Advance_Next

  procedure Change_Next
    parameters
      preserves r1 : Reference
      preserves r2 : Reference
    end parameters
    requires "r1 ≠ 0"
    ensures "Delta(Next,{r1}) and Next(r1) = r2 and
             Info = #Info and Unused = #Unused"
  end Change_Next
```

Figure 22 (continued)

```
      procedure Copy
        parameters
          preserves r1 : Reference
          alters r2 : Reference
        end parameters
        ensures "r2 = r1 and Unused = #Unused and
                 Info = #Info and Next = #Next"
      end Copy

      control Equal
        parameters
          preserves r1 : Reference
          Preserves r2 : Reference
        end  parameters
        ensures Equal iff "r1 = r2"
      end Equal

      control Is_Null
        parameters
          preserves r : Reference
        end  parameters
        ensures Is_Null iff "r = 0"
      end Is_Null
    end  interface

    description
      . . .
    end  description

  end Single_Link_Ref_Template
```

A variable of type Reference is modeled as a mathematical Integer. A Reference variable initially contains zero (which corresponds to 'nil' in the traditional view of pointers). Information (of type Item) and a next reference (of type Reference) are associated with each Integer by mathematical functions Info and Next, respectively. These functions must be defined globally to the module instance rather than locally to each Reference variable since they define the mappings for *all* Integers (i.e., all variables of type Reference). In other words, these functions (along with other items discussed shortly) define the state of the module instance, which is altered by operations provided by the facility.

The conceptual state of a facility is defined by variables declared in the auxiliary section of the conceptualization. In this example the state is defined by one Integer (Unused) and two functions over Integers (Info and Next).

Unused is the smallest positive Integer that has never been the value of a Reference variable. At module initialization, Unused = 1, and its value is non-decreasing throughout execution of the client. The actual value of Unused is not really important. What matters is that, at any given time during execution, no Reference variable has a value as large as Unused. This implies that when a Reference is given a new value by New_Ref, it is certainly a value not equal to any other Reference value at that time. Keeping track of Unused is simply one way to guarantee this property.

Info is a mapping from Integers to Items that associates each Reference value with some Item value. (In the traditional view of pointers, Info(p) corresponds to the data field of the record pointed to by p.) Info is formally defined as a total function, although only a portion of its domain is actually accessible.

Next is a function from Integers to Integers that associates each Reference value with another Reference value. (In the traditional view of pointers, Next(p) corresponds to the next field of the record pointed to by p.) Next is defined as a total function, although only a portion of its domain is actually accessible.

The initially clause in the auxiliary section is an assertion involving auxiliary variables that is true before any variable of type Reference is initialized, and before any operation defined by the conceptualization is invoked. In this example, Unused = 1, Info maps all Integers to an initial value of type Item, and Next maps all Integers to zero. Initialization code in the realization is responsible for accomplishing module-level initialization.

The initially clause for Reference indicates that each Reference variable initially contains zero, which corresponds to 'nil' in the traditional view of pointers. The initialization ensures clause specifies that the state of the facility does not change when a variable is initialized.

The distinction between these two clauses is subtle yet important. The initially clause is an assertion about the contents of a Reference variable, and is referenced in other

assertions as `Reference.init`. This assertion may contain references to state variables, but cannot reference 'old' and 'new' variable values (i.e., it may not reference variables with a '#'). The actual initialization of a variable is accomplished by executing an initialization routine defined in the realization (which is invoked automatically). The post condition of this routine is the conjunction of the initially clause with the initialization ensures clause. The initialization ensures clause relates the state after initialization to the state before initialization, and thus may reference 'old' and 'new' state variable values.

Variable finalization occurs when a variable is no longer accessible, such as at the end of the block in which the variable is declared. Actual finalization is accomplished by a finalization routine defined in the realization (which is invoked automatically). The post condition of this routine is specified in the finalization ensures clause, which relates the state before a variable is finalized to the state after. In this example, variable finalization does not change the state.

Variable finalization has no conceptual effect on the variables being finalized (since they are no longer accessible). The reason for including finalization in RESOLVE is to give a realization the opportunity to reclaim resources (such as memory) allocated to variables no longer needed.

Seven operations are defined by Single_Link_Ref_Template. New_Ref returns an Integer that has never been assigned, and sets Info to map to the Item passed. Swap_Info exchanges an Item with the Info associated with a Reference. Advance_Next advances a Reference to its Next reference. Change_Next changes the Next mapping of a Reference to another Reference. Copy makes a copy of a Reference. Equal and Is_Null return indications of equal References and a Null Reference, respectively.

3.6.3 Conceptualization ADO_Stack_Template

Modules designed using conventional "object-oriented" design guidelines conceptually incorporate the data object within the conceptualization. Although it is not recommended to design modules this way, these modules can be specified in RESOLVE using

conceptualization variables, as demonstrated by conceptualization ADO_Stack_Template, presented in Figure 23.

```
conceptualization ADO_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

    variables
      Stack : String
    end variables

    initially "Stack = Lambda"
  end auxiliary

  interface
    procedure Push
      parameters
        consumes x : Item
      end parameters
      ensures "Stack = Post(#Stack, #x)"
    end Push

    procedure Pop
      parameters
        produces x : Item
      end parameters
      requires "Stack ≠ Lambda"
      ensures "#Stack = Post(Stack, x)"
    end Pop

    control Is_Empty
      ensures Is_Empty iff "Stack = Lambda"
    end Is_Empty
  end interface

end ADO_Stack_Template
```

Figure 23

Specification for a Module Providing an "Object-Oriented" Stack

The similarities between this and Stack_Template (presented in Figure 2) are obvious. The differences are also quite obvious — every instance of ADO_Stack_Template has its own stack, and invoking the operations from an instance effects only that stack. It is also not possible to swap two ADO_Stacks or to pass an ADO_Stack as a parameter to an operation since there are no stack variables available to a client.

3.6.4 Summary

This section discussed conceptualization constants, variables, module initialization, and type finalization. Constants permit a realization to provide conceptual information to the client. Conceptual variables contain the state of a module instance, whose initial state is specified by an initially assertion. These variables can be used to implement "object-oriented" modules, though this design is not recommended. Finally, the effect that variable initialization and finalization has on the state of the facility is specified in the initialization ensures and finalization ensures clauses of the type definition.

These constructs were presented by discussing three examples — Bounded_Integer_Template that defines computational integers, Single_Link_Ref_ Template that defines a structure corresponding to traditional "singly-linked" data structures, and ADO_Stack_Template that defines an object-oriented stack. These examples also demonstrate how types traditionally built-in to languages can be formally specified using the same mechanism used to define "user-defined" types.

3.7 Realization Parameters, Constants, and Variables

There are situations where it is necessary for a client to pass information to the realization when an instance is declared, possibly in addition to what was passed to the conceptualization. This is accomplished by means of realization parameters in RESOLVE.

Also, it is often necessary for a facility to maintain realization state information during execution. In RESOLVE, this is accomplished by declaring variables and/or constants within the realization auxiliary section of a realization.

It is important to understand that realization parameters and realization state variables do not affect the conceptual behavior of the facility. For example, changing an actual realization parameter cannot alter the correctness of a client (assuming, of course, that the actual realization parameter is syntactically legal). These constructs are necessary only so performance goals can be met, and for other reasons dealing with *how* a conceptualization is implemented, not *what* it does.

This section discusses these RESOLVE constructs by presenting realization examples. Realization parameters are demonstrated by realization Copy_Stack_Real_1 for conceptualization Copy_Stack_Template, and realization state variables are demonstrated by realization List_Real_1 for conceptualization One_Way_List_Template.

3.7.1 Realization Copy_Stack_Real_1 of Copy_Stack_Template

Conceptualization Copy_Stack_Template, presented in Figure 18, defines procedure Copy_Stack that copies one stack into another, even if the implementations of the two stacks are different. Realization Copy_Stack_Real_1 for this conceptualization is presented in Figure 24.

```
realization of Copy_Stack_Template by Copy_Stack_Real_1

  conceptualization  parameters
    renaming
      Stack_Facility1.String_Theory.Reverse as Reverse
      Stack_Facility1.String_Theory.Cat as Cat
      Stack_Facility1.Item as Item
    end  renaming
  end  conceptualization  parameters

  realization  parameters
    procedure Copy_Item
      parameters
        preserves x : Item
        produces y : Item
      end  parameters
      ensures "y = x"
    end Copy_Item
  end  realization  parameters
```

Figure 24

Realization Copy_Stack_Real_1 of Copy_Stack_Template

Figure 24 (continued)

```
    interface
      procedure Copy_Stack
        parameters
          preserves s1 : Stack_Facility1.Stack
          produces s2 : Stack_Facility2.Stack
        end parameters

        begin
          local variables
            garbage : Stack_Facility2.Stack
            catalyst : Stack_Facility1.Stack
            temp : Item
            temp_copy : Item
          end local variables

          garbage :=: s2

          ensuring "catalyst = Cat(#catalyst,Reverse(#s1)) and
                    s1 = Lambda"
          while not Stack_Facility1.Is_Empty(s1) do
            Stack_Facility1.Pop (s1,temp)
            Stack_Facility1.Push (catalyst,temp)
          end while

          ensuring "s1 = Cat(#s1,Reverse(#catalyst)) and
                    s2 = Cat(#s2,Reverse(#catalyst))"
          while not Stack_Facility1.Is_Empty(catalyst) do
            Stack_Facility1.Pop (catalyst,temp)
            Copy_Item (temp,temp_copy)
            Stack_Facility1.Push (s1,temp)
            Stack_Facility2.Push (s2,temp_copy)
          end while
      end Copy_Stack
    end interface

    description
      . . .
    end description

  end Copy_Stack_Real_1
```

Procedure Copy_Stack must be able to make a copy of a value of type Item. Since nothing is known about this type, the client provides a copy procedure to the realization via realization parameter Copy_Item. The formal parameter declaration consists of the procedure signature (i.e., types and modes of all parameters) as well as a specification.

The compiler checks the actual procedure passed against the formal signature and specification.

The conceptual parameters section in a realization allows conceptual parameter items to be renamed for convenience. Additional conceptual parameters cannot be specified in this section!

It is also important to note that all items declared in the conceptualization can be referenced in the realization. For example, Stack_Facility1 is an instance of Stack_Template passed as a conceptual facility parameter, therefore procedure Stack_Facility1.Push is accessible within Copy_Stack_Real_1.

A client of Copy_Stack_Real_1 was presented earlier in Figure 19.

3.7.2 Realization List_Real_1 of One_Way_List_Template

Conceptualization One_Way_List_Template, presented earlier in Section 3.1.8 and Figure 4, defines a structure useful for storing information that is accessed sequentially in one direction only. Realization List_Real_1, presented in Figure 25, implements this conceptualization using Single_Link_Ref_Template, presented in Figure 22.

```
realization of One_Way_List_Template by List_Real_1

  realization  auxiliary
    facilities
      Item_Ref_Facility is Single_Link_Ref_Template (Item)
        realized by Single_Link_Real_1
        renaming
          Item_Ref_Facility.Reference as Item_Ref
          Item_Ref_Facility.New_Ref as New_Ref
          Item_Ref_Facility.Swap_Info as Swap_Info
          Item_Ref_Facility.Advance_Next as Advance_Next
          Item_Ref_Facility.Change_Next as Change_Next
          Item_Ref_Facility.Copy as Copy
          Item_Ref_Facility.Equal as Equal
          Item_Ref_Facility.Is_Null as Is_Null
        end  renaming
```

Figure 25

Realization List_Real_1 of One_Way_List_Template

Figure 25 (continued)

```
    Record_Facility is Record_2_Template (Item_Ref,Item_Ref)
      realized by Record_2_Real_1
      renaming
        Record_Facility.Record as List_Rep
        Record_Facility.Access_1 as Access_PreFirst
        Record_Facility.Access_2 as Access_Prev
      end renaming

    StringerRef is Single_Link_Ref_Template (Item_Ref)
      realized by Single_Link_Real_1
  end facilities

  variables
    Avail : StringerRef.Reference
  end variables

  operations
    procedure Get_Ref
      parameters
        produces r : Item_Ref
        consumes x : Item
      end parameters
      ensures "r ≠ 0"
      begin
        local variables
          FirstList : Item_Ref
        end local variables

        if StringerRef.Is_Null(Avail) then
          New_Ref (r)
        else
          StringerRef.Swap_Info (Avail,FirstList)
          Copy (FirstList,r)
          Swap_Info (r,x)
          Advance_Next (FirstList)
          if not Is_Null(FirstList) then
            StringerRef.Swap_Info (Avail,FirstList)
          else
            StringerRef.Advance_Next (Avail)
          end if
        end if
      end Get_Ref
    end operations
end realization auxiliary
```

Figure 25 (continued)

```
interface
  type List is represented by List_Rep
    exemplar L_rep
    correspondence "---"

    initialization
      performance "O(1)"
      begin
        local variables
          PreFirst : Item_Ref
          Prev : Item_Ref
          Init_Item : Item
        end local variables

        Get_Ref (PreFirst,Init_Item)
        Copy (PreFirst,Prev)
        Access_PreFirst (L_rep,PreFirst)
        Access_Prev (L_rep,Prev)
    end initialization

    finalization
      performance "O(1)"
      begin
        local variables
          FreeList : StringerRef.Reference
          PreFirst : Item_Ref
        end local variables

        Access_PreFirst (L_rep,PreFirst)
        StringerRef.New_Ref (FreeList,PreFirst)
        StringerRef.Change_Next (FreeList,Avail)
        FreeList :=: Avail
    end finalization
  end List

  procedure Reset
    parameters
      alters L : List
    end parameters
    performance "O(1)"

    begin
      local variables
        PreFirst : Item_Ref
        Prev : Item_Ref
      end local variables

      Access_PreFirst (L,PreFirst)
      Copy (PreFirst,Prev)
      Access_PreFirst (L,PreFirst)
      Access_Prev (L,Prev)
  end Reset
```

Figure 25 (continued)

```
procedure Advance
  parameters
    alters L : List
  end parameters
  performance "O(1)"

  begin
    local variables
      Prev : Item_Ref
    end local variables

    Access_Prev (L,Prev)
    Advance_Next (Prev)
    Access_Prev (L,Prev)
end Advance

procedure Add_Right
  parameters
    alters L : List
    consumes x : Item
  end parameters
  performance "O(1)"

  begin
    local variables
      newItem : Item_Ref
      Prev : Item_Ref
      Curr : Item_Ref
    end local variables

    Access_Prev (L,Prev)
    Copy (Prev,Curr)
    Advance_Next (Curr)

    Get_Ref (newItem,x)
    Change_Next (newItem,Curr)
    Change_Next (Prev,newItem)

    Access_Prev (L,Prev)
end Add_Right
```

Figure 25 (continued)

```
procedure Remove_Right
  parameters
    alters L : List
    produces x : Item
  end parameters
  performance "O(1)"

  begin
    local variables
      Prev : Item_Ref
      Curr : Item_Ref
    end local variables

    Access_Prev (L,Prev)
    Copy (Prev,Curr)
    Advance_Next (Curr)

    Swap_Info (Curr,x)
    Advance_Next (Curr)
    Change_Next (Prev,Curr)

    Access_Prev (L,Prev)
end Remove_Right

procedure Swap_Rights
  parameters
    alters L1 : List
    alters L2 : List
  end parameters
  performance "O(1)"

  begin
    local variables
      Prev1 : Item_Ref
      Curr1 : Item_Ref
      Prev2 : Item_Ref
      Curr2 : Item_Ref
    end local variables

    Access_Prev (L1,Prev1)
    Copy (Prev1,Curr1)
    Advance_Next (Curr1)
    Access_Prev (L2,Prev2)
    Copy (Prev2,Curr2)
    Advance_Next (Curr2)

    Change_Next (Prev1,Curr2)
    Change_Next (Prev2,Curr1)

    Access_Prev (L1,Prev1)
    Access_Prev (L2,Prev2)
end Swap_Rights
```

Figure 25 (continued)

```
      control At_Left_End
        parameters
          preserves L : List
        end parameters
        performance "O(1)"

        begin
          local variables
            PreFirst : Item_Ref
            Prev : Item_Ref
            temp1 : Item_Ref
            temp2 : Item_Ref
          end local variables

          Access_PreFirst (L,PreFirst)
          Access_Prev (L,Prev)
          Copy (PreFirst,temp1)
          Copy (Prev,temp2)
          Access_PreFirst (L,PreFirst)
          Access_Prev (L,Prev)
          if Equal(temp1,temp2) then
            return yes
          else
            return no
          end if
      end At_Left_End

      control At_Right_End
        parameters
          preserves L : List
        end parameters
        performance "O(1)"

        begin
          local variables
            Prev : Item_Ref
            Curr : Item_Ref
          end local variables

          Access_Prev (L,Prev)
          Copy (Prev,Curr)
          Advance_Next (Curr)
          Access_Prev (L,Prev)
          if Is_Null(Curr) then
            return yes
          else
            return no
          end if
      end Is_Empty
    end interface

end List_Real_1
```

List_Real_1 implements one-way lists using a standard "linked-list" representation discussed in most data structures texts. The items contained in a List are stored in a linked structure implemented using conceptualization Single_Link_Ref_Template. A List is represented by a record containing two references — PreFirst and Prev. PreFirst is a reference to the "dummy" node that is at the head of every linked list, and Prev is a reference to the item to the left of the fence.

Part of the complexity of the above realization is a result of implementing all operations to execute in constant time, including List initialization and finalization. This is a noble and worthwhile goal for any realization, and many issues are raised in reaching it. For instance, because every variable is initialized and finalized, it is especially important that these operations have efficient implementations. Initialization is normally not difficult to implement in constant time because the initial value for a type is usually defined to be one that is easy to construct.

Finalization is another matter because the finalization routine has no control over the values it must finalize. For example, one-way list finalization must be able to finalize empty lists as well as lists containing any number of items. Also, if the value is composite (e.g., a one-way list of Items) all components should also be finalized when the structure is finalized. Algorithms for constant time finalization are seldom obvious.[35]

Realization List_Real_1 accomplishes constant time finalization by using a trick that amortizes the cost of finalizing all items on a List over subsequent List insert operations. When a List is finalized it is placed on an internal list that contains finalized Lists, which requires a constant amount of time. This internal list is implemented by variable Avail. When an Item is inserted onto a List (by the client invoking Add_Right), an item from a previously finalized list is finalized if there is one, and its reference is reused for the new Item. If there are no finalized lists, a new reference is obtained and used for the inserted Item. A complete discussion of this approach to constant time initialization and finalization can be found in [Harms 89a].

---

35 Of course, since finalization usually has no conceptual effect, it need not do anything, which takes constant time! This approach makes no attempt to recover resources (such as memory) used to represent inaccessible variables, and is therefore neither advisable nor realistic.

Several interesting features of RESOLVE realizations are demonstrated by List_Real_1. Avail is declared as a realization auxiliary variable, and contains realization state information for the facility. It is a static variable in the C sense of the word, meaning its lifetime is for the entire program. Procedure Get_Ref is a local procedure to the realization. As is true of all items declared in the realization auxiliary section, Avail and Get_Ref are accessible only within the realization.

The initialization and finalization routines for Lists are defined within the type declaration. These routines are invoked automatically, and their purpose is to create an initial value and reclaim memory occupied by an inaccessible value, respectively.

3.7.3 Other Realization Sections

Although most of the sections of RESOLVE realizations are demonstrated by the examples in this chapter, several are not. For instance, a realization may need code to initialize the realization state of the facility. In realization List_Real_1 (presented in Figure 25) Avail is the only state variable, and it just happens that an initial value for its type is exactly what is needed as the initial state, so facility initialization code was not needed. However, this is not the case in general, so it is possible to define a facility initialization routine within the realization auxiliary section, which is automatically executed when the program begins execution.

It is also possible to declare constants within the realization auxiliary section. Constants are similar to variables, except that their value can be changed only during facility initialization (i.e., only by the facility initialization routine); to all other routines constants are "read-only" variables, and can only be used as actual preserves parameters.

In addition to operations, facilities and types can be passed as realization parameters. These kinds of parameters are useful for parameterizing performance characteristics of realizations, and are discussed more fully in [Muralidharan 90].

3.7.4 Summary

In this section some very interesting and powerful features of RESOLVE realizations were discussed — realization parameters, realization state variables, and initialization and finalization of types. This discussion centered around two non-trivial realizations for conceptualizations presented earlier — realization Copy_Stack_Real_1 for Copy_Stack_Template, and realization List_Real_1 for One_Way_List_Template.

3.8 Implementation Issues

One of the primary goals of RESOLVE is to provide the programmer with mechanisms enabling him or her to create efficient implementations of formally specified (and verifiable) software components. We have already discussed some implementation issues, such as implementing swap as a constant time operation in Section 3.3.1.2, and constant time initialization and finalization in Section 3.6.2.

In this section we'll examine three additional implementation issues — implementing primitive conceptualizations, lazy initialization, and efficient implementations of generic conceptualizations.

3.8.1 Primitive Realizations for Conceptualizations

Recall that all types referenced in a RESOLVE program must be formally defined in a conceptualization module. This holds even for types that are built-in to most languages, such as integers, characters, and booleans. In other words, there are absolutely no built-in types in RESOLVE. This approach to built-in types simplifies the definition of the language by making it very regular, and allows the programmer to select realizations that are appropriate for the constraints and performance goals of the particular client.

Realizations are a somewhat different matter. There must be a set of "primitive" realizations upon which other realizations can be built. These primitive realizations are defined in compilation units such as machine language or a "system" dialect of RESOLVE. Using one of these realizations is no different than using a regular realization, since the actual realization code is completely hidden from the client.

The programmer will have a library of "standard" realizations for commonly used conceptualizations such as integers, characters, booleans, and arrays. These will most likely be realizations built directly on the hardware, though this fact is transparent to the programmer.

3.8.2 Lazy Initialization of Variables

As mentioned in Section 3.6.2, it is desirable to implement type initialization and finalization efficiently (preferably as a constant time operations) because every variable is automatically initialized at the beginning of the block in which it is declared and finalized at the end of the block. However, the initial value in many variables is never actually referenced before the variable is finalized. For example, the initial values in variables temp and temp_copy in Figure 24 are never referenced. It might be a desirable characteristic of an implementation of RESOLVE to not waste time and memory initializing variables whose initial value is never referenced.

An implementation of RESOLVE might be able to take advantage of the fact that word addresses are even values in most modern machine architectures. If this is the case, the implementation could automatically place an odd value in each variable when it allocates memory for it. Let's assume that all variables are implemented as pointers to the actual representation of the value, and that representations always start on word boundaries. In this situation, all variables initially contain an illegal address that will cause a run-time trap when accessed. When a trap occurs, the trap routine invokes the appropriate initialization routine for the variable, and stores the address of the actual representation in the variable. All subsequent access to the variable will be legitimate.

When a variable is to be finalized, the implementation checks it for an odd value. If it is odd, the variable was never actually initialized, and thus does not need to be finalized. On the other hand, if the variable contains an even value, the appropriate finalization routine is invoked.

This implementation trick does not violate any conceptualization assertions that state all variables have an initial value. Indeed, the first time a variable's value is accessed, it is an initial value, which is exactly what the initially clause is for.

3.8.3 Efficient Implementation of Generic Modules

Implementing all RESOLVE variables as pointers has several advantages when coupled with the fact that the only data movement primitive is swapping the value of two variables. One advantage, discussed in Section 3.3.1.2, is that swap can be implemented as a constant time operation. Another advantage, discussed here, is that the object code for a realization can be shared by all instances of it.

Consider the object code produced by the swap statement 'x :=: y' where variables x and y are implemented as pointers. In most machine architectures this is effected by three MOVE instructions, where each MOVE moves a fixed-size bit sequence (e.g., a 32-bit address). Note that these statements do not depend on the type of x and y. So even if the type of x and y are unknown to the compiler (which is the case with type parameters in generic modules) it can still produce object code. Extending this one step further, there is no reason that each instance utilizing a particular realization must have its own copy of the object code, since the object code is identical.

The only type-specific operations that can be invoked within the generic facility are initialization and finalization of the type parameter. This can be easily implemented by creating a run-time structure for each facility that contains, among other items, the addresses of initialization and finalization routines for all types referenced in the module. When the initialization or finalization routine for a type parameter needs to be invoked, the instance-specific structure is used at execution time to invoke the proper routine.

This and other RESOLVE run-time structures are discussed more fully in [Hegazy 89].

3.9 Summary

RESOLVE is a programming language designed specifically to encourage the design and implementation of reusable software components, whose characteristics are described in Section 2.3. Specifically, each component's behavior is formally described using assertions in mathematical theories (that are themselves formally defined in theory modules). Every program type is modeled as a mathematical type, and every program variable is considered to be a value from the type's mathematical type for purposes of reasoning about behavior of the variable. Operations are formally specified with

requires and ensures clauses that specify the pre- and post-conditions, respectively. Although it is possible (and strongly encouraged) to also describe each component informally, the formal description is always used as the "final word" concerning the behavior.

The behavior of a component is specified in a conceptualization while the structures and code that implement a conceptualization are contained in a realization, thus separating component specification from implementation. In addition, it is possible to have multiple realizations for any conceptualization. A client gains access to the types and operations defined in a conceptualization by instantiating it (also called declaring a facility). A facility declaration binds a conceptualization with a realization, and binds actual with formal parameters. A different realization may be bound to each instance of a particular conceptualization .

A conceptualization is generic if it has one or more type parameters. It is thus possible to define structures where the type of certain components is passed as a parameter.

RESOLVE modules can be formally verified using techniques described in [Krone 88]. Verification is feasible here because of several characteristics of RESOLVE modules. First, all types and operations have formal mathematical specifications. Also, every type has a defined initial value, so the verification system need not incorporate the notion of an "undefined" value, thus simplifying verification. Second, all loop constructs have an associated assertion that formally defines the specification of the loop, which is similar to operation specification. Third, pointers are not a built-in type in RESOLVE (although they can be defined via a conceptualization), eliminating the possibility of aliasing and other problems relating to pointers, such as "dangling references," that typically complicate verification systems.

Finally, it is possible to efficiently implement RESOLVE modules, even generic ones. This is due largely to the fact that the only data movement primitive in RESOLVE is swapping the values of two variables, which takes constant time. All operation parameters are defined in terms of swapping actual with formal parameters, so parameter passing also takes constant time. Copying a value is accomplished by an operation defined by a conceptualization just like all other operations. It may not be possible to copy every type of value since RESOLVE does not require a copy operation be defined

for every type. Thus, generic modules are usually designed in such a manner that values are swapped into the structure, rather than copied, because the generic type may not have a copy operation.

RESOLVE supports the second point of the thesis, namely that it is possible to have a usable language incorporating constructs that encourage the design of reusable software parts.

CHAPTER IV

Interaction of Programming Language and Environment Design

A programming language and its program development environment (which includes, for example, components to create, modify, verify, compile, link, execute, and debug programs) are interrelated in the sense that the ease with which programs can be developed depends upon the facilities provided by the environment, which in turn depend upon the constructs defined in the programming language. For example, a structure editor is a useful tool to develop programs, but only if the programming language is block-structured.

This chapter investigates some of the influences that programming languages and editing environments have on each other. We'll start by studying the effect of programming languages on environments, and then explore the influence of environments on languages. In particular we'll examine a RESOLVE editing environment designed to execute on Macintosh personal computers, and some of the ways its design influenced RESOLVE's design.

4.1 Programming Language Influence on Environmental Design

Perhaps the biggest impact programming languages have had on editing environments in recent years has been the development of structure and syntax directed editors. Using a structure editor, the document being edited is considered to be a structure (e.g., a tree), not just a sequence of characters. Moving around the document is accomplished with commands such as "move to next sibling" or "move to parent." Creation and modification of a document is done by modifying the document's structure. A structure editor is useful for editing any structured document, such as programs and manuscripts.

Mentor [Donzeau-Gouge 84a, Donzeau-Gouge 84b] and Tioga [Teitelman 85, Swinehart 86] are examples of structure editors.

Syntax directed editors extend structure editors in that the structure imposed on the document is the structure of some programming language [Meyrowitz 82]. In other words, syntax-directed editors are structure editors that enforce (or at least have knowledge of) a particular programming language. The Cornell Program Synthesizer [Teitelbaum 81], Gandalf [Habermann 86] (including GNOME [Garlan 84] and GENIE [Chandhok 87, Chandhok 88]), and PECAN [Reiss 85] are examples of syntax directed editors.

Syntax directed editors have several potential advantages over traditional text editors. First, an editor can be designed in which it is impossible to create syntactically and static-semantically incorrect programs, thus reducing errors. Second, the editor can automatically take care of many of the mundane syntactic structures, such as keywords and "syntactic sugar", allowing more rapid program construction. Finally, the compiler is simplified if the editor produces an error-free parse tree; in fact, it is possible to have the editor translate the parse tree as it is created, eliminating the need for a separate compiler.

Despite the advantages of syntax directed editors, some researchers (e.g., [Waters 82]) argue that editors should have both structure and text capabilities. The reasons cited are that many programmers prefer to edit text, and that some editing (e.g., entering in-fix expressions and transforming an if-statement into a while-statement) are significantly easier to do by editing text. Others (e.g., [Cohen 82] and [Shani 83]) argue that the problems (if any) are in the existing structure and syntax directed editors, not the approach, and[Garlan 84] states that "students, as a whole, have *not* found the structure/infix mode of entry for expressions to be a problem" in referring to the GNOME environments. Also, the ability for "free typing" complicates the editor because it must include a parser to determine the structure of newly-entered text and insert it into the program structure; in addition the editor must specify what happens to text that is incorrect. Nonetheless, most structure and syntax directed editors include some ability for "free typing."

Although it is possible to develop a syntax directed editor for any programming language, they are most useful for "block structured" languages such as Pascal, Algol, and Ada. For instance, a syntax directed editor for BASIC would not be very useful because BASIC programs are organized as a sequence of statements rather than a tree. This suggests an important influence that a programming language has on its environment — whether or not a syntax directed editor is an appropriate tool for creating and modifying programs.

There are other ways that a programming language influences its environment. For instance, the hardware platform on which an environment executes is affected by the language's vocabulary. For example, environments that rely on standardized character sets for input and output (e.g., ASCII terminals) are not well suited for languages that use special symbols (e.g., APL).

Programming languages have influenced the design of machine architectures. Whereas early architectures were optimized for machine and assembly language programming, modern ones are designed for efficient implementation of programs coded in high level languages.

Also, a language that encourages development of separately-compiled modules (e.g., Ada, Eiffel, CLU, and RESOLVE) suggests that the environment include a component to help manage the module library. The librarian aids the programmer in selecting the appropriate module, and/or validates module use.

In fact, programming environments have evolved to a large extent in response to new developments in programming language design.

4.2 Environmental Influence on Programming Language Design

Historically, program development and execution environments have significantly influenced the design of programming languages. For example, the format of FORTRAN statements is directly related to the editing environment of the day — 80-column punch cards. Also, many of FORTRAN's control statements (e.g., the computed IF) are included in the language because of analogous statements in the machine language of the expected target computer, the IBM 709 [MacLennan 83]. The

goal of a simple one-pass compiler influenced the design of Pascal [Wirth 83], and more recently Ada includes only constructs whose implementation is clear and efficiently implementable using known techniques [DoD 83][36].

However, programming environments have not had as much influence on programming languages as programming languages have had on environments. It seems that often a language is completely designed before significant consideration is given to developing an environment for it, and therefore environmental concerns have little or no impact on the design of the language.

RESOLVE was designed concurrently with an editing environment, which influenced the syntax of the language. This editor is designed to meet several goals. For example, it is designed to execute on Macintosh personal computers, and takes advantage of a mouse and menus for syntax directed editing, makes minimal use of the keyboard (i.e., the keyboard is used only to name identifiers, and there is no option for "free typing" program text), and operates within a relatively small monochrome bit-mapped display. Also, the editor is responsible for processing several syntactic constructs that are usually defined in the syntax of contemporary programming languages, such as operator overloading and infix notation. (Portions of the editor have been implemented, and are described in Appendix A.)

Let's take a closer look at the editor's influence on the syntax of RESOLVE. First, keywords are inserted into a program using menus, and are never entered using the keyboard. Therefore, keywords in RESOLVE are not abbreviated because abbreviated keywords have no advantage over unabbreviated ones with respect to the ease of program construction, yet are potentially cryptic and confusing.

Second, because the display is narrow and the layout of the display is controlled by the editor (e.g., the programmer cannot insert carriage returns randomly in the program), RESOLVE's syntax is designed so statements do not become arbitrarily wide. For example, formal parameters are defined in a parameters section following the heading rather than within parentheses in the heading. Also, actual parameters are restricted to be identifiers, rather than permitting function invocations with potentially long actual parameter lists.

---

36 This demonstrates that terms such as "clear," "efficient," and "known" are relative!

Finally, RESOLVE's syntax does not include facilities for complex syntactic structures such as overloading and infix notation. Instead, these are handled by the editor using its "alternate syntax" feature. Operation invocations normally appear in a prefix syntactic form — the operation name is followed by the actual parameters enclosed in parentheses. Though this is adequate, some constructs are visually unattractive. For example, the usual assignment statement "`z := x + y`" is displayed as "`z := Add(x,y)`".

The RESOLVE editor allows any operation to have alternate syntax, which defines the format of all invocations of that operation. The alternate syntax for an operation is a sequence of characters which must include every formal parameter of the operation. The editor uses the alternate syntax definition as a template, substituting actual parameters for the formals in the alternate syntax definition.

For example, function Add defined in conceptualization Bounded_Integer_Template (see Figure 21 in Section 3.6.1) has two formal parameters — i and j — so every alternate syntax definition for Add must include i and j. Figure 26 demonstrates the effect of alternate syntax by showing several alternate syntax definitions for Add and the way the editor displays an example invocation.

Alternate Syntax	Example Invocation
(no alternate syntax)	z := Add(x,y)
"i + j"	z := x + y
"i j +"	z := x y +
"sum of i and j"	z := sum of x and y

Figure 26

Effect of Alternate Syntax

It is important to understand that alternate syntax only affects the way an invocation is *displayed* by the editor. Alternate syntax does not change the way a programmer inserts

an operation invocation into the program — she or he still uses the mouse and popup menus (see Appendix A).

This raises an interesting concern, though — what if the alternate syntax of two operations results in a program whose textual display is ambiguous? For example, if functions Add and Subtract both have "`i + j`" as alternate syntax, it is not clear from the display of a program whether the statement "`z := x + y`" is an invocation of Add or Subtract.

From the editor's viewpoint this ambiguity is not a problem. When the "ambiguous" statement was constructed the programmer explicitly indicated (via a popup menu) which operation to invoke, which is the operation inserted into the parse tree. In other words, the parse tree is constructed by the programmer using the editor; it is not created by parsing the textual form of a program. In fact, because the editor does not permit a program to be entered by "free typing," the textual form of the program is never parsed, and the fact that it might look ambiguous is not relevant for processing by the editor or compiler.

For obvious reasons of clarity it is not recommended that programmers define alternate syntax that might allow a program to be developed whose display is ambiguous. Therefore, when alternate syntax is defined the editor checks it for conflict with other definitions. If a conflict is detected, the programmer is appropriately warned, but is not prohibited from making the ambiguous definition.

In addition to permitting very general operator overloading, the alternate syntax facility can be used to display a procedure invocation to reflect its effect. For example, defining "`x :=: a[i]`" as alternate syntax of procedure Access in conceptualization Array_Template (see Figure 20 in Section 3.5.3) causes the invocation "`Access(arr,index,z)`" to be displayed "`z :=: arr[index]`", which accurately reflects the effect of the invocation as an apparent swap statement in which an array element acts like a variable.

Alternate syntax appears to be a very powerful syntactic mechanism, allowing the editor to effect syntactic constructs such as overloading and notation without complicating program parsing or the compiler.

4.3 Summary

In this chapter we've briefly explored the interaction between programming language design and program environment design. Some of the influences that programming languages have had on environments in recent years include structure and syntax directed editors, hardware platform designs for both program development and program execution, and the development of environment components such as librarians.

Although program development and execution environments influenced the design of early languages such as FORTRAN, their influence recently has been minimal. In fact it seems that environments are often developed after the programming language is designed. However, an editing environment was designed concurrently with RESOLVE, and influenced the language in several significant ways. In particular, RESOLVE keywords are not abbreviated, parameter lists are such that statements do not become arbitrarily wide, and several complex syntactic constructs such as operator overloading and infix notation are handled as alternate syntax by the editor rather than complicating the syntax of the language.

RESOLVE is an example of a programming language whose design was influenced in positive ways by the environment, and supports the third point of the thesis.

CHAPTER V

Conclusion

This chapter summarizes the research conducted for this dissertation, and presents the conclusions drawn from this work. Some open issues and future work are discussed, and the chapter concludes with a presentation of contributions to the field.

5.1 Summary and Conclusions

The following subsections summarize and draw conclusions from the three primary divisions of this work — reusability issues, definition of RESOLVE, and interaction between programming language and environment design. These subsections essentially summarize chapters 2, 3, and 4, respectively.

5.1.1 Software Reusability

Software quality and programmer productivity are two of the biggest challenges facing the software engineering community. Reusability, a mainstay of other engineering disciplines, is an approach to software development that addresses both of these issues. A designed-for-reuse software component is economically efficient to design and build, it most likely is of a higher quality than a "scavenged" part, and reusing it increases the productivity of client programmers. Despite these advantages, there are both technical and non-technical impediments to widespread software reuse.

A reusable software component is defined as one that can be incorporated into a variety of programs without modification (except possibly parameterization); furthermore, reusing a software component should not compromise other software engineering principles such as information hiding and data abstraction.

A reusable component has several important characteristics. First, it is formally specified by mathematical statements that provide a complete and unambiguous description of the component. Formal specification is important because it tells a potential client programmer exactly what the part does, it tells an implementor of the part exactly what the implementation must accomplish, it is necessary if formal verification is to succeed, and the process of writing a formal specification often helps reveal ambiguities and inconsistencies in the part's design. In addition to formal mathematical specification, it is important to include an informal natural language description of the component because it can be instrumental in helping a programmer (both client and implementation) understand the part's function.

A second characteristic of a well-designed reusable component is the separation of the part's specification and implementation into separately-compilable units. This is an effective method of realizing information hiding, it permits designers and programmers to consider a part's abstract function and performance as separate issues, it allows a specification to have multiple implementations, and it permits a client program to be compiled and verified before any implementation exists.

Third, a reusable component should be generic if possible. A generic part is one whose definition is parameterized somehow (e.g., by a component type). A generic part is a template for a family of parts, and must be instantiated to create a usable part. Generic parts achieve reusability in an obvious way.

Fourth, it should be possible for a specification to have multiple implementations, each with possibly different performance characteristics. This increases the likelihood that a part will be reused because clients place different performance requirements on a part. A client programmer should be allowed to choose an implementation for one part, and another implementation for a different instance of that same part. Also, it should be possible for a client programmer to select a different implementation of a part without having to recompile or reverify the client program.

A fifth and final characteristic of a well-designed reusable part is that efficient implementations must be possible, because a part that has no efficient implementation will be neither used nor reused. Perhaps the biggest implication of this is that an implementation should not rely on copying as a primary means of moving data.

Using these five characteristics, the features of a programming language can be characterized by how well they encourage and facilitate the design and implementation of reusable software components. For example, it is difficult to formally specify a program written in a language having implicit pointers and aliasing, and some parts have no efficient implementation in a language where copying is the only data movement primitive. A survey of nine representative modern programming languages — Anna/Ada, Modula-2, Euclid, Gypsy, Alphard, C++, Eiffel, Larch/CLU, and Z — reveals that none adequately supports the design and implementation of reusable components. This supports the first point of the thesis, namely that no modern programming language has all of the constructs necessary to encourage and facilitate the design of reusable software parts; in fact, existing languages have constructs (e.g., implicit pointers and copying as the only data movement primitive) that actually thwart the design of reusable software.

5.1.2 RESOLVE Programming Language

A language called RESOLVE (an acronym for REusable SOftware Language with Verifiability and Efficiency) was developed to demonstrate that it is possible to have a practical language incorporating constructs that encourage the design and implementation of reusable components. RESOLVE is an imperative language, with control structures similar to those found in most structured languages such as Pascal and Ada. A program is organized as a collection of separately-compiled modules. The behavior of each module is formally specified in a conceptualization, with the structures and code implementing each conceptualization contained in a realization. A conceptualization typically defines a type and a set of operations with one or more parameters of that type. Conceptualizations may be generic, and several realizations may exist for any conceptualization.

RESOLVE distinguishes between mathematical entities (types, domains, and variables) and program entities (types, domains, and variables). A mathematical domain is an anonymous set of anonymous values defined by an instance of a theory module, which defines a set of axioms that implicitly define the domain and a notation for expressing functions and relations among the values in this domain. A mathematical type is the name given to a mathematical domain by the client that instantiates the theory module. A

mathematical variable is a symbol that stands for some value of its type's domain. Two mathematical types are equivalent if and only if they are names for the same domain.

Similarly, a program domain is defined by an instance of a conceptualization. Each element in a program domain has a concrete representation (which may be as low-level as a configuration of bits in memory) and is modelled by an element from a corresponding mathematical domain. The elements in a program domain are those values that are "reachable" by executing the operations defined in the conceptualization. When a conceptualization is instantiated by a client, a marker is created for each program domain defined. A program type is the name for a marker, which in turn designates a program domain. Two program types are equivalent if and only if they are names for the same marker. A program variable is a symbol that stands for some value from the domain of its type; however, because every value in a program type's domain has a mathematical model, it is possible to *reason* about the value of a program variable as a mathematical value.

Program type equivalence and mathematical type equivalence are defined differently. Effectively, the mathematical types defined by two identical instances of a theory are equivalent, whereas the program types defined by two identical instances of a conceptualization are not equivalent; in fact, markers are introduced in the discussion of program types solely to make these two program types inequivalent. The justification for this is based in part upon our design philosophy that realizations determine performance characteristics of the operations and do not affect the contexts where those operations can be legally invoked. In other words, changing the realization of a facility should not alter the syntactic correctness of the module. Thus, program types defined by two identical instantiations are inequivalent, so that if the realization of one instance changes, the syntactic legality of the program can't change.

Every program type has an initial value specification defined for it. Every (program) variable is guaranteed to have a value that meets the initial value specification for its type before the variable's first reference. Initial values are important for the formal specification of a program, and to guarantee that variables always have some legitimate value from the domain of its type.

The effect of an operation is formally defined using pre- and post-conditions, which are assertions about the values of the operation's parameters before and after the operation invocation, written in first-order predicate calculus. Within these assertions parameters are considered to be values from the mathematical domain of the parameter's program type's model, and the assertions involve functions and relations in the theories defining those types. The pre-condition is specified in a requires clause, and is an assertion the operation assumes to be true when it is invoked. The post-condition is specified in an ensures clause, and is an assertion the operation guarantees to be true when it returns, provided the requires clause was true when the operation was invoked.

A realization contains the data structures and code that implement the program types and operations specified in a conceptualization. A correspondence is defined that maps values of a type's representation (considered as mathematical values) to the type's model. To accomplish initialization, an initialization routine is defined for every program type specified in the conceptualization, which is automatically invoked for every variable of that type. Also, variables whose representations involve dynamically-allocated memory should have that memory released when it is no longer needed. To accomplish this, a type has a finalization routine, which is automatically invoked for every variable of that type at the end of the block in which the variable is declared.

An interesting feature of RESOLVE is that no types are built-in to the language. Instead, every type is provided by some conceptualization, including those normally built-in such as integer, character, and boolean. Likewise, structured types such as arrays and records are not defined in RESOLVE, but are defined by conceptualizations. A similar approach is taken with respect to pointer variables. This design makes RESOLVE very regular. However, it also involves defining control operations, which are special in that they can be invoked only within if and while statements.

Another feature of RESOLVE is that data is moved by swapping the contents of two variables, rather than copying the contents of one variable to another. The ability to make a copy of a data value is not a built-in operation in RESOLVE, and RESOLVE does not define the traditional "copy" assignment statement. If it should be possible for a client to make a copy of a variable of a particular type, it is the responsibility of the writer of the conceptualization providing that type (or one that uses it) to specify an

operation that accomplishes this; a copy operation defined this way is invoked just like any other operation. All parameter passing is defined in terms of swapping actual parameters with corresponding formal parameters.

Defining swapping as the only data movement primitive offers two advantages over traditional languages defining copying as the data movement primitive — it is possible to implement swapping so it takes constant time to execute, and swapping guarantees that implicit aliasing never occurs. The former property is important for efficient implementation of generic components, and the latter for formal specification and verification.

RESOLVE is a programming language that has the constructs necessary to encourage and facilitate the design of reusable software parts, and is the "proof by demonstration" supporting the second point of the thesis, namely that it is possible to have a usable language incorporating constructs that encourage the design of reusable software components.

5.1.3 Interaction of Programming Language and Environment Design

A programming language and its program development environment are interrelated in the sense that the ease with which programs can be developed depends upon the facilities provided by the environment, which in turn depend upon the constructs defined in the programming language. Some influences that programming language design and environment design have on each other were explored. Structure and syntax directed editors, hardware platforms, and the development of environment components such as librarians are recent examples of the influence that programming language design has on programming environments.

Many recent programming languages were not significantly influenced by program environments. However, an editing environment was designed concurrently with RESOLVE, and influenced the language in several significant ways. In particular, RESOLVE keywords are not abbreviated, parameter and argument lists are such that statements do not become arbitrarily wide, and several complex syntactic constructs such as operator overloading and infix notation are handled as alternate syntax by the editor rather than complicating the syntax of the language. This is an example of a

programming language whose design was influenced in positive ways by the environment, and supports the third point of the thesis, namely that when a programming language and editing environment are designed at the same time, each can influence the other in positive ways.

5.2 Future Work

There are several areas that hold promise for productive future work involving both the language design and environment. Four of these are discussed here.

5.2.1 Enhancements

There are instances where a conceptualization, say A, is very similar to another one, say B, in the following sense: A provides identical definitions for all items defined in B, in addition to defining one or more new items. We could say that A enhances B. For example, a conceptualization defining type Stack with operations Push, Pop, Is_Empty, and Copy is similar to (i.e., an enhancement of) one defining everything but Copy.

Although this feature is included in many "object-oriented" languages, it is not currently in RESOLVE. The challenge is to define enhancements without sacrificing other desirable features of RESOLVE (e.g., formal specification and verifiability).

One proposal is to extend RESOLVE to allow a definition such as the following:

```
conceptualization Stack_With_Copy_Template
  enhances Stack_Template

  interface
    procedure Copy
      parameters
        preserves s1 : Stack
        produces s2 : Stack
      end parameters
      ensures "s2 = #s1"
  end interface

end Stack_With_Copy_Template
```

Conceptualization Stack_With_Copy_Template provides a client with all items defined in Stack_Template (i.e., type Stack and operations Push, Pop, and Is_Empty), plus operation Copy. An instance of Stack_With_Copy_Template can be used in any context

where an instance of Stack_Template is required (e.g., passed as an actual facility parameter to an instantiation); this is allowed because Stack_With_Copy_Template is a "superset" of Stack_Template.

Realizations of Stack_With_Copy_Template are obligated to implement *all* operations, including those defined in Stack_Template. Of course, a realization may want to implement some of these operations using a realization of Stack_Template, and syntax for this should be defined. Note, however, that new syntax is not required, as demonstrated here:

```
realization of Stack_With_Copy_Template by Stack_Copy_Real_1

  realization auxiliary
    facilities
      Stack_Fac is Stack_Template (Item)
        realized by Stack_Real_1
    end facilities
  end realization auxiliary

  interface
    type Stack is represented by Stack_Fac.Stack
      exemplar s_rep
      correspondence "s_rep = s"
    end Stack

    procedure Push
      parameters
        alters s : Stack
        consumes x : Item
      end parameters
      begin
        Stack_Fac.Push (s,x)
    end Push
      .
      .   similarly for Pop and Is_Empty
      .
    procedure Copy
      parameters
        preserves s1:Stack
        produces s2 : Stack
      end parameters
      begin
        .
        .   code to implement Copy in terms of Push, Pop, and
        .   Is_Empty
    end Copy
  end interface
end Stack_Copy_Real_1
```

Note that this realization has no more access to Stack_Real_1's representation of Stack_Template than any other client (i.e., information hiding is enforced).

There are several advantages of allowing enhancements. First, enhancements permit secondary operations to be encapsulated with the primary ones, allowing clients to reuse code for secondary operations. Second, very efficient implementation of some secondary operations may be possible if the realization implements all operations — both primary and secondary. For example, [Weide 86b] and [Pittel 90] describe an implementation of Copy for stacks that executes in constant time, but is not a secondary operation in the sense that it is not coded in terms of primary operations.

Enhancements appear to offer many of the advantages of inheritance, without adversely affecting the prospects for formal specification and verification.

5.2.2 Accessors

Currently in RESOLVE, accessing items within a structure (e.g., an array) is accomplished by invoking an operation that swaps an actual parameter with an item in the structure. For example, '`Access (a,i,x)`' swaps the ith element of array a with variable x. Swapping the jth element of a with the ith element requires three invocations of Access.

It might be useful to introduce a fourth kind of operation — an 'accessor' — into RESOLVE. An accessor would swap an item within a structure with some other value, and would be invoked within a swap statement. For example, an accessor called Arr_Access could be invoked by '`Arr_Access(a,i) :=: x`', which has the same effect as '`Access (a,i,x)`', or by '`Arr_Access(a,i) :=: Arr_Access(a,j)`', which swaps the ith with the jth element of a, or by '`Push(s,Arr_Access(a,i))`', which pushes the original value of the ith element onto stack s, leaving it with an initial value. In other words, an accessor could be invoked anywhere a variable is allowed.

5.2.3 Synoptic Comments

Comments often play a less-than-prominent role in programming language design, yet their importance in understanding and explaining a program should not be

underestimated. It might be useful to extend the notion of comment to be more than just white space. Specifically, the inclusion of "synoptic comments" that are explicitly associated with program code, similar to Kaelbling's scoped comments [Kaelbling 88], are proposed.

A synoptic comment would be bound to program structures as the program is constructed in the editor. A programmer would have several options regarding the display of synoptic comments and their associated code — both the comment and code could be displayed, only the comment and not the code, or only the code and not the comment. This would allow a programmer to hide irrelevant code when a comment summarizing the action of the code is sufficient.

Given an editor such as the one developed for RESOLVE, it might be possible to bind code to the synoptic comment by dragging a tool of some sort over the code. In this case, the binding is not textual, but visual.

5.3 Contributions

The focus of this research has been to investigate programming language features that either encourage or discourage the design of reusable software components — in other words, the influence of reusable software on programming language design. The following are the main contributions to the field:

Characterization of Reusable Software Components — Five characteristics of a well-designed reusable software component are: it is formally specified, its specification and implementation are in separately-compilable units, it is parameterized (i.e., generic) if possible, it may have multiple implementations, and it is possible to construct an efficient implementation. Evaluation of several modern programming languages with respect to how well they support components exhibiting these characteristics suggests that none has all of the features necessary to encourage and facilitate the design and implementation of reusable software components.

Definition of RESOLVE — A new programming language, called RESOLVE, is defined that has all of the necessary constructs to encourage and facilitate the design and implementation of reusable components. The next two contributions are actually features of RESOLVE.

Swapping as a Data Movement Primitive — Copying the contents of one variable to another is the only data movement primitive defined in modern programming languages, impacting the ability to develop efficient implementations of generic parts. In RESOLVE, the only data movement primitive is swapping the contents of one variable with another. Swapping is efficient to implement for all types (i.e., a constant time operation), and is safe in the sense that it does not introduce aliasing.

Simple Yet Powerful Definition of Types and Type Equivalence — Types and type equivalence (both mathematical and program) are defined in terms of abstract sets and functions. These definitions are unambiguous and powerful, yet easy to understand and implement.

Alternate Syntax — The alternate syntax mechanism allows the programmer to benefit from syntactic constructs such as overloading and infix notation without complicating the design of the programming language or its parser. This is accomplished by having the syntax directed editor process alternate syntax.

APPENDIX A

An Editing Environment for RESOLVE

The objectives in this project were to develop a RESOLVE structure editor for a small workstation that:

- permits only the creation of syntactically and static-semantically correct programs
- makes minimal use of the keyboard
- minimizes mouse movement

This appendix describes the prototype of this editor, implemented on the Macintosh. Two interesting characteristics are 1) the only use of the keyboard is to name an identifier, and 2) there is extensive use of hierarchical "popup" menus that minimize mouse movement.

This appendix is neither a user manual nor a tutorial, but rather a demonstration of the "look and feel" of the editor using a handful of sample editing sessions. A general knowledge of the Macintosh user interface is assumed (e.g., windowing, clicking, etc.).

Figure 27 is a snapshot of the Macintosh screen taken during an editing session for some realization module. The programmer has already instantiated two facilities — Int_Fac and Int_Stk_Fac[37] — and has declared a local operation called Copy_Stack that makes a copy of a variable of type Int_Stk_Fac.Stack. The programmer is in the process of creating the code for this procedure, using the algorithm presented in Figure 24.

---

37 The conceptualizations for these are given in Figures 21 and 2, respectively.

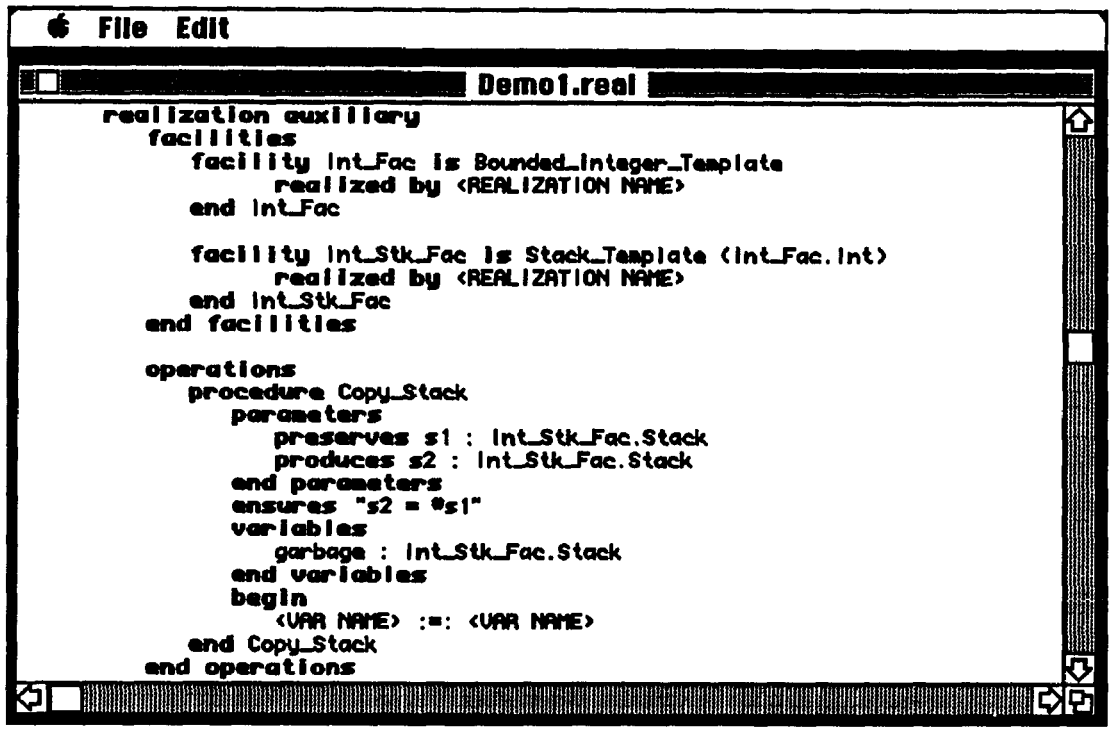

Figure 27

RESOLVE Editor Screen Snapshot

Items within a RESOLVE editor window are classified as *keywords*, *special symbols*, *placeholders*, *identifiers*, or *assertions*. Keywords appear bolded (e.g., **variables**), special symbols are sequences of one or more punctuation characters (e.g., :=:), placeholders begin and end with "<" and ">" (e.g., <VAR NAME>), identifiers appear as plain text (e.g., garbage), and assertions are surrounded by double quote marks (e.g., "s2 = #s1").

Editing is accomplished by pointing to an item in the edit window (using the mouse) and pressing the mouse button. If something interesting can be done with that item, a popup menu appears on the screen, providing the programmer with a set of allowable actions. The following subsections explore some interesting and useful actions.

Before we begin, it is interesting to note that there are only two menus in the main menu bar shown in Figure 27 — File and Edit. The contents of these menus is shown in Figure 28. The reason there are so few menus is the reliance on popup menus for editing, as demonstrated in the following subsections.

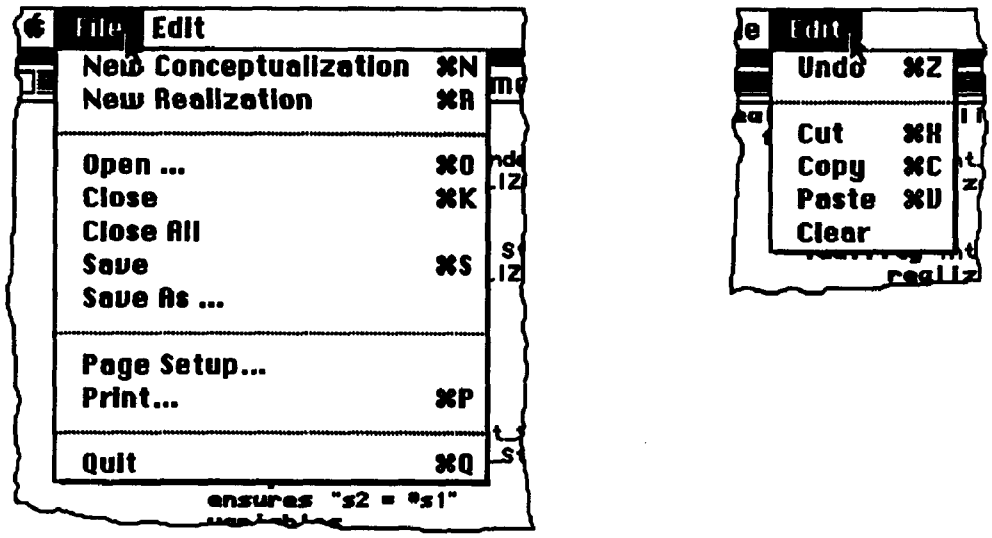

Figure 28

RESOLVE Editor Menus

A.1 <VAR NAME> Placeholders and Variables

Let's start with placeholders, which are items that must be "filled in" by the programmer before the program is considered complete. When the mouse is pressed in a placeholder, a popup menu appears that contains all legitimate replacements for the selected placeholder.

One such placeholder in Figure 27 is <VAR NAME>, which is a placeholder for a variable name. The popup menu for a <VAR NAME> placeholder contains all variables that can used there. If a type has already been bound to the placeholder (e.g., a swap statement with at least one variable selected), the popup menu contains only the variables of the

proper type; if a type has not been bound (e.g., a swap statement with neither variable selected), the popup menu contains all variables, organized hierarchically by type.

For example, Figure 29 shows the programmer replacing the left <VAR NAME> placeholder of the swap statement from Figure 27 with variable garbage (of type Int_Stk_Fac.Stack), and Figure 30 shows the programmer replacing the right placeholder with variable s2. Note that in this second situation the popup menu only contains variables of type Int_Stk_Fac.Stack because this type was bound to the placeholder when variable garbage was selected for the left <VAR NAME>. It does not matter which <VAR NAME> of a swap statement is replaced first.

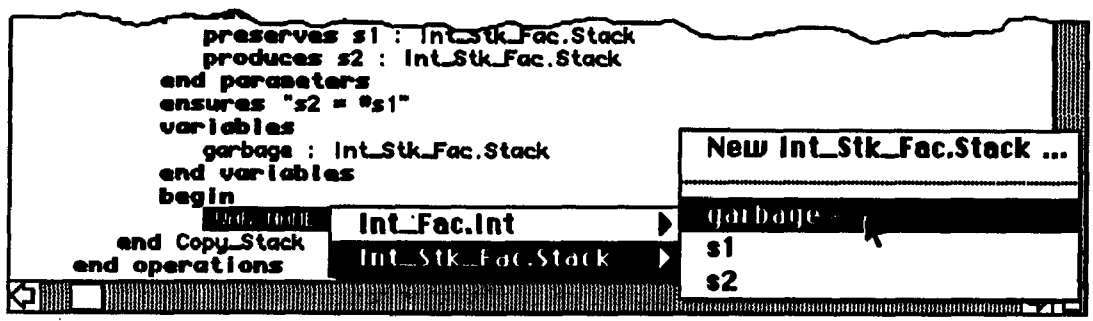

Figure 29

Replacing An Untyped <VAR NAME> Placeholder

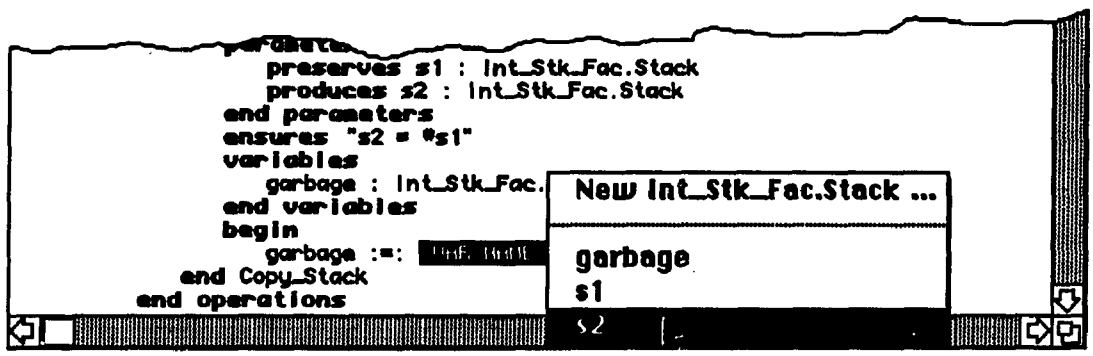

Figure 30

Replacing a Typed <VAR NAME> Placeholder

The first (and possibly only) menu item in every variable popup menu (e.g., New Int_Stk_Fac.Stack ...) is used to declare a new variable, allowing the programmer to conveniently declare a variable when it is first used. This saves the programmer from continually moving between the variable declaration section and code. Of course, a programmer can declare a variable in the variable declaration section if he or she chooses. Details of both techniques are discussed in Section A.3.

If the mouse is pressed on a variable name, a variable popup menu appears, allowing the programmer to replace the current variable with another of the same type. This is shown in Figure 31. (Note that the programmer decided not to change s2.)

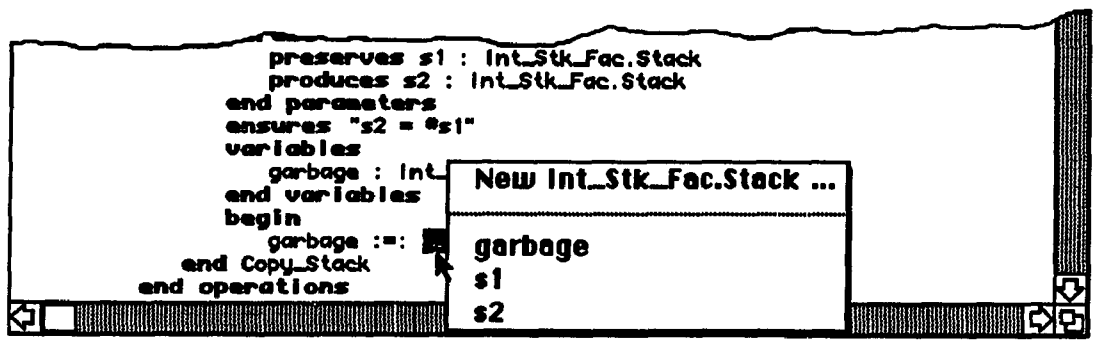

Figure 31

Changing a Variable

A.2 Inserting Statements and Control Invocations

Inserting a new statement (e.g., a while statement or variable declaration) or section (e.g., realization auxiliary section of a realization) is accomplished by placing the mouse in the white space between the program body and the left edge of the window and pressing the mouse button. When the mouse is in this white space (without the button pressed), an "insertion arrow" appears to its right if (and only if) it is possible to insert a statement or section there. An insertion arrow is displayed immediately to the left of the program body and is never *on* a statement, but always *between* two statements.

Figure 32 shows the insertion arrows displayed as the mouse is moved downward in the white space to the left of the parameters section of procedure Copy_Stack. The insertion

arrow changes its shape (i.e., ↗, →, or ↪) based on the indentation of the program, making it more noticeable. Moving the mouse horizontally in the white space does not change the shape or position of the insertion arrow.

Pressing the mouse button when an insertion arrow is displayed causes a popup menu to appear, containing the insertions possible at this location. For example, Figure 33 shows the insertion arrow located within the code section of Copy_Stacks, and the popup menu displayed when the mouse button is pressed. In this example, the programmer has chosen to insert a "while not" statement. Note that the popup menu appears next to the mouse location, not the insertion arrow.

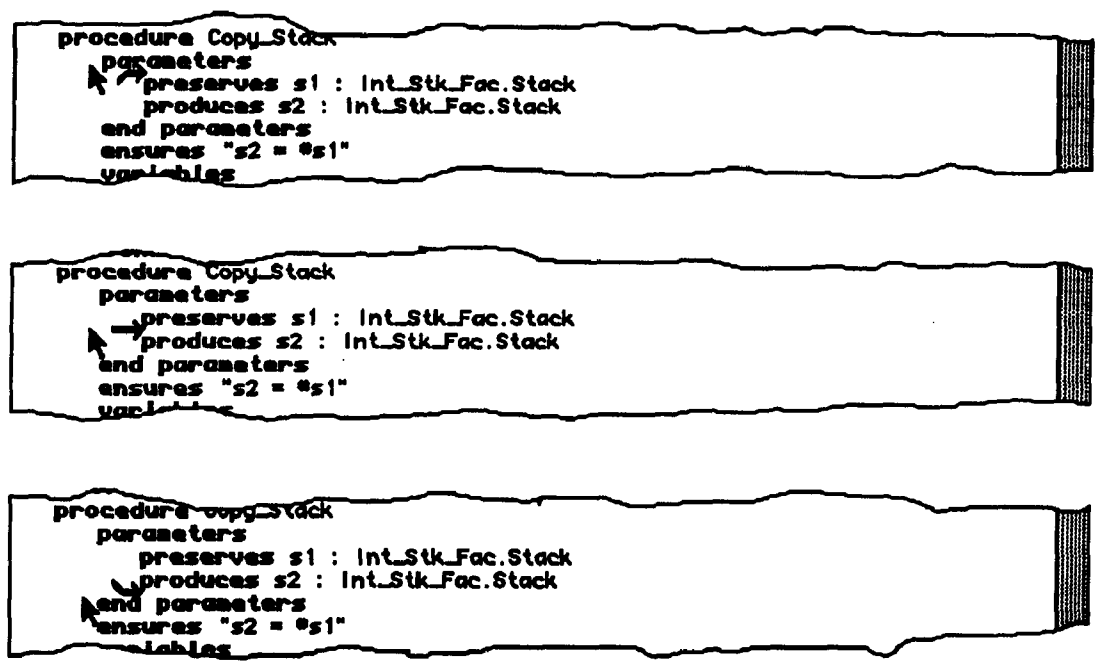

Figure 32

Insertion Arrow Positions

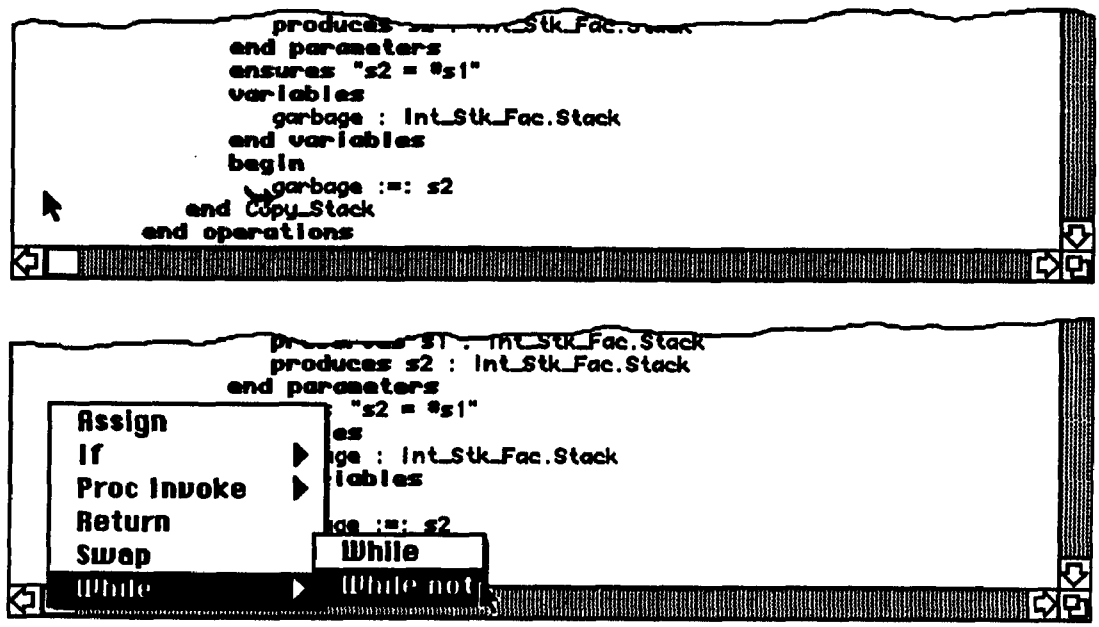

Figure 33

Inserting a While Statement Into Copy_Stack

Figure 34 shows the insertion arrow positioned within a code section, and the popup menu by which the programmer is inserting an invocation of procedure Pop provided by facility Int_Stk_Fac. Note that the procedure invocation menu is organized hierarchically by facilities providing procedures.

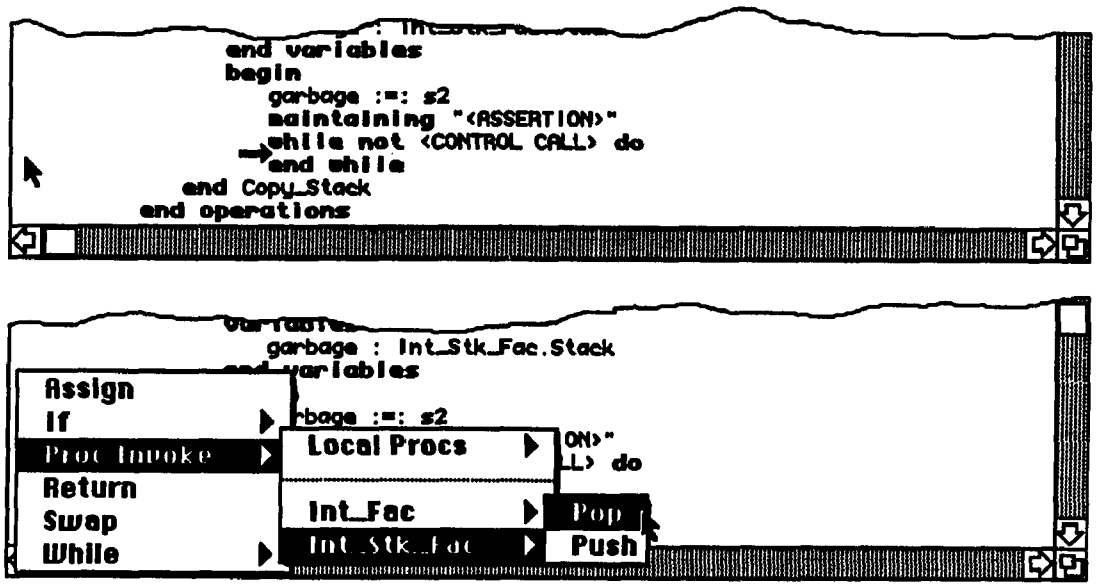

Figure 34

Inserting an Invocation of Procedure Pop

Figure 35 shows the popup menu that is displayed when the mouse is pressed in a <CONTROL CALL> placeholder. Here the programmer is inserting an invocation of control Is_Empty provided by facility Int_Stk_Fac. Note that the menu is organized hierarchically, similar to the procedure invocation menu in Figure 34. (Also note that the programmer previously inserted an invocation of procedure Push in the while loop.)

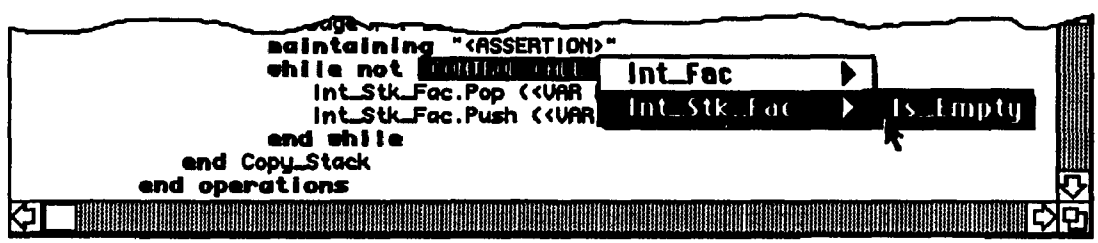

Figure 35

Inserting an Invocation of Control Is_Empty

A.3 Variable Declaration

The control and procedure invocations inserted into the example program in the previous section have a total of five actual parameters. Recall from Section 3.3.1.3 that all actual parameters are variables, so every actual parameter appears as a <VAR NAME> placeholder in the code. Note that a type is bound to each actual parameter[38]. Replacing these <VAR NAME> placeholders with variable names is done as discussed in Section A.1.

These five <VAR NAME> placeholders need to be replaced by three variables — s1, temp, and catalyst (see Figure 24 in Section 3.7.1). However, two of these variables — temp and catalyst — need to be declared first.

One way to declare a variable is to insert a variable declaration statement in the variable declaration section. Figure 36 shows the popup menu containing the available types that appears for a variable declaration. (The insertion arrow was between the declaration of variable garbage and the **end variables** statement when the mouse button was pressed.) The name of the variable is entered in a dialog box that appears, as shown in Figure 37. Note that the dialog box contains the names of all declared variables; this information may be useful to the programmer in determining a unique name for the new variable.

As shown in these figures, the programmer is declaring a variable named catalyst of type Int_Stk_Fac.Stack. (Also shown is the fact that our programmer can't spell very well! Hopefully this mistake won't go unnoticed for long.)

---

38 It is possible that the type of an actual parameter may not be known. This occurs when a type parameter to a facility declaration has not been filled in. For example, the type of the second parameter to procedure Push is whatever type is passed to the instantiation of Stack_Template. If an invocation of Push is inserted into the code before the facility's parameter is filled in, the <VAR NAME> placeholder for the second parameter to Push cannot be replaced.

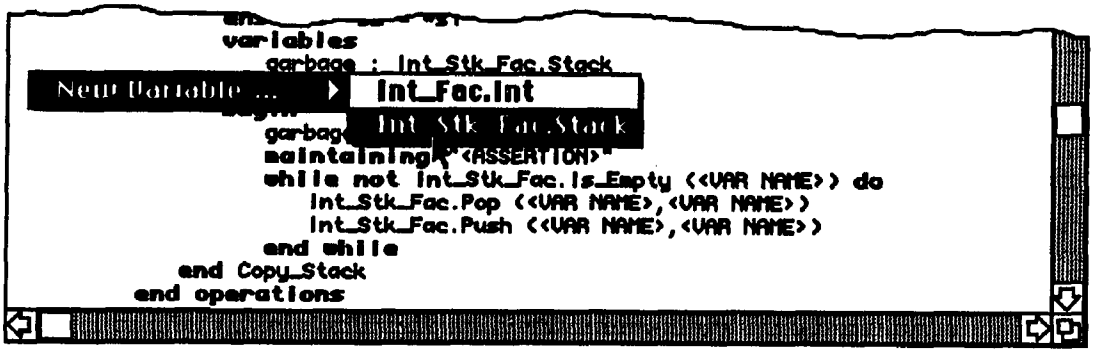

Figure 36

Type Declaration Menu

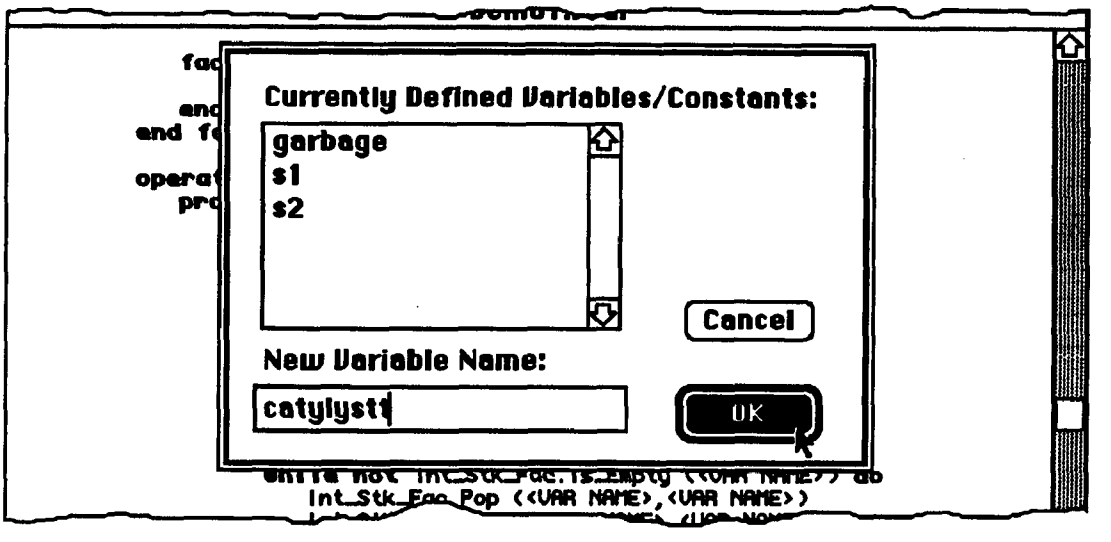

Figure 37

Dialog Box to Name a Variable

The second way to define a variable is to select the "New ..." menu item in the <VAR NAME> placeholder menu. Figures 38 and 39 show the menu and dialog box used by the programmer to declare variable temp of type Int_Fac.Int. Note in Figure 38 that the type of variable is known (i.e., Int_Fac.Int), and that no variables of this type are currently defined. The radio buttons in the dialog box in Figure 39 are used to indicate the scope

of the new variable (i.e., local to the operation or global to the module) and the position within the variable declaration list (i.e., either first in the list, or last)[39].

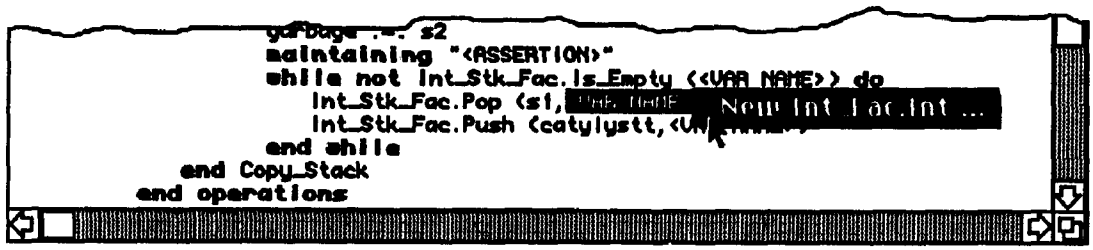

Figure 38

Menu for In-Place Variable Declaration

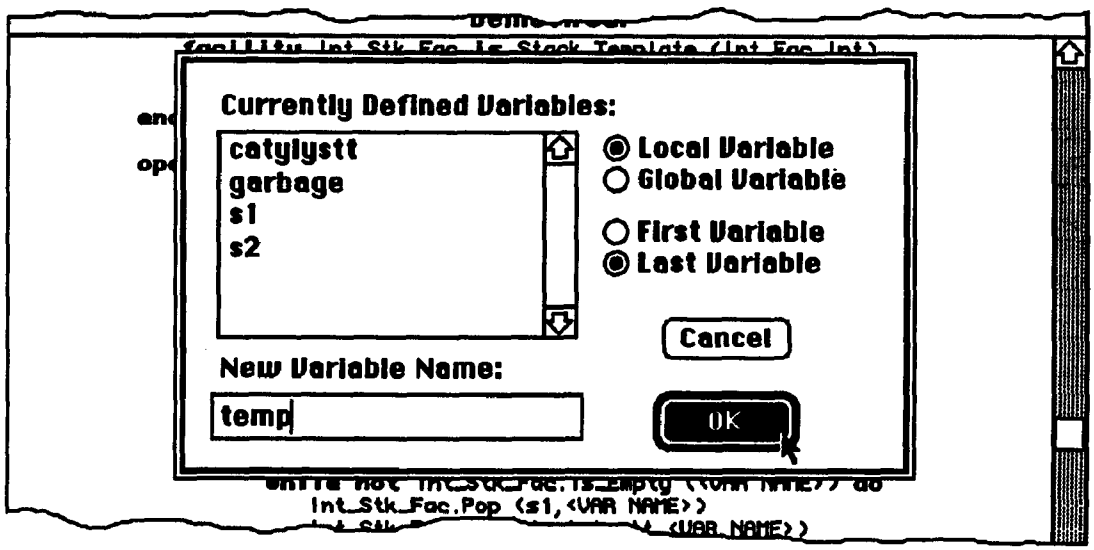

Figure 39

Dialog Box for In-Place Variable Declaration

The name of a variable can be changed by pressing the mouse button on the variable name in its declaration. When this is done a popup menu appears containing only the item "New Variable Name ...". If this item is selected, a dialog box very similar to the one in Figure 37 appears, allowing the programmer to enter a new name for the variable.

---

39 The position of the variable declaration in the list has no effect on the declaration but is purely cosmetic.

The editor then changes that variable's name everywhere it is used in the program. Thus, our klutzy programmer can easily change variable catylystt to catalyst.

A.4 Inserting Function Invocations

The righthand side of a function assignment statement is a function invocation, which appears as a <FUNCTION CALL> placeholder when a function assignment statement is inserted into a program. Pressing the mouse button on this placeholder causes a popup menu to appear containing all functions that can be invoked. Figure 40 shows the programmer selecting function Add provided by Int_Fac. Note that because the assignment variable k is of type Int_Fac.Int, the popup menu contains only functions of this type. The menu is hierarchically organized by facilities providing functions of the appropriate type (in this case Int_Fac is the only one), similar to the organization of the control invocation menu in Figure 35. If a type is not bound to the <FUNCTION CALL> placeholder the first hierarchy level contains all types for which functions exist.

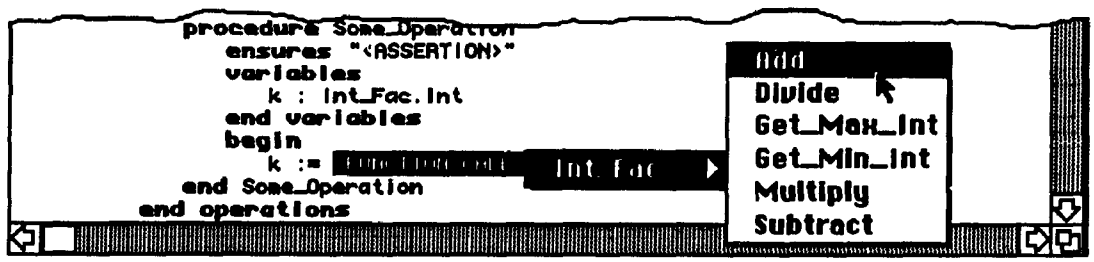

Figure 40

Replacing a Typed <FUNCTION CALL> Placeholder

If the mouse is pressed in a function identifier, a popup menu just like the one in the above figure is displayed, allowing the programmer to replace the current function invocation with another one of the same type.

A.5 If and Return Statements

Figure 41 shows the popup menu displayed for inserting an if...then...else statement into a program. Note that the four if statement forms are placed in a hierarchical menu, reducing the size of the first statement menu.

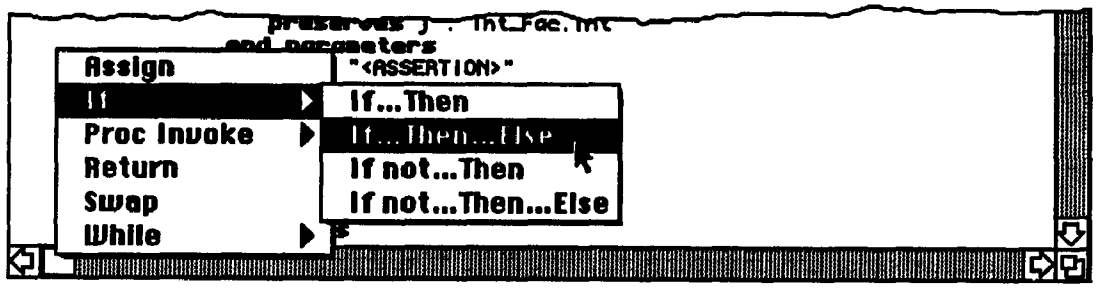

Figure 41

Inserting an If...Then...Else Statement

Recall from Section 3.3.3 that the simple return statement is allowed in functions and procedures, whereas a control must use either a return yes or return no statement. Figure 42 shows the programmer inserting a return yes statement in a control operation. Note that Return is a hierarchical menu, which is not the case in the menus shown in Figures 33, 34, and 41, because these figures showed the programmer inserting statements into procedures or functions, not controls.

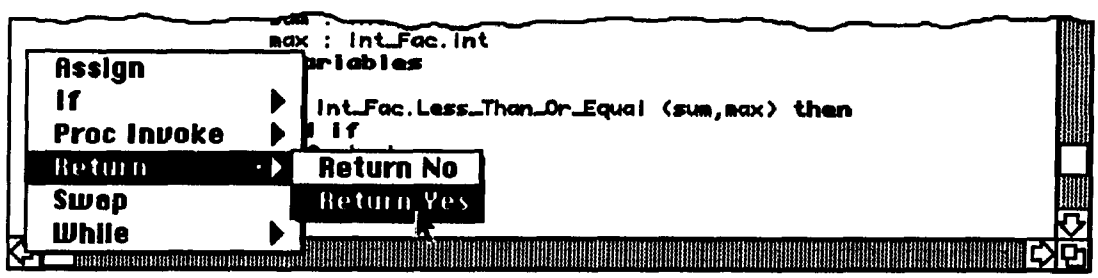

Figure 42

Inserting a Return Yes Statement

A.6 Editing Assertions

An assertion in a RESOLVE program currently consists of text surrounded by quote marks, and is meant ultimately for processing by a verifier. Assertions are treated as "free text" by the editor, and editing them is done using the standard Macintosh text editing facilities.

When the mouse is pressed within an assertion, a rectangular box is drawn around the text, as shown in Figure 43; all typing on the keyboard and mouse actions within this rectangle (i.e., clicking and dragging) are processed following the standard Macintosh user interface. Note that inserting and deleting return characters causes the rectangle to enlarge and shrink accordingly. When the mouse is pressed outside of the editing rectangle, the rectangle disappears, and RESOLVE editing resumes as normal.

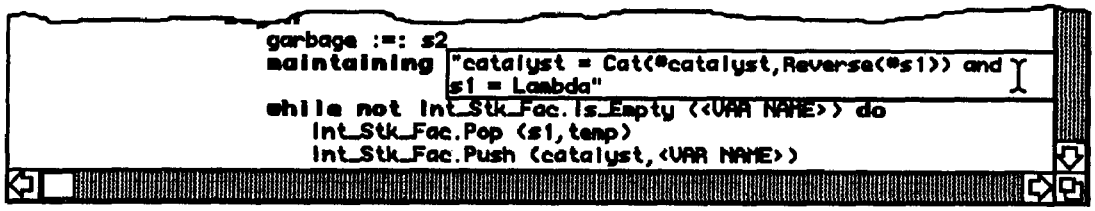

Figure 43

Editing an Assertion

A.7 Creating Conceptualizations, Types, and Operations

A conceptualization is created by selecting the appropriate menu item from the File menu (see Figure 28). A window is created containing the structure of the new conceptualization, as shown in Figure 44. The conceptualization is named by pressing the mouse button on the <CONCEPTUAL NAME> placeholder, selecting the "New Conceptual Name..." item from the popup menu that appears (it's the only item in the menu), and typing the name in the dialog box that is displayed. Conceptualization sections (e.g., parameters, auxiliary, interface, description) and items within them (e.g., facility parameters and operations) are inserted the same way that statements were inserted in Section A.2.

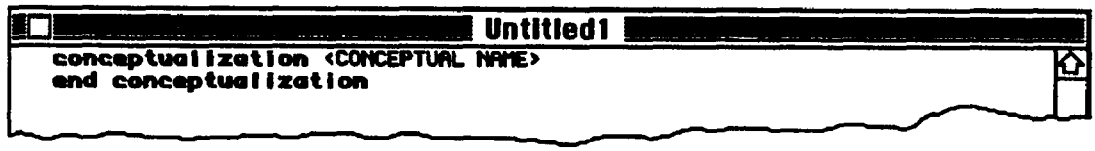

Figure 44

Newly Created Conceptualization

In a facility declaration, the conceptualization is specified by pressing the mouse button on the <CONCEPTUAL NAME> placeholder in the facility declaration, and selecting the "Select Conceptualization..." menu item that appears, as shown in Figure 45. A dialog box containing all available conceptualizations is displayed, and the programmer selects the desired one. (Note that the programmer is creating a conceptualization for bounded stacks, shown in Figure 15.)

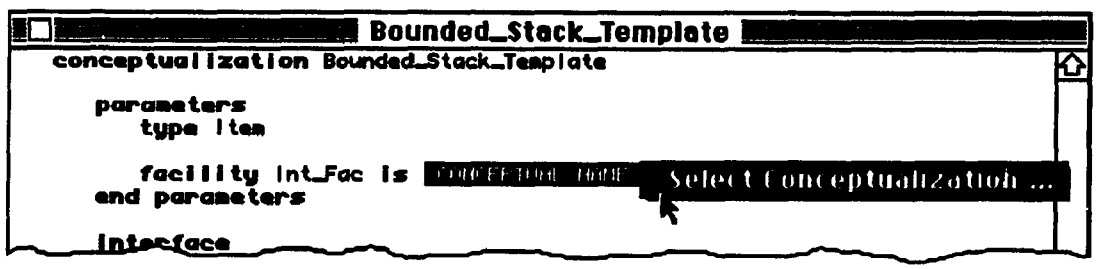

Figure 45

Replacing a <CONCEPTUAL NAME> Placeholder in a Facility Declaration

Type and operation declarations are inserted in the interface section using the same technique. For example, Figure 46 shows the programmer inserting a function of type Int_Fac.Int. A dialog box appears, in which the programmer types in the name of the function.

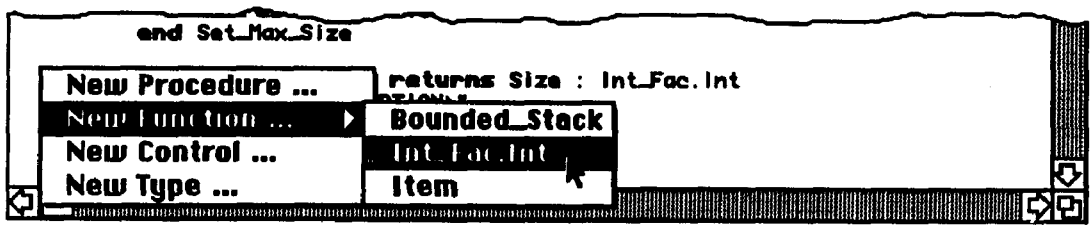

Figure 46

Inserting a Function Declaration

Figure 47 shows the programmer inserting a Bounded_Stack formal parameter declaration into the parameters section of function Get_Max_Size. The name and mode of the formal parameter are specified in a dialog box that appears, as shown in Figure 48. Note that the parameter in these figures is to a function, which must be a preserves parameter (see Section 3.1.6); this is why preserves is the only selectable parameter mode.

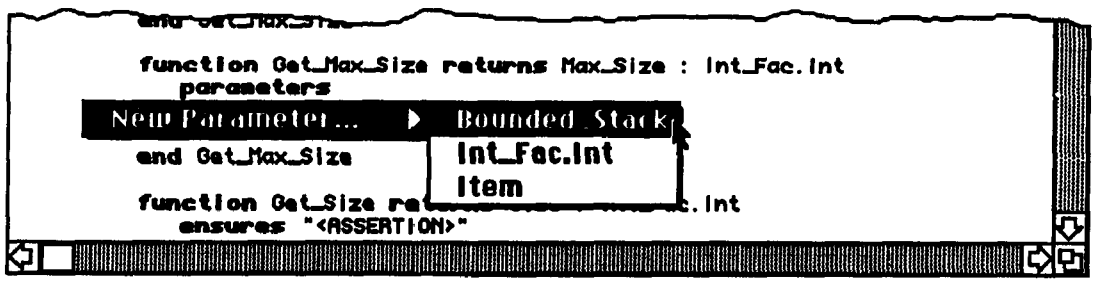

Figure 47

Inserting a Formal Parameter Declaration

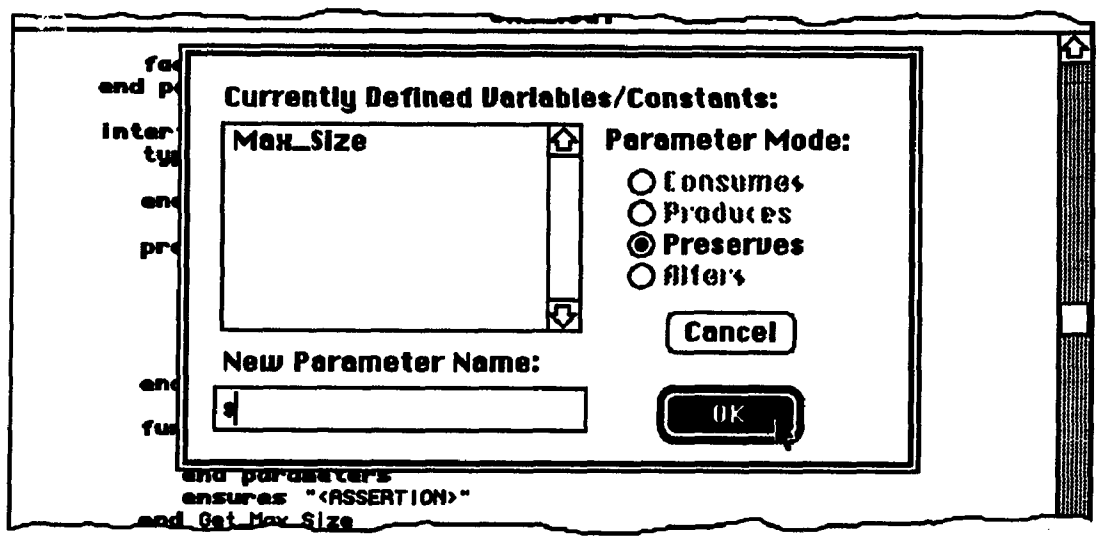

Figure 48

Dialog Box for Naming a Formal Parameter

A.8 Creating Realizations

A realization is created by selecting the appropriate menu item from the File menu (see Figure 28). A window is created containing the structure of the new realization, as shown in Figure 49. The realization is named by pressing the mouse button on the <REALIZATION NAME> placeholder, selecting the "New Realization Name..." item from the popup menu that appears (it's the only item in the menu), and typing the name in the dialog box that is displayed.

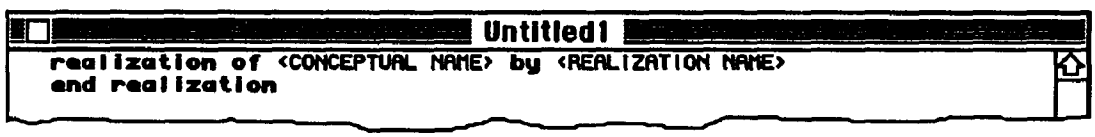

Figure 49

Newly Created Realization

The conceptualization that the realization implements is specified in the same manner as for a facility declaration discussed in the previous section — the mouse button is pressed in the <CONCEPTUAL NAME> placeholder, the item from the displayed popup menu is selected, and the conceptualization is selected from a list of available conceptualizations. When the conceptualization is selected, the editor uses the interface section of the conceptualization to automatically create the interface section in the realization, as shown in Figure 50. The items inserted into the interface section from the conceptualization (e.g., operation names and formal parameters) are not editable in the realization.

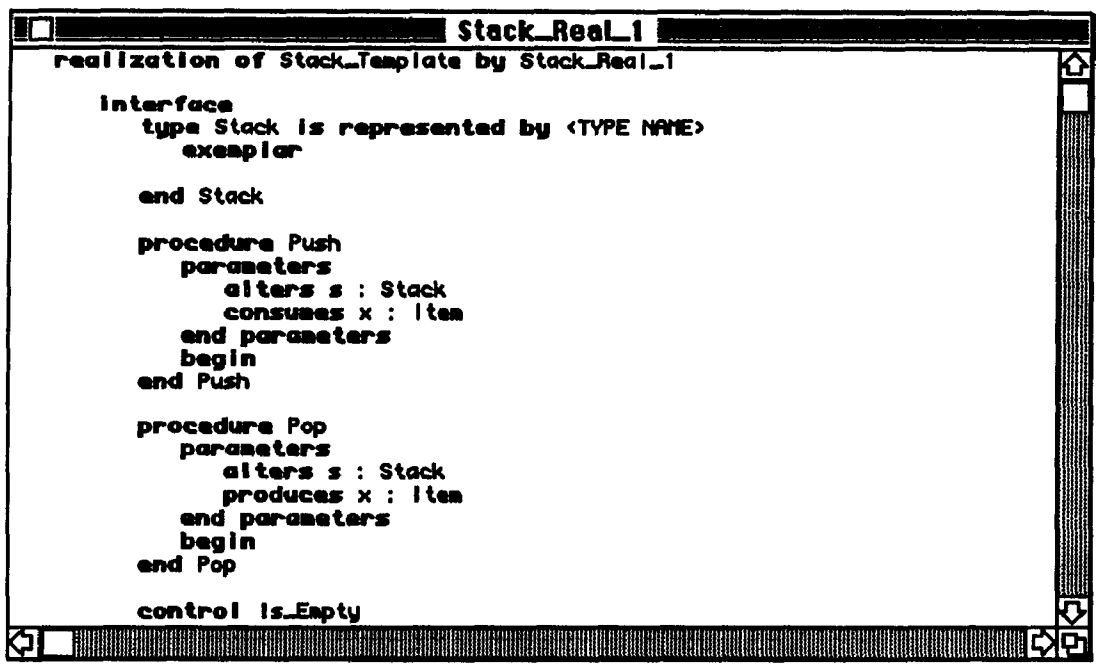

Figure 50

Interface Section of Realization Stack_Real_1

A.9 Selection and Deletion

When the mouse button is clicked on a keyword or special symbol, the structure enclosing that item is selected, and appears in reverse video. For example, Figure 51 shows the contents of the editor window after clicking on keyword "while" (the same

thing happens if the button is clicked on maintaining, not, or end while). Note that the entire while statement is selected, including all substatements.

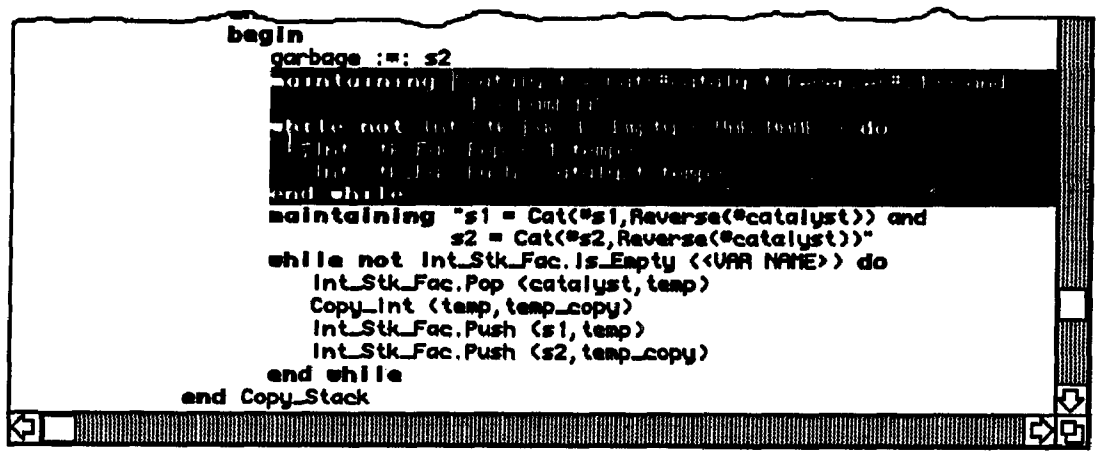

Figure 51

Selecting a Statement

Pressing on a keyword and then dragging the mouse adds multiple statements to the selection. It is not possible for a selection to contain statements outside of the block containing the originally selected statement. For example, Figure 52 shows the display after the programmer selected the Copy_Int procedure invocation statement and then dragged downward to the end Copy_Stack statement; note that only those statements within the body of the while loop (the block containing the originally selected statement) are selected.

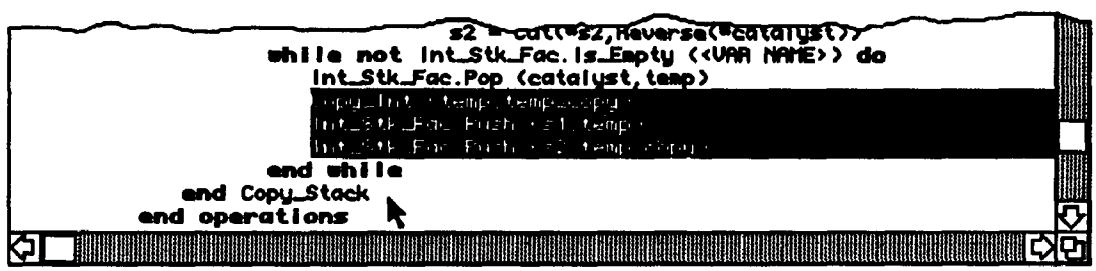

Figure 52

Selecting Multiple Statements By Dragging

Hitting the backspace key on the keyboard when statements are selected deletes those statements from the program. Presently this is the only action that can be taken with a selection, although work is underway for cutting/pasting selections.

BIBLIOGRAPHY

[Ada 79]
Preliminary Ada Reference Manual, June, 1979, ACM SIGPLAN Notices, vol. 14, no. 6.

[Aho 86]
A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[Ambler 77]
A.L. Ambler, D.I. Good, W.F. Burger, and C.G. Hoch, "GYPSY: A Language for Specification and Implementation of Verifiable Programs," *ACM SIGPLAN Notices*, vol. 12, no. 3, March 1977, pp. 1-10.

[Berard 89]
E.V. Berard, *Object-Oriented Life Cycle*, Berard Software Engineering, Inc., Germantown, MD, 1989.

[Biggerstaff 87]
T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, vol. 4, no. 2, March 1987, pp. 41-49.

[Biggerstaff 89a]
Software Reusability: Concepts and Models, T.J. Biggerstaff and A.J. Perlis, eds., ACM Press, New York, vol. 1, 1989.

[Biggerstaff 89b]
Software Reusability: Applications and Experience, T.J. Biggerstaff and A.J. Perlis, eds., ACM Press, New York, vol. 2, 1989.

[Boehm 87]
B.W. Boehm, "Improving Software Productivity," *IEEE Computer*, vol. 20, no. 9, September 1987, pp. 43-57.

[Boyle 84]
J.M. Boyle and M.N. Muralidharan, "Program Reusability through Program Transformation," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 574-588.

[Brooks 87]
F.P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, vol. 20, no. 4, April 1987, pp. 10-19.

[Carver 88]
D.L. Carver, "Acceptable Legal Standards for Software," *IEEE Software*, vol. 5, no. 3, May 1988, pp. 87-93.

[Chandhok 87]
R. Chandhok, D. Garlan, P. Miller, J. Pane, and M. Tucker, *Karel GENIE User's Manual*, Kinko's Academic Courseware Exchange, Ventura, CA, 1987.

[Chandhok 88]
R. Chandhok, D. Garlan, D. Goldenson, G. Meter, P. Miller, J. Pane, J. Carrasquel, J. Roberts, and E. Skwarecki, *Pascal GENIE User's Manual*, Kinko's Academic Courseware Exchange, Ventura, CA, 1988.

[COBOL 74]
American National Standards Institute, *American National Standard Programming Language COBOL, X3.23*, New York, 1974.

[Cohen 82]
E. Cohen, "Text-Oriented Structure Commands for Structured Editors," *ACM SIGPLAN Notices*, vol. 17, no. 11, November 1982, pp. 45-49.

[Currit 86]
P.A. Currit, M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, January 1986, pp. 3-11.

[Dahl 70]
O.J. Dahl, B. Mryhaug, and K. Nygaard, *Simula 67 Common Base Language*, Tech. Rept. S-22, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, October, 1970.

[Danforth 88]
S. Danforth and C. Tomlinson, "Type Theories and Object-Oriented Programming," *Computing Surveys*, vol. 20, no. 1, March 1988, pp. 29-72.

[DeMillo 79]
R.A. DeMillo, R.J. Lipton, and A.J. Perlis, "Social Processes and Proofs of Theorems and Programs," *Communications of the ACM*, vol. 22, no. 5, May 1979, pp. 271-280.

[DoD 83]
U.S. Department of Defense, Ada Joint Program Office, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, Government Printing Office, Washington, DC, February, 1983.

[Donahue 85]
J. Donahue and A. Demers, "Data Types Are Values," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, July 1985, pp. 426-445.

[Donzeau-Gouge 84a]
V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," In *Interactive Programming Environments*, D.R. Barstow, H.E. Shrode, and E. Sandwell, eds., McGraw-Hill, New York, pp. 128-140, 1984.

[Donzeau-Gouge 84b]
V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Mélèse, "Document Structure and Modularity in Mentor," *ACM SIGPLAN Notices*, vol. 19, no. 5, May 1984, pp. 141-148, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.

[Fitzgerald 87]
K. Fitzgerald, J.R. Devaney, and R. Thomas, "Faults & Failures: Lethal Dose," *IEEE Spectrum*, vol. 24, no. 12, December 1987, pp. 16.

[Fitzgerald 90]
K. Fitzgerald, "Vulnerability Exposed in AT&T's 9-hour Glitch," *The IEEE Institute*, vol. 14, no. 3, March 1990.

[Garlan 84]
D.B. Garlan and P.L. Miller, "GNOME: An Introductory Programming Environment Based on a Family of Structured Editors," *ACM SIGPLAN Notices*, vol. 19, no. 5, May 1984, pp. 65-72, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.

[Goldberg 83]
A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

[Gordon 79]
M. Gordon, *The Denotational Description of Programming Languages: An Introduction*, Springer-Verlag, New York, 1979.

[Griswold 71]
R. Griswold, J. Poage, and I. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, Second Edition, 1971.

[Guttag 77]
J.V. Guttag, "Abstract Data Types and the Development of Data Structures," *Communications of the ACM*, vol. 20, no. 6, June 1977, pp. 396-404.

[Guttag 78]
J.V. Guttag, E. Horowitz, and D.R. Musser, "Abstract Data Types and Software Validation," *Communications of the ACM*, vol. 21, no. 12, December 1978, pp. 1048-1064.

[Guttag 85]
J.W. Guttag, J.J. Horning, and J.M. Wing, "The Larch Family of Specification Languages," *IEEE Software*, vol. 2, no. 5, September 1985, pp. 24-36.

[Guttag 86]
J.V. Guttag and J.J. Horning, "A Larch Shared Language Handbook," *Science of Computer Programming*, vol. 6, no. 2, March 1986, pp. 135-157.

[Habermann 86]
A.N. Habermann and D. Notkin, "Gandalf: Software Development Environment," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 12, December 1986, pp. 1117-1127.

[Hamilton 76]
M. Hamilton and S. Zeldin, "Higher Order Software – A Methodology for Defining Software," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, March 1976, pp. 9-32.

[Hamilton 86]
M.H. Hamilton, "Zero-defect Software: The Elusive Goal," *IEEE Spectrum*, vol. 23, no. 3, March 1986, pp. 48-53.

[Harms 88]
D.E. Harms and B.W. Weide, *Swapping — A Desirable Alternative to Copying*, Tech. Rept. OSU-CISRC-1/88-TR2, Department of Computer and Information Science, The Ohio State University, Columbus, OH, January, 1988.

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

[Harms 89b]
D.E. Harms and B.W. Weide, *Types, Copying, and Swapping: Their Influences on the Design of Reusable Software Components*, Tech. Rept. OSU-CISRC-3/89-TR13, Department of Computer and Information Science, The Ohio State University, Columbus, OH, March, 1989.

[Hegazy 89]
W.A. Hegazy, *The Requirements of Testing a Class of Reusable Software Modules*, Ph.D. dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, June 1989.

[Hoare 73]
C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Progrmming Language Pascal," *Acta Informatica*, vol. 2, 1973, pp. 335-355.

[Hoare 83]
C.A.R. Hoare, "Hints on Programming Language Design," In *Programming Languages: A Grand Tour*, E. Horowitz, ed., Computer Science Press, Rockville, MD, pp. 31-40, 1983, Reprinted from SIGACT/SIGPLAN Symposium on Principles of Programming Languages, October 1973.

[Horowitz 84]
E. Horowitz, *Fundamentals of Programming Languages*, Computer Science Press, Rockville, MD, 1984.

[Ichbiah 79]
J.D. Ichbiah, B. Krieg-Brueckner, B.A. Wichmann, J.C. Heliard, J.G.P. Barnes, and O. Roubine, "Rationale for the Design of the Ada Programming Language," *ACM SIGPLAN Notices*, vol. 14, no. 6, June 1979.

[IEEE 84]
D. IEEE, *Special Issue on Software Reusability*, IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984.

[IEEE 87]
D. IEEE, *Special Issue on Software Reuse*, IEEE Software, vol. 4, no. 1, January 1987.

[Iverson 62]
K. Iverson, *A Programming Language*, John Wiley, New York, 1962.

[Jones 84]
T.C. Jones, "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 488-494.

[Joyce 85]
E.J. Joyce, "The Art of Space Software," *Datamation*, vol. 31, no. 22, November 15, 1985, pp. 30-34.

[Kaelbling 88]
M.J. Kaelbling, "Programming Languages Should NOT Have Comment Statements," *SIGPLAN Notices*, vol. 23, no. 10, October 1988, pp. 59-60.

[Kernighan 78]
B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Krone 88]
J. Krone, *The Role of Verification in Software Reusability*, Ph.D. dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, August 1988.

[Lampson 77]
B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.L. Popek, "Report on the Programming Language Euclid," *ACM SIGPLAN Notices*, vol. 12, no. 2, February 1977, pp. 1-77.

[Lampson 81]
B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, and G.J. Popek, *Report on the Programming Language Euclid*, Tech. Rept. CSL-81-12, Xerox Palo Alto Research Centers, Pala Alto, CA, October, 1981.

[Lanergan 84]
R.G. Lanergan and C.A. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September 1984, pp. 498-501.

[Lipow 82]
M. Lipow, "Number of Failures Per Line of Code," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, July 1982, pp. 437-439.

[Liskov 75]
B.H. Liskov and S.N. Zilles, "Specification Techniques for Data Abstractions," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 1, March 1975, pp. 7-19.

[Liskov 81]
B. Liskov, *CLU Reference Manual*, Springer-Verlag, Berlin, Lecture Notes in Computer Science, vol. 114, 1981.

[Luckham 79]
D.C. Luckham and N. Suzuki, "Verification of Array, Record, and Pointer Operations in Pascal," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 2, October 1979, pp. 226-244.

[Luckham 87]
D.C. Luckham, F.W. von Henke, B. Krieg-Brückner, and O. Owe, *ANNA: A Language for Annotating Ada Programs*, Springer-Verlag, Berlin, Lecture Notes in Computer Science, vol. 260, 1987.

[MacLennan 83]
B.J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt, Rinehart, and Winston, New York, 1983.

[Marcotty 76]
M. Marcotty, H. Ledgard, and G. Bochmann, "A Sampler of Formal Definitions," *ACM Computing Surveys*, vol. 8, no. 2, 1976, pp. 191-275.

[Meyer 85]
B. Meyer, "On Formalism in Specification," *IEEE Software*, vol. 2, no. 1, January 1985, pp. 6-26.

[Meyer 88]
B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Cambridge (U.K.), 1988.

[Meyrowitz 82]
N. Meyrowitz and A. van Dam, "Interactive Editing Environments: Part I," *ACM Computing Surveys*, vol. 14, no. 3, September 1982, pp. 321-352.

[Mills 87]
H.D. Mills, M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software*, vol. 4, no. 5, September 1987, pp. 19-25.

[Muralidharan 90]
S. Muralidharan, *Mechanisms and Methods for Performance Tuning of Reusable Software Components*, Ph.D. dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, 1990.

[Musser 80]
D.R. Musser, "Abstract Data Type Specification in the AFFIRM System," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, January 1980, pp. 24-32.

[Parnas 72]
D.L. Parnas, "A Technique for Software Module Specification with Examples," *Communications of the ACM*, vol. 15, no. 5, May 1972, pp. 330-336.

[Pittel 90]
T. Pittel, *Pointers in RESOLVE: Specification and Implementation*, Master's thesis, Department of Computer and Information Science, The Ohio State University, Columbus, OH, June 1990.

[Popek 77]
G.J. Popek, J.J. Horning, B.W. Lampson, J.G. Mitchell, and R.L. London, "Notes on the Design of Euclid," *SIGPLAN Notices*, vol. 12, no. 3, March 1977, pp. 11-19.

[Poston 87]
R.M. Poston and M.W. Bruen, "Counting Down to Zero Software Failures," *IEEE Software*, vol. 4, no. 5, September 1987, pp. 54-61.

[Pratt 84]
T.W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, Second Edition, 1984.

[Reiss 85]
S. Reiss, "Pecan: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, March 1985, pp. 276-285.

[Sedgewick 88]
R. Sedgewick, *Algorithms*, Addison-Wesley, Reading, MA, Second Edition, 1988.

[Shani 83]
U. Shani, "Should Program Editors Not Abandon Text Oriented Commands?," *ACM SIGPLAN Notices*, vol. 18, no. 1, January 1983, pp. 35-41.

[Shaw 81]
ALPHARD: Form and Content, M. Shaw, ed., Springer-Verlag, New York, 1981.

[Spivey 89]
J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, New York, 1989.

[Stovsky 90]
M.P. Stovsky, *A Framework and Environment Supporting Software Component Reuse for Teams of Developers*, Ph.D. dissertation, Department of Computer and Information Science, The Ohio State University, Columbus, OH, March 1990.

[Stroustrup 86]
B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[Swinehart 86]
D.C. Swinehart, P.T. Zellweger, R.J. Beach, and R.B. Hagmann, "A Structural View of the Cedar Programming Environment," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 4, October 1986, pp. 419-490.

[Teitelbaum 81]
T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, vol. 24, no. 9, September 1981, pp. 563-573.

[Teitelman 85]
W. Teitelman, "A Tour Through Cedar," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, March 1985, pp. 285-302.

[Waters 82]
R.C. Waters, "Program Editors Should Not Abandon Text Oriented Commands," *ACM SIGPLAN Notices*, vol. 17, no. 7, July 1982, pp. 39-46.

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

[Weide 86b]
B.W. Weide, *A New ADT and Its Applications in Implementing Linked Structures*, Tech. Rept. OSU-CISRC-TR-86-3, Department of Computer and Information Science, The Ohio State University, Columbus, OH, January, 1986.

[Welsh 77]
J. Welsh, W. Sneeringer, and C.A.R. Hoare, "Ambiguities and Insecurities in Pascal," *Software — Practice and Experience*, vol. 7, 1977, pp. 685-696.

[Wing 87]
J.M. Wing, "Writing Larch Interface Language Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 1, January 1987, pp. 1-24.

[Wirth 74]
N. Wirth and K. Jensen, *Pascal User Manual and Report*, Springer-Verlag, New York, Second Edition, 1974.

[Wirth 82]
N. Wirth, *Programming in Modula-2*, Springer-Verlag, New York, Second Edition, 1982.

[Wirth 83]
N. Wirth, "On the Design of Programming Languages," In *Programming Languages: A Grand Tour*, E. Horwitz, ed., Computer Science Press, Rockville, MD, pp. 23-30, 1983, Reprinted from Proceedings IFIP Congress 74.