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. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

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 9001960

The requirements of testing a class of reusable software modules

Hegazy, Wael A., Ph.D.

The Ohio State University, 1989

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

THE REQUIREMENTS OF TESTING A CLASS OF REUSABLE SOFTWARE MODULES

DISSERTATION

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

By

Wael A. Hegazy, B.S., M.S., M.S.

* * * * *

The Ohio State University

1989

Dissertation Committee

Dr. Bruce Weide

Dr. Stuart Zweben

Dr. Gary Perlman

Approved by

Bruce W. Weide

Adviser

Computer and Information Science

To My Parents

Acknowledgment

I am deeply grateful to Dr. Bruce Weide for his guidance and insight throughout the research. Thanks go to the other members of my research committee, Dr. Stuart Zweben and Dr. Gary Perlman, for their suggestions and comments. I would also like to thank the members of the reusability group for their active interest in my research. Finally, I offer sincere gratefulness to my parents for their continuous support.

Vita

August 14, 1958	Born – Alexandria, Egypt
1980	BS, Computer Science and Automatic Control, Alexandria University, Alexandria, Egypt
1980-1983	Instructor, Alexandria University, Alexandria, Egypt
1983	MS, Computer Science and Automatic Control, Alexandria University, Alexandria, Egypt
1985	MS, Computer and Information Science, The Ohio State University, Columbus, Ohio
1983-present	Teaching/Research Assistant, The Ohio State University, Columbus, Ohio

PUBLICATIONS

"The Extra Stage Gamma Network," *IEEE Trans. Comput.* **C-37** (Nov. 1988) 1445-1450, (with K. Lee). Also appeared in *ACM SIGARCH* **14**, 2 (June 1986) 158-168, and in *Proc. 13th Int. Symp Computer Architecture*, Tokyo, Japan, 1986.

"Optimal Line Breaking in Music," Tech. Rep., OSU-CISRC-10/87-TR33, The Ohio State University, 1987, (with J. Gourlay).

"On the Implementation of The MusiCopy Language Processor," Tech. Rep., OSU-CISRC-10/87-TR34, The Ohio State University, 1987.

"MusiCopy: An Automated Music Formating System," Tech. Rep., OSU-CISRC-10/87-TR29, The Ohio State University, 1987, (with A. Parrish, J. Gourlay, D. Roush, and F. Sola).

"Handling Context Sensitivity in Arabic Scripts," in *Proc. 3rd Int. Conf. Text Processing Systems*, Dublin, Ireland, 1986, (with J. Gourlay).

"Mixed Mode Sequential Machines," in *Proc. 15th Asilomar Conf. Circuits, Systems, and Computers*, 1981, (with M. El-Derini).

FIELDS OF STUDY

Major Field:

Software Engineering. Professor Bruce Weide

Minor Fields:

Programming Languages. Professor John Gourlay

Theory of Automata. Professor Eitan Gurari

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES PAGE

CHAPTER I
Introduction

The increasing dependence on computers and the widening gap between software demand and supply have resulted in a pressing need to improve the quality of software products and to increase the productivity of the people who develop them. A promising approach to achieving that is to construct software systems by assembling them from standard reusable software components. This results in increased productivity since building a software system, or even a new component, becomes primarily a matter of identifying the needed components and composing them properly, instead of developing them from scratch.

There is a two-way relationship between software quality and the approach of using reusable software components. On the one hand, software components are not likely to be widely reused unless they are certified to do what they are supposed to do. On the other hand, because the cost of producing a reusable software component can be amortized over many uses of it, the developer of this component can afford to spend more time, intellectual energy, and money to achieve a desired level of confidence that the part is done right.

In practice, testing is the primary method of engendering confidence in a software component. However, as will be shown in this dissertation, testing a class of software components, whose characteristics are believed to advance reusability, involves difficulties that are not encountered in testing conventional software. Overcoming those difficulties puts certain requirements on the implementation of the software components, the language in which the software components are written, and on the practice of testing the parts. It also requires the use of a software testing environment with particular core facilities. The thesis advanced here is that these requirements can be met, but that they demand careful attention to details beyond what have previously been available in programming languages and environments.

I.1. Software Reusability

It has become evident that there is a growing need to increase software productivity. Software demand is rising sharply on what seems to be an exponential growth curve [Boeh81], while supply is not keeping pace with the growing demand. In 1983, the shortfall between demand and supply was measured in terms of 50,000 to 100,000 programmers, and it was expected to rise to 1.2 million by 1990 [Stan84]. This labor shortage makes it impossible for many organizations to get all their work done and, with rising salaries, it makes any software development expensive [Horo84]. Improving software productivity is the key to reducing the rapidly-widening gap between the demand for software and the ability to supply it.

A concept that is believed to have the potential of increasing the software productivity by an order of magnitude is known as "reusable software." The basic idea is to avoid constructing every new software system from scratch. Instead, already existing software is used in building new software systems. The approach is to derive new software by applying a few, well-defined composition principles to existing software components. For example, the UNIX† pipe mechanism allows the construction of more complex programs from simpler ones by connection of one program's output to another program's input.

The promise of software reusability in improving software productivity derives from the fact that only a small portion of a typical program is unique, novel, and specific to an individual application. The rest appears to be common, generic, and concerned with the general task of putting applications into computers [Jone84]. Economic analysis [Boeh81] indicates that the effort of developing software grows faster than linearly with the software size; that is, halving the amount of the software which must be built more than halves the cost of building it. Reusability therefore holds great economic promise.

Different forms of software reusability appear in the literature [Bigg84, Horo84, Jone84, Lane84, Kern84, Gogu84, Yin87]. Throughout this dissertation we will take "reusable software" to mean object code that can be used, as is, in multiple applications (with possible parametrization). UNIX encompasses this meaning at the

---

† UNIX is a trademark of AT&T.

program level. We will focus on reusable software components that are "data-oriented" or "object-oriented," and that *must* be incorporated into client programs in order to be used at all.

Despite the fact that the benefits of reusable software have been acknowledged for years, software today is rarely reusable to any extent. It has been argued [Kron88] that a primary reason for this situation is that current programming languages and tools are simply inappropriate for the task. Other hindrances include the difficulty of cataloging the reusable components and the lack of mechanisms for identifying them [Horo84].

I.2. Characteristics of Reusable Software

There has been an ongoing effort by our research group at The Ohio State University to address various issues in software reusability and to develop a completely new programming system that deals with those issues [Weid86a, Weid86b, Weid86c, Weid87, Harm88a, Harm88b, Kron88]. We call this programming system RESOLVE (for REusable SOftware Language with Verification and Efficiency). A major result of this work has been the identification of certain characteristics that are believed to be necessary to make software truly reusable.

One of those characteristic is the use of abstract data types. In this paradigm of building software, the basic building blocks are *modules*, each of which provides one

or more types and operations involving parameters of those types. Programs (or other modules) declare variables of the provided types, and a variable has a value of its declared type. A variable can have its value changed when it is passed as parameter to an operation, where such an operation may be provided by the module providing the variable's type or by another module. Modules are reusable software parts in the sense that they can be composed in very general ways to construct more complex modules.

Another characteristic of reusable software is *genericity*. Modules can be made usable in more situations by parametrizing the underlying types of the types the modules provide. For example, at some level of abstraction, a stack of integers and a stack of stacks of reals are the same except that the elements contained in the former are of type integer and those contained in the latter are of type stack of reals. Other than this difference, the meanings of the operations are the same in both cases. In both cases the pop operation removes the top element off the stack, and the push operation adds an element of the underlying type at the top of the stack. Therefore, a generic stack module has a *type parameter* which tells what type of elements will be in the stack. Then, when a particular stack structure is needed, the generic stack module is *instantiated*, by supplying an actual type for the stack module type parameter.

A vital requirement for modules to be reusable is that they be formally specified. Specifications tell what the module is guaranteed to do. Without specifications, potential users cannot tell what a module does. Having to look at the module's implementation to know what it does is out of the question. Beside being very

impractical, it violates the principles of abstraction and information hiding, on which the entire idea of abstract data types is based.

The need for formalism in module specifications is twofold. First, there is ample evidence that natural-language descriptions of software are inherently ambiguous, not only in principle, but also in practice [Meye85]. Such ambiguity can lead to reuse of inappropriate software, which can be worse than not reusing software at all. Second, formal specifications are necessary for the ability to certify the correctness of software.

Reusable software parts must be certifiably correct; no one would likely reuse a software part that is not certified to do what it is supposed to. Verification and testing are the only apparent hopes of achieving some degree of certification, and both depend on formal specifications. Verification attempts to construct a formal proof that a software part meets its specification; hence, by its very nature, it requires that the part be specified formally. Although testing may only show the presence of errors but not their absence, it still demands that the tester know exactly what a software part is supposed to do so that errors can be identified when they are encountered. Moreover, the choice of the test cases can be influenced by the specifications [Gour83]; formal specifications would make it easier to identify appropriate test cases. Whatever the certification method is, formal specifications would also make it possible to have the certification process automated, at least partly.

Having pointed out the indispensability of formal specifications to the reusability of software modules, the next question is how to specify the modules formally. Two main approaches to formal specification of abstract data types are known: model-based specifications and algebraic specifications. In model-based specifications [Wulf81], the domain of values for a program type is described in terms of the domain of values for some mathematical type. Then, the operations of the mathematical type are used to describe the operations on the program type. On the other hand, in algebraic specifications [Wulf81, Gutt78], an abstract data type and its operations are mutually and implicitly defined by a set of axioms for a new mathematical theory.

The model-based specification approach seems to be better for use in the context of reusable software parts. It has significant advantages in terms of human comprehension. Model-based specifications appear to be easier to write, and the answers to the questions of whether the specifications completely describe an abstract data type, and nothing else, seem to be more intuitive than with algebraic specifications. These advantages of the model-based specification approach are based on two main factors. First, in the model-based approach it is possible to use a small number of well-known mathematical theories to model a wide variety of commonly used ADTs, ranging from conceptually simple structures to quite complex ones. In contrast, the theories arising from algebraic specifications are numerous and unfamiliar, and are usually more difficult to deal with. Second, the specifications of the provided operations in the model-based approach are expressed for each individual operation separately, making it easier to understand (and specify) what

each operation does. The mutual definition of the provided operations in the algebraic approach tends to impede comprehending the effect of each individual operation.

Module extensibility is another aspect of reusability. A module can be extended, to provide additional operations, by augmenting (not modifying) its specifications and implementation with the added operations. This aspect of reusability differs from incorporating the original module into a new module in that the augmented module does not have to be specified from scratch. Instead, the specifications of the original module are "reused" in the specifications of the augmented module.

Efficiency is also an important factor in software reusability. A software component that is not efficient might need to be redeveloped from scratch in order to meet certain performance requirements. That is, the more efficient the software component is, the "more reusable" it will be.

To summarize, reusable software parts must be:

- data-oriented (or object oriented)
- generic
- formally specified
- extensible
- efficient
- certifiably correct.

I.3. Software Certification By Testing

The basic quality goal for software is that it performs its functions in the manner that was intended by its developers and as described in its specifications. However, software development is a labor intensive task in which the opportunities for interjection of human fallibilities are enormous. The increase of complexity of software systems is accompanied by an increase in the possibility of their having errors, and the cost of those errors amplifies as the software systems are involved in more and more critical areas, such as health care and transportation. This has led to a growing interest in, and need for, means to assure that a given piece of software satisfies its specifications. Throughout this dissertation, the term *certification* will be used to mean assuring, with a certain level of confidence, that a final piece of software meets its specifications.

The most desirable outcome of certification would be an absolute guarantee that the certified software conforms to its specifications. However, experience indicates that this goal can be difficult to attain. Most of the time we settle for doing things to increase our confidence that a piece of software meets its specifications. The level of confidence to settle for often depends on the cost of software failure [Adri82]. For example, software used in the control of airplane landings or directing of substantial money transfers requires higher confidence in its proper functioning than does a car pool locator program.

Most of the techniques in software certification can be classified broadly into two categories: verification and testing. The purpose of software verification is to construct a formal proof that a piece of software satisfies its specifications. This must involve careful reasoning about the text of the software. It also requires that the software specification be expressed formally, and that the semantics of each construct in the programming language used be stated formally. On the other hand, the purpose of software testing is to develop confidence in the software through accumulated experience of its use on a selected set of test cases [Deut82]. This usually involves three major steps: selecting representative test cases according to some criteria, executing the test, and comparing the test results with the expected results.

Although formal verification is intended to place software certification on a more sound theoretical foundation than testing, it is often dismissed as impractical for application to large software products [Fair85]. The problems cited with formal verification include the amount of effort required, the opportunities for errors in the verification process, and the level of sophistication required of the programmer. It has been argued that verification tools can never totally automate the process of formal verification because of decidability limitations [Fair85, Lisk86, Huan77]; they just reduce the amount of effort and the chance of introducing errors. However, this argument is weak as an attack on practical verification efforts, and there is every reason to believe that eventually we will be able to use techniques such as those introduced in [Kron88] to verify real reusable software components. How soon that might be is open to question, but "proof of principle" seems already to exist.

In view of the impracticalities associated with formal verification at present, however, testing remains the most widely practiced method in software certification. But software testing has also been criticized as being inadequate since it can show the presence of errors but never their absence. This is because certification via testing is typically based on the successful execution of only a relatively small set of possible test cases. Nevertheless, a carefully chosen set of test cases can greatly increase our confidence that the software works as specified. If well done, testing can detect most of the errors in the software.

Software testing techniques can be classified in different ways. One important distinction is that of *black box* vs. *white box* testing [Chow85, Lisk86, Adri82]. In black box testing, the test cases are generated from the specifications alone, without regard for the internal structure of the software being tested. This has the advantages that the testing procedure is not influenced by the source code being tested, and that the generation of test cases is therefore robust with respect to changes in the implementation. In white box testing, on the other hand, the test cases are derived from properties of the source code. The test cases are usually chosen to force some execution coverage of the software (e.g., coverage of certain statements and/or paths). It is a common practice, however, for the generation of test cases to be a combination of the black box and the white box approaches.

I.4. The Focus of the Dissertation

Even with effective verification technology on the horizon, the practical importance of testing for reusable software components cannot be overemphasized. For one thing, practice often lags behind the state-of-the-art by many years, and complete verification is a still a bit beyond the state-of-the-art in 1989. This raises interests, from both the practical and the academic points of view, in the issues surrounding the testing of such components. The general discourse of the research presented in this dissertation is the testing of reusable software parts whose characteristics are summarized in section I.2, namely, RESOLVE modules. In fact, the problems and proposed solutions are applicable in testing similar modules in similar circumstances in other languages, such as Ada† [Ada83] and C++ [Stro86].

The testing of RESOLVE modules evokes various issues which, to date, do not seem to have been addressed in the literature, possibly because they are not typically encountered in this form in conventional program testing. Those issues include:

- **Testing Strategies.** In conventional program testing, the entity being tested is usually a single piece of code that is executed as a whole on a given input, and it produces some output. Even in integration testing, several pieces of code are put together in such a way that they constitute a larger piece of code, with a single

---

† Ada is a registerd trademark of the U.S. Government, Ada Joint Program Office.

interface, that is still executed as a whole. On the other hand, a RESOLVE module defines one or more data types and operations involving parameters of those types. That is, the unit to be tested (a RESOLVE module) has several pieces of code, each for an operation, and each is individually executable. How that group of operations can be tested in a way that enables us to assess the correctness of the whole module is a question that does not have a clear counterpart in conventional program testing.

- **Genericity.** Because of the genericity of RESOLVE modules, a module is, in effect, a template that is used to generate instances of modules, which are true components of a larger piece of software. Can we assure the correctness of a generic module by testing a carefully selected set of the module instances? If not, what will the ramifications of that be on the practice of testing RESOLVE modules?

- **Observability.** Whatever the testing strategy for RESOLVE modules is, it seems inevitable that the values of some variables will need to be observed, at least so that the test results can be analyzed and evaluated. The use of abstract data types raises a problem in this regard. The user of a module is not supposed to know the program representation of the abstract data type(s) it provides; so, how can the value of a variable whose representation is hidden be observed? In standard program testing, especially for numerical software where most research work has been concentrated, all variables are of built-in types or their representations are not hidden.

- **Controllability.** In some testing strategies, test cases for individual operations in a module are to be specified and produced. A test case consists of values for a set of variables. How can variables be correctly driven to have certain values?

- **Tools for Evaluating Test Results.** Since RESOLVE modules are formally specified, can we automate, at least partially, the evaluation of the test results? What would the tools for that be?

The primary goal of the research presented in this dissertation is to investigate the observability issue. Specifically, the question asked is how to provide the tester of a module with the mathematical-model views of the program variables. The answer to this question holds the key to "clean" observability; that is, observability that does not violate the principles of abstraction and information hiding. Moreover, since this enables the tester to view objects in the domain in which the abstract data types are specified, it should be expected to make it easier for the tester to evaluate the test results.

To achieve that goal, we have identified the requirements of "clean" observability on the implementation of modules providing abstract data types, and on the language in which they are implemented. To meet those requirements, new language constructs have been designed and incorporated into our original language, and the language implementation has been extended to accommodate the semantics of those constructs. Moreover, the basic facilities in an environment for testing RESOLVE modules, based on the idea of visualizing the mathematical-model view of program

variables, have been identified. The software components of the environment have been designed and partially implemented.

I.5. Organization of The Dissertation

The next chapter presents an overview of the language RESOLVE. It includes examples of implementing and specifying modules providing abstract data types in RESOLVE.

In Chapter 3, the testing of RESOLVE modules is discussed. A testing strategy is proposed and used to illustrate the difficulties of testing these modules. Basic requirements for overcoming those difficulties are then presented. The chapter ends with comparative comments on related work on testing abstract data types whose specifications are expressed using the algebraic approach.

Chapter 4 presents the language support needed for testing RESOLVE modules. First, the need for language support for testing is explained. Then, new language constructs are introduced and examples of their use are given.

In Chapter 5, we present two kinds of run-time structures for RESOLVE programs. The first kind is used for the actual execution of programs in the applications they are intended for, while the second kind is used by an environment for testing RESOLVE modules. Because the two kinds of run-time structures must

be compatible with respect to the object code of RESOLVE modules, their designs and the issues surrounding them had to be addressed together in a unified way. The chapter presents the design of each of the two kinds of run-time structures, the rationale behind the design, and how it is implemented.

Chapter 6 introduces an environment for testing RESOLVE modules, based on our approach to accommodating the requirements of testing these modules. First, the environment is described from its user's point of view. Then, the major components of the environment and their designs are overviewed.

The last chapter summarizes the work presented, highlights its contributions, and presents some suggestions for future research.

CHAPTER II
Overview of RESOLVE

The purpose of this chapter is to shed just enough light on the language RESOLVE to make the rest of this dissertation intelligible. The main features and constructs of the language, and their utility in constructing reusable software parts are illustrated by way of examples. More details about RESOLVE can be found in [Weid89].

II.1. What RESOLVE Is

Resolve is a language that supports the construction of reusable software parts and of systems that use these parts. The program "unit" in RESOLVE is called "part," which is a template for a generic abstract data type. Specifications of a part are provided in a *conceptual module*, and each conceptual module has at least one (parametrized) *realization module* containing code to implement those specifications. To use a part in constructing a program or implementing another part, the programmer refers to the conceptual module of the part to be used to see what it does,

and selects one of the available implementations of this part. Before using the selected realization module in the program, the programmer *instantiates* it, where this instantiation involves fixing its parameters if it has any. The result of instantiating a realization module is a *facility*, which provides one or more particular (as opposed to generic) abstract data types and operations thereon. The programmer, then, can declare variables of the type(s) provided by the facility and invoke the operations it provides.

The constructs and features of RESOLVE support the characteristics mentioned in Section I.2, which are deemed desirable in design of and with reusable software parts, as follows:

- **ADT paradigm.** RESOLVE supports the ADT paradigm by encapsulating in a single syntactic unit , the representations of one or more types and the operations which access the representations to manipulate data objects of these types. Access to the representations is limited to just those operations.

- **Genericity.** Genericity is provided for in RESOLVE modules by allowing them to have parameters, some of which may be types. Such modules are then instantiated by fixing their parameters.

- **Formal specifications.** Formal specifications of reusable software parts are supported in RESOLVE by requiring that the formal specifications of a part be

presented in a conceptual module. The form of these specifications is discussed in the next section.

- **Efficiency.** RESOLVE promotes writing efficient implementations by replacing the conventional assignment operation, as the primary means for moving data objects, by the swap operation, which interchanges the values of two variables. This operation works for arbitrarily large data objects in O(1) time without aliasing problems, provided that large data objects are referenced through one level of indirection. (This pointer is "hidden" from a client program, just as the rest of the representation data structure is hidden. RESOLVE has no concept of "pointer.") Copying large data objects can still be programmed in RESOLVE, but by not making the assignment operation part of the language, unnecessary copying is discouraged. By providing the swap operation, the cost of unnecessary copying is not imposed on the programmer. Surprisingly, copying is apparently not needed for most large objects in most applications [Harm88].

- **Certifiable correctness.** The most prominent support for certifiable correctness in RESOLVE stems from the requirement that every part be formally specified. This is why the specification language is part of RESOLVE. Any certification effort would then use the specifications of a part to assess the correctness of a given implementation. In addition, RESOLVE supports verification in two major aspects. First, the use of the ADT paradigm factors the verification into levels corresponding to the implementation levels. In verifying

the correctness of an implementation of a part constructed from other parts, we rely on the specifications of the component parts, provided that their implementations have been already verified. Second, the uniformity of treatment of types in RESOLVE, where there are no built-in types that the language semantics depends on, and the nonexistence of aliasing in RESOLVE tend to make the verification rules simpler.

Acknowledging that testing is the most widely practiced certification method at present and for the near future, RESOLVE provides language support for testing. Constructs in RESOLVE are introduced to make it easier and more reliable to implement operations that are used in presenting to the tester of modules the mathematical-model views of the program variables: the values that correspond to the program variables in the mathematical domains that are used in the model-based specifications to model the types provided. Those constructs are presented in Chapter IV.

II.2. Conceptual Modules

A conceptual module formally specifies the types and the operations which a part provides. The specifications are written by referring to known mathematical theories (e.g., number theory, string theory, set theory, and function theory) whose formal descriptions are presented in *theory modules*. Precise correspondences are established between domains and values in those theories and the program types and

values being specified. The mathematical entities are then used as models of their program counterparts. The effects of operations on program variables are described by writing **requires** and **ensures** *clauses*, which are assertions in first-order predicate calculus with equality, using functions and predicates from the mathematical theories being used. A **requires** clause is an assertion that must be true before the operation is invoked if any effect of the operation is to be guaranteed. An **ensures** clause is an assertion that is guaranteed to be true when the operation completes, provided that the **requires** clause was true when the operation was invoked. In the examples used in this dissertation, we will use shorthand notation from the available theories to make the assertions concise.

Following is an example conceptual module which presents the specifications of a part that provides a stack data type and operations thereon:

```
conceptualization Bounded_Stack_Template
   parameters
      type T

      facility Int_Facility is Integer_Template
         renaming
            Int_Facility.integer as integer
            Int_Facility.int as int
         end  renaming
   end  parameters

   auxiliary
      theories
         theory String_Theory is String_Theory_Template(T)
            renaming
```

```
            String_Theory.string as string
         end renaming
      end String_Theory

      theory Pair_Theory is 2_Tuple_Theory_Template(string,
integer)
         renaming
            Pair_Theory.pair as stack_model
            Pair_Theory.projection_1 as items
            Pair_Theory.projection_2 as max_size
         end renaming
      end Pair_Theory
   end theories
end auxiliary

interface
   type stack is stack_model
      exemplar s
      initialization
         items(s) = Λ and max_size(s) = 0
      end initialization
      lemmas
         |items(s)| ≤ max_size(s)
      end lemmas
   end stack

   procedure set_max_size
      parameters
         produces s : stack
         consumes max : int
      end parameters
      requires max > 0
      ensures items(s) = Λ and max_size(s) = max_size
   end set_max_size

   function get_max_size returns max : int
      parameters
```

```
      preserves s : stack
    end parameters
    ensures max = max_size(s)
  end get_max_size

  function get_size returns size : int
    parameters
      preserves s : stack
    end parameters
    ensures size = |items(s)|
  end get_size

  procedure push
    parameters
      alters s : stack
      consumes x : T
    end parameters
    requires |items(s)| < max_size(s)
    ensures items(s) = items(#s) • #x
  end push

  procedure pop
    parameters
      alters s : stack
      produces x : T
    end parameters
    requires |items(s)| > 0
    ensures items(#s) = items(s) • x
  end pop
end interface
```

description

This part provides the type "bounded stack of T," where T is any type. In the formal specifications above, a stack is

```
        modeled by a mathematical string of type T, with the top of
        the stack at the right end of the string, and an integer
        giving its maximum allowable size.  Initially, every stack is
        empty and has maximum size 0.

        • "set_max_size (s, max_size)" sets the maximum size of the
          stack to max_size and makes s an empty stack.  On return,
          max_size has an initial value for type int.
        • "get_max_size (s)" returns the maximum allowable size of
          stack s.
        • "get_size (s)" returns the current size of stack s.
        • "push (s, x)" pushes x onto stack s. On return, x has an
          initial value for its type.
        • "pop (s, x)" pops the top element off stack s and returns
          it in x.

    end  description
end Bounded_Stack_Template
```

The keyword **conceptualization** introduces a conceptual module, which generally has four main sections. The first section lists the **parameters**. In the above example there are two: a type parameter and a facility parameter. The type parameter provides more flexibility in reusing the module by allowing the user to adjust for the type of the data objects to be stacked. The facility parameter in the

example is restricted to be of a specific part; it is the part whose specifications are presented in a conceptual module named "Integer_Template." The facility passed as a parameter provides the type "int," which is referenced in the specifications of the stack because some of the parameters to the provided operations are of that type. Having the facility that provides the type int as a parameter makes the specifications of the stack independent of the choice of a particular realization of the part which is specified by the conceptual module "Integer_Template." The Int_Facility also defines the mathematical domain "integer" as the mathematical model of an int. (See Appendix A for the specifications of the Integer_Template.)

The second section contains **auxiliary** information needed to explain the third section, the **interface**. First, it defines mathematical **theories** needed to explain the interface. For example, "String_Theory **is** String_Theory_Template(T)" denotes by "String_Theory" the theory of strings over the domain which is used to model the program type T. (See Appendix B for a description of string theory.)

The **interface** section of a conceptual module defines the types and the operations provided by the module. In our example, the provided type is called "stack," and it is mathematically modeled by a tuple of two components: a string over the domain modeling T, and a (mathematical) integer. The **exemplar** is a name which is used to stand for an arbitrary value of the type throughout the reminder of the type declaration. The **initial_conditions** section specifies the initial value that a variable of type stack assumes when it is declared, in terms of the mathematical model

of the type stack. A **lemma** is a fact which must be true at all times except possibly *during* the execution of an operation — an invariant in this sense.

The specifications of a provided operation include the program types of its parameters, the conditions that the operation **requires**, if any, so that its specified effect is guaranteed, and the effect that the operation **ensures**. The "push" operation, for example, takes two parameters. The first is of type stack and the second is of type T. It requires that (at the time of invocation of the operation) the length of the string in the first component of the tuple which models the given stack be less than the integer which is the second component of that tuple. The assertion in the **ensures** clause of the push operation describes the operation's effect by relating values of the mathematical models of the parameters at invocation time and their values at the conclusion of the operation. In the **ensures** clause, the variables which are prefixed by "#" denote values at the time of invocation of the operation, while the variables which are not prefixed by "#" denote values at the conclusion of the operation. (In the **requires** clause, variables denote values at the time of the invocation.)

The **description** section of a conceptual module is intended just to facilitate human understanding of the specifications, and is not meant to be machine processable. It should be noted that the formal specifications are the final authority on what the module provides, not the informal description.

The stack example is simple enough that it does not illustrate everything that can be in a conceptual module. Among the syntactic slots that can be in a conceptual module, but which are not illustrated by the stack example, is one for declaring *local conceptual variables*. These local conceptual variables are used to identify internal state information that is affected by the provided operations or by the initialization of variables of the provided types. The local conceptual variables can then appear in the **requires** and **ensures** clauses, and in the **initialization** sections of the provided type definitions. The initial values of the local conceptual variables are specified in an **initialization** section associated with the declarations of those variables.

II.3. Realization Modules and Facilities

A *realization module* contains a description of the correspondence between conceptual and implementation types and values, performance information, and the implementation (code) for a particular conceptual module. There may be more than one realization module for any given conceptual module, since there may be many implementations of the same part with different performance characteristics.

A *facility* is created from a realization module by supplying fixed input parameters to the module. There can be several facilities created from the same realization module in a program (or in an implementation of a part). For example, in the stack part there is a type parameter T. We might have one facility of a realization of that part in which T is int (the facility provides stacks of integers), one in which T

is real (the facility provides stacks of reals), and one in which T is itself a stack of int (the facility provides stacks of stacks of integers).

The structure for realization modules can be illustrated using an array realization for bounded stack:

```
realization of Bounded_Stack_Template
   by Constant_Time_Using_Array

   conceptualization parameters
      type T

      facility Int_Facility is Integer_Template
         renaming
            Int_Facility.int as int
         end renaming
   end conceptualization parameters

   realization auxiliary
      facilities
         facility Array_Facility is
               Plastic_Array_Template (T, Int_Facility)
            realized by standard
            renaming
               Array_Facility.apply as get_element
               Array_Facility.array as arr
            end renaming

         facility Triple_Facility is
               Triple_Template (arr, int, int)
            realized by standard
            renaming
               Triple_Facility.projection_1 as stack_array
```

```
            Triple_Facility.projection_2 as stack_top
            Triple_Facility.projection_3 as stack_max_size
            Triple_Facility.triple as stack_rep
         end renaming
      end facilities
   end realization auxiliary

   interface
      type stack is stack_rep
         exemplar stk
         correspondence
            s = ( \I\pr(i=1,stack_top(stk),
get_element(stack_array(stk),i)) ,
                     stack_max_size(stk) )

         end correspondence
         .
         .
         .
         initialization
            variables
               stk : triple
            end variables
         end initialization
      end stack

      .
      .
      .
   procedure push
      parameters
         alters s : stack
         consumes x : T
      end parameters
      duration O(1)
      variables
```

```
        contents : arr
        top : int
    end  variables
    Triple_Facility.access1 (s, contents)
    Triple_Facility.access2 (s, top)
    Int_Facility.increment (top)
    Array_Facility.access (contents, top, x)
    Triple_Facility.access2 (s, top)
    Triple_Facility.access1 (s, contents)
  end push
  .
  .
  .
end Constant_Time_Using_Array
```

A realization module consists of sections which mirror those of a conceptual module. The **conceptualization parameters** section lists the parameters to the corresponding conceptual module. Although not illustrated by the stack example, a realization module may also have a **realization parameters** section listing module parameters which are specific to this particular realization of the specifications in the conceptual module. In most cases, a realization parameter is an operation. An example where an operation parameter is needed is one of a part for sorting values of a parameterized type T. The realization of such part would need to use a "less-than" operation on type T, where the implementation of this operation depends on the actual type for T. A realization parameter can also be a type or a facility.

The **realization auxiliary** section contains information and declarations used in the subsequent section, the **interface**. In the above example, the **realization auxiliary** section consists just of facility declarations. The example realization module implements a stack of type T as a triple whose first component is an array of type T, whose second component is an integer representing a top pointer, and whose third component is an integer representing the stack's maximum size. The facility declaration section (introduced by the keyword **facilities**) declares the needed facilities from the array and the triple parts by supplying the proper module parameters.

The **interface** section describes the implementation of the type(s) and operations provided by the module. In our example, the type bounded stack is implemented as the type triple provided by the Triple_Facility which is declared in the **realization auxiliary** section. The **correspondence** states a precise connection between the conceptual and realization views of a stack using the names of the stack exemplars in both the conceptual and the realization modules and using functions from the theories which model the components of the program representation of a stack. In our example, the conceptual view of a stack is a 2-tuple whose first component is a string of items and whose second component is an integer. The string of items is expressed as the concatenation of the entries in the array, starting at the first element and ending with the one whose index is the top pointer. To express that, we used the functions projection_1, projection_2, and projection_3 from 3-tuple theory, which is used to model the type triple, and the function "apply" from function theory, which is used to

model the type array. The second component of the pair that models the stack is an integer, which is expressed using the function "projection_3" from 3-tuple theory.

The **initialization** code for type stack constructs in a variable called stk (the name of the stack exemplar in the realization module) the initial value for a stack as specified in the conceptual module. The initialization code in our example turns out to be very simple (non-existent), by taking advantage of the initial values for the types integer, array, and triple.

The code of the push operation illustrates the style of programming used in RESOLVE, where the primary means for moving data objects is swapping their values rather than copying them. The code of the push operation declares two variables, which results in their containing the initial values for their types. The access1 (access2) operation provided by the triple facility swaps the value of the first (second) component of its first argument with the value of its second argument. In a similar fashion, the access operation provided by the array facility swaps the value of an array element whose index is given with a given value. Consequently, the result of executing the code of the push operation is to push a given value onto a stack, and replace the original argument value by the initial value for its type.

CHAPTER III
Testing RESOLVE Parts

The testing of ADTs which are specified using the model-based approach raises some serious issues which are not typically encountered in conventional program testing. The main purpose of this chapter is to illuminate those issues, show the problems they suggest, and discuss the requirements for overcoming these problems. RESOLVE parts are used as representatives of modules providing ADTs which are specified using the model-based approach; the issues are not confined to RESOLVE itself or unique to it in any sense.

The issues surrounding the testing of RESOLVE parts are elucidated through five questions related to the process of software testing in general. In Section III.1, these questions are answered for the case of conventional program testing, and the same questions are investigated for the case of testing RESOLVE parts in Section III.2. It is hoped that considering the same set of questions for both cases will illustrate the peculiarity of some testing problems raised by modules providing ADTs with model-based specifications. Section III.3 introduces our approach to handling the problems associated with testing RESOLVE parts. The chapter ends with a brief

review of related work on testing algebraically-specified ADTs along with some comparative remarks.

In this chapter, three RESOLVE parts are referred to in the examples: the "stack" part, the "list" part, and the "nilpotent function" part. Understanding the examples requires knowledge of the conceptualizations of those parts. The conceptualization of the stack part is given in Chapter II, and that of the list part and nilpotent function part can be found in Appendix A.

III.1. Conventional Program Testing

Testing an executable piece of software, be it a single operation, a subsystem, or a whole system, involves three major steps: 1) selecting a set of test cases (henceforth called "test points"); 2) executing the software for each test point; and 3) evaluating the test execution to determine whether or not the actual behavior of the software agrees with its expected behavior.

This work concentrates on steps (2) and (3), assuming the test points are already chosen. The process of performing and evaluating a test for a given test point evokes the following basic questions:

1) How is a test point expressed?
2) How is a test point actually produced?
3) How is it determined that an actually produced test point is the one intended?

4) How are the test results presented?

5) How is the test execution evaluated?

In what follows, the answers to these questions, as they apply to conventional program testing, are reviewed. We will use the term "software unit" to mean any executable piece of software that may be tested as a unit.

How is a test point expressed?

A test point consists of values for the variables that the subject software unit may access. For example, if the software unit is a procedure, a test point comprises values for the procedure's actual parameters and for the global variables that the procedure may access. To express a test point, then, is to express its constituent values. In the literature, it is usually assumed that the components of a test point are simple enough that they are expressed in a straightforward way (e.g., values of integers, character strings, and other "built-in" types). Even if the components of a test point are conglomerate structures, it is also assumed that the structure composition is known in terms of "built-in" types down to its primitive types; and hence, a value for a conglomerate structure can still be expressed as a composition of values of simple types. For example, a structure which is an array of records of integer arrays and character strings can have a value expressed as:

[{[7, 5, 4],"abc"} , {[2, 1, 5],"def"} , ...]

with the convention that square brackets enclose array elements separated by

commas, double quotes surround the characters in a string, and braces enclose record fields separated by commas.

How is a given test point actually produced?

The value of a test point is produced using program code that generates it, whether this code is written by the tester or automatically generated by a software tool from specifications of the test point. Usually, this code is a part of a driver program which produces the test point, invokes the subject software unit, and presents the test results. Typically, the code which produces a test point is conceptually trivial.

How is it determined that an actually produced test point is the one intended?

In conventional program testing, ensuring that a produced test point is indeed the desired one is not of pressing concern. Even if the code that produces the test point is not generated automatically, the code is still simple enough that reviewing the code itself (and possibly the data which the code might read) will not be much less effective in assuring its correctness than observing the values in the test point it produces. However, it might still be desirable to present the actually produced test point to the tester in an intelligible form, if for nothing else, for later use in evaluating the test execution.

The driver program may include code to present the values in a produced test point. There is nothing that makes such code more difficult than the code which produced the test point in the first place except, perhaps, the needed element of intelligibility, which can range from pretty printing to graphical representation of the structures and the values of the components of the test point [Myer80].

How are the test results presented?

By test results, we mean the values of the variables accessible to the subject software unit and the values of the variables produced by the software unit (e.g., if it is a function subprogram) immediately after control returns from it. The test results can be presented in the same way in which the test point is presented.

How is the test execution evaluated?

By evaluating a test execution, we mean determining whether or not the subject software unit performed functionally as expected on the given test point. Within the scope of the software we are considering, the functional behavior of a software unit consists of the changes it makes in the values of the variables accessible to it, and the values of the variables it produces, if any. In many applications, as long as the expected behavior is "sufficiently clear," it is not conceptually difficult (although it may be tedious) to evaluate the test execution by examining the test point and the results. However, this can be nontrivial for such applications as numerical analysis and optimization programs.

Because the process of evaluating a test execution is repeated for every test point used in testing a software unit, it is potentially so labor-intensive that it calls for automation. Unfortunately, the lack of formal specifications — and this lack is not uncommon in conventional programming — prevents the complete automation of the process of evaluating test executions. What may be aimed at in such a situation is to write code to evaluate test executions of a given software unit, based on a test point and its corresponding results. That is, such code is written once for the software unit, and is then used to evaluate its test executions. It should be noticed, however, that in some cases this code may not be much easier to write than the subject software unit itself. Consider, for example, the case of a procedure that solves a linear programming problem. Deciding whether the answer is correct is comparably or exactly as difficult as finding it in the first place.

III.2. Testing RESOLVE Parts

A RESOLVE part provides several operations, each of which has its own interface and can be invoked individually by a user of the part. This raises the basic question of how to test a RESOLVE part. Without ruling out the possibility of other strategies, the most immediate strategy is one in which each operation provided by a part (including the implicitly provided operations, which are the type initialization and finalization operations and the module initialization operation) is individually tested, and the correctness of the part is established by certifying the correctness of each individual operation that the part provides. This strategy takes advantage of the fact

that the provided operations are specified independent of each other to break down the problem of testing a whole RESOLVE *part* into the smaller problems of testing individual operations. Moreover, this strategy allows for leaning back on the accumulated experience from conventional program testing in choosing appropriate sets of test points for individual operations.

Having apparently reduced testing a RESOLVE part to testing individual operations, we revisit the five basic questions posed earlier in this Chapter — now in the context of testing the operations provided by RESOLVE parts — to illustrate the issues raised by testing those operations.

How is a test point expressed?

A test point consists of values for the arguments of the operation being tested, and values for the module variables, if it has any. The values in a test point must satisfy the condition expressed in the **requires** clause of the operation.

In black-box testing, where the program representations of the types of the values in a test point are not known, the values in a test point can be expressed only in terms of the mathematical models of their types. For example, the value of a "list of ints," where the type "list" is mathematically modeled by a pair of strings (see Appendix A), is expressed as a pair of strings of integers.

The situation is not much different in the case where the program representations of the types of the values in a test point are known. In a reusable software environment such as that supported by RESOLVE, the implementer of a type (say X) will ordinarily represent this type using some ADTs for which he does not know the program representations. In this case, if the implementer of type X describes a value of type X in terms of the program representation of X, he would still have to specify values of the other types used in the program representation of X in terms of the mathematical models for those other types. As an example, assume — just for the sake of the example — that a list is implemented as two stacks, where the implementer of this list uses a stack realization for which he does not know how a stack is represented. Expressing a list value in terms of the program representation of a list would then consist of a pair of stack values, but to describe a stack value, the implementer of the list has to use the mathematical model of a stack.

To summarize, it is inescapable that the values in a test point for an operation provided by a RESOLVE part have to be expressed, at one level or another, in terms of the mathematical models of the types of these values. This is a direct consequence of information hiding and abstraction.

Even if the top level(s) of the program representation for the type of a value in a test point is known, the following reasons suggest expressing the test point value entirely in the mathematical domain rather than partly in the program domain:

- Being described entirely in the mathematical domain, the test points can be reused in testing different implementations of the same part.

- The validity condition of the test points for an operation provided by a RESOLVE part is expressed in the operation's **requires** clause in terms of the mathematical models for the types of the values in the test points. Consequently, it is expected that checking the validity of a given test point will be easier if the test point is expressed in terms of those models.

- The **ensures** clause of a provided operation specifies the expected behavior of an operation on a valid test point by relating the mathematical views of the values of the variables in a test point and the mathematical views of the values of those variables at the conclusion of the operation. It is, therefore, most likely that evaluating a test execution of an operation will be easier in the case where the test point is described in the mathematical domain.

- By specifying the test points in the mathematical domain, the task of testing a part using given sets of test points can be assigned to someone other than the implementer of the part (e.g., as a production management discipline for more reliable quality assurance).

How is a given test point actually produced?

In contrast to conventional program testing, producing a given test point for an operation in RESOLVE is not in general conceptually trivial. In fact, it can conceivably be of a considerable degree of difficulty.

A test point is expressed in terms of the mathematical models of the types of its constituent variables. The problem of producing a given test point becomes one of maneuvering, using operations provided by the available RESOLVE parts, to produce variables, the values of whose corresponding mathematical models are given. To understand the reason for the potential difficulty of doing that, we may think of a procedure or function, conceptually, as operating on the mathematical values which correspond to its arguments. For example, we may think of the operation "push (s, x)" on stacks of ints as an operation which takes a string of integers and an integer, and appends this integer at the end of the string.

With this view in mind, we can then use operations provided by the available RESOLVE parts to produce variables whose mathematical values are given. However, this involves potential difficulty since those operations do not necessarily provide sufficiently precise control over, and access to, the mathematical views of their arguments. For example, the "nilpotent function" is mathematically modeled by three functions ("next," "count," and "data") and a given value for a nilpotent function variable consists of specific values for these three functions. However, the provided operations do not provide flexible control over these functions in a way that

allows us to individually set each function to a given value at each domain point. Instead, we have to maneuver using the available operations to set the three functions to the given values all together in an intricately complicated way.

How is it determined that an actually produced test point is the one intended?

Again, unlike the situation in conventional program testing, ascertaining the correctness of a test point is non-trivial and necessary, since the process of producing such a test point is quite error-prone. This is caused by two main problems. First, the potential difficulty of determining a sequence of operations to produce a test point can make producing the test point more error-prone than the operation being tested. This suggests that the correctness of the test point should be carefully checked. Second, even if the correct sequence of operations to produce a given test point is used, this sequence includes operations provided by the same part which provides the operation being tested. If those other operations have not been tested and certified correct yet, using the right sequence of operations can still result in a wrong test point.

The most obvious way for ascertaining that an actually produced test point is the one intended is to have the values of the variables in the produced test point presented to the tester so that he may compare the produced test point to the desired one. However, since the *desired* test point is expressed in the mathematical domain, the values of the variables in a *produced* test point should be presented in terms of the

mathematical models of their types. For example, a list of ints should be presented as a pair of strings of integers. Presenting the machine representation of the values of the variables in a produced test point is useless, since by virtue of the information hiding principle the tester may not be able to construct the mathematical views of the values of the variables from their machine representations. Even if the tester is himself the implementer of the module which provides the types of those variables, and he knows the correspondences between the program representations of those types and their mathematical-model representations, he might not know such correspondences for other (hidden) types which are used in the program representations of his types. This means a memory dump or a display of nodes and pointers is useless. Not only is it unwieldy; it does not contain enough information to enable the tester to decide the question in any case.

How are the test results presented?

In the context of an operation provided by a RESOLVE part, the test results are the values — at the conclusion of the operation — of the variables that are referred to in the operation's **ensures** clause. Following an argument similar to that in the investigation of the previous question, the test results should be presented in terms of the mathematical models of the variables.

How is the test execution evaluated?

Evaluating a test execution for an operation provided by a RESOLVE part entails checking whether or not the values in the test point and the values in the results satisfy the assertion in the operation's **ensures** clause. Given the mathematical views of the values in the test point and in the results, the test execution of an operation can be evaluated. The problem, therefore, reduces to one of presenting the mathematical views of data objects in the test point and in the results.

Because of the repetitive nature of the process of evaluating test executions, the tester of a RESOLVE operation might want to write code which checks that a given test point and the operation's results for that test point satisfy the specifications of the operation. However, such code can be difficult to write for reasons similar to those which make it potentially difficult to write code to produce a given test point. More specifically, while the specifications of an operation in RESOLVE are expressed in the mathematical domain, the operations available for use in writing code to evaluate a test execution do not necessarily provide sufficiently flexible access to the mathematical views of their arguments. The difficulty this might cause can be appreciated by trying to write code to check a condition as simple as two lists of ints (provided by the list part) being equal. Using the operations provided by the list part and the integer part, writing code to check the equality of two lists of ints is not straightforward. It can be done, but is a long enough program that there is plenty of opportunity for error.

Notwithstanding the potential difficulty in writing code to evaluate the test executions of an operation in RESOLVE, the use of formal specifications in RESOLVE provides fertile ground for automating the evaluation of test executions. An approach that takes advantage of the formal specifications in RESOLVE to advance the automation of evaluating test executions is introduced in the next section.

III.3. An Approach for Performing Tests in RESOLVE

From the preceding section, three basic requirements for testing RESOLVE parts are evident. First, there needs to be a way for presenting to the tester the mathematical-model views of variable values. This is needed both in ascertaining that a produced test point is indeed the desired one, and in evaluating the test executions of operations. Second, it is highly desirable that automatic tools be provided for use in assertion checking. This is motivated by the fact that the process of assertion checking is usually repeated many times for each operation being tested. Third, the inherent error-proneness of the process of producing a test point calls for a means for producing test points interactively in a step-by-step fashion, where after every step the tester can see the current mathematical views of the variables involved. In this way, if an unexpected value arises, the tester may be able to get clues on what went wrong, whether it was an incorrect method for producing the test point or the use of an erroneous operation.

Our approach to accommodating these three requirements for testing RESOLVE parts consists of the following:

- We provide a realization module corresponding to each mathematical theory in the set of theories which are used to express the specifications of RESOLVE parts. Each such module provides one or more types corresponding to the mathematical domains described by a theory, and operations corresponding to the functions and the predicates in that theory. For example, the realization module corresponding to string theory provides the type string of T (where T is a type parameter to the module), and functions such as *Length*, *Extend*, *Concatenate*, *Equal*, etc.

- We require that every realization of a RESOLVE part include a *model* operation for each type provided by this part. The model operation for a type takes a variable of that type and produces a corresponding value which is the mathematical model of the given variable. For example, in a realization of the stack part, where the type stack of T is provided, there is a model operation that takes a stack of T and produces a corresponding string whose elements are of the mathematical model of T. Note that this model operation uses the realization module corresponding to string theory in constructing the value it produces. Besides having a model operation for every provided type, if a RESOLVE part has conceptual module variables, it should also have a model operation that produces mathematical models of those variables. We call this model operation the "state model" operation.

- We provide a testing environment which uses the available realization modules (in compiled form) and allows its user (the tester of RESOLVE parts) to interactively instantiate realization modules, create new initialized variables of the types provided by the facilities resulting from instantiating realization modules, and invoke operations on the existing variables. The environment utilizes the model operations in the available modules to produce a representation of the mathematical view of each existing variable. These representations can then be used to provide visualization of the mathematical views of the variables. Moreover, those representations can also be used by the environment in automating the checking of a class of assertions (namely, assertions which do not involve quantifiers). To check an assertion involving given variables, the environment transforms the functions and predicates in the assertion into invocations of operations from the theory realization modules, and uses as arguments to these operations the representations of the mathematical views of the given variables.

The elements of this approach are detailed in the following chapters.

III.4. Related Work

Now that we have shown the problems inherent in testing modules providing ADTs which are specified using the model-based approach, and have outlined an approach for dealing with those problems, it is appropriate to compare our approach

to an approach for testing ADTs which are specified using the algebraic approach. The main result of the comparison is to reaffirm the appropriateness of using the model-based specification approach in specifying reusable modules, even with the testing issues taken into account.

In the DAISTS system [Gann81] (Data Abstraction Implementation, Specification, and Testing System), an ADT is specified using the algebraic approach. Testing an ADT in DAISTS involves compiling each axiom in its specifications into code calling on the operations of the ADT, with the free variables of the axiom as parameters to this code. Test points are chosen to "exercise" each branch in an axiom, and the constructed objects in a test point are fed as actual parameters to the code corresponding to the axiom. The code corresponding to an axiom, then, reports whether or not the axiom holds for the given test point. For example, the axiom

$pop(push(S, I))$ = **if** $Depth(S) = Limit$
then $Pop(S)$
else S;

for a bounded stack of integers has corresponding code which takes test points each of which consists of a stack value and an integer value, and checks for each test point whether the axiom holds.

In the approach taken in DAISTS, all the tester has to do is to choose and construct test points. The rest is automatic. With the cost of testing medium- to large-scale software systems amounting to about 50% of the development cost

[Jens79], it might seem that the apparent relative simplicity of testing ADTs in DAISTS would suggest using the algebraic specification approach to specify reusable software parts. However, the use of the algebraic specifications involves practical difficulties and hidden costs which, we believe, make them less attractive in practice. This conclusion is supported as follows:

- The specifications of a reusable software part should comprehensibly communicate to the potential user what the part does. Algebraic specifications score poorly in this regard since the mutual definitions of the operations provided by an ADT in the axioms of the algebraic specifications hardly communicate to the user what each operation does. The pattern of use of the operations in an ADT requires that the user know the effect of each individual operation, rather than the equivalence of combinations of operations, as is the case in the algebraic specifications.

- Algebraic specifications do not seem to be any easier to write than to read. The way in which the algebraic specification of an ADT is expressed may be quite far from the way the ADT is conceived by its designer. This opens the door wide for the incorrect specification of an ADT, possibly as wide as it is for wrong implementation. Therefore, in using algebraic specifications, there arises the idea of testing the specification against the concept it specifies [Gutt78]. The cost and reliability of such testing must be taken into account if algebraic specifications are to be practically useful.

- The approach taken in DAISTS for testing algebraically-specified ADTs is shown to work only for ADTs in which operations are side-effect free (i.e., they do not change the values of their parameters). Restricting the operations in ADTs to being side-effect free conflicts with the efficiency goals for reusable software parts. For example, it apparently disqualifies module designs based on swapping, as opposed to copying [Harm88].

- Writing algebraic specifications for an ADT may require defining operations not originally in the ADT [Majs77, Jone78, Gutt78c]. The testing approach in DAISTS requires that such operations actually be implemented [Gann81], which translates into more development cost.

- The ability to test individual operations provides more potential for better localization of the source of an error. Similarly, the possibility of visualizing the values of variables can give clues to what might have caused an error. Such abilities are not provided in DAISTS, because of the nature of algebraic specifications. Since there is no explicit mathematical model of an ADT except the (non-standard) mathematical type defined by the specifications themselves, there is generally no way even to *write down* the value of such an ADT except in terms of the constructor operations. New conventions for displaying values in a comprehensible notation would have to be invented for each new type.

- Although the problem of producing the actual test points in DAISTS is not addressed in [Gann81], it seems no easier than in RESOLVE. In DAISTS, to choose a test point which exercises a given branch of an axiom, one has to

"solve" the conditions leading to that branch for possible values of the variables involved. It is not clear how this can be done in general, and there is still the problem of determining a proper sequence of the available operations that produces the desired test point.

- Assessing the correctness of an implementation of an ADT in DAISTS is based on testing whether the implementation satisfies every individual axiom in the ADT specifications (using a limited set of test points, of course). However, no attempt is made in DAISTS to test whether theorems that can be derived from the axioms are also satisfied by the implementation. It can be shown that an implementation of an ADT may satisfy every axiom in the ADT specifications for *every possible* test point and still not satisfy theorems derived from these axioms (see Appendix D). Accordingly, such an implementation is incorrect, but its incorrectness can never be detected using the DAISTS testing method. It is not clear how the testing method in DAISTS may be extended so that there is at least one test case that uncovers the incorrectness of a given incorrect implementation of an ADT.

We conclude, as mentioned previously, that the problem of testing reusable modules providing ADTs is no easier if algebraic specifications are used. In fact, as we will now see, the model-based approach (for which testing problems have not previously been addressed) leads to cleaner and more understandable testing methodology.

CHAPTER IV
Language Support for Testing

Our approach to accommodating the requirements of testing RESOLVE parts was outlined in Chapter III. Central to this approach is the "model" operation which produces the mathematical-model view of a variable in a canonical representation. Because RESOLVE is a general purpose programming language, model operations can be programmed in RESOLVE. However, without certain extensions to the language, writing model operations in RESOLVE modules can have the undesirable side-effects of complicating the interfaces of RESOLVE modules, opening new doors for programming errors, and significantly expanding the RESOLVE run-time data structure.

The purpose of this chapter is to introduce extensions to RESOLVE to help avoid these problems. Section IV.1 illustrates the disadvantages that can arise by writing model operations in the original RESOLVE language. In section IV.2, we introduce extensions to RESOLVE and show how these extensions make it possible for model operations to be incorporated without undesirable side-effects. Type checking in RESOLVE, in light of the extensions to the language, is discussed in Section IV.3.

IV.1. The Model Operations and Their Requirements

The model operation for a type (say X) takes as an argument a variable of type X and returns a value of type Y, which has a canonical program representation of the mathematical domain which is used to model the type X. For example, if X is the type "stack" provided by the bounded-stack part, Y is the type "pair" whose first component is a string and whose second component is an integer. The type "pair" is provided by a unique realization of 2-tuple theory, the type "string" is provided by a unique realization of string theory, and the type "integer" is provided by a unique realization of number theory. The type of the elements of the string in this example is the type which represents the mathematical domain that is used to model the type of the elements of the stack.

To write the code for a model operation, one needs to refer to types and operations provided by facilities from the realization modules which implement the theories used to model the type of the model operation's argument. (Henceforth, such facilities will be referred to as "theory facilities.") The stack model operation, for instance, may refer to the types pair, string, and integer, and use operations involving these types. The stack model operation may also refer to the type which represents the mathematical model of the parametrized base type of the stack. In addition, the code of a model operation may need to refer to model operations for other types, some of which can be parametrized types. Using the stack example once again, the code of its model operation may use the model operation for ints and the model operation for the type parameter T.

Most of the aforementioned types, operations, and facilities can be made available to the model operation only by making them parameters to the module enclosing the model operation. This is best illustrated using a concrete example. In what follows we will consider a simple-minded realization of the "bounded list" part (whose conceptualization is given in Appendix A) in which the bounded list is represented by an array whose elements are of type T, and three integers representing the lengths of the left and right parts of the list and its maximum size. The items in the list occupy contiguous places in the array, and whenever an item is added to or removed from the list, the items to its right are shifted in the array.

The type "bounded list" is mathematically modeled by a triple whose first and second components are strings (from string theory) over the mathematical domain which models the type parameter T, and whose third component is an integer (from number theory). One way in which the model operation for the bounded list may produce the mathematical-model view of a bounded list from its program representation is as follows: The model operation starts with two empty strings for the left and right parts of the bounded list. It then accesses those elements of the array which hold the left part of the list, invokes the model operation for type T on each such element, and appends each object returned to the string of the left part of the list. The string of the right part of the list is similarly produced, and the (mathematical) integer for the maximum size is produced by invoking the model operation for the (program) integer which represents the maximum size. Finally, the model operation for the bounded list constructs a triple of the two strings and the

integer. The relevant parts of the bounded-list realization module, including the model operation, are given below.

```
realization of Bounded_List_Template
   by Simple_Array

   conceptualization  parameters
      type T

      facility Int_Fac is Integer_Template
         renaming
            Int_Fac.int as int
         end  renaming
   end  conceptualization  parameters

   realization  parameters
      type Math_T

      facility Num_Fac is Number_Theory
         renaming
            Num_Fac.integer as Math_Int
         end  renaming

      facility Math_String_Fac is String_Theory
         renaming
            Math_String_Fac.string as Math_String
         end  renaming

      facility Math_Triple_Fac is Triple_Theory
         renaming
            Math_Triple_Fac.triple as Math_Triple
         end  renaming

      function model_T returns mt : Math_T
         parameters
            preserves t : T
```

```
      end parameters
      ensures mt = t

   restrictions
      Math_Triple_Fac.T1 = Math_String
      Math_Triple_Fac.T2 = Math_String
      Math_Triple_Fac.T3 = Math_Int
      Math_String_Fac.T = Math_T
   end restrictions
end realization parameters

realization auxiliary
   facilities
      facility Array_Fac is
               Plastic_Array_Template (T, Int_Fac)
         realized by Standard
         renaming
            Array_Fac.array as arr
         end renaming

      facility Quad_Fac is
               Quad_Template (arr, int, int, int)
         realized by Standard
         renaming
            Quad_Fac.quad as list_rep
            Quad_Fac.access1 as access_array
            Quad_Fac.access2 as access_left_len
            Quad_Fac.access3 as access_right_len
            Quad_Fac.access4 as access_max_size
         end renaming
   end facilities
end realization auxiliary

interface
   type list is list_rep
      .
      .
      .
```

```
end list

.
.
.
function model returns l : Math_Triple
   parameters
      preserves s : list
   end parameters
   variables
      math_left : Math_String
      math_right : Math_String
      left_len : int
      right_len : int
      whole_len : int
      math_item : Math_T
      item : T
      i : int
      math_max_size : Math_Int
      a : arr
   end variables
   access_left_len(s, left_len)
   access_right_len(s, right_len)
   access_array(s, a)
   Int_Fac.add(whole_len, left_len)
   Int_Fac.add(whole_len, right_len)
   Int_Fac.increment(i)
   maintaining i ≤ left_len and
                          i-1
            math_left = ∏  a(j)
                          j=1
   while Int_Fac.less_than_or_equal(i, left_len) do
      Array_Fac.access(a, i, item)
      math_item := model_T(item)
      math_left := Math_String_Fac.Extend
                              (math_left, math_item)
      Array_Fac.access(a, i, item)
      Int_Fac.increment(i)
   end while
```

```
      maintaining i ≤ whole_len and
                                    i-1
                math_right =        ∏     a(j)
                                 j=left_len+1
      while Int_Fac.less_than_or_equal(i, whole_len) do
         Array_Fac.access(a, i, item)
         math_item := model_T(item)
         math_right := Math_String_Fac.Extend
                                 (math_right, math_item)
         Array_Fac.access(a, i, item)
         Int_Fac.increment(i)
      end  while
      access_array(s, a)
      access_left_len(s, left_len)
      access_right_len(s, right_len)
      access_max_size(s, i)
      math_max_size := Int_Fac.model(i)
      access_max_size(s, i)
      l := Math_Triple_Fac.set1(l, math_left)
      l := Math_Triple_Fac.set2(l, math_right)
      l := Math_Triple_Fac.set3(l, math_max_size)
   end model
   .
   .
   .
end Simple_Array
```

In the above example, having the realization module provide a model operation required adding five parameters to the module. These parameters are realization parameters (as opposed to conceptualization parameters) since different realizations of the same part can have different model operations each with different requirements. Explaining why the additional module parameters are needed is now in order.

The type parameter "Math_T" is intended to be the type which represents the mathematical model of the type parameter T. It is needed because the model operation for the bounded list requires declaring a variable of type "Math_T." Clearly, the actual type for the parameter "Math_T" varies according to the actual type for T, and hence, it is not known a priori (even in terms of T). Therefore, it cannot be defined inside the realization module. For a similar reason, the model operation for T, which is used by the model operation for the bounded list, has to be a realization parameter ("model_T") to the bounded list module.

The other three additional parameters in the above example are needed for a different reason. The structure of the type returned by the model operation for the bounded list involves the types integer, string, and triple which are provided by the realization modules of number theory, string theory, and triple theory, respectively. And since types in RESOLVE are compatible if and only if they are the same type provided by the same facility (not just structurally equivalent facilities), the type returned by the model operation for the bounded list can be known outside the module only if it is derived from the module parameters. This type needs to be known wherever the model operation is to be used so that the compiler can do the necessary type checking for invocations of the model operation. In the above example, the facilities providing the involved types, rather than the types themselves, are added to the module parameters since the model operation for the bounded list also uses operations provided by those facilities.

The additional module parameters required for the model operations can cause the disadvantage of complicating the code for declaring facilities from realization modules. For example, without the additional module parameters, the code for declaring a facility which provides a bounded list of bounded lists of integers is as follows:

```
facility Int_Fac is Integer_Template realized by Standard

facility Bl_Int_Fac is Bounded_List_Template
            (Int_Fac.integer, Int_Fac)
         realized by Simple_Array

facility Bl_Bl_Int_Fac is Bounded_List_Template
            (Bl_Int_Fac.list, Int_Fac)
         realized by Simple_Array
```

On the other hand, the code needed to declare the same facility from realization modules with the additional module parameters is:

```
facility Math_Int_Fac is Number_Theory realized by Standard
   renaming
      Math_Int_Fac.integer as M_int
   end  renaming

facility Int_Fac is Integer_Template
         realized by Standard (Math_Int_Fac)
   renaming
      Int_Fac.integer as int
      Int_Fac.model as model_int
   end  renaming

facility Str_Int_Fac is String_Theory (M_int)
         realized by Standard
   renaming
```

```
      Str_Int_Fac.string as M_str_int
   end  renaming

facility Trp_1_Fac is Triple_Theory
            (M_str_int, M_str_int, M_int)
         realized by Standard
   renaming
      Trp_1_Fac.triple as M_trp_1
   end  renaming

facility Bl_Int_Fac is Bounded_list_Template
            (int, Int_Fac)
         realized by Simple_Array
            (M_trp_1, Math_Int_Fac, Str_Int_Fac,
               Trp_1_Fac, model_int)
   renaming
      Bl_Int_Fac.list as list_int
      Bl_Int_Fac.model as model_list_int
   end  renaming

facility Str_Trp_1_Fac is String_Theory (M_Trp_1)
         realized by Standard
   renaming
      Str_Trp_1_Fac.string as M_str_trp_1
   end  renaming

facility Trp_2_Fac is Triple_Theory
            (M_str_trp_1, M_str_trp_1, M_int)
         realized by Standard
   renaming
      Trp_2_Fac.triple as M_trp_2
   end  renaming

facility Bl_Bl_Int_Fac is Bounded_list_Template
            (list_int, Int_Fac)
         realized by Simple_Array
```

```
(M_trp_2, Math_Int_Fac, Str_Trp_1_Fac,
   Trp_2_Fac, model_list_int)
```

By significantly complicating the code for declaring facilities, the additional module parameters required for the model operations could be expected to invite more programming errors in facility declarations. Although most such errors can be detected by the compiler or the program-writing environment, the more the chance that these errors may occur the more effort is needed to produce programs void of them. Furthermore, since the mathematical types corresponding to the actual type parameters to the module are supplied by the programmer, the door is open for undetectable errors in facility declarations. For example, in creating the facility providing a bounded list of bounded lists of integers as shown above, suppose the facility "Str_Int_Fac" were made to be for a string of program integers (instead of mathematical integers). Also suppose that instead of using the operation "Int_Fac.model" in the **renaming** clause associated with the declaration of "Int_Fac" another operation which takes and returns a program integer were used. Then all the **restrictions** on the actual realization parameters to the bounded-list module would be maintained in the Bl_Int_Fac declaration although the actual parameters were erroneous. Such errors would go undetected by the compiler.

Another disadvantage of providing the model operations and accommodating their requirements as described is that the space efficiency of RESOLVE programs is affected. Obviously, the object code of the model operations is not needed in a non-testing situation, but the model operations' being among the operations explicitly

provided by RESOLVE modules means that the object code of the model operations is always part of the object code of the modules. More significantly, because the run-time structure of RESOLVE programs includes representations of the facilities used in the program (as will be seen in the next chapter), all the theory facilities in a program (those facilities from the realization modules implementing the theories used) which are brought in just for use by the model operations, will needlessly consume space in the program run-time structure. Although the space inefficiency caused by the model operations and their requirements can be avoided by having two versions of each realization module, one for use in testing and one for production use, this may lead to other problems such as version control problems and ensuring that two versions of a realization module have the same behavior (except for model operations).

IV.2. Extensions to RESOLVE

It is evident from the discussion in the preceding section that more expressive power is needed in RESOLVE to allow for writing model operations without adding new module parameters and without significantly affacting the space efficiency of RESOLVE programs in non-testing situations. This section presents extensions to RESOLVE to solve these problems. A significant advantage of these extensions is that they fit nicely into the original framework of RESOLVE in the sense that they do not bring in new programming concepts which are significantly different from those in the original language. These extensions are explained in the following three subsections. An example of using the extensions is given in subsection IV.2.4.

IV.2.1. Theory Facilities

The first extension to RESOLVE is concerned with theory facilities: the facilities from the realization modules implementing the types, functions, and predicates in mathematical theories. A syntactic slot is added to the syntax of the realization module where the theory facilities used by the model operations in the module are declared. These theory facilities should include facilities corresponding to the theory "instances" described in the **auxiliary** section of the corresponding conceptualization module. Description of the mathematical representation of each provided type, in terms of the types provided by the declared theory facilities, is then attached to the type declaration in the realization module. The types provided by theory facilities will be referred to as "mathematical types" to distinguish them from the regular program types which are provided by regular (i.e., non-theory) facilities. (Terminology of model-based specifications is given in Appendix E).

Besides being declared in a separate section in a realization module, theory facilities must differ from regular facilities in the following respects:

- Type equivalence among the types provided by theory facilities is structural equivalence. That is, two mathematical types provided by theory facilities are equivalent if and only if they have similar structures, even if they come from two separately declared theory facilities.

- Theory realization modules can have only type parameters. In declaring a theory facility, the actual type parameters are limited to being mathematical types (i.e., types provided by theory facilities).

- Theory facilities and mathematical types cannot be used as actual module parameters in declaring non-theory facilities.

- If a type (say P) from the parameters of a realization module is used as an actual parameter in declaring a theory facility in the module, it denotes, in this context, the mathematical type corresponding to P rather than P itself. For example, in the declaration

```
facility Str is String_Theory (T)
```

  in a stack realization module, T denotes the mathematical type corresponding to T.

- The run-time representation of a RESOLVE program in a non-testing situation does not include anything about theory facilities. (This is explained in detail in the next chapter.)

IV.2.2. The "math" and "model" Constructs

The **math** construct enables the programmer to refer to the mathematical type corresponding to a given regular type. It has the syntax

math[*regular type*]

where *regular type* is any type that can be referred to in the original language. **math**[P], where P is a regular type, denotes the mathematical type corresponding to P. The **math** construct can be used in declaring mathematical variables. For example, the model operation for a stack may have the following variable declarations:

```
model_of_max_size : math[int]
model_of_x : math[T]
```

The **model** construct allows the programmer to refer to the model operation for a given regular type. It has the syntax:

model[*regular type*]

where *regular type* is any type that can be referred to in the original language. **model**[P], where P is a regular type, denotes the model operation for the type P. The **model** construct can be used to invoke model operations. For example, the model operation for a stack may include statements such as

```
model_of_x := model[T](x)
model_of_max_size := model[int](max_size)
```

IV.2.3. The Model Operations

Since the model operations are referred to just by the types whose model they produce (using the **model** construct), they can no longer be ordinary provided operations. Instead, they should be implicit operations (like type initialization and finalization operations). A model operation is therefore associated with the declaration of each provided type in a realization module. The syntax of the model operation can then be:

model

code

end model

The model operation has an implicit parameter whose type is the provided type with which the model operation is associated. The name of the parameter is taken to be the exemplar of this type in the realization module. The operation produces the value it returns in a variable whose name is the exemplar of the provided type in the conceptualization module. The type of the returned value is implicitly known to be the mathematical type corresponding to the provided type to which the model operation is attached. The *code*, as in any other operation in RESOLVE, consists of variable declarations and program statements.

IV.2.4. Example

With the language extensions presented above, the additional module parameters which were otherwise required for writing model operations are no longer needed. With structural type equivalence among the mathematical types, the theory facilities needed by a model operation can now be made available by declaring them in the realization module instead of passing them as module parameters. Likewise, the mathematical types and the other model operations used in a model operation can now be referred to using the **math** and **model** constructs, and hence, they need not be passed as module parameters. Here are the relevant parts of the bounded list realization module rewritten using the language extensions.

```
realization of Bounded_List_Template
   by Simple_Array

   conceptualization  parameters
      type T

      facility Int_Fac is Integer_Template
         renaming
            Int_Fac.int as int
         end  renaming
   end  conceptualization  parameters

   conceptualization  auxiliary
      theories
         theory Number is Number_Theory
            renaming
               Number.integer as Math_Int
            end  renaming
```

```
      theory String is String_Theory (T)
         renaming
         String.string as Math_String
         end  renaming

      theory Triple is
            Triple_Theory (Math_String, Math_String, Math_Int)
         renaming
            Triple.triple as Math_Triple
         end  renaming
   end  theories
end  conceptualization  auxiliary

realization  auxiliary
   facilities
      facility Array_Fac is
               Plastic_Array_Template (T, Int_Fac)
         realized by Standard
         renaming
            Array_Facility.array as arr
         end  renaming

      facility Quad_Fac is
               Quad_Template (arr, int, int, int)
         realized by Standard
         renaming
            Quad_Fac.quad as list_rep
            Quad_Fac.access1 as access_array
            Quad_Fac.access2 as access_left_len
            Quad_Fac.access3 as access_right_len
            Quad_Fac.access4 as access_max_size
         end  renaming
   end  facilities
end  realization  auxiliary

interface
   type list is list_rep modeled by Math_Triple
```

```
exemplar s
correspondence
.
.
.
   model
      variables
         math_left : Math_String
         math_right : Math_String
         left_len : int
         right_len : int
         whole_len : int
         math_item : math[T]
         item : T
         i : int
         math_max_size : Math_Int
         a : arr
      end  variables
      access_left_len(s, left_len)
      access_right_len(s, right_len)
      access_array(s, a)
      Int_Fac.add(whole_len, left_len)
      Int_Fac.add(whole_len, right_len)
      Int_Fac.increment(i)
      maintaining i ≤ left_len and
                                 i-1
                  left = ∏ a(j)
                                 j=1
      while Int_Fac.less_than_or_equal(i, left_len) do
         Array_Fac.access(a, i, item)
         math_item := model[T](item)
         math_left := String.Extend(math_left, math_item)
         Array_Fac.access(a, i, item)
         Int_Fac.increment(i)
      end  while
      maintaining i ≤ left_len and
                                      i-1
                  right =           ∏     a(j)
                                 j=left_len+1
      while Int_Fac.less_than_or_equal(i, whole_len) do
```

```
                Array_Fac.access(a, i, item)
                math_item := model[T](item)
                math_right := String.Extend
                                    (math_right, math_item)
                Array_Fac.access(a, i, item)
                Int_Fac.increment(i)
             end  while
             access_array(s, a)
             access_left_len(s, left_len)
             access_right_len(s, right_len)
             access_max_size(s, i)
             math_max_size := model[int](i)
             access_max_size(s, i)
             l := Triple_Fac.set1(l, math_left)
             l := Triple_Fac.set2(l, math_right)
             l := Triple_Fac.set3(l, math_max_size)
          end  model
       end  correspondence
       .
       .
       .
    end list
    .
    .
    .
end Simple_Array
```

IV.3. Type Checking

RESOLVE is a strongly typed language, where all type checking is done statically. Type checking in RESOLVE entails the following:

- checking that the two variables in a swap operation are of equivalent types, and
- checking that the types of the actual parameters to an operation are equivalent to the types of the corresponding formal parameters, after adjusting the latter types according to the actual module parameters used in the declaration of the facility to which the operation belongs. If the operation is a function whose returned value is assigned to a variable, the type of this variable should be equivalent to the type of the returned value.

Accordingly, type checking in RESOLVE consists of checking type equivalence.

With the extensions to RESOLVE, there are two kinds of type equivalence: the original type equivalence defined above and the structural type equivalence, which applies only to mathematical types. To be able to do type checking, the compiler must know when to use each definition of type equivalence. The complete separation between regular type structures and mathematical type structures, which derives from disallowing theory facilities and regular facilities to have in their declarations actual parameters from the other, makes it easy for the compiler to know when to use each kind of type equivalence. Structural type equivalence is used in type checking on the operations provided by theory facilities, while the type equivalence originally defined in RESOLVE is used in type checking on the operations provided by regular facilities. In a swap operation, the kind of type equivalence to be used depends on the types of the two variables involved. If both variables are of mathematical types, structural type equivalence is to be used, while if both variables are of regular types, the originally defined type equivalence is used. Obviously, if one variable is of a

mathematical type and the other is of a regular type, the compiler should report a type mismatch.

The only operation invocations which involve both regular and mathematical types are those of the model operations. The model operations can be invoked only using the **model** construct, and the type checking of their invocation is slightly different from that associated with the invocation of the provided operations. The invocation of a model operation is of the form

u := **model**[P](v)

where u is a variable of a mathematical type, P is a regular type, and v is a variable of a regular type. For the types involved in the operation to be consistent, v must have been declared to be of type P and the type of u must be structurally equivalent to the mathematical type corresponding to P. It should be noted, of course, that the type **math**[P] is structurally equivalent to the mathematical type corresponding to P.

CHAPTER V

The Run-Time Structure of RESOLVE Programs

The run-time structure of a RESOLVE program is an arrangement of the program representation, in executable form, in the memory of the target machine. It consists of the object code of the operations in the realization modules used in the program, and a structure representing the connections among the various facilities declared within the program. The design of the run-time structure of RESOLVE programs (henceforth called "RTS") is guided by two main objectives: efficiency of program execution, and the complete sharing of the code of a realization module by all the facilities which are instances of it.

We distinguish between two kinds of RTSs, the *plain* RTS and the RTS for testing RESOLVE parts. The former is used for the actual execution of a program in the application for which it is intended, while the latter, as its name suggests, is used for the interactive instantiation of realization modules and execution of operations in facilities from them, with the purpose of testing these realization modules. The two kinds of RTSs differ in two main aspects. First, while the plain RTS for a program

is completely known and set prior to the execution of the program, the RTS for testing is built incrementally. The structure is augmented whenever the user creates a new facility by instantiating a realization module. Second, the plain RTS does not include the code of the model operations in the realization modules being used, nor the code of any realization of a theory module. On the other hand, the RTS for testing includes the code of the model operations in every realization module used. Moreover, whenever the representation of a new facility is added to the existing structure upon the user's creation of the new facility, a corresponding structure of theory facilities is automatically constructed and linked properly to the existing structure so that the model operations, which call on operations from these theory facilities, can be executed.

Despite the differences between the two kinds of RTSs, the RTS in both cases is constructed from the same realization modules (in compiled form), and the same code in a compiled realization module should work with both kinds of RTSs. This implies that the two kinds of RTSs must be compatible with respect to the object code of the realization modules (except for the model operations).

The purpose of this chapter is to present the two kinds of RTSs and show how they are constructed. Because the RTS for testing is an extension to the plain RTS, we first present the plain RTS and then build on the ideas that will have been established to present the RTS for testing.

V.1. The Plain Run-Time Structure

A program in RESOLVE consists of a group of realization modules, where any realization module that is used in one of those in the group must also be in the group. One of the realization modules composing a program is designated as the *main module*. The main module must have no module parameters and must include a module initialization operation. A realization module designated as the main module of a program is not explicitly instantiated like the other realization modules are when facilities from them are declared. Instead, a single facility is implicitly created from the main module, and the rest of the facilities in the program derive from this facility: the facilities declared in the main module are created, then the facilities declared in the realization modules of the just created facilities are also created, and so on. When the execution of a program starts, control is at the beginning of the module initialization operation of the single facility of the main module.

Preparing a program for execution entails producing a memory image of its RTS, ready to be loaded into the memory of the target machine. To produce a memory image of the RTS of a program, the individual realization modules in the program must have been compiled into "compiled realization modules" (or CRMs for short). A program called *linker*, which has a means for retrieving CRMs by the names of their realization modules, is then used to construct the RTS of the program. Given the name of the main module of a program, the linker follows through the rest of the CRMs of the realization modules used in the program and constructs a memory image of the program's RTS.

Subsection V.1.1 explains the components of the plain RTS. An example RTS for a program is given in subsection V.1.2. Subsection V.1.3 describes how the linker constructs the RTS of a program from the CRMs of the realization modules composing the program.

V.1.1. Components of the Plain RTS

The RTS of a program is a representation of the program, in an executable form, in the memory of the target machine. The RTS includes the following:

- *Object code.* This is the object code of all the operations in the realization modules used in the program. There is only one copy of the object code of every operation in a realization module regardless of the number of facilities created from this realization module. That is, the code for the operations in a realization module is completely shared among all the facilities which are instances of it.

- *A structure representing the facilities used in the program.* This structure contains a record for each facility used in the program. Such a record will be referred to as the "run-time facility record" (or RFR for brevity). An RFR consists of the addresses of object code for operations and references to other RFRs as follows:

  - The RFR of a facility (say F) contains pointers to the object code of the operations in the realization module of F. These operations include the

explicitly provided operations, the implicitly provided operations (the initialization and finalization operations), and the private operations (i.e., those which are not available outside the module) in the realization module. In addition, the RFR of F contains pointers to the object code of other operations (in other realization modules) which are called in the code of F's realization module.

- The RFR of facility F contains pointers to the RFRs of the facilities providing the operations called in F.

The structure of the run-time representation of a program in RESOLVE is mostly dictated by the genericity of RESOLVE modules. Because of genericity and separate compilation, the same object code generated for the operations in a realization module can and should be used for all the facilities created from the realization module. Therefore, an operation call in RESOLVE specifies not only the operation to be called, but also the facility in the context of which the called operation is to be executed; otherwise, the effect of the called operation would be independent of the actual parameters to the module to which the operation belongs. Accordingly, the execution of RESOLVE programs takes place on a virtual machine which has, in addition to the conventional program counter, a "facility register" which points to the RFR of the facility providing the operation being executed. The "CALL" instruction in this virtual machine specifies both the address of an operation and the address of the RFR of the facility to which the operation belongs. The CALL instruction, then, has the effect of loading these two addresses into the program counter and the facility

register, respectively saving the current values on the run-time stack for restoration by a "RETURN" instruction.

However, also because of the genericity of RESOLVE modules, the effect of the CALL instruction itself must depend on the facility in the context of which it is being executed. For example, the effect of a CALL instruction generated for an invocation in realization module R of the initialization operation for type T, where T is a module parameter of R, should depend on the actual type for T (i.e., the effect of the CALL instruction should depend on the particular facility of R to which the instruction belongs). Therefore the addresses of the operation and RFR specified in a CALL instruction cannot be absolute; instead they have to be relative to the current RFR. This is made possible by having the CALL instruction in RESOLVE's virtual machine be of the form "CALL X, Y" where both X and Y are offsets from the beginning of the RFR pointed to by the (current) facility register. This instruction has the effect of making the facility register point to the RFR whose address is at offset X in the current RFR, and making the program counter point to the instruction whose address is at offset Y in the current RFR. This is why every facility used in a RESOLVE program is represented at run time as a set of pointers to RFRs and to object code. With this design, the object code generated for an operation invocation in RESOLVE (other than preparing the arguments) consists of a single call instruction, in spite of the possibility of genericity. The effect of this choice on efficiency becomes evident when it is noted that most of the operations in a RESOLVE program are procedure and function invocations.

The general form of an RFR is shown in Figure 1, and the components of this form are explained below:

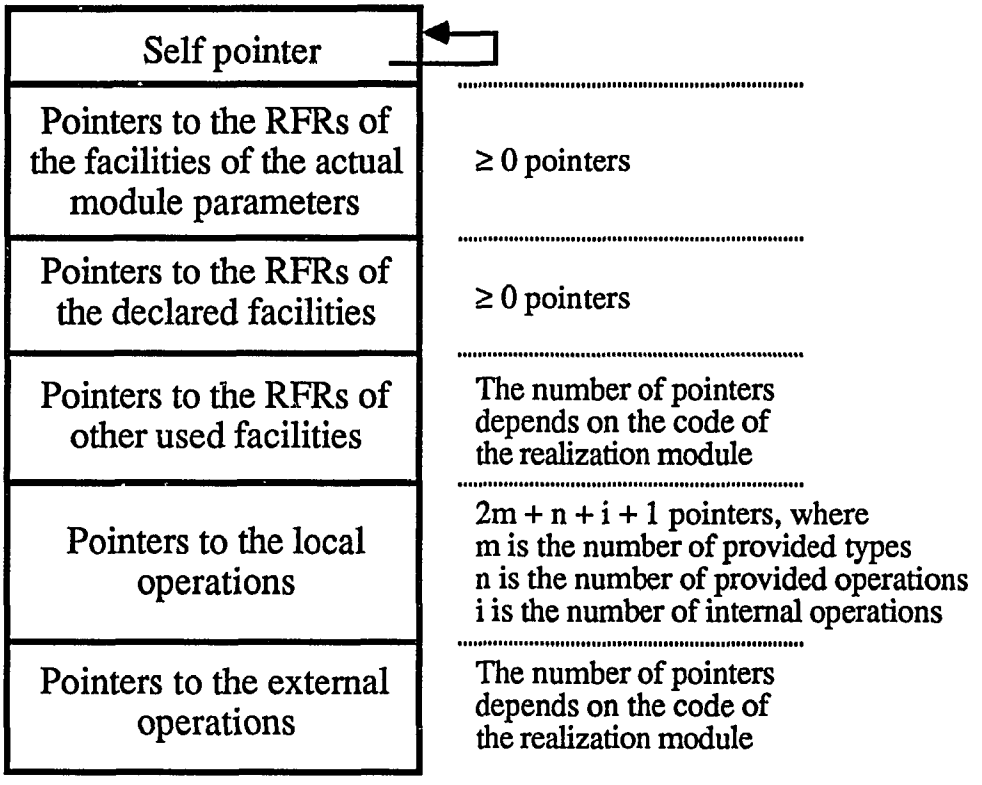


Figure 1. The contents of the RFR

- *Self pointer.* This is a pointer to the RFR itself. It exists to make it possible that the CALL instruction be used for calling operations which belong to the facility of this RFR. In this case the call instruction is of the form "CALL 0, Y."

- *Pointers to the RFRs of the facilities of the actual module parameters.* In the RFR of facility F, these are pointers to the RFRs of the facilities of the actual module parameters used in creating F. An actual module parameter can be a facility, a type, or an operation. If it is a facility, a pointer to the RFR of this facility is placed in the RFR of F. If it is a type (or operation), a pointer to the RFR of the facility which provides this type (or operation) is placed in the RFR of F.

- *Pointers to the RFRs of the declared facilities.* In the RFR of facility F created from realization module R, these are pointers to the RFRs of the facilities declared in R. Note that some actual parameters in the declarations of these facilities may be from the module parameters to R. Here, those actual parameters are adjusted to account for the actual parameters of R which are used in declaring F itself.

- *Pointers to the RFRs of other used facilities.* The code of a realization module may also use facilities other than those from the module parameters and the declared facilities. RESOLVE allows the code in a realization module to refer to parameters to facility parameters to the module. This is not limited to one level of depth; in general, the code in a realization module (say R) may refer to a parameter to a facility parameter to a facility parameter ... to a facility parameter

to R. We will refer to such a sequence of facility parameters as a *facility chain.* If the code of realization module R refers to facilities at the end of facility chains, pointers to the RFRs of these facilities are set in the RFRs of the facilities created from R. Similarly, if the code of R refers to type parameters to facilities at the end of facility chains, pointers to the RFRs of the facilities providing these types are set in the RFRs of the facilities created from R.

- *Pointers to the local operations.* These are pointers to the object code of the various operations in the realization module from which the facility of the current RFR is created. These operations include the provided operations, the initialization and finalization operations for the provided types, the module initialization operation, and the private operations which are not used outside the realization module. The pointers to these operations are set in the RFR in a specific order.

- *Pointers to the external operations.* These are pointers to the object code of operations provided by facilities created from other realization modules, where those operations are used by the code of the operations in R.

V.1.2. Example

In this subsection, we consider a simple RESOLVE program and illustrate its RTS. Because an illustration showing the complete details of the RTS for even a simple program may not fit into one page and still be intelligible, it will suffice for the

purpose to show only those parts of the RFRs which have pointers to RFRs. The pointers to operations' object code are not shown.

All our example program does is to create an empty stack of stacks of integers. To do that, the main module of the program might have the following facility declarations:

```
facilities
   facility Int_Fac is Integer_Template realized by Standard

   facility Stk_Int_Fac is Bounded_Stack_Template
               (Int_Fac.int, Int_Fac)
         realized by Constant_Time_Using_Array
      renaming
         Stk_Int_Fac.stack as stk_int
      end  renaming

   facility Stk_Stk_Int_Fac is Bounded_Stack_Template
               (stk_int, Int_Fac)
         realized by Constant_Time_Using_Array
end  facilities
```

Note that the choice of the realizations of the integer and the bounded stack modules in the above declarations is arbitrary. Recalling that the chosen realization of the bounded stack module uses the standard realizations of both the "plastic array" and the "triple" modules (as given in Chapter 2), and noting that the realizations of these two modules use no other modules, we conclude that the whole program consists of instances of the following realization modules:

- the main module,

- the standard realization of the integer module,
- the standard realization of the plastic array module,
- the standard realization of the triple module, and
- the constant-time-using-array realization of the bounded stack module.

Figures 2, 3, 4, 5 and 6 show partial forms for the RFRs of the facilities from the realization modules used in the program. The partial RTS for the program is shown in Figure 7, where the meaning of each pointer in an RFR is understood from the form for that RFR.

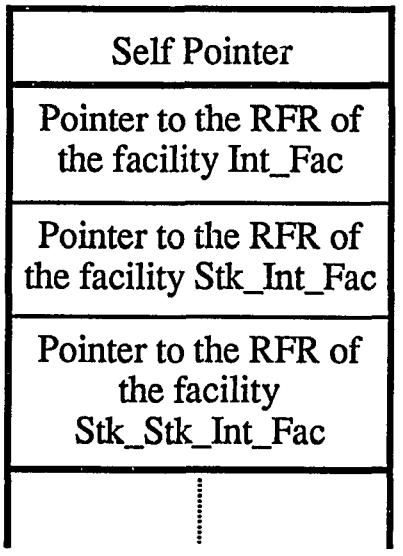


Figure 2. Partial form for the RFR of the main module

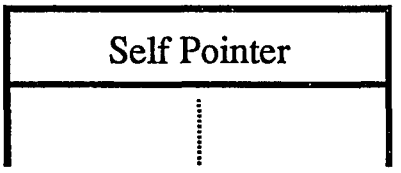


Figure 3. Partial form for the RFR of facilities created from the Integer module

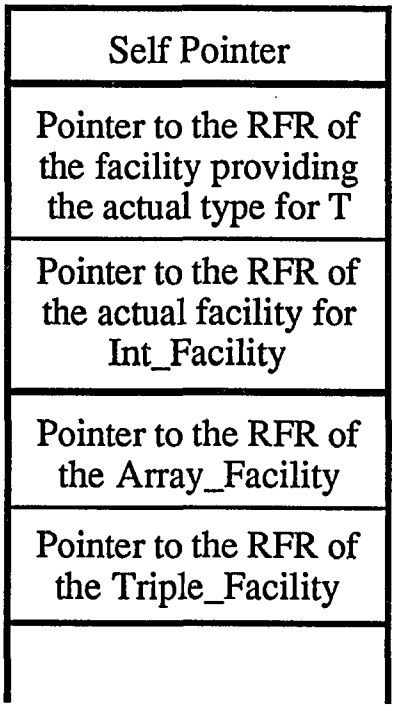


Figure 4. Partial form for the RFR of facilities created from the Bounded_Stack

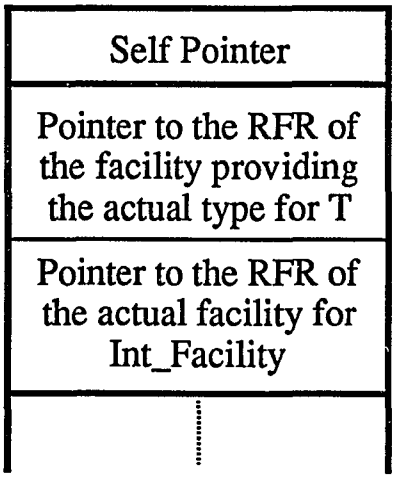


Figure 5. Partial form for the RFR of facilities created from the Plastic_Array

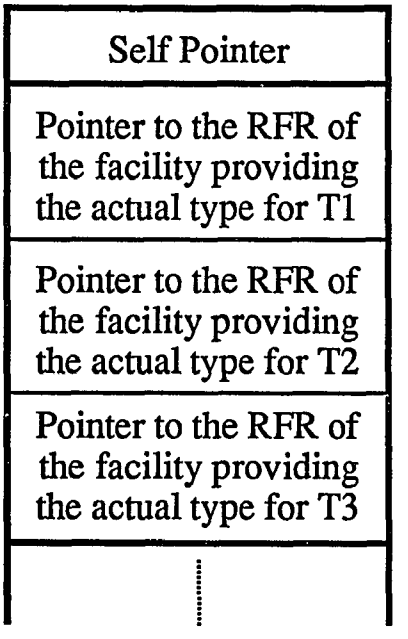


Figure 6. Partial form for the RFR of facilities created from the Triple module

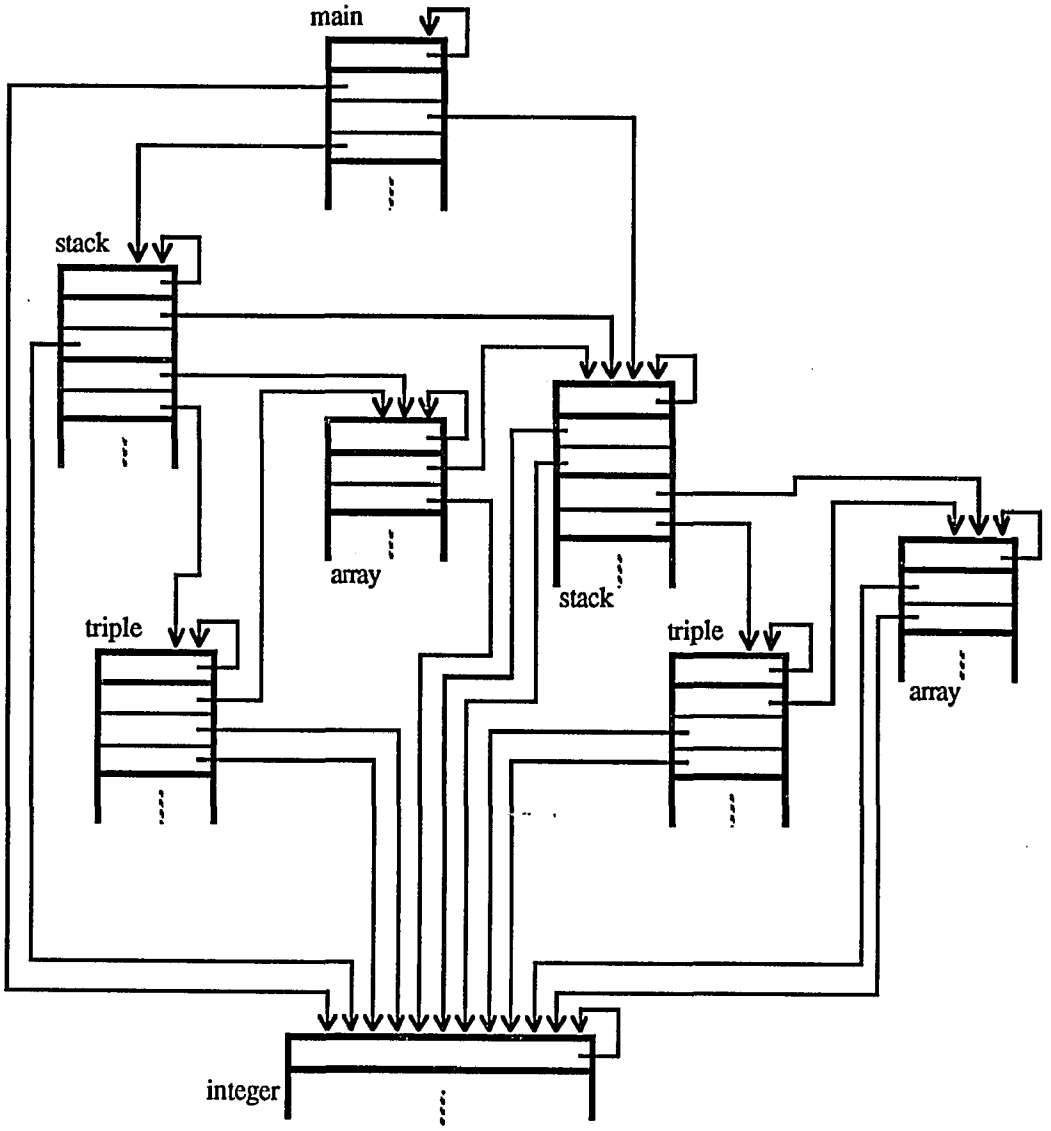


Figure 7. A partial RTS for a RESOLVE program

V.1.3. Constructing The Plain Run-Time Structure

The linker builds the RTS of a program using information in the CRMs of the realization modules which make up the program. Before showing how the linker constructs the RTS, we describe the information in the CRM.

V.1.3.1. The CRM

Beside having the object code of the various operations in a realization module, the CRM contains information about the module interface (for use by RESOLVE's compiler), linkage information to be used by the linker in constructing the RTS for a program involving the realization module, and additional information that is used in constructing the RTS in a testing environment. The linkage information consists mainly of:

- Descriptions of the facilities which are used by the code of the operations in the realization module. In general, such facilities are described in terms of the module parameters.

- Descriptions of the external operations (i.e., those operations in other realization modules) which are called by the code of the operations in the realization module. The description of each external operation includes both the operation

and the facility to which it belongs. In general, the description of an external operation is in terms of the module parameters or the facilities declared in the realization module.

For better manageability of the information in a CRM, this information is organized into a hierarchical structure of tuples, sequences, and discriminating unions, ending up with integer values and character strings. (This means that each CRM can be stored as an ordinary text file for easy portability.) Each CRM has its own hierarchical structure of information depending on the code in its realization module. However, all possible structures of information in CRMs can be characterized by a grammar whose non-terminal symbols correspond to components of the structures and whose terminal symbols are integers and character strings. Each production rule of the grammar characterizes a structural component of the information in a CRM in terms of tuples, sequences, and unions of other components, using a fairly standard version of the BNF notation. For example, a production rule such as:

<A> —> <B> <C>* | <D>

states that the structure of the component A can be either of two alternatives: a 2-tuple whose first component is the structural component B and whose second component is a (possibly empty) sequence of the structural component C, or the structural component D. The complete grammar which characterizes the structure of the information in CRMs, with embedded explanatory comments, is given in Appendix F.

The structured information in a CRM is produced by the compiler as the result of compiling a realization module, and this information is later accessed for processing by both the compiler and the linker. An abstraction which we call *GPD* (for General-Purpose Data structure) is used to hold the structured information in a CRM, and operations for constructing a GPD and accessing its components are provided for use by the linker (and by the program which constructs the RTS for testing). The GPD implementation is a recursive data structure which reflects the organization of the structured information in the CRMs: it can be made, through operations thereon, to be an integer, a character string, a record of GPDs, a sequence of GPDs, or a union of GPDs. Other operations on a GPD include those to access components of the structure, write the structure (in a specific internally-known format) into a file, and read a GPD from such a file. This approach relieves us from specifying a "file format" for a CRM, since all access to a CRM file (by the compiler and the linker) is through these procedures.

V.1.3.2. The Linker

Given the name of the main module of a program, the linker constructs an image of the RTS of the program in two integer arrays (which will ultimately become the contents of "instruction memory" and "facility record memory" of the RESOLVE virtual machine). The first array (the *code array*) contains the object code of the operations in the realization modules which make up the program. The second array (the *RFR array*) contains the RFR structure of the program, where each RFR consists

of offsets of RFRs in the same array and offsets of object code in the code array. Running a program, then, becomes a matter of loading data from these two arrays into the memory of the target machine, and starting execution at the first instruction in the module initialization operation of the main module, with the facility register initially pointing to the RFR of the main module's facility.

The workhorse of the linker is a recursive procedure which takes, as arguments, a CRM (in GPD form) and actual module parameters, and proceeds by loading the object code in the CRM into the code array (if the code is not already in the array) and constructing, in the RFR array, an RFR for the facility which corresponds to the given CRM and actual module parameters. The actual module parameters (if any) which are passed as arguments to this procedure are in terms of the part of the RTS that has been already constructed by the time the procedure is invoked. Specifically, if an actual module parameter is a facility, it is represented as the offset of that facility's RFR in the RFR array, and if the actual module parameter is a type (or an operation), it is represented as a pair of values: 1) the offset of the RFR of the facility providing the type (or the operation); and 2) the order of the type (or the operation) among those provided by the facility. The recursive nature of the procedure stems from the fact that in the process of constructing an RFR, it identifies the facilities declared in the realization module being used and calls itself to construct RFRs for these facilities.

Expressing the structure of the information in a CRM using a grammar plays a vital role in the design of the linker's procedure which constructs an RFR for a CRM

and actual module parameters. As pointed out earlier, the linkage information in a CRM consists mainly of descriptions of the facilities and the external operations used in a realization module, in terms of the module parameters. Using this information in setting an RFR requires that the information be translated, depending on the actual module parameters, into offsets of RFRs in the RFR array and offsets of object code in the code array. Using a grammar to characterize the structure of the information in a CRM makes such translation quite manageable by making it possible to express the translation in terms of the structure of information in a CRM, in a way similar to those used in syntax-driven translation schemes for programming languages.

Most of the non-terminals in the grammar have semantic values associated with them, where the semantic value of a non-terminal reflects its meaning in terms of the RTS. For example, the semantic value of <xtrnl op> (which describes a structure of information specifying an external operation) is an integer representing the index at which the code of the operation starts. Associated with each non-terminal which has a semantic value is an "evaluation procedure" (analogous to a "semantic routine" in syntax-driven translation) which takes a GPD representation of the structured information in an instance of the non-terminal and produces the corresponding semantic value. The evaluation procedure for a non-terminal calls on the evaluation procedures of the non-terminals in the right-hand side of the former's production rule. This has the desirable effect of decomposing the translation of the information in a CRM into a group of fairly simple procedures calling each other.

V.2. The Run-Time Structure for Testing

The RTS for testing is used in a testing environment where facilities are interactively created from the available realization modules, and operations in these facilities are interactively invoked. The RTS for testing differs from the plain RTS in two main aspects:

- The realization modules and the facilities used in the RTS for testing are not known prior to the construction of the RTS. Rather, the user of the testing environment may interactively refer to the realization modules which are available to the testing environment and create facilities from them, using actual module parameters from the facilities that have already been created. The creation of a new facility causes the proper RFRs to be set up in the memory of the target machine. An operation in a created facility can then be invoked interactively by setting up the operation's actual parameters, setting the facility register to point to the facility's RFR, and passing control to the designated operation.

- In a testing environment, the model operations are used to produce the representations of mathematical values of variables for the purpose of visualizing these values and checking assertions involving them. Consequently, the object code of the model operations in the realization modules being used is included in the RTS for testing. And since the code of the model operations uses theory facilities, whenever a regular facility is created, the theory facilities used by the

model operations in this regular facility are automatically created and their RFRs are constructed and linked properly to the current RTS.

Subsection V.2.1 presents an extension to the RFR structure, where the extension is used in the RTS for testing to accommodate the presence of the model operations. An example RTS is given in subsection V.2.2. In subsection V.2.3, we show how the RTS for testing is constructed.

V.2.1 Extension to the RFR Structure

The use of the model operations in a testing environment for testing RESOLVE modules requires using extended versions of the RFRs. The extension consists of additional pointers to RFRs and to object code. These pointers are referenced only by the object code of the model operations and by the linking procedure which incrementally constructs the RTS for testing. The extension recognizes the code of the model operations in the RTS for testing, the use of theory facilities, and the possibility of referencing mathematical types through the regular types that they model (using the **math** construct). The extension does not apply to the RFRs of the theory facilities since they do not have model operations. The general form of the extension to the RFR structure is shown in Figure 8, and the components of this form are explained below.

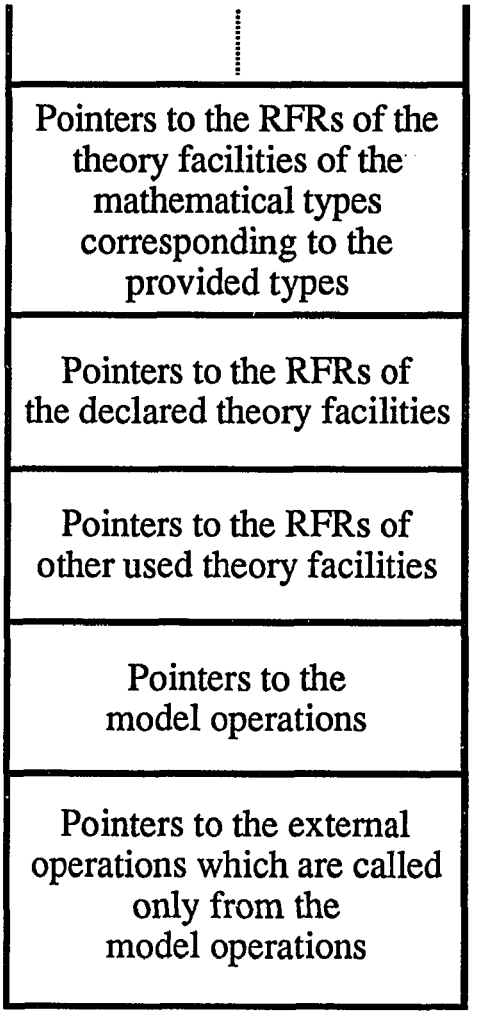


Figure 8. The contents of the extension to the RFR

- *Pointers to the RFRs of the facilities providing the mathematical types corresponding to the provided types.* In the RFR of facility F of realization module R, these are pointers to the RFRs of the theory facilities which provide the mathematical types corresponding to the types provided by R, after adjusting the formal module parameters of R to their settings in F's declaration. These pointers are needed so that if another realization module (say M) refers to the mathematical type corresponding to a type provided by R (using the **math** construct), a pointer to the RFR of the theory facility providing this mathematical type can be set in the RFR of a facility from M.

- *Pointers to the RFRs of the declared theory facilities.* In the RFR of facility F of realization module R, these are pointers to the RFRs of the theory facilities declared in R. Note that some actual parameters in the declarations of these theory facilities may be in terms of the module parameters to R. Here again, those actual parameters are adjusted according to the actual parameters of R which are used in declaring F.

- *Pointers to the RFRs of other used theory facilities.* In the RFR of a facility of realization module R, these are pointers to the RFRs of the theory facilities that are used by the object code of the model operations in R, but are not declared in R. Specifically, these theory facilities are those which provide the mathematical types that are referred to in the realization module using the **math** construct.

- *Pointers to the model operations.* These are pointers to the object code of the model operations in the realization module from which the facility of the current

RFR is created. (There may be more than one because a module can provide more than one type.)

- *Pointers to the external operations called only from the model operations.* These are separated from the pointers to the external operations which are called from other operations in the realization module for more space efficiency in a non-testing situation where the model operations are not used.

V.2.2. Example

We consider an example in a testing situation where first an integer facility is created, and then a bounded stack-of-integer facility is created. The corresponding facility declaration code is:

```
facility Int_Fac is Integer_Template
      realized by Standard

facility Stk_Int_Fac is Bounded_Stack_Template
                                  (Int_Fac.int, Int_Fac)
      realized by Constant_Time_Using_Array
```

Parts of the RTS are shown after each of these two steps. For simplicity, only those parts (in an RFR) which include pointers to other RFRs are shown. Figure 9 shows the RTS upon the creation of the integer facility, and Figure 10 shows it after the bounded-stack facility is created. For better intelligibility, each RFR for a regular

facility is divided, by a thick line that extends slightly out of the RFR boundaries, into two parts: the original part of the RFR and its extension part. Regular facility RFRs are on the left, and those of theory facilities are on the right. Note that the creation of the bounded stack facility automatically triggers the creation of the array and the triple facilities which are declared in the bounded stack realization module. Also, whenever a regular facility is created (explicitly by the user or implicitly as just mentioned), the theory facilities declared in it are automatically created. For example, as shown in Figure 9, when the "int" facility is created, the "math. int" facility is automatically created. The RFR of the math. int facility is pointed to by two entries in the RFR of the int facility; the first entry represents the mathematical type corresponding to type int, and the second entry represents the theory facility declared in int.

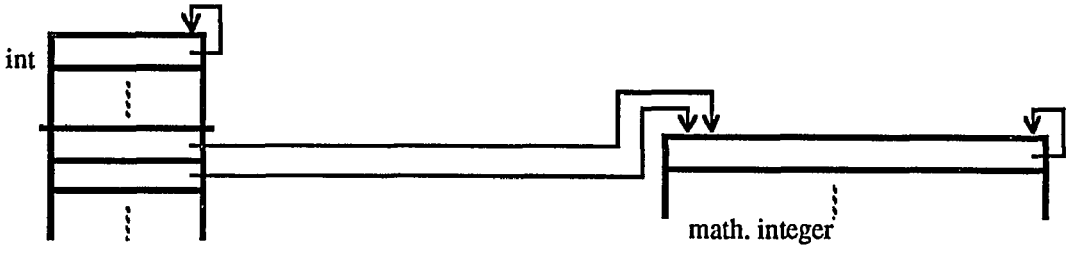


Figure 9. Partial RTS upon creating an integer facility

Recall (from the preceding subsection) that the extension part of an RFR includes, in general, three sets of pointers to RFRs. These pointers are shown in the

stack RFR, for example, in Figure 10 in three distinct parts below the extended thick line. The first part contains one pointer which points to the RFR of the theory facility that provides the mathematical type corresponding to the type stack (i.e., the RFR of the pair facility). The second part contains pointers to the RFRs of the theory facilities declared in the stack module (the string facility, the mathematical integer facility, and the pair facility). Noting that the stack realization module refers to the mathematical type corresponding to the type int (through the **math** construct), the third part contains a pointer to the RFR of the mathematical integer facility.

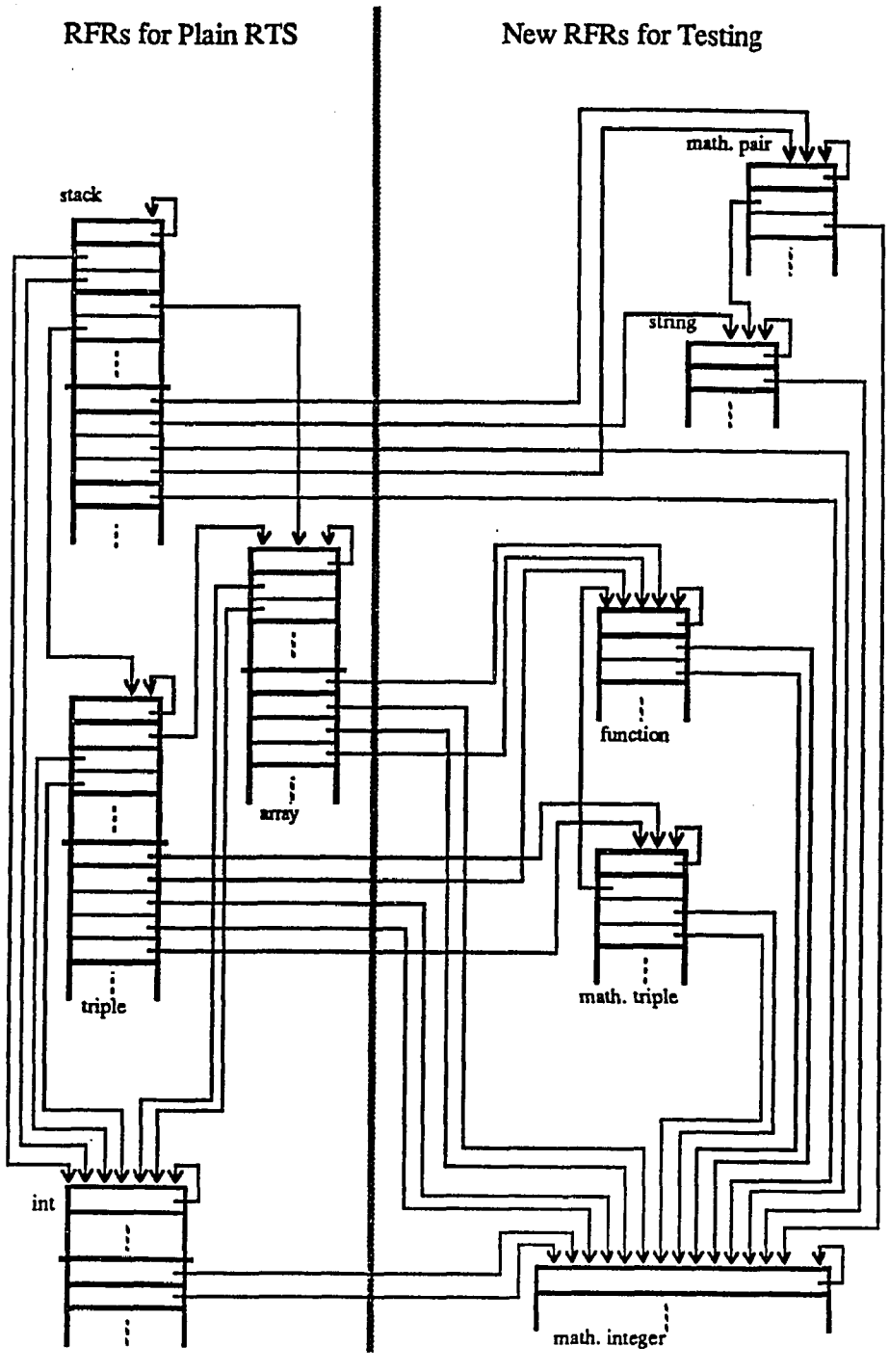


Figure 10. Partial RTS after creating a bounded stack facility

V.2.3. Constructing the RTS for Testing

Constructing the RTS for testing is quite similar, in principle, to constructing the plain RTS. The structure of the information in the CRMs is extended to include the information needed to set up the extensions to the RFRs of the regular facilities and to set up RFRs for the declared theory facilities. The grammar describing the structure of information in the CRMs is extended accordingly, and additional evaluation procedures are associated with the non-terminals introduced by this extension.

When the user of a testing environment creates a facility from a given realization module R, with actual module parameters from the facilities which have been already created, the CRM of R is passed to a procedure similar to that described in 5.1.3.2, along with the actual module parameters in terms of the already constructed parts of the RTS. This procedure loads the object code of R into the memory of the target machine (if the code is not already there) and constructs an extended RFR for the facility being created. In constructing the RFR of this facility, the procedure identifies the theory facilities and the regular facilities declared in the realization module being used, and calls itself to have the necessary run-time structure for these facilities constructed and linked properly to the rest of the RTS.

CHAPTER VI

An Environment for Testing RESOLVE Modules

This chapter introduces an environment for testing RESOLVE modules. We call this environment TERM (for Testing Environment for RESOLVE Modules). Our main purpose for designing and introducing TERM is to demonstrate the feasibility and utility of our approach to accommodating the requirements of testing RESOLVE modules. Therefore, the key factor behind most of the design decisions in TERM has been to maintain simplicity while attaining a certain degree of practical usefulness.

A software environment, in general, is a set of integrated tools which help human individuals (or teams) perform certain tasks more effectively. Accordingly, an environment for testing RESOLVE modules should provide tools to assist in the process of testing these modules. Many such tools and different possible user interfaces associated with them are conceivable. However, we have focused in TERM on those tools which are directly related to our approach for accommodating the requirements of testing RESOLVE modules. Namely, these tools allow for 1) producing test points interactively in a step-by-step fashion, 2) visualizing the

mathematical-model views of RESOLVE variables at arbitrary points of time during their use, and 3) having certain assertions involving mathematical-model views of RESOLVE variables checked automatically. From our discussion (in chapter III) of the problems involved in testing ADTs which are specified using the model-based approach, and of the requirements for solving these problems, it is apparent that any testing environment for RESOLVE modules must provide the abovementioned tools in order for it to be practically useful.

The rest of this chapter is organized into two main sections. Section VI.1 describes TERM from its user's point of view, and section VI.2 explains the implementation of TERM and explains how it works.

VI.1. User Perspective of TERM

TERM is a graphical environment which allows its user to interactively do the following:

- Create facilities from the available realization modules.
- Create new initialized variables of the types provided by the already created facilities.
- Perform operations provided by the created facilities using, as actual parameters, variables from among those which have been created already. The variables

involved in an operation may then have new values according to the actual effect of the operation. Meanwhile, the mathematical values corresponding to the old values of the variables involved become available to the user so that he may check assertions involving these values.

- Have the mathematical value corresponding to a given variable automatically produced. Such mathematical values may then be used as actual parameters to operations provided by theory facilities. This can prove helpful in checking assertions involving large conglomerate structures. For example, checking the equality of two large sets of strings of pairs of integers can be accomplished by performing the *Equal* operation provided by the theory facility which provides the mathematical type of the two sets.

- Graphically visualizing, at different levels of detail, the mathematical values corresponding to variables. For example, a mathematical value which is a tree of strings of integers is rendered, at the highest level of abstraction, as an icon indicating that the value it represents is a tree. *Opening up* this icon results in displaying a tree of icons, each of which indicates that the value it represents is a string. In turn, opening up any of these icons causes a string of icons to be displayed, where each icon in the string has an integer value displayed inside it.

TERM is designed to be used on the Macintosh®. The user interface of TERM, therefore, takes advantage of such features as icons, windows, and pull-down and pop-up menus.

RESOLVE variables are presented in TERM as icons. There are two visually different kinds of icons in TERM: regular icons and math icons. Regular icons correspond to variables of program types and math icons correspond to variables of mathematical types. The user can attach a user-given name to an icon for later identification of the variable denoted by the icon. Such names may be changed by the user at any time. In addition, the facility name and the type name of the variable represented by a regular icon are automatically attached to the icon. For math icons, only the type names of the variables they represent are attached to the icons. Moreover, if a math icon represents a mathematical variable of a *primitive* type (e.g., integer or boolean), the value of the variable is displayed inside the icon.

Regular icons and those math icons which are not part of a structure of icons (e.g., a tree of icons) can be dragged around. On the other hand, math icons which are part of a structure of icons cannot be dragged. This is essential since a structure of math icons is regarded as the details of a value, rendered as icons representing components of the value and graphical representations of relations among the value's components (e.g., lines representing tree branches connecting icons). Moving icons in a structure of icons would, then, undermine the graphical representation of the structure. However, the immobility of such icons is compensated for by making it possible for the user to make a copy of the mathematical value corresponding to any math icon, where the icon of the duplicate variable can be dragged freely. The icons which can be dragged around will be referred to as *free* icons, while those which cannot will be referred to as *fixed* icons.

Icons in TERM are displayed in a *main* window and as many *math-view* windows as is needed. All icons of the user-created variables and new icons which result from executing operations are always in the main window. When the user opens up a math icon representing a mathematical value of a non-primitive mathematical type, the next level of detail of this value is displayed as a structure of icons in a new *math-view* window. Icons may not be moved from a window into a math-view window. On the other hand, copies of fixed icons in math-view windows may be moved into the main window.

In addition to the standard Macintosh system menus (the , **File**, and **Edit** menus), TERM has four pull-down menus:

- *The Realization Modules Menu.* This menu contains the names of the realization modules which have been loaded into the environment. Choosing a realization module from this menu causes the display of information about this realization module, including the module's header, the names of the types it provides, and the headers of the operations it provides. This menu also contains the command **Load RM** which is used to load realization modules into the environment so that facilities from them can be created and used.

- *The Facilities Menu.* This menu has the names of the facilities which have been created by the user. Choosing a facility from this menu results in the display of information about this facility, including the actual module parameters used in creating it. This menu also has the command **Instantiate** which is used to

create a facility from a realization module by supplying its name and actual module parameters.

- *The Program Menu.* This menu includes commands corresponding to the user actions that involve program variables (as opposed to mathematical variables). These commands are:

  - *Initialize.* This is used to create a new initialized variable of a type to be specified by the user.
  - *Model.* This is used to produce a mathematical variable whose value is the mathematical-model value corresponding to the value of a program variable.
  - *Finalize.* This finalizes a given variable.
  - *Regular Op.* This is used to perform an operation provided by a regular facility that has been created. The facility, the operation, and the actual parameters to the operation are to be specified by the user.

- *The Math Menu.* This menu includes commands corresponding to the user actions that involve mathematical variables. These commands are:

  - *Theory Op.* This is used to perform an operation provided by a theory facility.
  - *Open Up.* This displays, in a new window, the next level of details of the value represented by a given math icon.
  - *Duplicate.* This produces a copy of a given mathematical variable.
  - *Finalize.* This finalizes a given mathematical variable.

Following is a description of the scenario involved in each major user action.

Loading a Realization Module

When the user chooses **Load RM** from the realization modules menu, a dialog box (as shown in Figure 11) displays the names of the realization modules that may be loaded. The user may then select a realization module and click in **Load** to have the module loaded. Once a realization module has been loaded, its name is added to the list of realization modules in the realization modules menu, and facilities from it may be created.

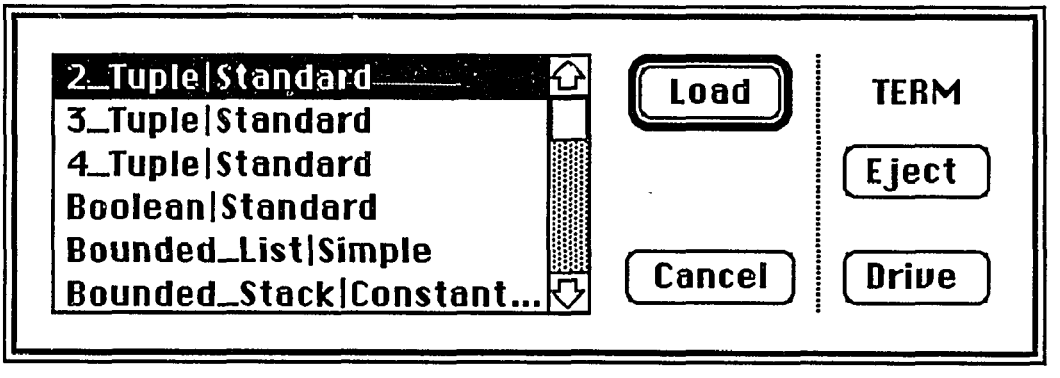


Figure 11. A dialog box for selecting a realization module

Creating a Facility

Choosing **Instantiate** from the facilities menu causes a dialog box (like the one shown in Figure 12) to appear, through which the user can select a template and a realization module, set the actual module parameters, and a give a name to the facility to be created. The user should select a template from the **Conceptualization** pop-up menu (as shown in Figure 13), and choose a realization of the selected template from the **Realization** pop-up menu. (Upon selecting a template, the **Realization** pop-up menu contains the available realization modules of the selected template.) After a conceptualization and a realization are selected, the module's formal conceptual and realization parameters are displayed in two parameter boxes, one for the conceptual parameters and the other for the realization parameters. Figure 14 shows the dialog box of Figure 12 after a realization of the bounded stack template is selected.

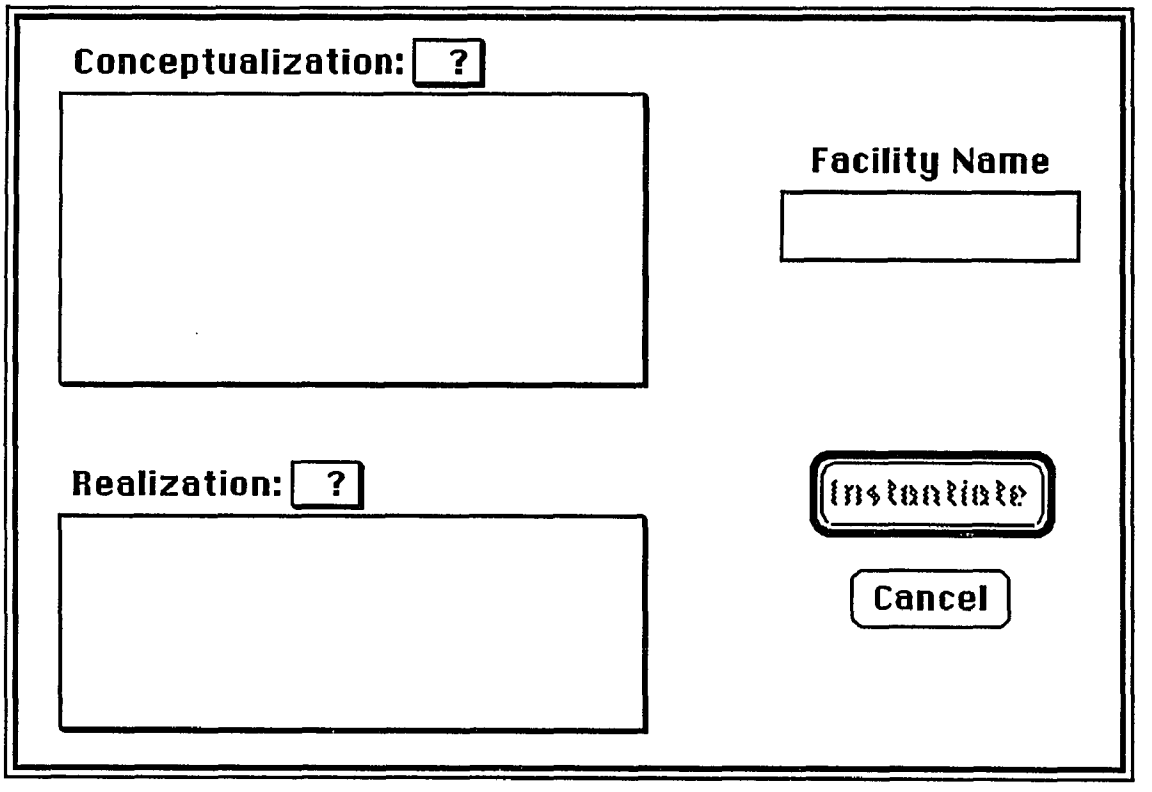


Figure 12. Instantiation dialog box

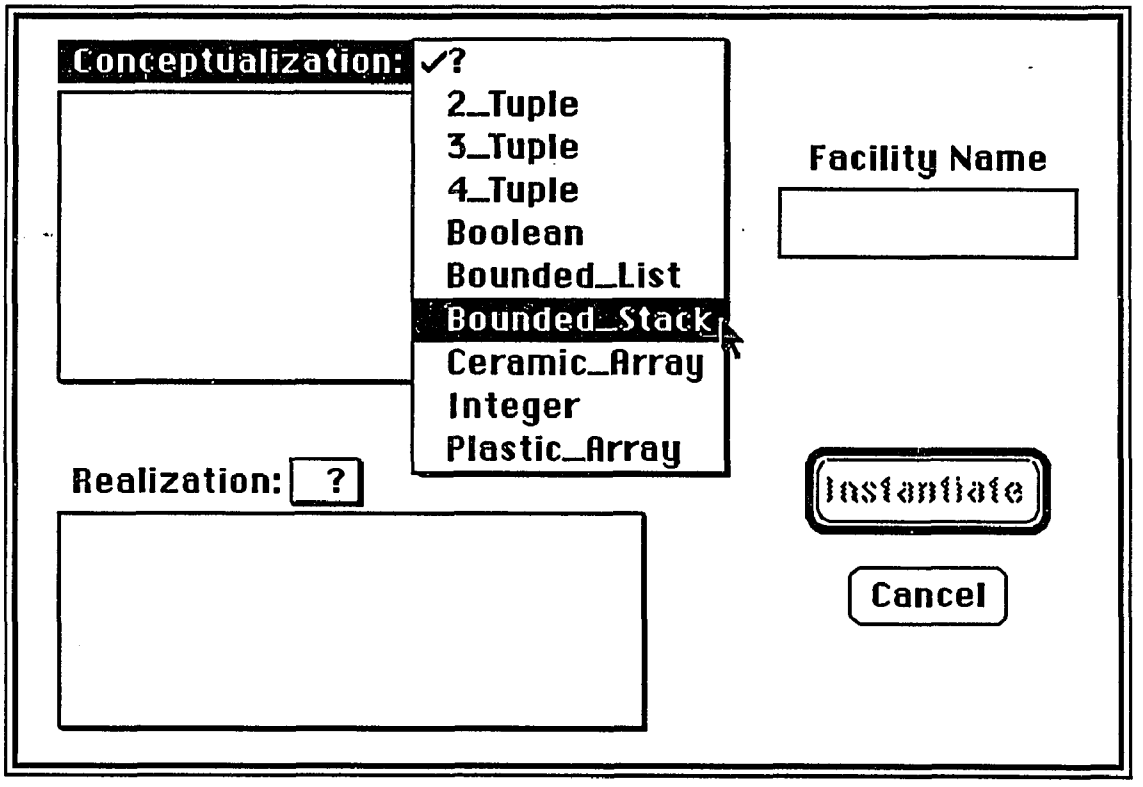


Figure 13. Selecting a template in the instantiation dialog box

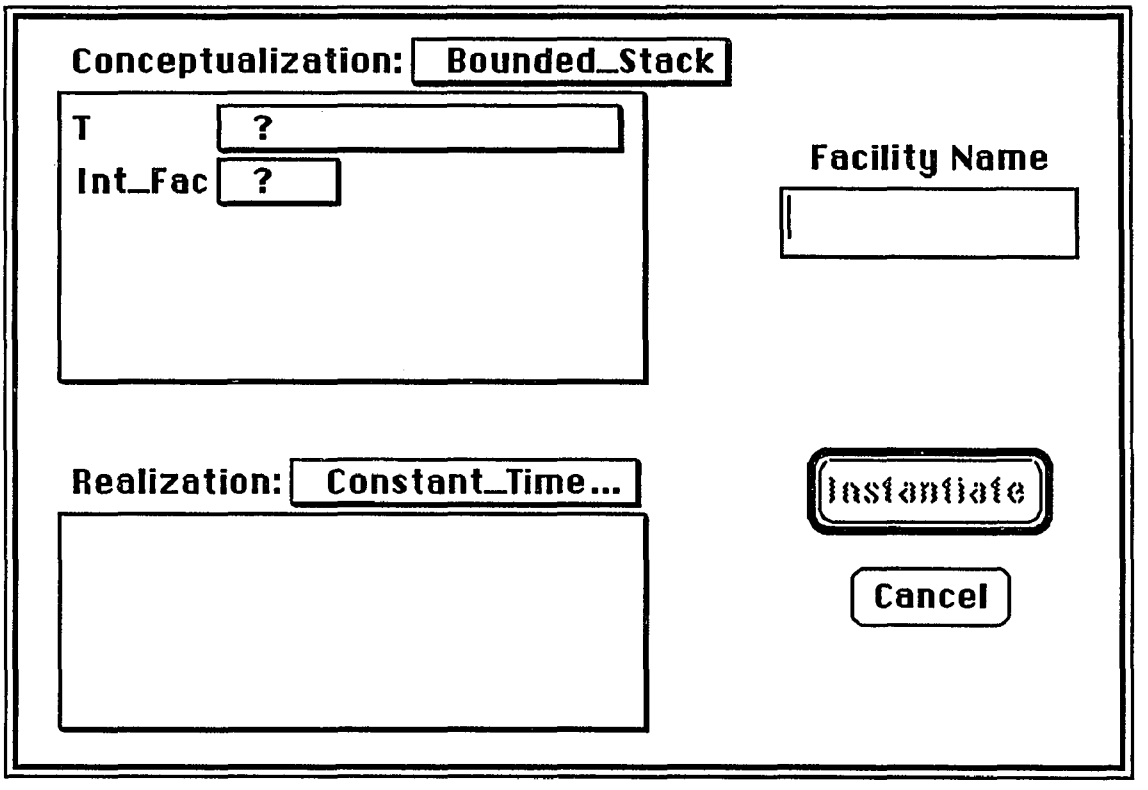


Figure 14. Instantiation dialog box with a template and a realization module selected

Now, the user has to supply actual module parameters for type T and facility Int_Fac. There is a pop-up menu next to each formal parameter, listing the possible actual parameters corresponding to the formal parameter, and the user can choose one of these actual parameters. If the formal parameter is a facility, the corresponding pop-up menu lists all the facilities that have been created from the template of the formal parameter. If the formal parameter is a type, the corresponding pop-up menu is a two-level hierarchical menu. The first-level menu lists all the available facilities and the second-level menu lists the types provided by a facility (only those types which do not violate the restrictions imposed on the actual parameters in the

restrictions section of the module are listed). Similarly, if the formal parameter is an operation, the corresponding pop-up menu is a two-level hierarchical menu where the first-level menu lists the available facilities and the second-level menu lists the operations provided by a facility. (Only those operations which match the formal parameter and which do not violate the restrictions are listed.)

After an actual parameter is selected, it is displayed in the box of the pop-up menu of the corresponding actual parameter. Figure 15 shows the dialog box of Figure 14 after setting the two module parameters. Also, after an actual parameter is selected, the corresponding pop-up menu contains only one item (**Cancel**). The user may cancel the setting of an actual parameter at any time by selecting **Cancel** from the corresponding pop-up menu. Canceling the setting of an actual parameter causes **?** to be displayed in the box of the corresponding pop-up menu.

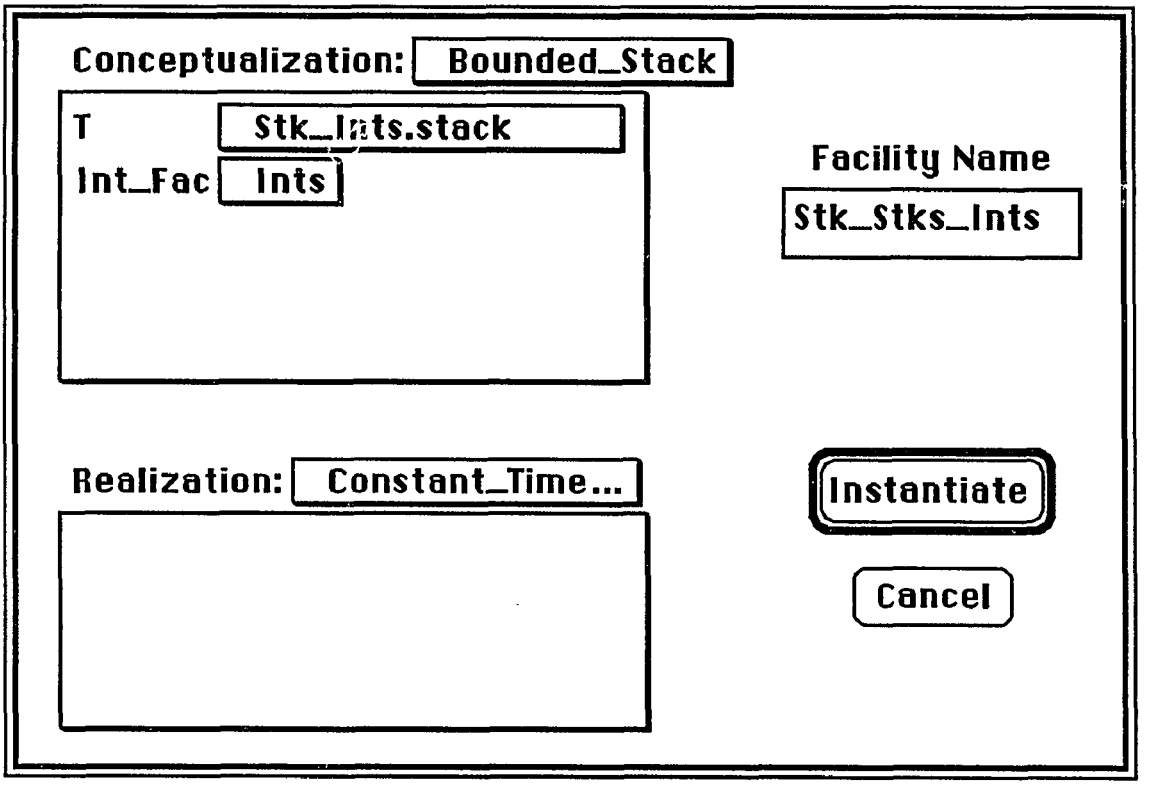


Figure 15. Instantiation dialog box with formal parameters set

After setting all the actual parameters to the module, the user should type in, in the **Facility Name** box, a name for the facility to be created. Clicking in **Instantiate**, then, triggers the creation of the facility, and the facility's name is added to the facilities' menu.

Creating a New Initialized Variable

When the user chooses **Initialize** from the program menu, a dialog box (as shown in Figure 16) appears, through which the user can select the type of the variable to be initialized and give a name to this variable. The **Type** pop-up menu is a two-level hierarchical menu. The first-level menu lists the available facilities (i.e., those facilities which have been created), and the second-level menu lists the types provided by a facility. After selecting a type, the user must give a name to the variable to be initialized. Initializing the new variable is then triggered by clicking in **Initialize**. An icon for the new variable then appears near the upper right corner of the main window.

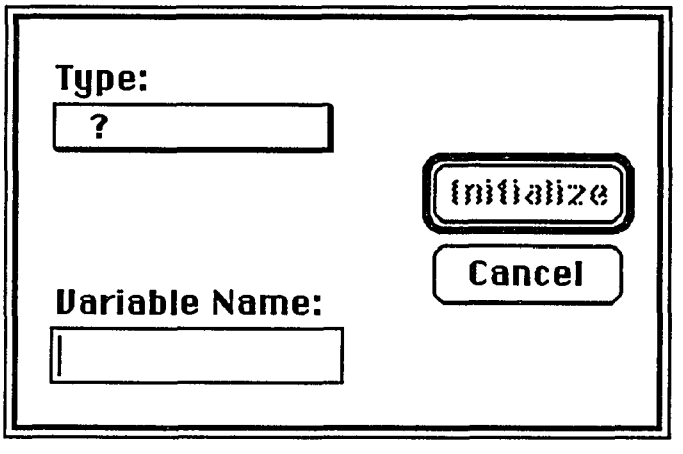


Figure 16. A dialog box for variable initialization

Performing a Regular Operation

Upon choosing **Regular Op** from the program menu, a dialog box (similar to that used for selecting a type for creating a new variable) is used so that the user selects an operation provided by a facility that has been created. After an operation is selected, the dialog box goes away and an empty *operating room* for the selected operation appears. An empty operating room for a regular operation which has two parameters is shown in Figure 17. The upper part of this operating room has places for two regular icons to be brought into, and the variables represented by these two icons will be taken as the actual parameters to the operation, in order. **Go** is used for triggering the execution of the operation, and **Cancel** for canceling it. The lower part of an empty operating room has nothing in it.

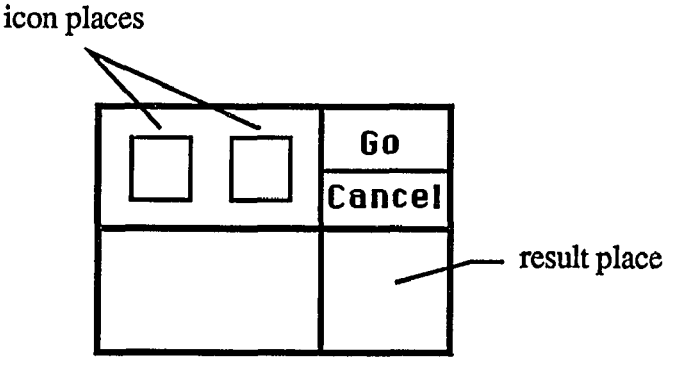


Figure 17. An empty operating room for a regular operation

The user provides actual parameters for the operation by dragging icons into the icon places in the operating room. However, an icon representing a variable whose

type is different from the expected type for a parameter will not go into the icon place corresponding to this parameter. After all the actual parameters for the operations are provided, the operation execution may be triggered by clicking in **Go**. Upon the execution of the operation, the following happens in the operating room:

- The icons representing the actual parameters move to the lower part of the operating room. They retain their user-given names, but now they represent the variables of the actual parameters as they stand at the conclusion of the operation (i.e., these variables may have different values now, depending on the actual effect of the operation).

- In place of each actual parameter icon in the upper part of the operating room, there appears a math icon representing the mathematical value corresponding to the value of the actual parameter before executing the operation. Each such icon is given the name of its corresponding regular icon, prefixed by M_, and the user may change this name later.

- If the operation is a function, an icon representing the variable returned by the function appears in the *result place* (at the right lower corner of the operating room), and the user is asked to give a name for this icon through a dialog box. If the operation is a control, either **Yes** or **No**, depending on the result of the operation, is displayed in the result place. If the operation is a procedure, nothing is displayed in the result place.

Afterwards, the user must drag all icons out of the operating room into the main window. When the last icon has been removed, the operating room goes away.

Producing the Model of a Regular Variable

Model in the program menu is active only if a regular icon is selected. Choosing **Model**, then, causes a new math icon to be produced next to the selected regular icon. It represents a mathematical variable whose value is the mathematical-model value corresponding to the value of the regular variable represented by the regular icon. The produced math icon is automatically given the name of the corresponding regular icon, prefixed by M_, but this name may be changed later by the user.

Performing a Theory Operation

A theory operation can be either a function or a control. **Theory Op** in the math menu is active only when a math icon is selected. Choosing **Theory Op** from the math menu, then, signifies a request to perform an operation from the theory facility which provides the type of the mathematical variable represented by the selected icon. A theory facility is selected in such a way because theory facilities are neither created nor named by the user; that is, they can only be referred to through variables of the types they provide.

Upon choosing **Theory Op** from the math menu, a dialog box listing the operations provided by the implicitly selected theory facility appears. Figure 18 shows an example of such dialog box. After one of these operations is selected, the dialog box goes away and an empty operating room for the selected operation appears. Figure 19 shows an empty operating room for a theory operation which has two parameters. This operating room has places for two math icons to be brought into, and the mathematical variables represented by these two icons will be taken as the actual parameters for the operation, in order. Unlike regular operating rooms, theory operating rooms do not have lower parts. The reason is that theory operations are "trusted" operations, and since according to their semantics the values of their actual parameters are not changed, there is no need for different icons representing values of the actual parameters both before and after the execution of an operation.

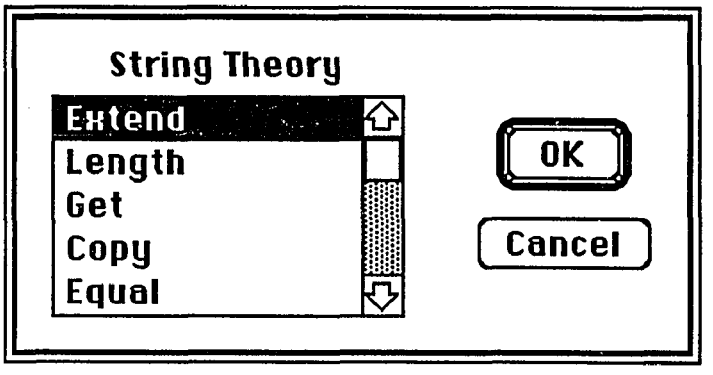


Figure 18. A dialog box for selecting a theory operation

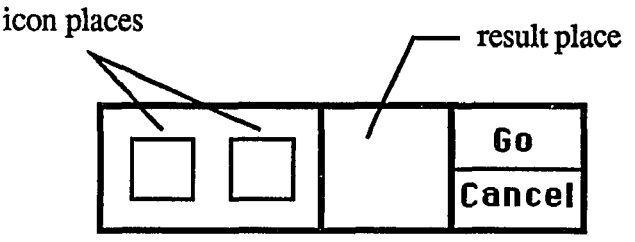


Figure 19. An empty operating room for a theory operation

The user provides actual parameters for the operation by dragging icons into the icon places in the operating room, and an icon representing a mathematical variable whose type is different from the expected type for a parameter will not go into the icon place corresponding to this parameter. After all the actual parameters for the operation are provided, the execution of the operation may be triggered by clicking on **Go**. Upon the execution of the operation, the operation's result appears in the result place in the operating room. If the operation is a control, either **Yes** or **No** is displayed in the result place. If the operation is a function, an icon representing the value returned by the function appears in the result place, and the user is prompted to give a name for this icon through a dialog box. As in regular operations, the user must then drag all icons out of the operating room into the main window, and when the last icon has been removed, the operating room goes away.

Opening Up a Math Icon

Open Up in the math menu is active only when a single math icon is selected, to which **Open Up** will apply. Opening up a math icon representing a value of a

primitive mathematical type (e.g., integer) results in a dialog box displaying the primitive mathematical value. This can be useful in case the primitive value is too long to fit into the math icon. On the other hand, opening up a math icon representing a value of a composite type (e.g., a string of pairs of integers) results in the opening of a new math-view window whose title is the same as the name of the math icon being opened up. In this window is displayed a structure of fixed math icons representing the components of the value represented by the opened-up icon. Figure 20 shows an example of such a structure for a value of string of pairs of integers. Each icon in the structure may be further opened up.

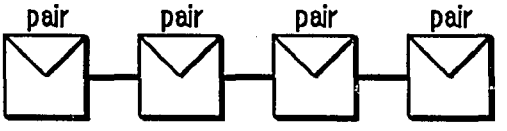


Figure 20. An icon structure representing a string of pairs

Duplicating a Mathematical Value

The possibility of making a copy of a mathematical value is provided for in TERM so that a free math icon representing a value equal to that represented by a fixed icon may be produced. **Duplicate** in the math menu is active only when a math icons is selected. Choosing **Duplicate**, then, causes a new math icon to be produced next to each selected math icon, where the new math icon represents a mathematical value equal to that of the selected math icon.

Finalizing a Variable

Finalize in the program menu is active only when a regular icon is selected. Choosing **Finalize**, then, causes the selected icon to disappear, and the variable it represents to be finalized. Similarly, **Finalize** in the math menu is active only when a free math icon is selected, and it has the effect of causing this icon to disappear and having the variable it represents finalized. It should be noted that closing a math-view window finalizes all the variables represented by the icons in the window.

VI.2. Implementation

The implementation of TERM is based on four major components: an interpreter for RESOLVE modules, a set of display operations, an icon-to-variable mapping, and user interface operations. Each of these components provides operations which are used by some of the other components and by the top level of the environment. Programming the top level of the environment is fairly straightforward using these operations and operations from the Macintosh User Interface Toolbox. Description of the major components on which the implementation of TERM is based follows.

The Interpreter

This interpreter is a piece of software which provides operations that are used to interactively create facilities from RESOLVE modules, create new variables of the types provided by the created facilities, and perform operations in these facilities. In this sense, the interpreter does not *interpret* RESOLVE modules; rather, it interprets such actions on RESOLVE modules as creating facilities and variables, and invoking operations. This kind of interpretation is, in fact, the one which is needed by an environment like TERM where the user activities are allowed to be in an order which does not necessarily mirror legitimate RESOLVE code. For example, TERM allows creating facilities after variables of types provided by other facilities are created and after operations from other facilities are performed, whereas in RESOLVE code all facilities must be declared before any of them may be used to declare variables of types provided by it or to invoke an operation in it.

Realization modules from which facilities are to be created are brought into the interpreter by calling an operation in the interpreter which reads in a compiled realization module, given its name, from a file. Reading in a compiled realization module causes the compiled realization modules of the theories referred to in this module to be automatically read in too. It should be noted that the source code of realization modules is not needed at all for the functioning of the interpreter. Using realization modules in compiled form, then, is advantageous in that it does not assume that the source code of realization modules is always available, and in that it

allows for testing realization modules in the form in which they are actually used: as object code.

Conceptually, the interpreter may be viewed as a repository of realization modules, facilities, and variables, and operations which manipulate the contents of this repository as follows:

- Adding realization modules to the repository by reading them in, given their names.
- Adding facilities to the repository by instantiating realization modules that are already in the repository, and using actual module parameters from the facilities already in the repository.
- Adding new variables to the repository by creating new initialized variables of types provided by the facilities in the repository, or by performing functions provided by these facilities.
- Changing values of variables in the repository through performing procedures provided by the facilities in the repository.
- Removing variables from the repository by finalizing them.

The interpreter has other supporting operations also. A list of the interpreter's operations along with brief informal description of each is given in Appendix G.

The Display Operations

TERM uses a set of display operations, one corresponding to each mathematical theory in the set of theories which are used in model-based specification in RESOLVE. These operations are used to produce the effect of "opening up" a math icon in TERM. This is accomplished by accessing the components of the value of the mathematical variable corresponding to the icon being opened up and assigning copies of the values of these components to new variables. New math icons corresponding to these new variables are displayed in a graphical structure which mirrors the mathematical domain of the variable whose icon is opened up. For example, if the variable is a string, the new icons are displayed as a horizontal sequence of icons connected by short lines. Appendix H outlines the display operation for strings, as an example. It should be noted that the theory modules must provide the necessary operations needed by the display operations to access the components of mathematical variables. Moreover, in some cases, the theory modules should provide operations which provide information that may be needed in formatting the structure of icons to be displayed (e.g., the number of nodes in level i of a given tree).

The Icon-Variable Mapping

This is a mapping from the icons in TERM to the variables they represent. It provides operations to add icon-variable pairs to the mapping, remove a pair from the mapping, and get the variable that a given icon maps to.

The User Interface Operations

These are operations that handle the dialog with the user which is involved in the user's actions in TERM. As an example, one of these operations is invoked when the user chooses **Instantiate** from the action menu. This operation, then, administers the instantiation dialog with the user and returns identification of the realization module to be instantiated, the actual module parameters that are selected by the user, and the name which the user gives to the facility to be created.

CHAPTER VII
Conclusions

VII.1. Summary

A promising approach to achieving better software quality and productivity is one in which software systems are constructed by assembling them from standard reusable software components. Among the desirable characteristics of such software components is that they be modules providing generic ADTs that are formally specified using the model-based specification method, and that they be certifiably correct. The research presented in this dissertation addresses the requirements of certifying such software components via testing, using RESOLVE modules as representatives of these components.

Three basic requirements for testing modules have been identified. First, because the representations of program types are not necessarily known or accessible to the tester due to the information hiding principle inherent in ADTs, and because the specifications of RESOLVE modules are expressed in terms of the mathematical-

model domains, there needs to be a way for presenting to the tester the mathematical model views of values of variables. This is needed in two contexts: 1) evaluating the test executions of operations, and 2) ascertaining that a produced test point is indeed the one intended. Second, because the process of assertion checking is usually repeated many times for each operation being tested, it is highly desirable that automatic tools be provided for use in assertion checking. Third, the error-prone process of driving a set of variables in a test point to have specific values calls for a means for producing test points interactively in a step-by-step fashion, where after every step the tester can see the current mathematical views of the variables involved.

An approach to accommodating these requirements has been presented. The essence of this approach is to have a "model operation" associated with each type provided by a module. The model operation for a type takes a variable of that type and produces a corresponding value which is the mathematical model of the given variable. (A model operation's being a part of a RESOLVE module under test may be thought of as analogous to a test driver routine's being a part of a piece of software under test in conventional program testing.) A testing environment is used to allow the tester to interactively create facilities from the available realization modules, create new variables of the types provided by the created facilities, and invoke operations on variables. The testing environment uses the model operations in the available modules to produce representations of the mathematical views of variables. These representations can then be used to provide visualization of the mathematical views of variables.

It has been shown that more expressive power was needed in RESOLVE to allow for writing model operations without significantly complicating the interfaces of RESOLVE modules or crippling the space efficiency of RESOLVE programs. The **math** and **model** constructs, along with allowing the declaration of theory facilities in realization modules, provide the needed expressive power. These extensions to the language fit nicely into the original framework of RESOLVE, and do not bring in new programming concepts which are significantly different from those in the original language.

A testing environment encompassing our approach to accommodating the requirements of testing RESOLVE modules has been designed and partially implemented (the linker, run-time system and part of the interpreter have been implemented). The testing environment allows its user to interactively create facilities from realization modules, create new variables, and invoke operations provided by the created facilities. RESOLVE program variables are rendered graphically as icons, and their mathematical views are rendered as identifiably different kinds of icons (math icons). The user can "open up" a math icon to visualize the next level of detail of the mathematical value represented by the math icon, as a graphical structure of other math icons.

The testing environment is based on an interpreter which provides for, among other things, the interactive creation of facilities and the performance of operations in these facilities. Consistent with the notion of reusable software components as object code that is used without modification in different applications, the interpreter uses

RESOLVE modules in compiled form. Because the same compiled realization modules are used both by the linker (to produce memory images of the run-time structure of RESOLVE programs) and by the interpreter, we had to address the design of the run-time structure of RESOLVE programs, the information in the compiled realization modules and its form, and the construction of the run-time structure using this information. This has led to the design of a run-time structure for efficient execution of RESOLVE programs. A linker, which constructs the run-time structure of RESOLVE programs, has also been designed and implemented. A similar program incrementally constructs the run-time structure which is used by the interpreter, on which the testing environment is based.

VII.2. Contributions

The major contributions of the research presented in this dissertation are:

- Identifying the problems inherent in testing software modules which provide ADTs that are specified using the model-based specification approach. These problems arise for RESOLVE, but are not unique to it. Other languages such as Ada and C++, if they admitted formal specifications, would also face these problems. Even without formal specifications, modules must be tested and these problems must be addressed.

- Presenting an approach to overcoming these problems. Again, while RESOLVE has been used as example here, similar solutions would apply to similar languages.

- Identifying the language support needed for this approach, and introducing language constructs which provide the needed support. These extensions could also be added to similar languages.

- Design and partial implementation of a testing environment which demonstrates a "proof of principle" for our approach for testing modules providing ADTs whose specifications are given using the model-based approach.

In conclusion, the research presented in this dissertation shows that testing a class of software components, whose characteristics are believed to advance reusability, places certain requirements on the implementation of the software components, the language in which they are written, and the practice of testing these components. These requirements can be met, but they demand careful attention to details beyond what has previously been available in programming languages and environments. Current languages such as Ada and C++ lack the features needed to support rigorous testing of modules.

VII.3. Future Research

The first part of this chapter has summarized the research presented in this dissertation. As is always the case in research, there is room for further development of the issues addressed, and many related issues remain to be considered.

We have focused on the requirements for testing modules providing ADTs, but we have not touched on the issue of selecting significant test cases for the operations provided by such modules. A complementary research effort would investigate criteria and techniques for the selection of such test cases. Seemingly, techniques for test case selection based on black-box testing can be easily adapted from their counterparts in conventional program testing (e.g., selecting test cases to exercise different cases in the specifications and to test boundary conditions).

On the other hand, selecting test cases based on white-box testing strategies raises a new problem that is worthy of study. In RESOLVE, a test case that exercises a given statement or group of statements depends on the statements leading to the *control* operation enclosing the statement to be exercised. For example, consider the following piece of code:

proc (x, a)

y := func (x)

if contr (y) **then** ... statement...

A test case to exercise "statement" consists of values for x and a which lead to contr(y) having the result *yes*. How can the tester determine initial values of x and a

which lead to the statement to be exercised? It should be noted that the analogous problem in conventional program testing has been explored, particularly for programs which involve mainly numerical values, and a technique known as *symbolic evaluation* has been proposed for dealing with special cases of the problem [King76, Clar76, Howd77]. However, the proposed symbolic evaluation technique does not offer much help in our case. A typical program in RESOLVE consists mainly of invocations of operations on ADTs, and only the specifications of these operations are known (due to information hiding). This results in two difficulties regarding symbolic evaluation. First, because the effects of an operation in RESOLVE are specified in terms of *relations* among the models of the old and new values of the parameters (and not necessarily with the new values as *functions* of the old values), operation invocations in RESOLVE are not amenable to the symbolic evaluation method described in [King76, Clar76, Howd77]. Second, even if a suitable symbolic evaluation method for RESLOVE programs is identified and used, the resulting *path conditions* will consist of compositions of predicates and functions from the mathematical theories used for specification in RESOLVE. Solving such arbitrary path conditions may be much more difficult than solving the path conditions which arise in conventional programs that involve only numerical values.

The testing environment whose design has been presented in this dissertation was meant as a "proof of principle" — to demonstrate the feasibility and potential utility of our approach to overcoming the problems involved in testing RESOLVE modules. There is room for more features and tools in such environment. These include the following:

- Allowing the tester to define assertions, and at any time bind mathematical variables to the free variables of a defined assertion and trigger assertion checking.

- Providing for the reproducibility of generated test points; e.g., allowing the tester to give names to test points (which are sets of variable values) and to be able, at later sessions, to recall these test points. This might require that the history of operations leading to values of variables be kept track of and stored.

- Making it possible to save the "state" of the environment so that the tester may continue later from the point where the previous session was terminated.

- Having the assertions in the **ensures** clauses of operations automatically checked when the operations are performed and reporting to the tester what might have caused an assertion not to hold. Checking an assertion which does not involve quantifiers can be automated by generating code corresponding to the assertion, where this code calls operations in the theory modules of the theories used in the assertion. Where quantifiers are involved, the problem is far more difficult, however.

Another issue of interest for further research is the visualization of values from mathematical domains of infinite size (e.g., infinite sets and functions). A promising idea in this regard is to give the tester the illusion that an infinite structure of values is being viewed through a small window, whereas only the part of the structure which can fit into the window actually exists in the representation system. Other parts of the

infinite structure may be generated on demand when the window is moved by the tester. A question that remains to be answered, however, is how to internally represent conceptually infinite structures.

In the course of discussing the problems involved in testing modules providing ADTs which are specified using the model-based specifications approach, a qualitative comparison between the use of algebraic specifications and model-based specifications in reusable software has been presented. However, a detailed quantitative comparison of the two specification approaches, with experiments involving subjects using the two approaches, is a problem worthy of future study.

There is also the evaluation of the *usability* of the testing environment, in whole or in part, which has yet to take place.

APPENDIX A
Conceptual Modules

```
conceptualization Integer_Template
   auxiliary
      theories
         theory Integers is Number_Theory
            renaming
               Integers.integer as integer
            end renaming
         end Integers
      end theories

      constants
         min_int : integer
         max_int : integer
      end constants

      constraints
         min_int ≤ 0 ≤ max_int
      end constraints
   end auxiliary

   interface
      type int is integer
```

```
   exemplar i
   initialization
      i = 0
   end  initialization

   constraints
      min_int ≤ i ≤ max_int
   end  constraints
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 = #i + 1
end increment

procedure add
   parameters
      alters i : int
      preserves j : int
   end  parameters
   requires min_int ≤ i + j ≤ max_int
   ensures i = #i + j
end add

procedure subtract
```

```
   parameters
      alters i : int
      preserves j : int
   end parameters
   requires min_int ≤ i - j ≤ max_int
   ensures i = #i - j
end subtract

procedure multiply
   parameters
      alters i : int
      preserves j : int
   end parameters
   requires min_int ≤ i·j ≤ max_int
   ensures i = #i·j
end multiply

procedure divide
   parameters
      alters i : int
      preserves j : int
   end parameters
   requires j ≠ 0 and
            ( ((i < 0) and (j < 0))
                     ⇒ (i > j·(max_int + 1)) ) and
            ( ((i > 0) and (j < 0))
                     ⇒ (i < j·(min_int - 1)) ) and
            ( ((i < 0) and (j > 0))
                     ⇒ (i > j·(min_int - 1)) )
   ensures ( ((#i ≥ 0) and (j > 0))
                  ⇒ (i·j ≤ #i < j·(i+1)) ) and
           ( ((#i ≤ 0) and (j < 0))
                  ⇒ (j·(i+1) < #i ≤ i·j) ) and
           ( ((#i ≥ 0) and (j < 0))
                  ⇒ (i·j ≤ #i < j·(i-1)) ) and
           ( ((#i ≤ 0) and (j > 0))
```

```
                    ⇒ (j·(i-1) < #i ≤ i·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
```

description

This template provides type int, whose mathematical model is just an integer. However, an int is a "computational integer" in the sense that it has a lower and upper bound given respectively by min_int and max_int, which are constants defined by the realization.

Initially, a variable of type int is zero.

The primary operations are provided here:

- "get_min_int ()" returns min_int.
- "get_max_int ()" returns max_int.
- "increment (i)" adds one to i.
- "add (i, j)" adds j to i.
- "subtract (i, j)" subtracts j from i.
- "multiply (i, j)" multiplies i by j.

- "divide (i, j)" divides i by j (i.e., returns in i the integer part of the result of dividing the original i by j).

- "less_than_or_equal (i, j)" returns yes iff i ≤ j.

end description

end Integer_Template

```
conceptualization Plastic_Array_Template
   parameters
      type T

      facility Integer_Facility is Integer_Template
            renaming
               Integer_Facility.int as int
            end  renaming
      end Integer_Facility
   end  parameters

   auxiliary
      theories
         theory Integers is Number_Theory
            renaming
               Integers.integer as integer
            end  renaming
         end Integers

         theory Functions is Function_Theory (integer, T)
            renaming
               Functions.function as integer_to_T
               Functions.apply as apply
            end  renaming
         end Functions

         Pairs is 2_Tuple_Theory (integer, integer_to_T)
            renaming
               Pairs.pair as array_model
               Pairs.projection_1 as size
               Pairs.projection_2 as map
            end  renaming
         end Pairs
      end  theories
   end  auxiliary
```

```
interface
   type array is array_model
      exemplar a
      initialization
         size(a) = 0 and
         ∀ i : integer, T.init (map(a)(i))
      end  initialization
      lemmas
         ∀ i : integer, (i < 0 or i ≥ size(a))
                        ⇒ T.init (map(a)(i))
      end  lemmas
   end array

   procedure set_size
      parameters
         alters a : array
         consumes n : int
      end  parameters
      requires n ≥ 0
      ensures size(a) = #n and
              ∀ i : integer, T.init (map(a)(i))
   end set_size

   function get_size returns s : int
      parameters
         preserves a : array
      end  parameters
      ensures s = size(a)
   end get_size

   procedure access
      parameters
         alters a : array
         preserves i : int
         alters x : T
      end  parameters
```

```
      requires 0 ≤ i < size(a)
      ensures size(a) = size(#a) and x = map(#a)(i) and
              Δ (map(a), {i}, map(a)(i) = #x)
   end access
end  interface
```

description

This template encapsulates a function from the first size non-negative integers to T, and effectively captures the concept of an array whose size can be set dynamically. The conceptual model is a pair: the current size, and a function from the integers to T. In the abstract, the function is total, but the only operation that can evaluate or change it requires that the index into the array (the point at which the function is to be evaluated and changed) must be non-negative but less than the array's current size. Therefore, the function value is meaningful only on this interval of its domain.

Initially, an array's size is zero and its function maps every integer to an initial value of type T.

- "set_size (a, n)" sets the size of array a to n. Notice that it also resets a's function so it maps every integer to an initial value of type T, and that it consumes n.

- "get_size (a)" returns the current size of the array a.

- "access (a, i, x)" swaps the previous value of a(i) with the previous value of x.

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

```
   end  description
end Plastic_Array_Template
```

```
conceptualization List_Template
   parameters
      type T
   end  parameters

   auxiliary
      theories
         theory Strings is String_Theory (T)
            renaming
               Strings.string as string
            end  renaming
         end Strings

         theory Pairs is 2_Tuple_Theory (string, string)
            renaming
               Pairs.pair as list_model
               Pairs.projection_1 as left
               Pairs.projection_2 as right
            end  renaming
         end Pairs
      end  theories
   end  auxiliary

   interface
      type list is list_model
         exemplar L
         initialization
            left(L) = Λ and right(L) = Λ
         end  initialization
      end list

      control atend
         parameters
            preserves l : list
         end  parameters
         ensures atend iff right(l) = Λ
```

```
end atend

procedure reset
   parameters
      alters l : list
   end  parameters
   ensures left(l) = Λ and
            right(l) = left(#l) o right(#l)
end reset

procedure advance
   parameters
      alters l : list
   end  parameters
   requires right(l) ≠ Λ
   ensures left(l) o right(l) = left(#l) o right(#l) and
            ∃ x : T, left(l) = left(#l) o (Λ•x)
end advanve

procedure add
   parameters
      alters l : list
      consumes x : T
   end  parameters
   ensures left(l) = left(#l) and
            right(l) = (Λ • #x) o right(#l)
end add

procedure remove
   parameters
      alters l : list
      produces x : T
   end  parameters
   requires right(l) ≠ Λ
   ensures left(l) = left(#l) and
            right(#l) = (Λ • x) o right(l)
```

```
   end remove

   procedure swapright
      parameters
         alters l1 : list
         alters l2 : list
      end  parameters
      ensures left(l1) = left(#l1) and right(l1) = right(#l2)
              left(l2) = left(#l2) and right(l2) = right(#l1)
   end swapright
end  interface
```

description

A "list" conceptually consists of an ordered pair of strings of items of type T. To visualize the effects of the list operations, think of these strings as being separated by a "fence" (which is not explicit in the formal specifications). The fence is able to "advance" (move forward, i.e., from left to right) one item at a time, and can be "reset" to the beginning (far left end) of the list. The first of the two string components of a list is called "left", as it represents the items that are to the left of the fence; the second string is called "right", and represents those items to the right of the fence. Notice that the fence is *between* two items in the list. It is not a cursor that is *on* some item. Initially, a list has an empty left string and an empty right string, so there is nothing on either side of the fence.

As an example, suppose we have a list "s" containing the items (10, 20, 30, 40, 50), in that order, and that the fence is after the 20. Then we might visualize the list as:

```
     10   20 ǂ 30   40   50
```

Here, "‡" represents the fence, the left string holds the items (10, 20), and the right string consists of the items (30, 40, 50). The list operations available, and their effects on this example list — if applied sequentially — are given below:

- "atend (s)" is true iff the fence is at the far right end of the list and therefore cannot move forward; i.e., iff the right string is empty. For the example list above, atend (s) is false.

- "reset (s)" moves the fence to the beginning of the list, but does not affect the items in the list nor their order. It does affect the left and right strings. In the example, "reset (s)" would result in s becoming:

  ‡ 10 20 30 40 50

- "advance (s)" moves the fence one position forward. Applying "advance" to the list just above leaves s as:

  10 ‡ 20 30 40 50

- "add (s, x)" puts x at the beginning of the right string of s, and x returns with an initial value for its type. The fence is not affected. In the example, with x = 80, "add (s, x)" leaves x = 0 and s as:

  10 ‡ 80 20 30 40 50

- "remove (s, x)" returns in x the value of the first item of the right string of s, and removes that item from s, but does not affect the fence. In the example, "remove (s, x)" will result in x = 80, and s looking just as it did prior to the add operation:

```
        10 ‡ 20   30   40   50
```

- "swapright (s1, s2)" exchanges the right components of s1 and s2. Let's suppose r is the following list:

```
        55   65   75 ‡ 85   95
```

  Then with s as above, "swapright (r, s)" leaves r looking like:

```
        55   65   75 ‡ 20   30   40   50
```

  and s as:

```
        10 ‡ 85   95
```

There is no operation here to examine an item in the list before removing it. This effect can be accomplished by removing an item, and then adding it back into the list if it should not have been removed. An enhanced version of this basic facility might have the convenient "access (s, x)" operation that swaps x with the first item in the right string of s, thus permitting a perhaps more natural "examine before removing" approach to list processing. Another useful additional operation is "prefix (s, x)", which puts x at the beginning of the left string of s (as opposed to "add", which inserts at the beginning of the right string).

```
   end  description
end List_Template
```

```
conceptualization Bounded_List_Template
   parameters
      type T

      facility Int_Fac is Integer_Template
         renaming
            Int_Fac.int as int
         end renaming
      end Int_Fac
   end parameters

   auxiliary
      theories
         theory Integers is Number_Theory
            renaming
               Integers.integer as integer
            end renaming
         end Integers

         theory Strings is String_Theory (T)
            renaming
               Strings.string as string
            end renaming
         end Strings

         theory Triples is 3_Tuple_Theory
                              (string, string, integer)
            renaming
               Triples.triple as list_model
               Triples.projection_1 as left
               Triples.projection_2 as right
               Triples.projection_3 as max_size
            end renaming
         end Triples
      end theories
   end auxiliary
```

```
interface
   type list is list_model
      exemplar L
      initialization
         left(L) = Λ and right(L) = Λ and max_size(L) = 0
      end initialization
      lemmas
         |left(L)| + |right(L)| ≤ max_size(L)
      end lemmas
   end list

   procedure set_max_size
      parameters
         alters l : list
         consumes max : int
      end parameters
      requires max > 0
      ensures left(l) = Λ and right(l) = Λ and
              max_size(l) = max
   end set_max_size

   function get_max_size returns max : int
      parameters
         preserves l : list
      end parameters
      ensures max = max_size(l)
   end get_max_size

   function get_size returns s : int
      parameters
         preserves l : list
      end parameters
      ensures s = |left(l)| + |right(l)|
   end get_size
```

```
control atend
   parameters
      preserves l : list
   end  parameters
   ensures atend iff right(l) = Λ
end atend

procedure reset
   parameters
      alters l : list
   end  parameters
   ensures left(l) = Λ and
           right(l) = left(#l) o right(#l)
end reset

procedure advance
   parameters
      alters l : list
   end  parameters
   requires right(l) ≠ Λ
   ensures left(l) o right(l) = left(#l) o right(#l) and
           ∃ x : T, left(l) = left(#l) o (Λ•x)
end advanve

procedure add
   parameters
      alters l : list
      consumes x : T
   end  parameters
   requires |left(l)| + |right(l)| < max_size(l)
   ensures left(l) = left(#l) and
           right(l) = (Λ • #x) o right(#l)
end add

procedure remove
```

```
      parameters
         alters l : list
         produces x : T
      end parameters
      requires right(l) ≠ Λ
      ensures left(l) = left(#l) and
              right(#l) = (Λ • x) o right(l)
   end remove

   procedure swapright
      parameters
         alters l1 : list
         alters l2 : list
      end parameters
      requires |left(l1)| + |right(l2)| ≤ max_size(l1) and
              |left(l2)| + |right(l1)| ≤ max_size(l2)
      ensures left(l1) = left(#l1) and right(l1) = right(#l2)
              left(l2) = left(#l2) and right(l2) = right(#l1)
   end swapright
end interface
```

description

A "bounded list" conceptually consists of an ordered triple whose first and second components are strings of items of type T, and whose third component is an integer representing the maximum allowable size for the bounded list. To visualize the effects of the list operations, think of the two strings in the list as being separated by a "fence" (which is not explicit in the formal specifications). The fence is able to "advance" (move forward, i.e., from left to right) one item at a time, and can be "reset" to the beginning (far left end) of the list. The first of the two string components of a list is called "left", as it represents the items that are to the left of the fence; the second string is called "right", and represents those items to the right of the fence. Notice that the fence is *between* two items in the list. It is not a

cursor that is *on* some item. Initially, a list has maximum length 0 and has an empty left string and an empty right string, so there is nothing on either side of the fence

As an example, suppose we have a list "s" containing the items (10, 20, 30, 40, 50), in that order, and that the fence is after the 20. Then we might visualize the list as:

```
10   20 ‡ 30   40   50
```

Here, "‡" represents the fence, the left string holds the items (10, 20), and the right string consists of the items (30, 40, 50). The list operations available, and their effects on this example list – if applied sequentially – are given below:

- "set_max_size (l, max_size)" sets the maximum size of l to max_size and makes both the left and right parts of l empty.

- "get_max_size (l)" returns the maximum size of l.

- "get_size (l)" returns the current size of l (i.e., the sum of the lengths of the left and right parts of l).

- "atend (s)" is true iff the fence is at the far right end of the list and therefore cannot move forward; i.e., iff the right string is empty. For the example list above, atend (s) is false.

- "reset (s)" moves the fence to the beginning of the list, but does not affect the items in the list nor their order. It does affect the left and right strings. In the example, "reset (s)" would result in s becoming:

```
‡ 10   20   30   40   50
```

- "advance (s)" moves the fence one position forward. Applying "advance" to the list just above leaves s as:

  10 ǂ 20 30 40 50

- "add (s, x)" puts x at the beginning of the right string of s, and x returns with an initial value for its type. The fence is not affected. In the example, with x = 80, "add (s, x)" leaves x = 0 and s as:

  10 ǂ 80 20 30 40 50

- "remove (s, x)" returns in x the value of the first item of the right string of s, and removes that item from s, but does not affect the fence. In the example, "remove (s, x)" will result in x = 80, and s looking just as it did prior to the add operation:

  10 ǂ 20 30 40 50

- "swapright (s1, s2)" exchanges the right components of s1 and s2. Let's suppose r is the following list:

  55 65 75 ǂ 85 95

  Then with s as above, "swapright (r, s)" leaves r looking like:

  55 65 75 ǂ 20 30 40 50

  and s as:

  10 ǂ 85 95

```
        There is no operation here to examine an item in the list
        before removing it. This effect can be accomplished by
        removing an item, and then adding it back into the list if it
        should not have been removed. An enhanced version of this
        basic facility might have the convenient "access (s, x)"
        operation that swaps x with the first item in the right
        string of s, thus permitting a perhaps more natural "examine
        before removing" approach to list processing. Another useful
        additional operation is "prefix (s, x)", which puts x at the
        beginning of the left string of s (as opposed to "add", which
        inserts at the beginning of the right string).
    end  description
end Bounded_List_Template
```

```
conceptualization Nilpotent_Function_Template
   parameters
      type T
   end  parameters

   auxiliary
      theories
         theory Integers is Number_Theory
            renaming
               Integers.integer as integer
            end  renaming
         end Integers

         theory Functions1 is Function_Theory (integer, integer)
            renaming
               Functions1.function as integer_to_integer
            end  renaming
         end Functions1

         theory Functions2 is Function_Theory (integer, T)
            renaming
               Functions2.function as integer_to_T
            end  renaming
         end Functions2
      end  theories

      variables
         next : integer_to_integer
         count : integer_to_integer
         data : integer_to_T
      end  variables

      initialization
         ∀ p : integer, next(p) = 0
         ∀ p : integer, (p ≠ 0 ⇒ count(p) = 0) and
                  count(0) = 2
```

```
      ∀ p : integer, T.init(data(p))
   end  initialization

   lemmas
      next(0) = 0
      (∀ p : integer, count(p) ≥ 0) and count(0) = 2
      ∀ p : integer, ∃ k : integer, (k ≥ 0 and next^k(p) = 0)
   end  lemmas
end  auxiliary

interface
   type token is integer
      exemplar p
      initialization
         p = 0 and
         next = #next and count = #count and data = #data
      end  initialization

      finalization
         next = #next and
         Δ (count, {p}, p ≠ 0 ⇒ count(p) = #count(p) - 1) and
         data = #data
      end  finalization
   end token

   procedure newposition
      parameters
         alters p : token
      end  parameters
      ensures #count(p) = 0 and
         Δ (next, {p}, next(p) = default) and
         Δ (count, {#p, #next(p), p},
               (#p ≠ 0 ⇒ count(#p) = #count(#p) - 1) and
               (#next(p) ≠ 0 ⇒ count(#next(p)) =
                                 #count(#next(p)) - 1) and
               (count(p) = 1)
```

```
        ) and
      Δ (data, {p}, T.init(data(p)))
end newposition

procedure changenext
   parameters
      preserves p1 : token
      preserves p2 : token
   end parameters
   requires not ∃ k : integer such that
      (k ≥ 0 and next^k(p2) = p1)
   ensures Δ (next, {p1}, next(p1) = p2) and
      Δ (count, {#next(p1), p2},
            ((p2 ≠ #next(p1) and #next(p1) ≠ 0) ⇒
               count(#next(p1)) = #count(#next(p1)) - 1) and
            ((p2 ≠ #next(p1) and p2 ≠ 0) ⇒
               count(p2) = #count(p2) + 1) and
            (p2 = #next(p1) ⇒ count(p2) = #count(p2))
         ) and
      data = #data
end changenext

procedure move
   parameters
      alters p1 : token
      preserves p2 : token
   end parameters
   ensures p1 = p2 and
      next = #next and
      Δ (count, {p1, #p1},
            ((p1 ≠ #p1 and p1 = 0) ⇒
               count(p1) = #count(p1) + 1) and
            ((p1 ≠ #p1 and #p1 ≠ 0) ⇒
               count(#p1) = #count(#p1) - 1) and
            (p1 = #p1 ⇒ count(p1) = #count(p1))
         ) and
```

```
      data = #data
end move

procedure follownext
   parameters
      alters p : token
   end  parameters
   ensures p = next(#p) and
      next = #next and
      Δ (count, {#p, p},
            (#p ≠ 0 ⇒ count(#p) = #count(#p) - 1) and
            (p ≠ 0 ⇒ count(p) = #count(p) + 1)
         ) and
      data = #data
end follownext

control equal
   parameters
      preserves p1 : token
      preserves p2 : token
   end  parameters
   ensures equal iff p1 = p2
end equal

control isdefault
   parameters
      preserves p : token
   end  parameters
   ensures isdefault iff p = 0
end isdefault

control isunique
   parameters
      preserves p : token
   end  parameters
   ensures isunique iff count(p) = 1
```

```
   end isunique

   procedure access
      parameters
         preserves p : token
         alters x : T
      end parameters
      ensures next = #next and
         count = #count and
         Δ (data, {p}, data(p) = #x and x = #data(p))
   end access
end interface
```

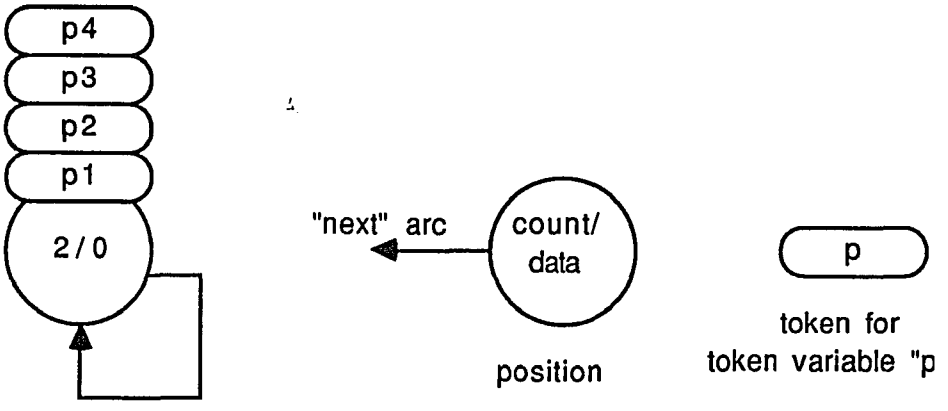
description

A "nilpotent function" is a function with the property that if it is applied repeatedly to any argument in its domain, the value eventually is zero. More formally, f is a nilpotent function iff there is a zero value (designated "0") in its domain (which is also the range) such that $f(0) = 0$, and for every x in the domain, there is a $k \geq 0$ such that $f^k(x) = 0$.

The importance of the concept in this module is that the "next" function (whose domain, integer, is the mathematical model of type "token") is nilpotent; the zero value is called "default". Schematically, one may depict the possible token *values* as the vertices of a directed graph, which we will henceforth call "positions". The "next" function is represented schematically as this graph's directed edges (arcs), where there is an arc from position x to position y iff next(x) = y. Since "next" is nilpotent, this graph is acyclic (except for the edge from the default position to itself). Also, since f is a total function, each position has exactly one arc leaving it. An integer-valued "count" function is defined for each token value. Finally, each token value has a data value of type T associated with it (the "data" function).

There are infinitely many positions (one corresponding to each possible token value – i.e., to each integer), and each initially has an arc to the default. But without any token variables, no position is "accessible" to a program. Suppose we declare a token variable. Schematically, this is depicted by placing a "token" representing that variable (hence, the type name) on the position whose token value it has. Initially, the token is placed on the default position, but operations are available to move it around (i.e., change the variable's value). The "count" function for every position (except default) is equal to the number of tokens on it, plus the number of arcs leading into that position. Only for the default position is this interpretation incorrect – there are infinitely many arcs into it conceptually, but "count (default)" is always 2. The "count" value is shown before the "/" in each position of the diagrams in this module. The "data" value of type T (here assumed to be integer for purposes of illustration) associated with a position is shown after the "/".

Here is a picture of the situation following declaration and initialization of token variables p1, p2, p3, and p4, showing only positions accessible to the program (i.e., only the default at this stage):



The operations are now explained in terms of token placement and movement in this graph. Each operation that moves a token or changes the "next" function also appropriately updates the "count" function so that its interpretation remains "the number of tokens on a position plus the number of arcs into that position". By explicit exception, "count (default)" is always 2.

- "newposition (p)" moves p's token to a position whose previous "count" was 0, makes that position's "next" arc go to the default position, and makes sure the "data" function value for the position has an initial value for type T.

- "changenext (p1, p2)" changes the arc leading from the position of p1's token so that it now goes to the position occupied by p2's token. The requires clause says that by following "next" arcs, one cannot get from p2 to p1; if this were possible, invoking "changenext (p1, p2)" would introduce a circularity in the "next" function, and it would no longer be nilpotent.

- "move (p1, p2)" moves p1's token to the position occupied by p2's token.

- "follownext (p)" moves p's token to the position reached by following the (unique) arc leaving the position where it resided before the operation.

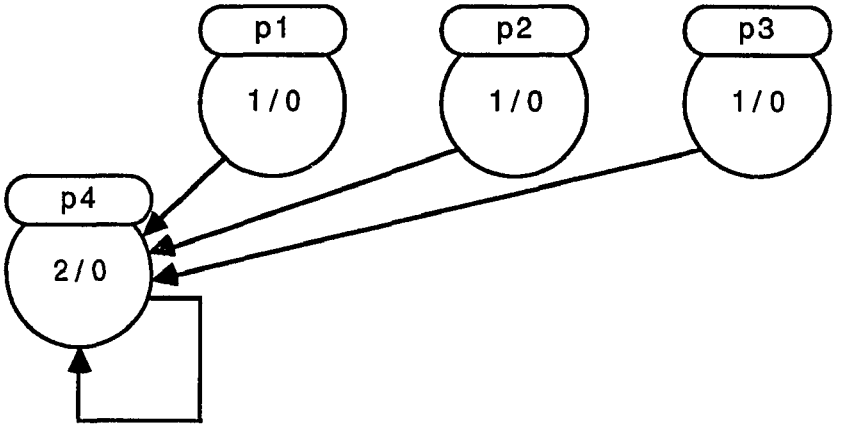
- "equal (p1, p2)" returns true iff p1's token and p2's token are on the same position. (Of course, this implies that the "data" values associated with them are also equal, but the converse is not true. To check equality of "data" of two different token variables, access each of them and use the equality test for type T, if any.)

- "isdefault (p)" returns true iff p's token is on the default position.

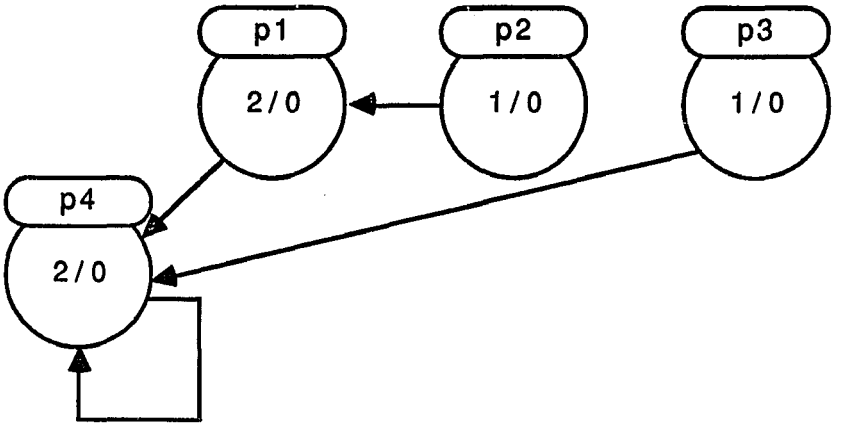
- "isunique (p)" returns true iff p's token is on a position for which the "count" function equals 1. This can only happen if there are no other tokens on that position and no arcs into it. Note that if p is on the default, "isunique (p)" is false, since count (p) = 2 in that case.

- "access (p, x)" exchanges x with the "data" value associated with the position occupied by p's token.

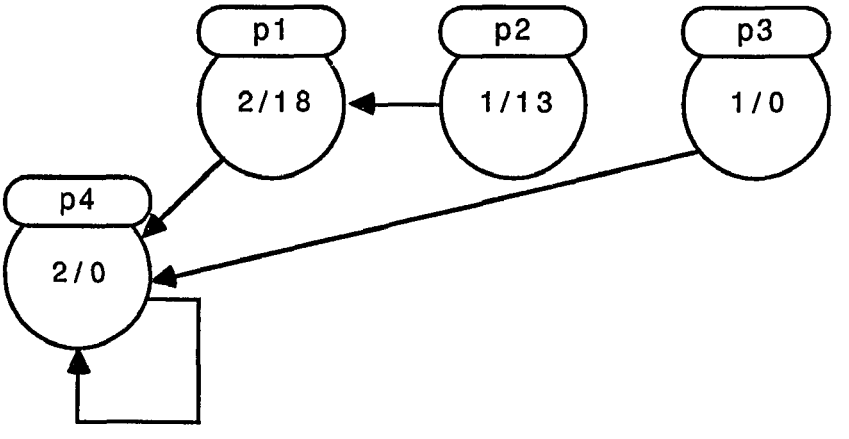
In the diagram below, we've applied "newposition" to each of p1, p2, and p3, following initialization depicted in the previous schematic:



Now, if we "changenext (p2, p1)", we end up in the situation below. Note how the "next" and "count" functions have been affected. Also, "equal (p1, p2)", for instance, is not true since p1 and p2 are not equal as tokens (they are not on the same position) even though their "data" function values are equal.

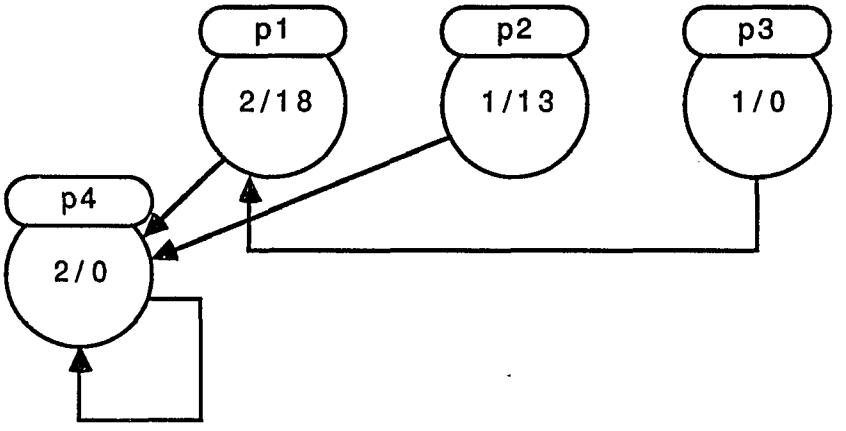


We can get and set "data" function values using "access". Suppose x and y are integers, with x = 18 and y = 13. Following "access (p1, x)" and "access (p2, y)", we have x = 0 and y = 0 and the situation below:

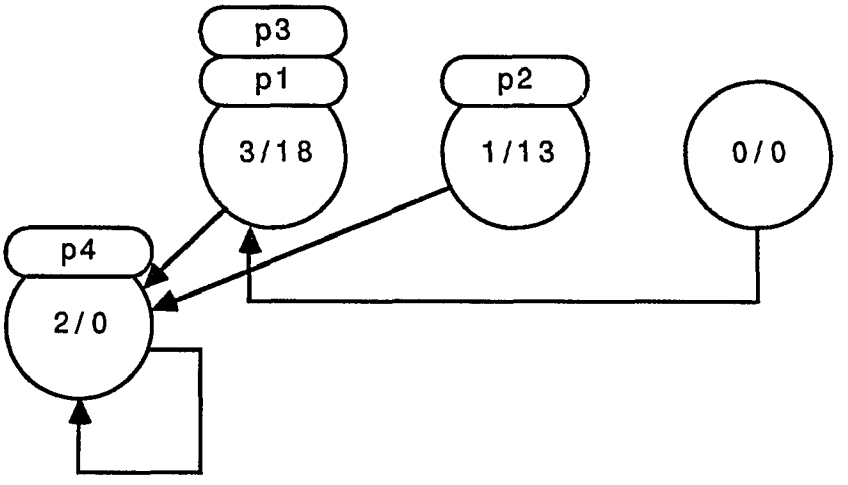


At this stage, "isunique (p2)" is true, but "isunique (p1)" is false. Of course, "isdefault (p4)" is true, and "isdefault (p3)" is false. These are very straightforward.

Suppose we now "changenext (p3, p1)" and "changenext (p2, p4)", resulting in the following:



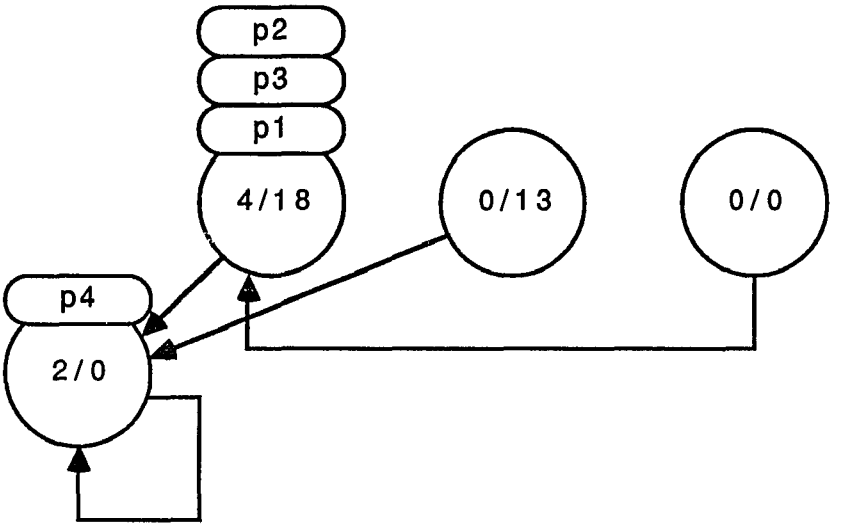
After "follownext (p3)" we are left with:



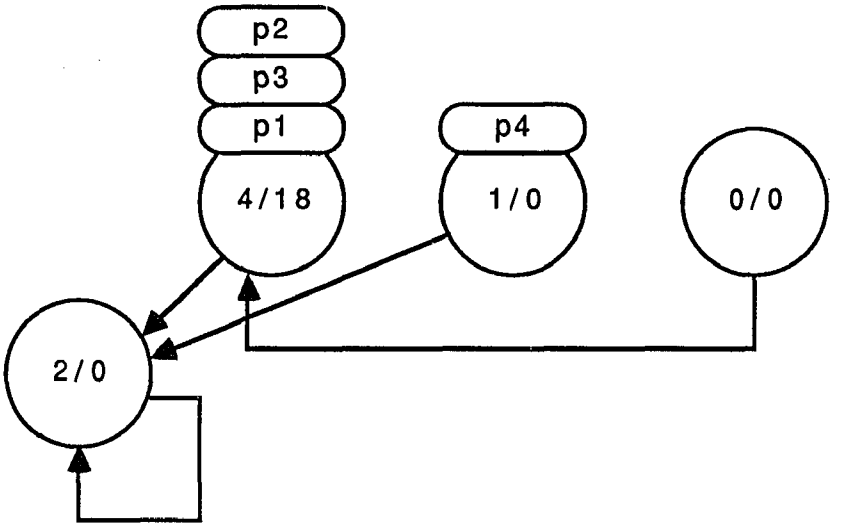
Note that while the far right position is now inaccessible, the value of its "next" function still affects the position occupied by p1 and p3, since the latter's "count" function includes that incoming arc.

Suppose we now "move (p2, p1)". This leaves the configuration below. Note again that a position has become inaccessible. Its "data" function value remains equal to 13, but this is totally irrelevant since the program can never again access

that value. Positions can become inaccessible as a result of the token-moving operations "newposition", "followne xt", "changenext", and "move", but not as a result of "swap" (not illustrated in the diagrams).



Finally, imagine that we execute "newposition (p4)". There are many possible outcomes in terms of effects on the graph, but all are (almost; see below) indistinguishable from the standpoint of the program. Token p4 can be placed on any position with "count" = 0. This can be one of the inaccessible positions above, or one of the infinitely many virgin positions not shown in the diagrams. Suppose the middle position above is selected. Then we end up with the following situation:



Notice that "newposition" results in the "data" function value of the selected position being an initial value for type T, and the "next" function value being the default. (Here, the latter did not cause a change in the "next" function, but in general it might do so.)

It is important to note that the only way the program can distinguish the above configuration from the selection of the rightmost position is by the "count" function of the position occupied by p1, p2, and p3. For suppose we now finalize p1 and p2 (thereby resulting in the removal of those tokens from the graph), so that p3 is the only token on that position. Invoking "isunique (p3)" after the outcome depicted in the diagram, we get the answer "false". If the rightmost position had been selected for the "newposition" operation, "isunique (p3)" would be true. In either case, though, token variable p3 would be the only way for the program to access the position in question. In this sense, "isunique (p3)" gives a conservative answer to the question, "Is p3 on a position that is uniquely accessible via p3?" If "isunique" is true, you may be sure that there is no other way to access that position. But if "isunique" is false, you may *not* be sure

```
      that there is another way to access the position. In short,
      "isunique" implies unique accessibility, but not vice versa.

      The nilpotent function facility is an important basic
      facility, as it can serve as the basis for a variety of
      "singly linked data structures". One may observe the
      correspondence between positions and "nodes" in classical
      linked data structures. The "next" arcs, of course, take the
      place of "next" pointers, and token variables replace pointer
      variables. The major difference here is that the effects of
      operations on the "next" arcs and on token variables are
      quite restricted, and these restrictions prevent various
      dangerous effects obtained with undisciplined use of
      pointers.

      It should be noted that it is possible to design a similar
      facility in which there are multiple "next" functions per
      position, and where all taken together must be nilpotent.
      This is valuable in implementing more complex data
      structures, such as trees.
   end description
end Nilpotent_Function_Template
```

APPENDIX B
String Theory

String over alphabet Σ

Signature: $\langle Str(\Sigma), \Lambda, Ext \rangle$

Λ: $Str(\Sigma)$

Ext: $Str(\Sigma) \times \Sigma \rightarrow Str(\Sigma)$

Axioms:

I. $\forall\, \alpha$: $Str(\Sigma)$, $\forall$ z: Σ, $\quad Ext(\alpha, z) \neq \Lambda$

II. $\forall\, \alpha, \beta$: $Str(\Sigma)$, $\forall$ y, z: Σ, $\quad (Ext(\alpha, y) = Ext(\beta, z)) \Rightarrow (\alpha = \beta \wedge y = z)$

III. $\forall\, S \subseteq Str(\Sigma)$,

$$((\Lambda \in S) \wedge (\forall\, \alpha: Str(\Sigma), (\alpha \in S \Rightarrow \forall\, z: \Sigma, Ext(\alpha, z) \in S))) \Rightarrow S = Str(\Sigma)$$

Definition: $\alpha \circ \beta$

$\circ$: $Str(\Sigma) \times Str(\Sigma) \rightarrow Str(\Sigma)$

(i) $\alpha \circ \Lambda = \alpha$

(ii) $\alpha \circ Ext(\beta, z) = Ext(\alpha \circ \beta, z)$

Definition: $|\alpha|$

$|\bullet|: Str(\Sigma) \rightarrow N$

(i) $|\Lambda| = 0$

(ii) $|Ext(\alpha, x)| = |\alpha| + 1$

Definition: α^R

$\bullet^R: Str(\Sigma) \rightarrow Str(\Sigma)$

(i) $\Lambda^R = \Lambda$

(ii) $(Ext(\alpha, x))^R = Ext(\Lambda, x) \circ \alpha^R$

APPENDIX C
String Theory Template

```
conceptualization String_Theory_Template
   parameters
      type T
   end  parameters

   auxiliary
      theories
         theory Integers is Number_Theory
            renaming
               Integers.integer as integer
            end  renaming
         end Integers

         theory Strings is String_Theory(T)
            renaming
               Strings.string as str
            end  renaming
         end Strings
      end  theories
   end  auxiliary

   interface
```

```
type string is str
   exemplar s
   initialization
      s = Λ
   end  initialization
end string

function Extend returns s2 : string
   parameters
      preserves s1 : string
      preserves e : T
   end  parameters
   ensures s2 = s1•e
end Extend

function Length returns n : integer
   parameters
      preserves s : string
   end  parameters
   ensures n = |s|
end Length

function Concatenate returns s : string
   parameters
      preserves u : string
      preserves v : string
   end  parameters
   ensures s = u ○ v
end Concatenate

function Reverse returns r : string
   parameters
      preserves s : string
   end  parameters
   ensures r = s^R
end Reverse
```

```
      function Get returns e : T
         parameters
            preserves s : string
            preserves i : integer
         end  parameters
         requires i > 0 and i ≤ |s|
         ensures ∃ u, v: string,
                        ( ((u•e) o v) = s and |u| = i-1 )
      end Get

      function Copy returns s2 : string
         parameters
            preserves s1 : string
         end  parameters
         ensures s2 = s1
      end Copy

      control Equal
         parameters
            preserves s1 : string
            preserves s2 : string
         end  parameters
         ensures Equal iff s1 = s2
      end Equal
   end  interface
end String_Theory_Template
```

APPENDIX D

An Example of the Insufficiency of Testing the Implementation of an ADT Against the Individual Axioms in Its Algebraic Specifications

In this appendix we present an example in which an incorrect implementation of an algebraically specified ADT satisfies the individual axioms in the ADT specifications for every possible test point. The ADT used in this example captures the notion of "complex integers," and its specifications are taken from [Gogu78]. The implementation language is the one used in [Gann81].

Specifications of the Complex Integer Data Type

```
type complex_int

syntax
    Make(integer, integer) → complex_int
    Re(complex_int)   →  integer
    Im(complex_int)   →  integer
    Add(complex_int, complex_int)   →   complex_int
```

```
    Mul(complex_int, complex_int)  →  complex_int
    Equal(complex_int, complex_int)  →  boolean

semantics
    Make(Re(C), Im(C))  = C
    Re(Make(N, N'))  =  N
    Im(Make(N, N'))  =  N'
    Add(Make(N, N'), Make(M, M'))  =  Make(N+M, N'+M')
    Mul(Make(N, N'), Make(M, M'))
                     =  Make((N*M)-(N'*M'), (N*M')+(N'*M))
    Equal(C, D)  =  if  Re(C) = Re(D)
                        then if  Im(C) = Im(D)
                                 then TRUE
                                 else FALSE
                        else FALSE
```

Implementation of the Complex Integer Data Type

```
class complex_int = Make, Re, Im, Add, Mul, Equal

/* representation */
unique int real
unique int imaginary
unique bool just_made

/* operations */
complex_int func Make(int r, int i)
   complex_int result
   result.real := r
   result.imaginary := i
   result.just_made := True
   return(result)

int func Re(complex_int c)
   return(c.real)

int func Im(complex_int c)
   return(c.imaginary)
```

```
complex_int Add(complex_int a, complex_int b)
   complex_int result
   result.real := a.real + b.real
   if  a.just_made and b.just_made
      then
         result.imaginary := a.imaginary + b.imaginary
      else
         result.imaginary := 0
      end
   result.just_made := False
   return(result)

complex_int Mul(complex_int a, complex_int b)
   complex_int result
   result.real := (a.real * b.real) - (a.imaginary * b.imaginary)
   if  a.just_made and b.just_made
      then
         result.imaginary :=
                  (a.real * b.imaginary) + (a.imaginary * b.real)
      else
         result.imaginary := 0
      end
   result.just_made := False
   return(result)

bool func Equal(complex_int a, complex_int b)
   return((Re(a) = Re(b)) and (Im(a) = Im(b)))
```

Discussion

What is incorrect in the above implementation of the complex integer data type is that the imaginary part of the result of the Add or the Mul operation is always zero whenever either of the operands is not a direct result of the Make operation. This

implementation satisfies all the axioms in the specifications of the complex integer data type because none of the axioms involves the Add operation (or the Mul operation) with operands which are not direct results of the Make operation. Although this implementation satisfies all the individual axioms in the specifications, there are theorems derivable from these axioms which are not satisfied by the implementation. For example, the theorem

```
Add( Add(Make(A, A'), Make(B, B')),
     Add(Make(C, C'), Make(D, D')) )
         = Make( (A+B)+(C+D), (A'+B')+(C'+D') )
```

is not satisfied by the given implementation.

It should be noted that deriving theorems from the axioms of the algebraic specifications depends on the properties of *equality* (reflexivity, symmetry, transitivity, and substitutivity). For example, the theorem given above could not be derived without using the transitivity and substitutivity of equality on complex integers. Therefore, an implementation that does not satisfy the properties of equality can satisfy the axioms of the algebraic specifications of an ADT, but fails to satisfy theorems derivable from these axioms. To remedy this, the properties of equality must be expressed as additional axioms in the specifications of an ADT.

In ANNA [Luck87] (ANNotated Ada), the axioms of the algebraic specifications of an ADT are automatically augmented with axioms for the properties of equality. These axioms are then used in checking the consistency of an implementation of an ADT with its specifications at run time. However, from the testing point of view, if

the implementation language allows the "abstract" values of the parameters to a function to be changed by the function's code, it is possible to have an implementation of an ADT that satisfies all the axioms in the ADT's algebraic specifications (including the axioms of the equality properties) but does not satisfy theorems derivable from these axioms.

APPENDIX E

Terminology of Model-based Specifications

In model-based specifications, a *program type* is modeled by a *mathematical domain*. For example, the program type "stack" is modeled by the mathematical domain "string." A program type is known in a program; program variables are declared to be of such a type. A mathematical domain is known in a specification; mathematical variables which *model* program variables in assertions are from mathematical domains.

A *correspondence* assertion relates the mathematical model of a program type to its concrete representation. This "abstraction mapping," from concrete representations to mathematical models, is generally many-to-one since the same abstract mathematical value my have several different representations. The correspondence is needed for formal verification of programs (or modules) which are specified using the model-based specification method (to map the effects of the operations used in programs from the program domain to the mathematical domain).

For testing and visualization purposes, canonical program representations of the mathematical domains which model program types are used (e.g., a canonical program representation of mathematical strings). The types represented by such canonical representations are called *mathematical types*. It should be noted that mathematical types are program types, but they happen to be canonical representations of mathematical domains. Whenever, in a given context, it is necessary to distinguish between mathematical types and program types which are not canonical representations of mathematical domains, the latter types are referred to as *regular types*.

Since a mathematical type is itself a program type, it is modeled by a mathematical domain (which happens to be the mathematical domain for which the mathematical type is a canonical representation). The mathematical type and its mathematical domain are therefore also related by a correspondence assertion. Figure 21 depicts the relations between the mathematical model and the concrete representations of program entities, and Figure 22 shows an example of these relations. Note that a *model operation* for a given program type is used to map values of this program type into corresponding values of the corresponding mathematical type directly, without first mapping the values of the program type into values in the corresponding mathematical domain.

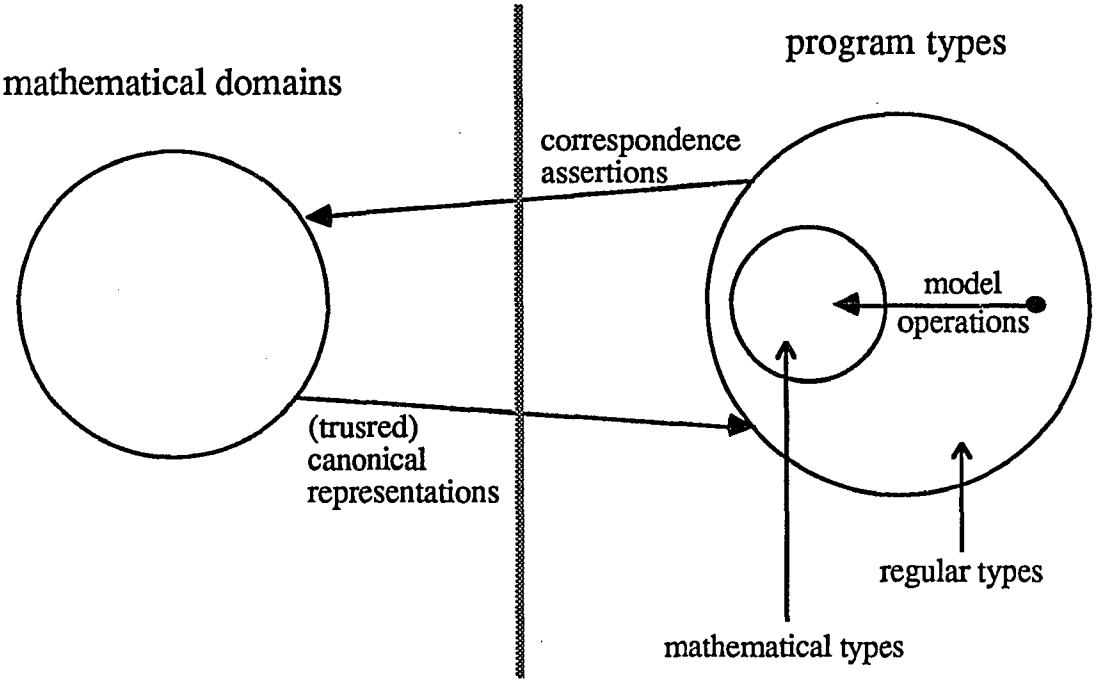


Figure 21. Relations between the mathematical model and the concrete representations of program entities

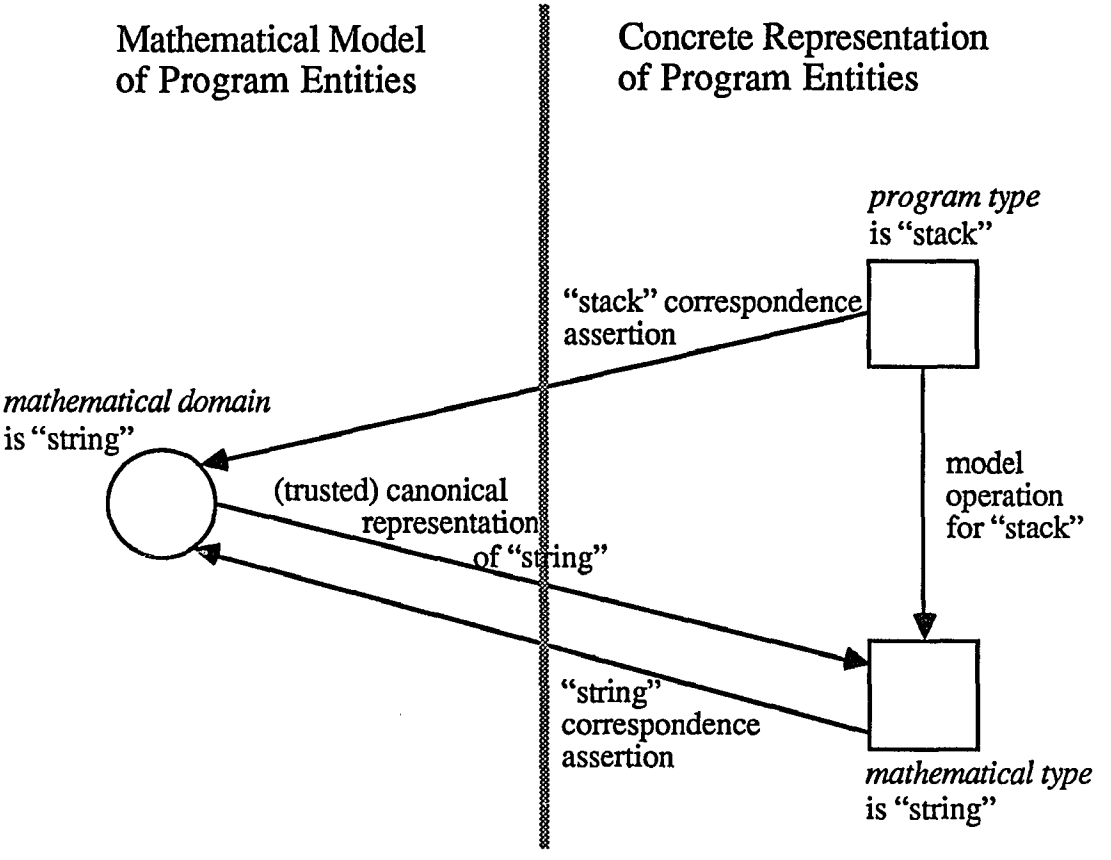


Figure 22. Example of the relations between the mathematical model and the concrete representations of program entities

APPENDIX F
CRM Grammar

Notation:

We use an extended BNF notation to define the structure of information in compiled realization modules. The general form of a production is:

<non-terminal> —> alternative
| alternative
| ...
| alternative

The following extensions are used:

<a>* a sequence of zero or more *a*'s

<a>+ a sequence of one or more *a*'s

The only terminal symbols are INTEGER and STRING. Keywords, which are in bold face, bear no significance on the structure of information described by the grammar; they are used just to provide more human comprehensibility of what each alternative in a production stands for. (In fact, each alternative is "tagged" with an integer in implementation, and this keyword just explains the intuitive meaning of the tag.) Comments are enclosed between braces.

Definitions:

Following are definitions of terms used in describing the information in a CRM:

- *P-number*: Every module parameter (conceptual or realization) has a *p-number*. The p-number of the i^{th} conceptual parameter is i. The p-number of the j^{th} realization parameter is (c+j), where c is the number of the module's conceptual parameters.

- *V-number*: Every provided type has a *v-number*, which tells the order of the provided type among all the types provided by a module (starting with 1), and each provided operation has a v-number which tells the order of the provided operation among all the operations provided by a module (starting with 1).

- *D-number*: Every declared regular facility has a *d-number*, which tells the order of the declared facility among all the regular facilities declared in a module (starting with 1). Similarily, every declared theory facility has a d-number, which tells the order of the declared facility among all the theory facilities declared in a module (starting with 1).

- *Facility chain*: A facility chain is a sequence (possibly empty) of p-numbers. In the context of a given facility F, a facility chain specifies a facility as follows: If the chain is empty, it specifies the facility F itself. If the chain is not empty, say of the form $p_1 p_2 \ldots p_{(e-1)} p_e$, it specifies the facility which is a facility parameter with p_number p_e to the facility which is a facility parameter with p_number $p_{(e-1)}$ to the facility which is a facility parameter with p_number ... to the facility which is a facility parameter with p_number p_1 to F.

- *"Self" facility*: Among the facilities that can be referred to in a realization module is the facility created from the realization module itself. In the context of explaining the meanings of the pieces of information in a CRM, such facility will be referred to as "self."

The Grammar:

<CRM> —> <template name>
{The name of the corresponding conceptual module}
<RM name>
{The name of the compiled realization module}
<sizes>
<formal C par>*
{The formal conceptual parameters}
<formal R par>*
{The formal realization parameters}
<restriction>*
{All the restrictions are given here, including those restrictions which relate types from the conceptual parameters only}
<dclrd fac>*
{The regular facilities declared in the realization module}
<other used fac>*
{These are the regular facilities to which belong operations that are called by the code of this realization module, other than the following facilities:
- "self,"
- the facilities of the module parameters, and
- the declared facilities}
<local code>

{The code of the various operations in the realization module (other than the model operations)}

<xtrnl op>*

{These are operations in other realization modules, which are called by the code of this realization module}

<provided type>*

<provided op>*

<dclrd theory fac>*

{The theory facilities declared in this realization module. This information is needed only in constructing the RTS for testing}

<math of prvdd type>*

{These are the mathematical types corresponding to the types provided by this realization module}

<other used theory fac>*

{These are the theory facilities used in the realization module, other than the declared ones; specifically, they are the theory facilities which provide the mathematical types which are referred to using the "math" construct}

<special xtrnl op>*

{These are operations in other realization modules which are called by the code of the model operations in this realization module, but are not called by other operations in this realization module; this includes the model operations in other modules}

<code of conc ops>

{The code of the model operations in this realization module}

<aux in>

{This tells whether or not the
information needed to construct the
RTS for testing is included in the
CRM}

<template name> —> STRING

<RM name> —> STRING

<sizes> —> <no of c_pars>
<no of r_pars>
<no of dclrd facs>
<no of prvdd types>
<no of conc mod vars>
<no of OUFs>
{The number of "other used facilities"
described above}
<no of prvdd ops>
<no of private ops>
<no of xtrnl ops>
<no of module vars>
<no of dclrd th facs>
{The number of theory facilities
declared in this realization module}
<no of OUTFs>
{The number of "other used theory
facilities" described above}
<no of special xtrnl ops>

<no of c_pars> —> INTEGER

<no of r_pars> —> INTEGER

<no of dclrd facs> —> INTEGER

<no of prvdd types>
—> INTEGER
<no of conc mod vars>
—> INTEGER
<no of OUFs> —> INTEGER
<no of prvdd ops>
—> INTEGER
<no of private ops>
—> INTEGER
<no of xtrnl ops> —> INTEGER
<no of mod vars> —> INTEGER
<no of dclrd th facs>
—> INTEGER
<no of OUTFs>
—> INTEGER
<no of special xtrnl ops>
—> INTEGER

<formal C par> —> **type**
<formal par name>
| **fac**
<template name>
<formal par name>
<formal R par> —> **type**
<formal par name>
| **fac**
<template name>

<formal par name>

| **op**

<op par type>*

{The types of the parameters to an operation parameter to the realization module}

<optional result>

{The type of the result of an operation parameter to the realization module}

<formal par name>

<formal par name>

—> STRING

<op par type> —> **dummy**

{The type of the parameter to the operation parameter will be equated to another type in a "restriction" clause; so, no further information about it is needed}

ε

| **FromPar**

{The type of the parameter to the operation is in terms of a type from another parameter to the module}

<type from par>

| **BuiltIn**

{For future use}

<Id_num>

<v_num>

| **control**

{The type is "control" (this is only for results)}

ε

<optional result> —> <op par type>*

<restriction> —> <rest_type>
<rest_type>

<rest_type> —> **FromPar**
{The restricted type is in terms of a type from a module parameter}
<type from par>
| **BuiltIn**
{For future use}
<Id_num>
<v_num>
| **TypeOfParToOpPar**
{The restricted type is the type of a parameter to an operation parameter to the realization module}
<p_num>
{The p-number of the operation parameter to the module}
<op par number>
{The order (1st, 2nd, ...etc) of the parameter to the operation}
| **TypeOfResOfOpPar**
{The restricted type is the type of the result of an operation parameter to the realization module}
<p_num>
{The p-number of the operation parameter to the module}

<op par number> —> INTEGER

<type from par> —> **PrvddBy**

{This alternative specifies a type that is provided by a facility at the end of a facility chain}

<fac_chain>
{The chain here has length ≥ 1; that is, the type is not provided by "self"}

<v_num>

| **TypeParTo**

{This alternative specifies a type that is a type parameter to a facility at the end of a facility chain}

<fac_chain>
{The length of the chain here can be 0}

<p_num>
{The p-number of the type parameter}

<dclrd fac> —> <template name>

<RM name>

<actual c_par>*
{Actual conceptual parameters}

<actual r_par>*
{Actual realization parameters}

<actual c_par> —> **fac**

<regular fac>
{The regular facility here is never "self" or a facility not declared yet}

| **type**

<regular type>
{The regular type here is never a type provided be "self"}

<actual r_par> —> **fac**

<regular fac>

{The regular facility here is never "self" or a facility not declared yet}

| **type**

<regular type>

{The regular type here is never a type provided by "self"}

| **OpPar**

{This alternative specifies an actual operation parameter in the facility declaration, where the operation is a parameter to the realization module}

<p_num>

{The p-number of the operation parameter}

| **OtherOp**

{This alternative specifies an actual operation parameter in the facility declaration, where the operation is not a parameter to the realization module}

<regular fac>

{The regular facility here is never "self" or a facility not declared yet}

<v_num>

{v-number of the operation}

<other used fac> —> **BuiltIn**

{for future use}

<Id_num>

| **FromParChain**

{This alternative specifies a facility at the end of a facility chain}

<fac_chain>

{The chain here has length > 1 since an OUF is never "self" or a facility of a parameter to the module}

| **FacPrvdgTPar**

{This alternative specifies a facility that provides an actual type parameter to a facility at the end of a facility chain}

<fac_chain>

{The chain here has length ≥ 1 since an OUF is never providing a type parameter to "self"}

<p_num>

{The p-number of the type parameter to the facility at the end of the chain}

<local code> —> <op offsets>

{For each operation in this realization module, this is the offset of the beginning of its code from the beginning of the code generated for the whole module. Clearly, the offset for the first operation is 0. The order of the operations is:

```
init 1, fin 1,
:
init m, fin m,
module init,
provided op 1, ...,
      provided op n,
private op 1, ...,
      private op i}
```

<code>

<op offsets> —> INTEGER+

<code> —> INTEGER+

<xtrnl op> —> **OpFromPar**

{This alternative specifies an operation which is a parameter to the realization module}

<p_num>

{The p-number of an operation parameter to the module}

| **PrvddOp**

{This alternative specifies an operation provided by a facility from the module parameters or by a declared facility}

<regular fac>

<v_num>

{The v-number of the operation}

| **InitOp**

{This alternative specifies the initialization operation for a given type}

<regular type>

{This is never a type provided by "self" since this would make the operation local}

| **FinOp**

{This alternative specifies the finalization operation for a given type}

<regular type>

{This is never a type provided by "self" since this would make the operation local}

| **FacInitOp**

{This alternative specifies the initialization operation for a given facility}

<regular fac>

{This is never "self" since this would make the operation local}

<provided type> —> STRING

{The type name}

<provided op> —> **Proc**
<op name>
<par>*
| **Func**
<op name>
<par>*
<par type>
{The type of the result of the function}
| **Cntr**
<op name>
<par>*

<op name> —> STRING

<par> —> <mode>
<par type>

<mode> —> INTEGER

<par type> —> **FromPar**
<type from par>
| **FromTheoryFac**
{This alternative may be used in theory modules only}
<d_num>
<v_num>

<dclrd theory fac> —> <theory name>
<math type>*

{These mathematical types are the actual parameters in the declaration of the theory facility}

<theory name> —> STRING

<math of prvdd type>

—> <d_num>

{This is the d-number of a declared theory facility, which provides the mathematical type corresponding to a provided type}

<v_num>

{The v-number of the mathematical type in the theory realization module}

<other used theory fac>

—> **FacPrvdg**

<regular type>

{This means the theory facility which provides the math type corresponding to the given regular type}

<code of model ops>

—> <op offsets>

{For each model operation in the realization module, this is the offset of the beginning of its code from the beginning of the code of the model operations in the module}

<code>

<special xtrnl op> —> **Regular**

<xtrnl op>

| **Math**
<math xtrnl op>

<math xtrnl op> —> **PrvddOp**
<theory fac>
<v_num>
| **InitOp**
<math type>
| **FinOp**
<math type>
| **ModelOp**
<regular type>
| **FacInitOp**
<theory fac>

<math type> —> **PrvddBy**
<theory fac>
<v_num>
| **MathOf**
{This alternative specifies a type that is the math type corresponding to a given regular type}
<regular type>

<theory fac> —> **Dclrd**
<d_num>

<regular fac> —> **FromParChain**

{This alternative specifies the facility at the end of a facility chain}

<fac_chain>

| **dclrd**

{This alternative specifies a declared facility.}

<d_num>

| **BuiltIn**

{For future use}

<Id_num>

<regular type> —> **PrvddBy**

<regular fac>

<v_num>

| **TypeParTo**

{This alternative specifies a type that is a parameter to a facility at the end of a given facility chain}

<fac_chain>

<p_num>

{The p_number of the type parameter to the facility at the end of the chain}

<fac_chain> —> <p_num>*

{This specifies a facility; "self," facility parameter to "self", facility parameter to a facility parameter to "self", ...etc. If the length of the chain is 0, this means "self." Otherwise, the specified facility can be reached by following the chain of facility parameters starting at "self." E.g., the sequence of p_numbers 5 2 4 (each is for a facility parameter) specifies the facility which is the 4th parameter to

the 2nd parameter to the 5th parameter to "self." Note that only the first p_number in the sequence (5 in the example) can be of a realization parameter. All the other p_numbers are of conceptual parameters}

<aux in> —> INTEGER
{This has the value 1 if the information needed in constructing the RTS for testing is included in the CRM, and the value 0 otherwise}

<v_num> —> INTEGER

<p_num> —> INTEGER

<d_num> —> INTEGER

<Id_num> —> INTEGER

APPENDIX G
The Interpreter's Operations

Preliminary Notes

- The interpreter can be viewed as a repository of realization modules, facilities, and variables, and operations which manipulate the contents of this repository. Every element in the repository (a realization module, a facility, or a variable) is assigned an identification number by the interpreter when it first gets into the repository. Such identification number will be referred to as RM-id, facility-id, or variable-id, respectively. When an element in the repository is involved in an operation among those of the interpreter, it is referred to by its identification number.

- When a facility is created (and added to the repository), the theory facilities which are declared in the realization module from which the facility is created are implicitly created, assigned identification numbers, and added to the repository.

- The *v-number* of a type provided by a module is defined to be the order of the provided type among all the types provided by this module (starting with 1). Similarily, the *v-number* of an operation provided by a module is defined to be the order of the provided operation among all the operations provided by this module (starting with 1).

- Every module parameter has a *p-number*. The p-number of the i^{th} conceptual parameter is i, and the p-number of the j^{th} realization parameter is (c+j), where c is the number of the module's conceptual parameters.

- The following types are used in the interpreter's operations:
  - integer.
  - boolean.
  - string: This is a character string.
  - int_list: This is a sequence (possibly empty) of integers.
  - act_par: This type describes an actual module parameter. It is a discriminated union of facility, type, and operation. When it is a facility, its value is the identification number of a facility. When it is a type (or an operation), its value consists of a pair of integers for the identification number of the facility providing the type (or the operation) and the v-number of the type (or the operation).
  - act_par_list: This is a sequence (possibly empty) of act_par's.
  - partial_par_list: This is like an act_par_list, except that some of the act_par's in the sequence may be unspecified (*null*).

- gpd: This is the general-purpose-data type, which has been described in the context of chapter 5.

The Interpreter's Operations

```
Load_RM (RM_name: string): integer
```

This operation takes the name of a realization module and adds the named realization module to the interpreter's repository. The theory realization modules used by the given realization module are implicitly added to the repository. The value returned by the operation is the RM-id assigned by the interpreter to the realization module named in the operation's argument.

```
Instantiate (RM_id: integer, act_conc_pars: act_par_list,
                    act_real_pars: act_par_list): integer
```

This operation creates a facility from a realization module whose RM-id is given along with actual module parameters. The value returned is the facility-id assigned by the interpreter to the created facility.

```
Initialize_Var (fac: integer, v_num: integer): integer
```

This operation creates a new initialized variable of some type. The type is specified in the arguments as a facility-id and a v-number. The variable-id of the new variable is returned.

```
Finalize_Var (var_id: integer)
```

The variable whose identification number is given is finalized and removed from the repository.

```
Conceptualize (var_id: integer): integer
```

This operation produces a mathematical variable whose value is the mathematical model of the value of a given regular variable. The variable-id of the new mathematical variable is returned.

```
Perform_Proc (fac: integer, v_number: integer, args: int_list)
```

This operation performes a RESOLVE procedure on given arguments. The procedure is given as a facility-id and a v-number, and the arguments are given as a sequence of variable-id's. At the conclusion of the Perform_Proc operation, the values of the given variables may have been changed according to the actual effect of the given procedure.

```
Perform_Func (fac: integer, v_number: integer,
                                args: int_list): integer
```

This operation performes a RESOLVE function on given arguments. The function is given as a facility-id and a v-number, and the arguments are given as a sequence of variable-id's. The value returned by the Perform_Func operation is the variable-id of a new variable that has the value returned by the given RESOLVE function.

```
Perform_Cntrl (fac: integer, v_number: integer,
                                args: int_list): boolean
```

This operation performes a RESOLVE control operation on given arguments. The control operation is given as a facility-id and a v-number, and the arguments are given as a sequence of variable-id's. The value returned by the Perform_Cntrl operation is *true* if the result of the control operation is *yes* and *false* if the result of the control operation is *no*.

```
Facility_Num (var: integer): integer
```

This operation takes the variable-id of a variable and returns the facility-id of the facility which provides the variable's type.

```
V_Num (var: integer): integer
```

This operation takes the variable-id of a variable and returns the v-number of the variable's type.

```
RM_Id (fac: integer): integer
```

This operation takes the facility-id of a facility and returns the RM-id of the realization module from which the facility is created.

```
Act_Par (fac: integer, p_num: integer): act_par
```

This operation takes the facility-id of a facility and the p-number of a parameter to this facility, and returns the actual parameter.

```
Theory_Id (RM_id: integer): integer
```

Every mathematical theory in the set of theories which are used in the specification of RESOLVE modules is assigned a unique positive number. The Theory_Id operation takes the RM-id of a realization module and returns 0 if the realization module is not a theory module, and the number of the corresponding theory if it is a theory module.

```
Value (var: integer): string
```

This operation takes, as an argument, the variable-id of a variable whose type must be mathematical integer (i.e., the type provided by the theory module implementing number theory). The returned value is a character string representing the value of the given variable.

```
Reg_Op_Par_Ok (fac: integer, v_num: integer,
                                    i: integer, var: integer): boolean
```

This operation checks whether the type of the variable whose variable-id is var is consistent with the type of the i^{th} formal parameter of the regular operation provided by the facility whose facility-id is fac, with v-number v_num.

```
Thry_Op_Par_Ok (fac: integer, v_num: integer,
                                    i: integer, var: integer): boolean
```

This operation checks whether the type of the variable whose variable-id is var is consistent with the type of the i^{th} formal parameter of the theory operation provided by the facility whose facility-id is fac, with v-number v_num.

```
Mod_Pars_Ok (RM_id: integer, act_conc_pars: partial_par_list,
                          act_real_pars: partial_par_list): boolean
```

This operation checks whether the actual module parameters in the partial parameter lists are consistent with their corresponding formal module parameters.

```
Interface (RM_id: integer): gpd
```

This operation returns a gpd representation of the interface of the realization module whose RM-id is given.

APPENDIX H
The Display Operation for Strings

The display operation for strings takes the variable-id of a variable of type string and displays in a new window a string of icons corresponding to the components of the string value of the given variable. Following is a pseudocode for the string display operation in terms of the interpreter's operations.

```
String_Display (v: integer)
        /* v is the variable-id of a string variable */

f := Facility_Num(v)  /* f is the facility-id of the facility
                         providing the type of the given
                         variable; i.e., the string theory
                         facility */
length := ⟨the v-number of the "length" operation
                        in the string-theory module⟩
v':= ⟨an int_list value consisting of just the variable-id v⟩
l := Perform_Func(f, length, v')
len := ⟨string_to_int of Value(l)⟩
                 /* len is the length of the given string value */
⟨make a new window⟩
if len = 0 then ⟨display "Λ" in the new window⟩
 else
```

```
a := Act_Par(f, 1)
a_fac := ⟨the facility component of a⟩
                /* a_fac is the facility-id of the theory
                   facility providing the math type of the
                   components of the given string value */
a_v_num := ⟨the v-number component of a⟩
                /* a_v_num is the v-number of the math type of
                   the components of the given string value */
thry := Theory_Id(RM_Id(a_fac))
type_name := ⟨the name of the math type provided by the theory
                module whose RM-id is RM_Id(a_fac) with
                v-number a_v_num⟩
if ⟨thry correspond to a theory of primitive domain⟩
   then primitive := true
   else primitive := false
endif
get := ⟨the v-number of the "Get" operation in the
          string-theory module⟩
index := ⟨the variable-id of a variable whose type is
             mathematical integer and whose value is 0⟩
for i := 1 to len do
   ⟨prepare a math icon whose attached type name is the
      character string in type_name⟩
   ⟨increment the value of the variable whose variable-id
      is index by 1⟩
   index′:= ⟨an int_list value consisting of just the
                variable-id index⟩
   e := Perform_Func(f, get, index′)
   if primitive
      then ⟨get the primitive value of the variable whose
                variable-id is e and write the value inside
                the prepared icon⟩
   endif
   ⟨make the prepared icon map to e in the
      icon-variable mapping⟩
```

```
      ⟨display the prepared icon at the proper position (depending
         on i) in the new window⟩
      if l < len
         then ⟨display a link at the proper position (depending
                  on i) in the new window⟩
      endif
   endfor
endif
Finalize(l)
Finalize(index)
end String_Display
```

References

[Ada83] *The Ada Programming Language Reference Manual.* US Department of Defense, US Government Printing Office, 1983.

[Adri82] Adrion, W., Branstad, M., Cherniavsky, J. "Validation, Verification, and Testing of Computer Software, " *ACM Comp. Surv.* **14**, 2 (June 1982), 159-192.

[Bigg84] Biggerstaff, T., Perlis, A. "Foreward," *IEEE Trans. Softw. Eng.* **SE-10**, 5 (Sept. 1984), 474-477.

[Boeh81] Boehm, W. *Software Engineering Economics.* Prentice-Hall, 1981.

[Boeh83] Boehm, W., Standish, T. "Software Technology in the 1990's: Using an Evolutionary Paradigm," *IEEE Computer* **16**, 11 (Nov. 1983), 30-37.

[Chow85] Chow, T. *Tutorial: Software Quality Assurance, A Practical Approach.* IEEE Computer Society, 1985.

[Clar76] Clarke, L. "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Trans. Softw. Eng.* **SE-2**, 5 (Sept. 1976), 215-222.

[Deut82] Deutsch, M. *Software Verification and Validation.* Prentice-Hall, 1982.

[Fair85] Fairley, R. *Software Engineering Concepts.* McGraw-Hill, 1985.

[Gann81] Gannon, J., McMullin, P., Hamlet, R. "Data-Abstraction, Implementation, Specification, and Testing," *ACM TOPLAS* **3**, 3 (July 1981), 211-223.

[Gogu78] Goguen, J. "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in R. Yeh (Ed.), *Current Trends In Programming Methodology, vol. IV—Data Structuring*, Prentice-Hall, 1978, 80-149.

[Gogu84] Goguen, J. "Parameterized Programming," *IEEE Trans. Softw. Eng.* **SE-10**, 5 (Sept. 1984), 528-543.

[Gogu86] Goguen, J. "Reusing and Interconnecting Software Components," *IEEE Computer* **19**, 2 (Feb. 1986), 16-28.

[Gour83] Gourlay, J., "A Mathematical Framework for the Investigation of Testing," *IEEE Trans. Softw. Eng.* **SE-9**, 6 (Nov. 1983), 686-709.

[Gutt78a] Guttag, J., Horning, J. "The Algebraic Specifications of Abstract Data Types," *Acta Informatica* **10**, 1 (1978), 27-52.

[Gutt78b] Guttag, J., Horowitz, E., Musser, D. "Abstract Data Types and Software Validation," *Comm. ACM* **21**, 12 (Dec. 1978) 1048-1064.

[Gutt78c] Guttag, J., Horowitz, E., Musser, D. "The Design of Data Type Specifications," in R. Yeh (Ed.), *Current Trends In Programming Methodology, vol. IV—Data Structuring*, Prentice-Hall, 1978, 60-79.

[Harm88] Harms, D., Weide, B. *Swapping—A Desirable Alternative to Copying.* OSU-CISRC-1/88-TR2, The Ohio State University, January, 1988.

[Horo84] Horowitz, E., Munson, J. "An Expansive View of Reusable Software," *IEEE Trans. Softw. Eng.* **SE-10**, 5 (Sept. 1984), 477-487.

[Howd77] Howden, W. "Symbolic Testing and the Dissect Symbolic Evaluation System," *IEEE Trans. Softw. Eng.* **SE-3**, 4 (July 1977), 266-278.

[Huan77] Huang, J. "Error Detection Through Program Testing," in R. Yeh (Ed.), *Current Trends In Programming Methodology, vol. II—Program Validation*, Prentice-Hall, 1977, 16-43.

[Jens79] Jensen, R., Tonies, C. *Software Engineering.* Prentice-Hall, 1979.

[Jone78] Jones, D. "A Note on Some Limits of the Algebraic Specification Method," *SIGPLAN Notices* **13**, 4 (April 1978), 64-67.

[Jone84] Jones, T. "Reusability in Programming: A Survey of the State of the Art," *IEEE Trans. Softw. Eng.* **SE-10**, 5 (Sept. 1984), 488-494.

[Kart86] Kartashev, S., Kartashev, S. "Guest Editors' Introduction," *IEEE Computer* **19**, 2 (Feb. 1986), 9-13.

[Kern81] Kernighan, B., Mashey, J. "The UNIX Programming Environment," *IEEE Computer* **14**, 4 (April 1981), 12-24.

[Kern84] Kernighan, B. "The UNIX Stystem and Software Reusability," *IEEE Trans. Softw. Eng.* **SE-10**, 5 (Sept. 1984), 513-518.

[King76] King, J. "Symbolic Execution and Program Testing," *Comm. ACM* **19**, 7 (July 1976) 385-394.

[Kron88] Krone, J. *The Role of Verification in Software Reusability*. PhD Th., The Ohio State University, 1988.

[Lane84] Lanergan, R., Grasso, C. "Software Engineering with Reusable Design and Code," *IEEE Trans. Softw. Eng.* **SE-10**, 5 (Sept. 1984), 498-501.

[Lisk86] Liskov, B., Guttag, J. *Abstraction and Specification in Program Development*. The MIT Press, 1986.

[Luck87] Luckham, D., von Henke, F., Krieg-Bruckner, B., Owe, O. *ANNA: A Language for Annotating Ada Programs*. Springer-Verlag, 1987.

[Majs77] Majster, M. "Limits of the Algebraic Specification of Abstract Data Types," *SIGPLAN Notices* **12**, 10 (Oct. 1977), 37-42.

[Meye85] Meyer, B. "On Formalism in Specifications," *IEEE Software* **2**, 1 (Jan. 1985), 6-26.

[Myer80] Myers, B. "Displaying Data Structures for Interactive Debugging," Xerox PARC CSL-80-7 (June, 1980).

[Ogde87] Ogden, W. *CIS 680 Class Notes*, The Ohio State University, 1987.

[Oste81] Osterweil, L. "Software Environment Research: Directions for the Next Five Years," *IEEE Computer* **14**, 4 (April 1981), 35-44.

[Stan84] Standish, T. "An Essay on Software Reuse," *IEEE Trans. Softw. Eng.* **SE-10**, 5 (Sept. 1984), 494-497.

[Stro86] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.

[Teit81] Teitelman, W., Masinter, L. "The Interlisp Programming Environment," *IEEE Computer* **14**, 4 (April 1980), 25-34.

[Wall80] Wallis, P. "External Representations of Objects of User-Defined Type," *ACM TOPLAS* 2, 2 (April 1980), 137-152.

[Weid86a] Weide, B. *Design and Specification of Abstract Data Types Using OWL*. OSU-CISRC-TR-86-1, The Ohio State University, January, 1986.

[Weid86b] Weide, B. *A Catalog of OWL Conceptual Modules*. OSU-CISRC-TR-86-2, The Ohio State University, January, 1986.

[Weid86c] Weide, B. *A New ADT and Its Application in Implementing "Linked" Structures*. OSU-CISRC-TR-86-3, The Ohio State University, January, 1986.

[Weid87] Weide, B. *CIS 680 Class Notes*, The Ohio State University, 1987.

[Weid89] Weide, B. *RESOLVE Reference Manual.* In preparation.

[Wulf81] Wulf, W., Shaw, M., Hilfinger, P., Flon, L. *Fundamental Structures of Computer Science.* Addison-Wesley, 1981.

[Yin87] Yin, W., Tanik, M., Yun, D., Lee, T., Dale, A. "Software Reusability: A Survey and a Reusability Experiment," *Proceedings of the 1987 Fall Joint Computer Conference*, Dallas, TX, Oct. 1987, 65-70.