MIT/LCS/TR-299

A Two-Tiered Approach to Specifying Programs

Jeannette Marie Wing

This blank page was inserted to preserve pagination.

A Two-Tiered Approach to Specifying Programs

by

Jeannette Marie Wing

© Massachusetts Institute of Technology 1983

This research was supported in part by the National Science Foundation under grant MCS-8119846 and by the Defense Advanced Research Projects Agency monitored by the Office of Naval Research under Contract No. N0014-83-K-0125.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

A Two-Tiered Approach to Specifying Programs

by

Jeannette Marie Wing

Abstract

Current research in specifications is beginning to emphasize the practical use of formal specifications in program design. This thesis presents a specification approach, a specification language that supports that approach, and some ways to evaluate specifications written in that language.

The two-tiered approach separates the specification of underlying abstractions from the specification of state transformations. In this approach, state transformations and target programming language dependencies are isolated into an *interface* language component. All interface specifications are built upon *shared* language specifications that describe the underlying abstractions. This thesis presents an interface specification language for the CLU programming language and presumes the use of the Larch shared language.

This thesis also suggests a number of kinds of analyses that one might want to perform on two-tiered specifications. These are related to the consistency, completeness, and strength of specifications, and are all presented in terms of the theories associated with specifications.

Thesis Supervisor: John V. Guttag
Title: Associate Professor of Computer Science and Engineering
Keywords: Formal Specifications, Program Design, Specification Languages, Specification Analysis, Algebraic Specifications, Abstract Data Types, Programming Methodology.

This report is a revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on 19 May 1983 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Acknowledgments

I would like to express my foremost thanks to my advisor, John Guttag, for his sustaining guidance and encouragement throughout my graduate career. His keen intuition helped me separate the good ideas from the bad, the important from the trivial, the interesting from the dull. I am deeply grateful to Jim Horning for his invaluable technical help on my thesis and for the opportunities to share ideas with him at Xerox Parc. I would like to thank both John and Jim for their help in formulating my thesis topic, and their constant faith and interest in it. I would also like to thank Barbara Liskov for her perceptive comments on my final draft. All three deserve many thanks for their careful and thorough readings of my thesis and their suggestions for improving its presentation and organization. John deserves special credit for patiently editing numerous drafts of many of the chapters.

I owe thanks to many people for their friendship and their help in providing a bearable working environment. In particular, I am indebted to Bill Weihl for his technical expertise, and his willingness to discuss problems and suggest solutions. I am also thankful to Kathy Yelick for diligently reading drafts of my entire thesis; to Maurice Herlihy, Gene Stark, Karen Sollins, and Joe Zachary for various enlightening conversations and their feedback on some of my chapters; to Pierre Lescanne for listening to my ideas, answering my questions, and heightening my awareness of MIT parochialism. I would also like to express my appreciation to Srivas Mandayam and Sriram Atreya for their open ear and encouragement at the start of my thesis research; to Julie Lancaster, Randy Forgaard, Ron Kownacki, and Brian Oki, at the end.

Finally, I wish to thank my parents with all my heart for their unceasing moral support, optimism, and confidence in me.

CONTENTS

1. Introduction 7

1.1 The Problem 9
1.2 The Two-Tiered Approach 10
1.2.1 The Approach 10
1.2.2 Two-Tiered Specifications 11
1.2.3 Following the Approach 13
1.3 A Glimpse at a Particular Two-Tiered Specification Language 14
1.3.1 A Preview of the CLU Interface Language 15
1.3.2 An Overview of Larch 18
1.4 Related Work 21
1.4.1 Program Verification 21
1.4.2 Program Development 22
1.4.3 Abstract Data Types 23
1.4.4 Specification Languages 24
1.5 What is in this Thesis 26
1.5.1 Approach to the Formalization 26
1.5.2 A Guide to the Rest of the Thesis 28

2. Kernel Interface Language 30

2.1 Classes of Models 31
2.1.1 Traits and Algebras 32
2.1.2 Objects 33
2.1.3 State 33
2.1.4 Procedures and Operations 34
2.1.5 Clusters and Abstract Data Types 36
2.1.6 Computations 36
2.2 Kernel Interface Language and Models 39
2.2.1 Interface Assertion Language 40
2.2.2 Procedure Specifications 47
2.2.3 Cluster Specifications 58
2.3 Summary 61

3. Theories 62

3.1 Definitions 62
3.2 Satisfaction 64
3.3 Theory of a Specification 65
3.3.1 Theory of a Trait 65
3.3.2 Theory of a Procedure Specification 68

3.3.3 Theory of a Cluster Specification 71
3.4 Theory of an Implementation 71
3.4.1 Theory of a Procedure 71
3.4.2 Theory of a Cluster 74
3.5 Type Induction 74
3.5.1 Computational Induction 75
3.5.2 Type Induction Principle 76
3.6 Summary 87

4. Extended Interface Language for CLU 88

4.1 Simple Extensions 88
4.1.1 Default Used Trait 89
4.1.2 Mutates Clause 90
4.1.3 Default Termination Condition Value 92
4.1.4 Multiple Pre- and Post- Conditions 93
4.2 Handling Other CLU Features 94
4.2.1 Memory Objects 95
4.2.2 Iterators 98
4.2.3 Parameterized Specifications 105

5. Evaluating Specifications 113

5.1 Properties of Specifications 114
5.1.1 Consistency 115
5.1.2 Full-Coverage 118
5.1.3 Determinism 120
5.1.4 Protection 123
5.2 Comparing Specifications 126
5.2.1 Comparing Strength 127
5.2.2 Definition of Strength 128
5.2.3 Modifying a Specification With Respect To Strength 131
5.3 Essentiality 134
5.3.1 Definitions 135
5.3.2 Situations for Determining Inessentiality 136

6. Conclusions, Contributions, and Further Work 138

6.1 Summary of Conclusions and Contributions 138
6.2 Directions for Further Work 140
6.2.1 Development of Interface Languages 140
6.2.2 Evaluating Collections of Specification 141
6.2.3 Machine Support 142
6.2.4 Experimentation 143

References **144**

Appendix I. Interface and Trait Specifications **152**

Appendix II. Proofs **156**

II.1. Validity of a Type Induction Rule 156
II.2. Proof of Satisfaction 157

1. Introduction

The goal of this thesis is to help people write formal specifications of pieces of large software. To achieve this goal, we propose a two-tiered approach for formally specifying the behavior of sequential programs, we describe a language that supports this approach, and we suggest ways to evaluate specifications written in this language.

A specification describes a program's behavior; it is independent of the program itself. It is formal if it is written in a language with explicitly and precisely defined syntax and semantics. Two virtues of formal specifications are their precision and amenability to machine-manipulation.

Current research in specifications is beginning to emphasize the practical use of formal specifications in the programming process. People have already benefited from using informal specifications in most phases of this process. Writing informal specifications is widely accepted as a useful way of organizing ideas, documentating design decisions, and informally arguing the correctness of programs. Software design methods that include some form of informal specification have been in use in industry for some time [Caine75, Jackson75, Katzan76, Yourdon78].

Thus far, formal specifications have played a less influential role in the programming process than informal specifications. People have used them with limited success in program verification, and have just begun using them in program design. We believe that formal specifications can and should play a more important role in the programming process than they do now.

Using formal specifications early in the programming process, i.e., the design phase, should reduce the time, effort, and resources spent in the overall process, especially in the costly testing, debugging, and maintenance phases. It is often the act of specifying and not the final product that is most useful in the design phase. Uncovering bugs early can save the

cost of uncovering them later in the testing and debugging phases. Also, as with informal specifications, a formal specification serves as a valuable piece of documentation--a means of communicating between a client and a specifier, between a specifier and programmers, and among programmers.

There are many problems with trying to use formal specifications during program design. Ironically, one is that the need to be precise intimidates many programmers. The problem of programmers learning how to read and write formal specifications can be gradually overcome. Every programmer has already learned to deal with at least one formal language--a programming language. We need to make formal specifications more accessible to programmers by supplying an easy-to-learn and easy-to-use specification language, and by suggesting guidelines for reading and writing specifications.

Another problem is that much of the past research in formal specifications focused on theory and not practice, so that specifications of small examples pervade literature, e.g., the ubiquitous stack. The result of this theoretical focus is a collection of small and self-contained specifications of the behavior of well-understood data structures or of small and simple programs. Small examples are not convincing and the lack of larger ones reinforces people's reluctance to accept the use of formal specifications. We need to demonstrate the use of formal specifications on larger examples.

The problem of size has been addressed in programming. In the same way a large program is constructed from program modules, the specification of a large program should be constructed from specifications of the program modules. This technique introduces the two subproblems of how to specify the pieces and how to combine them; this thesis focuses on the former.

Finally, another problem is that in the development of a specification the specifier is usually not provided with any feedback as to whether the specification is in some sense "correct." We need to identify and check for properties of the specification that relate to its utility. Ideally, we would check individual components of the specification for local properties, like sufficient-completeness [Guttag75], expressive-richness [Kapur80b], and implementation-bias [Jones80], and the entire specification for global properties, like modularity [Parnas72b] and coupling [Myers75]. Since we expect specifications to grow incrementally, feedback needs to be provided on incomplete specifications.

We organize the rest of this chapter as follows. Section 1.1 contains a statement of the problem and the essence of our solution. The next two sections describe in some detail, but not formally, the key aspects of the specification approach, and the key features of a particular specification language. We define the language precisely in later chapters. Section 1.4 contains a discussion on related work. Section 1.5 presents the approach we take for providing a formal basis for defining the specification language. It also contains a guide to the rest of this thesis.

1.1 The Problem

The main problem specifiers face is that formal specifications are hard to write. The effort involved in writing them has thus far been disproportionate to the benefit gained from having written them. We propose one step towards a solution to this problem by providing the specifier with:

1. A specification approach,
2. A specification language, and
3. Ways to evaluate specifications.

The most significant contribution of this thesis is the specification approach, *the two-tiered approach*. It motivates the design of the specification language whose precise definition constitutes the bulk of this thesis. In this chapter, we discuss the approach and give an overview of the language; in Chapter 5, we address the evaluation of specifications.

We keep in mind the following two goals. First, we want to make specifications easier for programmers to understand. This goal greatly affected our language design. Second, we want to make it easier to reason about specifications with sufficient machine support. Machine support, such as that provided by a theorem-prover, allows us to infer properties about not only the specification, but also what it specifies. This goal greatly affected our approach to our formalization.

1.2 The Two-Tiered Approach

Sections 1.2.1 and 1.2.2 describe, in general terms, the two-tiered approach and two-tiered specifications, respectively; Section 1.2.3 outlines how a specifier would follow our approach to write a specification.

1.2.1 The Approach

The two-tiered approach to specifying programs separates the specification of underlying abstractions from the specification of state transformations. We use a *shared* specification language to describe underlying abstractions, and an *interface* specification language to describe state transformations. The specification of a program module is written in an interface language and consists of two parts: a *shared language component* (bottom tier) and an *interface language component* (top tier). These two components correspond to the two tiers in our approach.

The interface specification language is programming language dependent, while the shared language is programming language independent. This allows us to keep separate the description of programming language independent issues from the description of programming language dependent ones, e.g., side effects, error handling, and resource allocation. For example, if we were to implement arithmetic, we would describe ideal arithmetic in the shared language, and we would describe boundary conditions constrained by word and memory size in an interface language.

Since the invention and description of key abstractions is done in the shared language, we expect most of the effort involved in writing a specification to be invested in the shared language component. The interface language component should deal only with state transformations and programming language dependent issues. One reason for separating the two language components is that we expect many shared language components to be reuseable by different interface language components. Some of them will be developed for particular applications; a few central ones will be useful in many applications.

We use the term "interface" because an interface specification describes all the information about the behavior of the program module. Any user of a program module need only look at its interface specification to understand the module's behavior. We use the term "shared" because in the design of a family of interface languages, each interface language is derivable from a subset of a target programming language, and a common subset, which is the shared language.

1.2.2 Two-Tiered Specifications

In this thesis we focus on the description of an interface language for the programming language CLU [Liskov77, Liskov81]. In this section, however, we discuss, in general terms, syntactic and semantic properties of interface and shared language components.

An interface language component has three parts: a *header*, a *body*, and a *link* to the shared language component of the specification. The syntax of the header is based on the syntax of the programming language. For example, the types of the input and output arguments to a procedure are listed in the header information of a procedure specification as they would be in an implementation. The body contains first-order *assertions* written in a language based on its shared language component, plus special assertions, which are introduced to handle issues dependent on the semantics of the programming language. The meaning of the assertions is based on first-order predicate logic with equality, where equality is defined by its shared language component. The link identifies the shared language component to be used.

The crucial syntactic information provided by a shared language component to an interface language component is a set of *sort* identifiers, and a set of *function* identifiers and function signatures. The function identifiers are composed to build *terms*, which are used to write the assertions appearing in the body of an interface language component. The sort identifiers and function signatures are used to *sort-check* terms much in the same way as type identifiers are used to type-check programs. The crucial semantic information provided by a shared language component to an interface language component is a theory of equality for terms.

By explicitly including a shared language component in an interface specification, we gain the advantage that every symbol in an assertion is precisely defined within a specification. In some other specification methods [Hoare72, Parnas72a], there is a reliance on an interpretation for symbols in an assertion, where the interpretation comes from outside the specification. For example, the meanings of symbols like $\in$ and $\subseteq$ might come from textbooks on set theory. In contrast, some other methods [Robinson77, Jones81] provide an assertion language defined within the specification, but restrict the symbols to come from a fixed set of primitives. We gain the advantage that the user is able to provide just the symbols

necessary to write the assertions in the body of a specification.

1.2.3 Following the Approach

When a designer begins to write specifications early in the programming process, the act of specifying intertwines with the act of designing. One helps the other. We sketch below a typical top-down design strategy that could be used in following the two-tiered approach.

1. Develop an approximate intuition of the problem to be solved. This requires close, often verbal, interaction with the client who is posing the problem.

2. Decide on the major abstractions.

    1. *Top tier*: Write the header information of the interface language components.

    2. *Bottom tier*: Write the syntactic information of the shared language components of the specification, i.e., the sort identifiers, and function identifiers and signatures.

3. Fill in the blanks.

    1. *Top tier*: Fill in the information in the bodies of the interface language components of the specification, e.g., write the assertions in the body of a procedure specification. Simultaneously generate additional function and sort identifiers needed from the shared language components.

    2. *Link between top and bottom tiers*: Define the explicit link to the shared language components of the specification.

    3. *Bottom tier*: Fill in the semantic information in the bodies of the shared languages components of the specification, i.e., the theory of equality for terms.

4. Check one's understanding of the problem and its formalization; repeat previous steps until convergence is achieved.

There are two points worth observing in regard to following this approach, especially for large pieces of software. First, as with any overall design method, many iterations over these steps may be necessary. Writing a specification sharpens a specifier's intuition of the problem. Hidden design decisions surface. Addressing postponed decisions often requires modifications of decisions made earlier. Second, the specifier should be willing to discard large chunks of a specification in the process of refining the abstractions. This is especially true after the first iteration. Often after a large investment in time and effort, the specifier (or designer or programmer) is reluctant to start anew or to try an alternate strategy. With sufficient machine support the specifier should be able to save time and effort often spent in managing and maintaining the consistency of a large specification.

During the process of writing a specification, the specifier should also evaluate it for certain properties, e.g., consistency and completeness. Checking for these properties as a specification develops can increase one's confidence that a specification is in some sense "good." We discuss the evaluation of specifications in Chapter 5. Finally, as with any design, the specifier should evaluate the overall structure of the specification, e.g., analyze the interconnectivity among its components. We do not address this kind of specification evaluation in this thesis.

1.3 A Glimpse at a Particular Two-Tiered Specification Language

In this section we provide an overview of the two-tiered specification language we define more precisely in the rest of this thesis. By considering a specific programming language and a specific shared language we gain the advantage of being concrete in defining our interface language.

The interface language we describe is for the programming language CLU. Section 1.3.1 gives a preview of the CLU interface language with those concepts from CLU required to understand the interface language presented as needed.

The shared language we choose is the Larch Shared Specification Language [Guttag83a], henceforth referred to as "Larch." Enough similarity between Larch and other axiomatic specification languages (see Section 1.4.4 on related work) exists so that a different specification language could be used as the shared language. Section 1.3.2 gives an informal overview of Larch. We describe only the minimal subset of constructs in Larch needed to understand the examples presented in this thesis. Details on Larch can be found in [Guttag83b].

1.3.1 A Preview of the CLU Interface Language

CLU has the primitive notions of *object* and *state*. An object is an entity that can be manipulated by a program. Two important properties of an object are its *type*, which never changes, and its *value*, which may change. A state consists of a set of objects, a mapping from program variables (object identifiers) to objects, and a mapping from objects to values. Two important observable state changes are when a new object is created and when the value of an existing object changes. An object whose value can change is said to be *mutable*. A type is mutable if objects of that type are mutable.

It is important not to confuse an object and its type, which are CLU concepts, with a term and its sort, which are shared language concepts. The connection between the CLU and the shared language concepts is that (typed) objects have values that are denotable by (sorted) terms. Through the interface specifications of procedures and clusters, we establish a link between the values that objects can have and the terms defined by shared language components. We establish this link explicitly in the text of the interface specifications.

A CLU program consists of a set of modules, each of which is either a *procedure* or *cluster*. A procedure performs an action on a set of objects, and terminates returning a set of objects. Communication between a procedure and its invoker generally occurs through these objects. A cluster names a type and defines a set of procedures that create and manipulate objects of that type. Users of this type are constrained to treat objects of the type abstractly. That is, objects can be manipulated only via the procedures defined by the cluster so, in particular, information about how objects are represented in storage may not be used.

A procedure specification consists of a header, a link to its shared language component, and a body. Header information includes the types of the input and output arguments to the procedure and a list of possible termination conditions. The link is the name of a shared language component. Since the unit of encapsulation in Larch is called a *trait*, we call the link in an interface specification the *used trait*. The body of the specification contains two assertions that correspond to a pre-condition on the state when the procedure is invoked and a post-condition on the state when the procedure terminates. Terms in these assertions are constructed from function identifiers provided by the used trait. The pre- and post-conditions may also contain other special assertions particular to CLU's semantics.

Figure 1 gives an example of a procedure specification. The identifiers, *s* and *i*, that appear in the header denote objects of type *set* and *int*, respectively. The name of the shared language component is *SetOfInt*, which is *choose*'s used trait. The pre-condition is satisfied if the initial value of the input argument is not empty. The post-condition contains an assertion

---

```
choose = proc (s: set) returns (i: int)
  uses SetOfInt
       pre ~isEmpty(s↑)
       post has(s↑,i↓) ∧ s↓ = remove(s↑,i↓) ∧ mutates s
  end
```

Figure 1. Choose Procedure Specification

about the initial and final values of the *set* object and the final value of the *int* object. An object identifier that is followed by an up arrow (↑) denotes the value of that object in the state upon procedure invocation, i.e., the *initial* state; one followed by a down arrow (↓) denotes the value in the state upon procedure termination, i.e., the *final* state. The function identifiers, *isEmpty*, *has*, *remove*, and ∧, and the meaning of the equality symbol, =, all come from *SetOfInt*. The last conjunct in the post-condition, **mutates** s, is an example of a special assertion; it states that the *choose* procedure may mutate no object other than that denoted by *s*.

A cluster specification consists of a header, a link to the shared language component, and a body. The header is a list of procedure identifiers. The body of the specification consists of a set of procedure specifications. The link from the interface component to the shared component is given by a used trait and a **provides** clause. The used trait supplies all function identifiers that appear in the assertions of the procedure specifications of the cluster specification. The **provides** clause gives a mapping from a type identifier to a sort identifier. This mapping determines the values over which objects of the type defined by the cluster can range. All objects of the type are restricted to values denotable by terms of that sort. The sort identifier must appear in the used trait. The **provides** clause also indicates whether the type is mutable or not.

Figure 2 gives a skeleton of a cluster specification that defines the type, *set*. The used trait is *SetOfInt*. The **provides** clause gives a mapping from the type identifier, *set*, to the sort identifier, *SI*, which comes from *SetOfInt*. The keyword **mutable** indicates that objects of the *set* type are mutable. Specifications for *create*, *insert*, *remove*, and *member* are of the form described for procedure specifications.

```
set = cluster is create, insert, remove, member
  uses SetOfInt
  provides mutable set from SI

        create = proc () returns (s: set)
                 ...
          end

        insert = proc (s: set, i: int)
                 ...
          end

        remove = proc (s: set, i: int)
                 ...
          end

        member = proc (s: set, i: int) returns (b: bool)
                 ...
          end

  end
```

Figure 2. Set Cluster Specification

1.3.2 An Overview of Larch

The unit of encapsulation in Larch is called a *trait*. The identifier appearing before the keyword **trait** is the name of the trait and is distinct from the sort and function identifiers appearing in the trait. We will refer to Figures 3 and 4 to help illustrate the meanings of constructs appearing in traits. We repeat these figures in Appendix I for future reference.

```
Equivalence: trait
  introduces
        eq: E, E → Bool
  constrains [eq] so that for all [x, y, z: E]
        eq(x,x) = true
        eq(x,y) = eq(y,x)
        ((eq(x,y) ∧ eq(y,z)) ⇒ eq(x,z)) = true
```

Figure 3. Equivalence Trait

```
SetOfE: trait
  includes Integer, Equivalence
  introduces
        empty: → C
        add: C, E → C
        remove: C, E → C
        has: C, E → Bool
        isEmpty: C → Bool
        card: C → Int
  closes C over [empty, add]
  constrains [C] so that for all [s: C, e, e1: E]
        remove(empty, e) = empty
        remove(add(s,e), e1) = if eq(e,e1) then remove(s,e1) else add(remove(s,e1),e)
        has(empty, e) = false
        has(add(s,e), e1) = if eq(e,e1) then true else has(s,e1)
        isEmpty(empty) = true
        isEmpty(add(s,e)) = false
        card(empty) = 0
        card(add(s,e)) = if has(s,e) then card(s) else 1 + card(s)

SetOfInt: trait
  includes SetOfE with [SI for C, Int for E]
```

Figure 4. SetOfE and SetOfInt Traits

A trait contains a set of function declarations, which follows the keyword **introduces**, and a set of axioms, which follows a **constrains** clause. A function is declared by giving its name (an identifier) along with its signature, i.e., a domain and range. A domain is a list of sort identifiers, and a range is a single sort identifier. In the *Equivalence* trait (Figure 3), the *eq* function has two arguments of sort *E*, and returns a result of sort *Bool*. All traits may use boolean connectives, e.g., $\wedge$ and $\Rightarrow$ in *Equivalence*, with their usual first-order propositional logic meanings. Functions can be declared to be mixfix or prefix. For example, if *.eq* is to be used as an infix function, we would write "*#* .eq *#*: E, E → Bool" in its declaration.

There are two kinds of axioms that can appear after a **constrains** clause. One kind of axiom is an equation relating two *terms*. The " = " symbol denotes an equivalence relation on terms. The second kind of axiom, not seen in either Figure 3 or Figure 4, is of the form "τ **exempt**" where τ is a term. This indicates that the lack of an equation is not an oversight and

is an aid to "completeness" checking. An example of an axiom of this form is "*pop(null)* **exempt**," which might appear in a trait that defines a theory of stacks.

A function identifier is *constrained* if it appears in the bracketed list following the keyword **constrains**. If a sort identifier appears in the bracketed list (e.g., in the *SetOfE* trait of Figure 4), each function identifier whose signature contains that sort identifier is constrained. A **constrains** clause indicates the function identifiers that are intended to be constrained in the equations.

A trait denotes a theory, i.e., a set of formulae closed under a set of inference rules. Each equation appearing in a trait is a formula in the trait's theory. An axiom of the form "τ **exempt**" adds nothing to a trait's theory. We can enrich the theory denoted by a set of equations by adding **closes** clauses (explained below). Together the **introduces**, **constrains**, and **closes** clauses, the "inequation" ~(true = false), and propositional and quantified tautologies define a first-order theory of a trait.

A **closes** clause adds an inductive rule of inference to a trait. Closing a sort, S, over a set of function identifiers, F, asserts that there is a representative member, τ, of each equivalence class of terms of sort S, where each function identifier with range sort S appearing in τ is in F. The inductive rule of inference is used to add formulae to a trait's theory that cannot be shown using purely equational logic. For example, the **closes** clause in the *SetOfE* trait asserts that each term of sort *C* is equal to a term, τ, where each function identifier with range sort *C* appearing in τ is either *empty* or *add*. The associated inductive rule of inference can be used to derive theorems like $\forall s{:}C\ card(s) \geq 0$.

Larch also provides ways of putting traits together, one of which is an **includes** clause. A trait that includes another trait is textually expanded to contain all function declarations, **constrains** clauses, **closes** clauses, and axioms of the included trait. The meaning of the including trait is the meaning of the textually expanded trait. In *SetOfE*, the signature of *eq*,

which is used in the axioms of *SetOfE*, comes from that given in the included *Equivalence* trait.

Finally, function and sort identifiers that appear in an included trait can be renamed. An explicit renaming is given in brackets following the keyword, **with**. In the *SetOfInt* trait the sort identifiers *C* and *E* of *SetOfE* are respectively renamed to be *SI* and *Int*. Renaming is used both to collide identifiers intentionally and to prevent identifiers from colliding.

1.4 Related Work

Work related to this thesis falls into two broad categories: specification languages and uses of formal specifications. Various specification languages have developed in parallel with different roles of formal specifications in the programming process and with the evolution of higher-level languages. We now discuss each of the following topics as they relate to this thesis: using specifications in program verification, using specifications elsewhere in program development, specifying abstract data types, and specification languages.

1.4.1 Program Verification

Origins of the use of formal specifications can be traced to early work done on proofs of program correctness [Floyd67, Hoare69], and later work done on machine-aided program verification (e.g., see [King69, Deutsch73, Boyer75, Good75, vonHenke75, London75, Suzuki75]). Most of the work is based on Floyd's inductive assertions technique [Floyd67] and on Hoare's axiomatic approach to specifying the meaning of programs [Hoare69] (for an excellent review of subsequent developments based on Hoare's approach, see [Apt81]). Early proofs were of programs written in simple programming languages (e.g., **while** programs) or manageable subsets of higher-level languages like Pascal. Most of the work does not focus on the approach for the construction of specifications nor on the specification language itself; in contrast, our work focuses on both.

In the mid 1970's, the focus of program verification turned to problems of specifying programs using data structures like pointers, arrays, and records [Suzuki76, Luckham76, Wegbreit76, Reynolds77], and using shared data [Burstall72, Oppen75, Yonezawa77, Schaffert81]. Of these, Schaffert's work is most closely related to ours.

Schaffert studies the problem of specifying and verifying programs that use abstract data types and shared data with an emphasis on verification. Although his specification language is not particular to CLU, its design is motivated by CLU semantics. One difference between his specification language and ours is that he combines the specification of properties of objects of an abstract data type with the specification of properties of their values into one specification rather than separating them into two parts as in our two-tiered approach. Another difference is that his assertions are not restricted to first-order logic so mechanization of his proofs would be more difficult than of ours.

1.4.2 Program Development

Philosophical discussions on the practical use of formal specifications can be found in [Parnas77] and, more recently, in [Guttag82]. Guttag and Horning advocate the use of formal specifications in the design phase of program development in [Guttag80b], where they hint at the two-leveled approach to specifying programs. They specify routines using weakest-preconditions [Dijkstra76], but the main example of their paper contains no specifications of routines. More importantly, they do not make explicit, as we do, programming language dependencies in their routine specifications nor do they make explicit a connection between routine specifications and their algebraic specification components. Jones also advocates the use of formal specifications for program development; his formal method stems from the Vienna Definition Method (VDM) (see [Bjorner78] for extensive coverage and related references on VDM).

The use of specifications to enforce "modular" programming gave rise to the distinction between a "specification part" and "implementation part" in the encapsulation units of programming languages such as Mesa modules [Mitchell78] and Ada packages [Ada79]. Each encapsulation unit has a specification part that defines how implementation parts of other encapsulation units can use it. Specification parts contain syntactic information that the compiler can use, such as the types of input and output arguments, and possible termination conditions of a procedure, but no formal semantic information about the encapsulation unit, such as the input-output behavior of a procedure. The design of the CLU library includes this kind of specification information as well. Specifications in CLU, however, are not part of the syntax of the language. Specifications written in our interface language are like "specification parts" except that we provide not only syntactic, but also semantic, information about program modules.

1.4.3 Abstract Data Types

Formal specifications have been used extensively to describe abstract data types, leading to two different approaches, sometimes referred to as "operational" and "definitional." A survey of these approaches can be found in [Liskov79]. In the operational approach, one gives a method of constructing the abstract data type. Examples of the operational approach include Parnas's work on state-machines [Parnas72a], Robinson and Roubine's extensions to them with V-, O-, and OV-functions [Robinson77], Berzins's abstract models [Berzins79], and Jones's model-oriented specifications [Jones80].

In the definitional approach, one gives a list of properties of the abstract data type, not a method of constructing the type. The definitional approach can be broken into two categories, sometimes referred to as "axiomatic" and "algebraic." The axiomatic approach stems from Hoare's work on proofs of correctness of implementations of data types [Hoare72], where predicate logic pre- and post-conditions are used for the specification of each operation of the type. Other work using the axiomatic approach is in [Standish73] and

[Nakajima80]. In the algebraic approach data types are defined to be hetereogeneous algebras [Birkhoff70]. This approach uses axioms to specify properties of abstract data types, but the axioms are restricted to equations. Much work has been done on the algebraic specification of abstract data types [Goguen75, Guttag75, Zilles75, Burstall77, Ehrich78, Wand79, Kamin83] including the handling of error values [Goguen77, Goguen78, Kapur80a], nondeterminism [Kapur80a], and parameterization [Thatcher78, Goguen81, Ehrig80].

Our work is related to both the axiomatic and algebraic approaches. At the interface language level, a cluster specification that defines a data type is written in an axiomatic style since pre- and post-conditions are associated with each of the procedure specifications. At the shared language level, a trait specification is written in an algebraic style where axioms appearing in a trait are restricted to be primarily equational.

One significant difference between the axiomatic part of our approach and other axiomatic approaches is that we define the truth of an assertion with respect to two states. Since a program is normally viewed as an input-output relation, a post-condition often needs to refer to both the initial and final values of objects. Usual Hoare logic, in which each predicate in a triple is interpreted with respect to a single state [Hoare69], uses a standard trick of introducing free variables in pre-conditions to "save" the initial values. Jones avoids this by defining pre-conditions on one state and post-conditions on two [Jones80]. We also avoid this by interpreting all assertions, found in both pre- and post-conditions with respect to two states.

1.4.4 Specification Languages

Much of the work on specification languages has evolved from work done on the specification of abstract data types. The more widely-known specification languages that have resulted from this research are CLEAR [Burstall77, Burstall81], Iota [Nakajima80], Z [Abrial80], SPECIAL [Robinson77], and VDM's Meta-IV [Bjorner78]. CLEAR, Iota, and Z stem

from the definitional approach of describing abstract data types. SPECIAL and Meta-IV stem from the operational approach, so we discuss them separate from the other three.

CLEAR, Iota, and Z distinguish between a "syntactic part" and a "semantic part" where the syntactic part defines the signatures of functions. The semantic part of a CLEAR specification is a set of equations with universally quantified variables, and a possible induction rule. Models of a theory in CLEAR are based on initial algebras. The semantic part of an Iota specification is a set of axioms written in first-order predicate logic, and a possible induction rule. A model for an Iota specification is also an algebra, but since Iota does not restrict axioms to be equations, the existence of an initial algebra is not guaranteed. The semantic part of a Z specification is a set of predicates on sets, relations, and functions. A model for a Z specification is a set that satisfies those predicates together with an interpretation of the relation and function symbols.

One important difference between these three specification languages and ours is that specifications written in CLEAR, Iota, and Z have no simple way of specifying side effects and error handling of procedures that implement the specified functions. As stated in Section 1.2.1 we use the interface language component of a two-tiered specification to deal with issues like side effects and errors. As an intended consequence of our separation of concerns, CLEAR, Iota, and Z can be substituted for Larch as a shared language although doing so would correspondingly change the underlying models of interface specifications. Each, however, provides the required syntactic and semantic properties of the shared language that we discussed in Section 1.2.2.

SPECIAL's viewpoint is similar to our two-tiered viewpoint; it separates the "assertion" part, analogous to our shared language component, from the "specification" part, analogous to our interface language component. A major difference between SPECIAL and our work is that in SPECIAL, types used in the specification part are defined in the assertion part. A type is restricted to be either a primitive type, a subtype, or a structured type, each of which comes

with a set of pre-defined functions. Hence, since the assertion language is so restricted, most of the work of writing a specification is done in the specification part, where their O-, V-, and OV-function definitions correspond to our procedure specifications. We take the opposite viewpoint and expect most of the work of writing a specification to be done in the "assertion" part (shared language component).

The most significant difference between Meta-IV, which is the language of the Vienna Definition Method, and our language is that we do not use an operational approach to writing specifications. In Meta-IV, a model of an abstract data type is given in terms of previously defined types. Constraints on the properties of such a model are given in terms of "meta-programs," which include the use of declarations, assignment statements, and conditionals.

1.5 What is in this Thesis

We reemphasize that the most important contribution of this thesis is the two-tiered approach and the particular separation made between the two components of a specification. This thesis lays out a basis for this approach by formally defining a two-tiered specification language (Chapters 2, 3, and 4), and describes ways to evaluate two-tiered specifications (Chapter 5). In Section 1.5.1 we discuss our approach to defining the language formally, and in Section 1.5.2 we give a guide to the rest of this thesis.

1.5.1 Approach to the Formalization

This thesis deals with specifications, i.e., strings of symbols. A string of symbols may be viewed in two ways: as a sentence of a language, or as the meaning of that sentence. Logicians sometimes call the first point of view "syntactic" and the second point of view "semantic." From the syntactic viewpoint, a precise description of sentences is given by defining a *formal system*: a set of symbols, a set of well-formed formulae, a set of axioms, and a set of rules of inference. A *theory* associated with a formal system is the set of well-formed

formulae derivable from the axioms and rules. From the semantic viewpoint, a precise description of sentences is given by defining a *model* for the language. A model consists of a universe of mathematical entities such as sets and functions, and a mapping (sometimes called an *interpretation*) from sentences in the language to the mathematical entities. These mathematical entities are called *meanings* of the sentences.

The syntactic and semantic views are related. A sentence, σ, in a language, L, is *valid* if it is *true* in every model for L. We write "$M \models \sigma$" to denote that the sentence σ is true in the model M (or equivalently, "σ holds in M," "M satisfies σ," and "M is a model of σ"). M is a model for a set of sentences, Σ, if it is a model for each $\sigma \in \Sigma$. Since a theory is a set of sentences in a language, it also makes sense to talk about a *model of a theory.*

In this thesis, we concentrate on describing specifications and implementations from a syntactic viewpoint because we can treat them as concrete objects, i.e., text written down on a piece of paper, as opposed to abstract mathematical entities. Furthermore, we define a *satisfies* relation between an implementation and a specification in terms of their theories. Chapter 3 contains the definitions of *satisfies* and the formal systems associated with specifications and implementations.

It is important to establish the soundness of these formal systems. Informally, a formal system, F, is sound if no invalid formula is deducible from the axioms and rules of inference of F. That is, any theorem in the theory, T, specified by F is valid in all models of T. Formally, F is sound if all the axioms of the formal system are valid and the rules of inference are sound. A rule is sound if the validity of each of its hypotheses implies the validity of the conclusion.

Therefore, to show the soundness of the formal systems we will define, it is necessary to define (1) the classes of models of the theories of the formal systems and (2) the validity relation ($\models$) between models and theories. Chapter 2 contains the definitions of these classes of models, which are the same for specifications as for implementations, and the

definition of the validity relation for specifications. Although we lay out the foundations to be able to prove the soundness of the formal systems we describe, it is outside the scope of this thesis to present the proof.

We choose to present the semantic viewpoint first (Chapter 2) and the syntactic one later (Chapter 3) because we believe that it is easier to understand the meanings of specifications and implementations in terms of familiar mathematical entities such as sets, functions, and relations, rather than in terms of strings of symbols and rules that manipulate them. We hope that it is easier for the reader to compare whether his intuition matches ours, i.e., whether the models we define reflect the same intuitive concepts he has about the meaning of a program and its behavior.

1.5.2 A Guide to the Rest of the Thesis

In Chapters 2 and 4, we view specifications semantically. We give meanings to specifications in terms of mathematical entities that include, among other things, algebras and relations. In Chapter 2, we define a kernel interface language, and in Chapter 4, we define extensions to the kernel. The kernel language is defined to serve as a basis for other interface languages and also to reduce the number of linguistic constructs to consider when viewing specifications syntactically. The extensions in Chapter 4 are syntactic amenities to the kernel and additional constructs to handle particular features in CLU, e.g., iterators.

In Chapters 3 and 5, we view specifications syntactically. The formal systems associated with specifications are defined by using the axiomatic semantics of CLU, which associates proof rules with individual CLU statements and expressions, and the semantics of Larch. In Chapter 3, we define the theory denoted by a specification written in the kernel interface language. In Chapter 5, we describe evaluation properties of specifications in terms of these theories.

Chapters 2 and 3 can be read together for a formal description, in terms of both models and theories, of the kernel interface language. Chapters 2 and 4 can be read together for a description of the entire interface language for CLU. Chapters 3 and 5 can be read together for an idea of the benefits gained from treating the meanings of specifications as pure text.

Finally, in Chapter 6 we summarize our conclusions and main contributions of this research, and discuss directions for future work.

2. Kernel Interface Language

This chapter defines a kernel language that can be used to write specifications of CLU programs consisting of procedures and clusters. A procedure specification specifies the set of procedures that implement it; a cluster specification specifies the set of clusters that implement it.

We would like the kernel language to have the following properties:

1. Rich enough to allow us to specify any operation or type one might want to implement in CLU.

2. A small number of constructs. In Chapter 4, in order to make reading and writing specifications easier, we introduce some syntactic sugar and add other constructs to the kernel. The additions will be defined by translating them into constructs of the kernel language.

3. A syntax that maps easily into the well-formed formulae of the theory that a specification denotes. This is to simplify the formal definitions presented in Chapters 3 and 5.

A goal for the entire interface language, not just the kernel, is that it be adaptable to programming languages other than CLU. The particular concrete syntax presented, not surprisingly, borrows heavily from CLU, but the abstract syntax of the interface language can serve as a basis for an interface language for other programming languages.

Section 2.1 presents the classes of models for theories associated with specifications and implementations. Section 2.2 presents the (kernel) interface language. The two main objectives of Section 2.2 are (1) to define the validity relation ($\models$) between a model and a specification, and (2) to present the precise syntax and (model-oriented) semantics of procedure and cluster specifications. The presentation is bottom-up. Assertions constitute the body of a procedure specification, and procedure specifications constitute the body of a cluster specification. Hence, we start by defining an assertion language based on Larch, then

procedure specifications, then special assertions that are additions to the assertion language particular to CLU, and finally, cluster specifications. We warn the reader that we sometimes digress from our two main objectives of Section 2.2 in order to present some necessary detail for the sake of precision.

2.1 Classes of Models

A theory defines a class of models. In this section, we are interested in describing the classes of models for the theories of specifications and implementations. To do so we use the basic mathematical entities of values, functions, and relations to define the notions of objects, states, operations, and abstract data types.

Let us first motivate the kinds of models we will introduce to model the computation of a CLU program. The execution of a program begins with the invocation of some operation in some initial state. The execution of the operation and of subsequent operations invoked in a computation can change the state. We thus need to characterize carefully what information is in a state and what possible changes to a state may arise because of the execution of an operation. An operation can change a state by creating new objects and changing the values of existing ones. Each CLU object can be accessed only through certain operations, depending on the abstract data type it belongs to.

We present our classes of models in a bottom-up fashion: we start off by describing values, then objects, states, operations, abstract data types, and finally, computations. In Section 2.1.1, we define when an *algebra* is a model of a trait theory. In Sections 2.1.2 and 2.1.3, we discuss the domains of *objects* and *states*, which underlie the models of procedures and clusters. In Sections 2.1.4 and 2.1.5, we define the classes of models for procedures and clusters, respectively. We call these models *operations* and *abstract data types*. The classes of models for specifications are the same as for their implementations. The chart in Figure 5 summarizes the syntactic and semantic domains we will be dealing with. Finally, in Section

2.1.6 we define our model of computation.

Syntactic Conventions

For an n-tuple, $x = \langle v_1, ..., v_n \rangle$, we write $x.v_i$ for the ith component of x. For a function of one argument, f, we write *dom*(f) for the domain of f and *ran*(f) for its range.

2.1.1 Traits and Algebras

A trait defines a set of equations, propositional formulae, and first-order quantified formulae that makes up the trait's first-order theory with equality. The class of models of the theory of a trait is a set of many-sorted algebras. We use the usual definition of satisfaction between an algebra and a first-order theory that has equality [Birkhoff70, Enderton72]. We define an algebra to be a model of a trait Tr if it satisfies the theory of Tr.

A many-sorted algebra is a pair consisting of a set of values, *Val*, partitioned according to their sorts, and a set of total functions, *Fun*, over these values. We use the set of terms, *Term*, to denote values in *Val*. Terms are of the form "x" where x is in the set of (sorted) variable identifiers, *VarId*, or of the form "f(t1, ..., tn)" where f denotes a function in *Fun*, and t1, ..., tn are terms. Let *SortId* be an infinite set of sort identifiers (not associated with any

---

Syntax (text)	***Semantics(models)***
Specifications	
Trait	Algebra = <values, functions>
Procedure specification	Operation = <relation, algebra>
Cluster specification	Abstract Data Type = <objects, operations>
Implementations	
Procedure	Operation
Cluster	Abstract Data Type

Figure 5. Syntax and Semantics

particular algebra). Henceforth, when we say "algebra," we mean a many-sorted algebra.

2.1.2 Objects

Let *Obj* be an infinite set of objects partitioned into subsets according to their *types*. Each object has exactly one type, which cannot be changed. We call *Obj* the *universe*; it is the set of all potentially existing objects. A state (defined below) defines a value for each object. When an object's value changes, we say the object is "mutated." Let *TypeId* be an infinite set of type identifiers (not associated with any particular universe), and let TtoS be a many-to-one function that maps type identifiers to sort identifiers. For an object, x, of type T, the sort of the value of x is TtoS(T).

In CLU, an object, A, can be the value of another object, B, in which case we say "A contains B." Sharing of objects arises when two or more objects contain the same object. Because of sharing of mutable objects, it is not sufficient that the value of a containing object refer to the value of the contained object; it must refer to the contained object itself, i.e., its identity.

In order to treat a contained object as part of the value of the containing object, we treat objects as special kinds of values. We always include implicitly in every trait a trait defining this infinite set of objects. Therefore, any model (i.e., an algebra, A = $\langle Val, Fun \rangle$) of the theory of a trait will have the property that $Obj \subseteq Val$. Treating objects as values raises a sticky technical issue: what is the sort of a term that denotes an object? We answer this question in Section 2.2.1 where we carefully define how to sort check terms.

2.1.3 State

Objects can be created and manipulated in the course of program execution. We model the state of a program at an instant in time by a *state*. We model CLU states as follows, where *P*(*Obj*) is the powerset of the set *Obj*.

$State = P(Obj) \times Env \times Store$
$Env = ObjId \rightarrow Obj$
$Store = Obj \rightarrow Val$

Def: A *state*, $\sigma = \langle O, e, s \rangle$, is a triple consisting of a finite set of existing objects, O, which is a proper subset of *Obj*; an *environment*, e, which is a mapping from *ObjId* to O; and a *store*, s, which is a mapping from O to *Val*.

We call *Val*, the *value set* of σ. The identifiers in *ObjId* are CLU program variables, which always range over objects. Whenever we refer to "an object in σ" we mean an object in $\sigma.O$.

We use $\Sigma(Val)$ to denote the set of states with *Val* as their value set. That is, $\Sigma(Val) = \{\langle O, e, s \rangle \mid s: O \rightarrow Val\}$. We do this to avoid having four components in a state. A particular state, σ, is an element of some set of states, $\Sigma(Val)$, and thus each state is always associated with some fixed set of values.

A state can change over time in three ways: the set of existing objects grows because new objects are added from the universe; the environment changes because the mapping from CLU program variables (i.e., object identifiers) to objects changes; or the store changes, because the values of existing objects change.

2.1.4 Procedures and Operations

We model a procedure as an operation, where an *operation* is a pair, $\langle R, A \rangle$, consisting of a relation and an algebra. We refer to the relation of an operation modeling a procedure as the *input-output behavior* of the procedure. A relation, R, is a set of pairs of states:

$R \subseteq \Sigma(Val) \times \Sigma(Val)$ where $A = \langle Val, Fun \rangle$

We call the first component of a pair in the relation the *input state*; the second, the *output state*. Let *dom*(R) be the set of input states of R; *ran*(R) be the set of output states of R. The relation viewed as a set of pairs of states is more general than we need. In particular, we can and should be specific about the arguments passed to and from a procedure.

Def: The object identifiers in a procedure heading are *input formals* of the procedure. The objects the formals denote are *input arguments* of the procedure. The objects returned by a procedure are *output arguments*.

A relation, R, which is a component of an operation, has the following properties:

1. $dom(R) = \{\langle O, e, s\rangle \mid dom(e) = \text{set of input formals} \wedge ran(e) = \text{set of input arguments}\}$
2. $ran(R) = \{\langle O, e, s\rangle \mid ran(e) = \text{set of output arguments}\}$

where *dom*(e) is the domain of the environment e, and *ran*(e) is the range. The first property states that the environment of all input states is the set of bindings from input formals (object identifiers) of a procedure to the arguments passed to it. The second property states that the range of the environment of all output states is the set of output arguments. (CLU procedures do not list identifiers for output arguments. Since our specifications do, we will strengthen the second property when we define a model of a procedure specification.)

The algebra A of a model of a procedure provides the set of values, *Val*, over which objects manipulated by the procedure can range. *Val* is the same set as the value set of each state of the pairs in the relation.

Procedures can terminate in more than one way. Let *TermCond* be a set of special values called *termination conditions*, and let **terminates** be a special object in the state that can take on a value from *TermCond*. For simplicity, we henceforth view that included implicitly in all traits is the trait defining the values in *TermCond* and that $\mathbf{terminates} \in O$ for all states $\langle O, e, s\rangle$. We reserve the special value **normal** for the normal termination condition. A procedure may also never terminate. For a given input state, if the set of output states is non-empty, then the procedure must terminate for that input state.[1]

---

1. In CLU, a procedure may also terminate because of an unhandled exception thereby signaling **failure**. We view this situation as a programmer error and we choose not to provide the ability to specify such procedures. Hence, a procedure that signals **failure** satisfies no specification.

2.1.5 Clusters and Abstract Data Types

We model a cluster as an abstract data type, where an *abstract data type* is a pair, T = <Obs, Ops>, consisting of a set of objects and a set of operations. The set of objects, Obs, is the subset of the objects of *Obj* whose elements are of type T. An operation in Ops is a pair consisting of a relation and an algebra, as previously defined. We require that all the operations of the type have the same algebra.

2.1.6 Computations

We model a computation as an alternating sequence of states and statements starting in some *initial state*, σ_0. Each statement, S, of a computation sequence is a partial function on states:

$$S: \Sigma(Val) \rightarrow \Sigma(Val)$$

For the states, σ_i, and the statements, S_i, $1 \leq i \leq n$, let a computation sequence be:

$$\sigma_0\, S_1\, \sigma_1, \ldots, \sigma_{n-1}\, S_n\, \sigma_n$$

and for all $1 \leq i \leq n$ $\langle \sigma_{i-1}, \sigma_i \rangle \in S_i$. We refer to the states $\sigma_0, \ldots, \sigma_n$ above as "states of a computation sequence." We could also view a computation sequence as a sequence of states, and dispense with references to individual statements. However, in defining computational induction, which we do in Chapter 3, we need to be able to refer to the statements that cause the changes to states.

We are interested in only two kinds of CLU statements: assignment and procedure invocation. All other statements can be defined in terms of these two. In CLU, a simple assignment statement can change the environment of a state by changing the mapping from an object identifier to an object. A procedure invocation can change the set of existing objects of a state by adding new objects to it, and it can change the store of a state by

changing the values of objects. All objects returned from a procedure as a result of a procedure invocation can be assigned to object identifiers in an assignment statement. So, when assignment is combined with procedure invocation, an assignment statement, in general, can change all components of a state.

Properties of Computations

1. *Successive states*: A property that holds between two successive states of all computation sequences is:

$$\forall 1 \leq i \leq n \; \sigma_{i-1}.O \subseteq \sigma_i.O.$$

This property states that new objects can possibly be added to, but not removed from, a state as a result of a procedure invocation.

2. *Procedure invocation*: For all $1 \leq i \leq n$, if S_i is or contains the invocation of a procedure, Pr, the following two properties hold. Let Op = ⟨R, A⟩ be the operation modeling Pr. For all ⟨in, out⟩ pairs of states in R (recall that the range of an environment is a set of objects):

2.1. *ran*(in.e) ∪ {Pr} ⊆ $\sigma_{i-1}.O$
2.2. *ran*(out.e) ⊆ $\sigma_i.O$

The first property states that all input arguments and the procedure Pr are in the set of existing objects of the state before the invocation of Pr. Pr is included because a procedure is also an object in CLU and must exist before it is invoked. The second property states that all output arguments are in the set of existing objects upon the termination of Pr.

We summarize the models we have described in Section 2.1 in Figure 6.

Syntax	*Semantics*
Trait	A model of a trait is a (many-sorted) algebra, where for an algebra A = ⟨*Val*, *Fun*⟩, *Val* is a set of values and *Fun* is a set of functions.
Procedure	A model of a procedure is an operation, where for an operation Op = ⟨R, A⟩, R is an input-output relation on pairs of *states* (see below), and A is an algebra.
Cluster	A model of a cluster is an abstract data type, where for a type T = ⟨Obs, Ops⟩, Obs is a set of objects (of type T), and Ops is a set of operations.

Some Syntactic Domains

SortId	set of sort identifiers
TypeId	set of type identifiers
ObjId	set of object identifiers

Some Semantic Domains

State = $P(Obj) \times Env \times Store$

$\Sigma(Val)$	set of states over value domain, *Val*.
Obj	set of all potentially existing objects
TermCond	set of termination conditions

Facts

For all *states*, $\sigma = \langle O, e, s \rangle$, where $\sigma \in \Sigma(Val)$,

$O \subset Obj$	set of existing objects
$e: ObjId \rightarrow O$	an environment
$s: O \rightarrow Val$	a store

$TermCond \subseteq Val$
terminates$\in O$
normal$\in TermCond$

Figure 6. Summary of Models, Syntactic and Semantic Domains

2.2 Kernel Interface Language and Models

We now turn to describing in detail the interface language. We have already defined the underlying models for traits, described the domains of objects and states, and described the underlying models for procedures and clusters. What remains is to present the syntax of the kernel language and to define the validity relationship ($\models$), which we do in Section 2.2.2 for procedure specifications and in section 2.2.3 for cluster specifications.

Syntactic Conventions

We use extended BNF to define the syntax of our language with the following syntactic conventions:

|	alternative separator
a+	one or more a's
a+,	one or more a's separated by commas
<a>	an optional a

Nonterminals are italicized. Terminal symbols include parentheses, square brackets, curly braces, and boldface items. Comments in specifications begin with "%" and end with a newline.

In the next three sections, 2.2.1 through 2.2.3, we describe the interface assertion language, procedure specifications, and cluster specifications. Section 2.2.1 contains the basis of the assertion language for writing the bodies of procedure specifications. Section 2.2.2 on procedure specifications is further broken down into five subsections describing various parts of the interface language that are germane to procedures. It introduces special assertions that are additions to the base assertion language described in Section 2.2.1. In Sections 2.2.2 and 2.2.3, for each part of the interface language we will present four sections: its syntax, its syntactic checks, its meaning, and an example. Some of the syntactic checks that we require would be unnecessary if we added more complexity to the grammar that we present. We choose not to put the complexity in the grammar in order to simplify our

description of the meanings of the various parts of the language.

2.2.1 Interface Assertion Language

In this section we describe the language we use to make assertions about objects and their values in a state. These assertions appear in the bodies of specifications and can refer to both initial and final values of objects. After presenting the syntax of interface assertions, we present a lengthy section on the syntax checking of assertions. It is long because we discuss in depth the issue of sort checking a term that refers to an object. Finally, we present the meaning of an interface assertion by giving a truth value function. Since an assertion can refer to the initial and final value of an object, the truth function is defined with respect to two states, corresponding to the input and output states of an input-output relation.

Syntax

Assn :: = true | false | ~*Assn* | *Assn Connective Assn* | (*Assn*)
| *Quantifier VarId*: *SortId Assn*
| *Term* = *Term*
Term :: = *VarId* | *ObjId* | *OpId*<(*Term* + ,)> | *Term*↑ | *Term*↓
Connective :: = ∧ | ∨ | ⇒ | ⇔
Quantifier :: = ∀ | ∃

We allow parentheses to be omitted by relying on the following conventions:

1. Outermost parentheses may be dropped.
E.g., "A ∧ B" is "(A ∧ B)."

2. The precedence of the operators and quantifiers from highest to lowest is ~, ∀, ∃, ∧, ∨, ⇒, ⇔.
E.g., "∀x A ⇒ B" is (∀x A ⇒ B), and not "∀x (A ⇒ B); "~A ∧ B ⇒ C" is "((~A) ∧ B) ⇒ C."

3. When one connective is used repeatedly, the expression is grouped to the right.
E.g., "A ⇒ B ⇒ C" is "A ⇒ (B ⇒ C)."

We allow the use of other delimiters, such as square brackets, for parentheses. An assertion of the form τ = true is abbreviated to τ; τ = false, ~τ, where τ is in *Term*.

Assertions in specifications can refer to both the initial and final values of objects. We use x↑ to denote the initial value and x↓ to denote the final value of object x. The interpretation of these terms will be defined rigorously in the *Meaning* section.

In order to define precisely how to sort check an assertion we need to define the subterms of an assertion or term:

Def: The *subterms* of an assertion, α, in *Assn* are defined as follows:

1. α is a subterm of itself.
2. If α is of the form *t1 = t2*, the subterms of both *t1* and *t2* are subterms of α.
3. If α is of the form *~a*, the subterms of *a* are subterms of α.
4. If α is of the form *a1 # a2*, where *#* is in *Connective*, the subterms of both *a1* and *a2* are subterms of α.
5. If α is of the form *(a)*, the subterms of *a* are subterms of α.
6. If α is of the form ∀v:S *a* or ∃v:S *a*, the subterms of *a* are subterms of α.

Def: The *subterms* of a term, τ, in *Term* are defined inductively as follows:

1. τ is a subterm of itself.
2. If τ is of the form *(f(t1, ..., tn))*, where *f* is in *OpId* and *t1, ..., tn* are in *Term*, the subterms of *t1, ..., tn* are subterms of τ.
3. If τ is of the form *t↑* or *t↓*, the subterms of *t* are subterms of τ.

Checking

We check that all assertions sort check, where all trivial subterms, i.e., terms that are in either *VarId* or *ObjId*, sort check. The second definition below relies on understanding the discussion, *Sorts for Objects and Values*; we present it here to keep the definitions involving the syntax checking of an assertion together.

Def: An assertion, α, *sort checks*:

1. If α is of the form *t1 = t2*, the sorts of both *t1* and *t2* are the same.
2. All subterms of α sort check.

Def: A term, τ, *sort checks* if and only if:

1. All subterms of τ sort check.
2. If τ is of the form *g(s1, ..., sm)*, where *g* is in *OpId* and *s1, ..., sm* are in *Term*, the domain of *g* must be a sequence of the sorts of the *m* terms in *s1, ..., sm* where
   a. The sort of a term of the form *f(t1, ..., tn)*, is the range of *f*, where *f* is in *OpId* and *t1, ...,tn* are in *Term*,
   b. The sort of a term of the form *v* is *S*, where *v* is in *VarId* and is bound in an assertion of the form ∀v:S *a* or ∃v:S *a*, for *a* in *Assn*,
   c. The sort of a term of the form *o* is the sort T_obj where *o* is in *ObjId* and T

is the type of the object denoted by *o*, and

d. The sort of a term of the form $t\uparrow$ or $t\downarrow$ is the sort TtoS(T) where *t* is in *Term* and T is the type of the object denoted by *t*.

3. If τ is of the form $t\uparrow$ or $t\downarrow$, *t* must denote an object, where *t* is in *Term*.

Sorts for Objects and Values

We now address the sticky technical issue raised earlier in Section 2.1.2 where we discussed objects: if an object is a value, what is the sort of a term denoting such a value? Before we answer this, let us look at an example. Let the value of some array (of sets) object be denoted by the term, *addh(addh(create(1),s1),s2)*, where the signatures of *addh* and *create* are (*addh* and *create* are trait function identifiers):

create: Int → A
addh: A, ? → A

What sort is "?"? The object identifiers *s1* and *s2* denote objects since the value of an array object refers to the set objects the array contains, not just the values of the set objects.

We introduce a special subset of *SortId* called *ObjSortId*. For each different type in the set, *Obj*, there is a sort identifier in *ObjSortId*. Each sort identifier in *ObjSortId* is called an *obj sort*; each in *SortId* is called a *value sort*. (Just as an object is a special kind of value, an obj sort is a special kind of value sort.) So, in our array example, *s1* and *s2* are of some obj sort.

Therefore, an object has two sorts associated with it: its obj sort and its value sort. The sort of a term denoting the value of an object is a value sort--it can be an obj sort since objects can contain other objects. The sort of a term denoting the object itself must be an obj sort. There is a one-to-one correspondence between the type of an object and its obj sort. We use the naming convention that T_obj is the name of the obj sort for objects of type T. In our array value example, *s1* and *s2* are of the obj sort, *set_obj*. There is a one-to-one correspondence between the type of an object and the sort of a term denoting its value. The function, TtoS, gives us this mapping from type names to (value) sort names. (TtoS can be many-to-one

because more than one type can be defined with respect to the same sort.) In our array example, the term *addh(addh(create(1),s1),s2)* is of (value) sort, *A*.

We emphasize that the reason we introduce an obj sort of the form "T_obj" instead of simply using the type identifier "T" is to keep the set of sort identifiers disjoint from the set of type identifiers. We do this to be consistent with the facts that the set of values, *Val*, is partitioned by sorts and the set of objects, *Obj*, is partitioned by types. We also emphasize that the only reason we need to introduce obj sorts for objects is that objects are treated as values (because of sharing and mutability); for sort checking to work, we need to be able to refer sensibly to "the sort of an object," or more precisely, "the sort of a term denoting an object."

Def: A term *denotes an object* if and only if the sort of the term is some obj sort.

Figure 7 summarizes the various sets of identifiers for objects, values, obj sorts, value sorts, and types; some facts relating these sets; and some questions that are reasonable to ask of objects and values, and their answers.

Returning to the array example, the signature of the addh function is:

addh: A, set_obj → A

Suppose we also have a fetch function for arrays with the following signature:

fetch: A, Int → set_obj

with TtoS defined as follows:

TtoS(array[set]) = A
TtoS(set) = S
TtoS(integer) = Int

Syntactic Domains

VarId	variable identifiers denoting values, some of which may be objects
ObjId	object identifiers denoting objects, which are special kinds of values
SortId	value sort identifiers
ObjSortId	obj sort identifiers, each of the form T_obj, for type identifier T
TypeId	type identifiers

Facts

$VarId \cap ObjId = \emptyset$
$SortId \cap TypeId = \emptyset$
$ObjSortId \subseteq SortId$
$|TypeId| = |ObjSortId|$, where "|X|" is the cardinality of set X.
$\exists$ bijection: $TypeId \leftrightarrow ObjSortId$
$\forall T \in TypeId \; \exists S \in SortId$ TtoS(T) = S

Questions	***Answers***
For an object, x, of type T:	
What is the *type* of x?	T
What is the *value* of x in a state, $\sigma = \langle O, e, s \rangle$?	σ.s(x).
What is the *obj sort* of *object* x?	T_obj
What is the *value sort* of the *value* of x?	TtoS(T)

Figure 7. Sorts and Types, Objects and Values

For an array[set] object, a, let a↑ be the value of a, and for an integer object, i, let i↑ be the value of i:

The type of a is array[set].
The obj sort of a is array[set]_obj.
The (value) sort of the value of a is A.

The type of the object denoted by fetch(a↑,i↑) is set.
The obj sort of fetch(a↑,i↑) is set_obj.
The (value) sort of fetch(a↑,i↑)↑ is S.

Suppose instead that addh and fetch were declared as:

addh: A, S → A
fetch: A, Int → S

In this case, it would not make sense to ask for the type of fetch(a↑,i↑) since fetch(a↑,i↑) does not denote an object. It does make sense to ask for the sort of fetch(a↑,i↑); the sort is S.

An Important Shorthand

It is important to realize that we can quantify over objects because we are treating objects as values. It makes sense to write an assertion ∀x:T_obj α or ∃x:T_obj α, where x ranges over objects of type T and α is in *Assn*. In our examples, we abbreviate these to the forms ∀x:T α and ∃x:T α.

Meaning

Assertions are well-formed formulae in first-order predicate calculus with equality, where equality is denoted by the symbol, =. We will define the truth of an assertion with respect to two states, an algebra, and a variable-to-value mapping. Before we define the truth function, ***T***, we explain why we need these various pieces of information.

As mentioned in the beginning of Section 2.2.1, we need to interpret interface assertions with respect to two states because assertions in specifications can refer to both the initial and final values of objects. The two states correspond to the input state and the output state in a relation of an operation.

A model of a procedure specification is an operation that includes the same algebra used to interpret an interface assertion. The algebra provides a set of values, *Val*, and a set of functions, *Fun*, to which we refer below.

Finally, in order to handle the free variables in an assertion, we include a variable-to-value mapping. This is a standard "trick" used to keep track of the variable identifiers that are introduced in quantified assertions. (The following definition is adapted from [deBakker80].)

Def: Let *VarMap* be the set of functions, μ: *VarId* $\rightarrow$ *Val* (the same *Val* as for the algebra discussed above). For all $\mu \in$ *VarMap*, v$\in$*VarId*, x$\in$*Val*, we write "μ[x/v]" (read "substitute x for v in μ") for the element of *VarMap* that satisfies, for each y$\in$*VarId*:

1. $\mu[x/v](y) = x$, if $y = v$
2. $\mu[x/v](y) = \mu(y)$, if $y \neq v$

We are now ready to give the truth function, ***T***.

$$T: Assn \times \Sigma(Val) \times \Sigma(Val) \times Alg \times VarMap \rightarrow \{TRUE, FALSE\}.$$

We write "T[P](σ, σ', A, μ)" for the truth of an assertion P in states, σ, σ'; algebra, A; and variable-to-value mapping, μ. The states σ and σ' are elements of Σ(*Val*), where *Val* is the same set *Val* as for the algebra A. For all a, a1, a2 $\in$ *Assn*, and t1, t2 $\in$ *Term*,

T[true](σ, σ', A, μ) = *TRUE*
T[false](σ, σ', A, μ) = *FALSE*
T[~a](σ, σ', A, μ) = ~T[a](σ, σ', A, μ)
T[a1 # a2](σ, σ', A, μ) = T[a1](σ, σ', A, μ) # T[a2](σ, σ', A, μ),
where # is in *Connective*.
T[(a)](σ, σ', A, μ) = T[a](σ, σ', A, μ)
T[$\forall$v:S a](σ, σ', A, μ) = $\forall$x:S T[a](σ, σ', A, μ[x/v]),
where x is of sort S and does not appear free in a.
T[$\exists$v:S a](σ, σ', A, μ) = $\exists$x:S T[a](σ, σ', A, μ[x/v]),
where x is of sort S and does not appear free in a.
T[t1 = t2](σ, σ', A, μ) = *TRUE*, if V[t1](σ, σ', A, μ) = V[t2](σ, σ', A, μ);
FALSE, otherwise;
where " = " between values is the equality relation on values in algebra, A.

The value of a term is defined by the following function,

$$\mathcal{V}: Term \times \Sigma(Val) \times \Sigma(Val) \times Alg \times VarMap \rightarrow Val.$$

For all $y \in VarId$, $x \in ObjId$, $f \in OpId$, and t, t1, ..., tn $\in Term$,

$\mathcal{V}[y](\sigma, \sigma', A, \mu) = \mu(y)$
$\mathcal{V}[x](\sigma, \sigma', A, \mu) =$ x, where x is neither an input nor output formal
$\mathcal{V}[x](\sigma, \sigma', A, \mu) = \sigma.e(x)$, where x is an input formal
$\mathcal{V}[x](\sigma, \sigma', A, \mu) = \sigma'.e(x)$, where x is an output formal
$\mathcal{V}[f(t1, ..., tn)](\sigma, \sigma', A, \mu) = f!(\mathcal{V}[t1](\sigma, \sigma', A, \mu), ..., \mathcal{V}[tn](\sigma, \sigma', A, \mu))$
where f! is the function $\in A.Fun$ denoted by f.
$\mathcal{V}[t\uparrow](\sigma, \sigma', A, \mu) = \sigma.s(\mathcal{V}[t](\sigma, \sigma', A, \mu))$
$\mathcal{V}[t\downarrow](\sigma, \sigma', A, \mu) = \sigma'.s(\mathcal{V}[t](\sigma, \sigma', A, \mu))$

Example

As an example, let us apply the value function, $\mathcal{V}$, to the term, fetch(a↑,i↑), where a and i are input formals of a procedure specification.

$\mathcal{V}[fetch(a\uparrow,i\uparrow)](\sigma, \sigma', A, \mu)$
$= fetch!(\mathcal{V}[a\uparrow](\sigma, \sigma', A, \mu), \mathcal{V}[i\uparrow](\sigma, \sigma', A, \mu))$
$= fetch!(\sigma.s(\mathcal{V}[a](\sigma, \sigma', A, \mu)), \sigma.s(\mathcal{V}[i](\sigma, \sigma', A, \mu)))$
$= fetch!(\sigma.s(\sigma.e(a)), \sigma.s(\sigma.e(i)))$

Here, fetch! is a function in A.*Fun*; $\sigma.s(\sigma.e(a))$ and $\sigma.s(\sigma.e(i))$ are values in A.*Val*.

2.2.2 Procedure Specifications

A procedure specification specifies a subset of the set of all the possible operations that are models of procedures. In this section, we define when an operation is a model of a procedure specification.

In the next five subsections we will describe the language and the validity relation for procedure specifications. First we consider procedure specifications ignoring exceptional termination; second, we consider those with exceptional termination. In the subsequent three sections, we describe special assertions to handle the creation of new objects, the mutation

of existing objects, and procedure objects.

2.2.2.1 Procedure Specifications Without Signals

A procedure specification includes a name, a heading, a link, and a body. The heading specifies the types of the input and output arguments. The link identifies the name of the trait that defines an algebra that provides the values over which the input and output arguments can range. The body is a pair of assertions that specify conditions relating the initial and final values of the input and output arguments.

Syntax

ProcSpec ::= *ProcId* = *ProcHead Link ProcBody* **end**
ProcHead ::= **proc** *Args* <*Rets*>
Link ::= **uses** *TraitId*
ProcBody ::= *PreC PostC*
PreC ::= **pre** *Assn*
PostC ::= **post** *Assn*

Args ::= (<*Decl* + ,>)
Rets ::= **returns** (*Decl* + ,)
Decl ::= *ObjId* + ,: *TypeSpec*
TypeSpec ::= *TypeId*

Some definitions:

Def: The object identifiers in a procedure heading are *formals* of the procedure specification. The objects the formals denote are *arguments*.

Def: Object identifiers in an *Args* are called *input formals*, and their objects, *input arguments*; object identifiers in a *Rets* are called *output formals* and their objects, *output arguments*.

Def: The trait named in a procedure specification, pr, is called the *used trait* of pr.

Checking

For a procedure specification to be syntactically well-formed, we check that:

1. Each object identifier appearing in a pre-condition or post-condition appears in the list of formals. The sets of input formals and output formals are disjoint.

2. The assertions appearing in the pre- and post-conditions sort check according to the function declarations of the used trait.

3. Output formals appear only in a post-condition.

4. Terms of the form $\tau\downarrow$, where $\tau \in Term$, appear only in the post-condition.

The header of a procedure specification is the same as that for a CLU procedure except that identifiers are introduced in the **returns** clause for output arguments.

Meaning

Informally, the pre-condition of a procedure specification defines a subset of the universe of states over which the procedure must terminate. The procedure specification does not say anything about those states which do not satisfy the pre-condition. The post-condition defines for any valid initial state the final states that are acceptable.

Formally, a model of a procedure specification, Pr, is an operation. An operation is a pair, <R, A>, where R is a relation on pairs of states, and A, is an algebra. Each relation, R, of an operation has the following properties (compare with Section 2.1.4):

1. $dom(R) = \{\langle O, e, s\rangle \mid dom(e) = \text{set of input formals} \wedge ran(e) = \text{set of input arguments}\}$
2. $ran(R) = \{\langle O, e, s\rangle \mid dom(e) = \text{set of output formals} \wedge ran(e) = \text{set of output arguments}\}$

The first property states that the environment of all input states is the set of bindings from input formals (object identifiers) of a procedure specification to input arguments (objects). The second property states that the range of the environment of all output states is the set of

bindings from output formals (object identifiers) to output arguments (objects).

We now define when an operation is a model of a procedure specification, Pr. Let Pr have a pre-condition P, post-condition Q, and used trait Tr.

Def: For an operation, Op = ⟨R, A⟩, *Op is a model of Pr*, i.e., Op ⊨ Pr, if and only if:
1. A is a model of Tr, and
2. ⟨R, A⟩ ⊨ ⟨P, Q⟩ (defined below).

Def: Let A = ⟨*Val*, *Fun*⟩. ⟨R, A⟩ ⊨ ⟨P, Q⟩ if and only if:

$$\forall \mu{:}\mathit{VarId} \rightarrow \mathit{Val}$$
$$\forall \sigma\ T[P](\sigma, \rho, A, \mu) \Rightarrow [\exists \sigma'\ \langle \sigma, \sigma' \rangle \in R \wedge \forall \sigma' [\langle \sigma, \sigma' \rangle \in R \Rightarrow T[Q](\sigma, \sigma', A, \mu)]]$$

This says that for all variable-to-value mappings (needed to handle free variables that appear in assertions), for all states in which the pre-condition is satisfied, there exists some output state in the relation (this gives us termination) and for all such output states (reached from an input state in which the pre-condition is satisfied), the post-condition is satisfied. In the above predicate, we define ρ to be some constant state (e.g., the null state) because although all assertions are interpreted with respect to two states, it makes sense to refer to only initial values of objects in a pre-condition. By the syntactic restrictions we place on what assertions may appear in pre-conditions, the evaluation of an assertion in a pre-condition can ignore the second state.

Example

```
choose = proc (s: set) returns (i: int)
  uses SetOfInt
       pre ~isEmpty(s↑)
       post has(s↑,i↓)
       end
```

This procedure specification specifies that the choose procedure takes in one input object of type *set* and returns one output object of type *int*. The pre-condition is satisfied only when the value of the input set object is not empty. The post-condition asserts that the value of the output integer object is in the value of the input set object. The function identifiers, *isEmpty*

and *has*, appear in the *SetOfE* trait, which is included in the *SetOfInt* trait (Appendix A).

2.2.2.2 Termination Conditions

A CLU procedure may terminate in more than one way, depending on the input state. We distinguish *exceptional termination* from *normal termination* by including in the procedure heading all possible exceptional termination conditions of the procedure and each of their associated returned objects.

Syntax

We add to the procedure specification heading a **signals** clause:

ProcHead :: = **proc** *Args* ⟨*Rets*⟩ ⟨*Sigs*⟩
Sigs :: = **signals** (*Exception* + ,)
Exception :: = *SigId* ⟨(*Decl* + ,)⟩

and to the assertion language:

Assn :: = ... | **returns** | **signals** *SigId*

As with a *Rets* clause, object identifiers in a *Sigs* clause are called *output formals* and their objects, *output arguments*.

Checking

We additionally check for a well-formed procedure specification that:

1. Each signal identifier appearing in some **signals** assertion in the post-condition appears in the heading.

2. **signals** and **returns** assertions appear only in the post-condition.

Meaning

Recall that a special **terminates** object is included as part of the set of existing objects of all states. Upon normal termination of the procedure, the value of **terminates** is equal to **normal**; upon exceptional termination, the value of **terminates** is equal to the *SigId* in some signals assertion. Formally, we extend the truth function, T, such that for all x∈*SigId*:

$$T[\textbf{returns}](\sigma, \sigma', A, \mu) = \sigma'.s(\textbf{terminates}) = \textbf{normal}$$
$$T[\textbf{signals } x](\sigma, \sigma', A, \mu) = \sigma'.s(\textbf{terminates}) = x$$

The set, *TermCond*, is the union of *SigId* and **{normal}**.

Example

```
choose = proc (s1: set) returns (i: int) signals (emptySet(s2: set))
  uses SetOfInt
       pre true
       post    [~isEmpty(s1↑) ⇒ has(s1↑,i↓) ∧ returns] ∧
               [isEmpty(s1↑) ⇒ signals emptySet ∧ s2 = s1]
       end
```

When *choose* terminates normally, **terminates↓ = normal** and returns an *int* object; when it terminates exceptionally, **terminates↓** = emptySet and returns a *set* object.

2.2.2.3 New Objects

Procedures can create new objects. When a new object is created, the set of existing objects, O, of the input state is extended by adding an element from the universe to O that was previously not in O.

Syntax

Assn :: = ... | **new** Ø | **new** *Term* + ,

Checking

A **new** assertion can appear only in a post-condition. Let *a* be an assertion of the form **new** *t1, ..., tn*, where *t1, ..., tn* are in *Term*. Subterms of *a* are the subterms of each term in the list *t1, ..., tn*. We check that for the assertion *a*:

1. Each subterm of each term listed in *t1, ..., tn* sort checks.
2. Each term listed in *t1, ..., tn* denotes an object.

Meaning

Recall that a state has three components, one of which is the set of existing objects, O. We extend the truth function, $\mathcal{T}$, such that for all terms t1, ..., tn in *Term*:

$$\mathcal{T}[\textbf{new } \emptyset](\sigma, \sigma', A, \mu) = \sigma.O = \sigma'.O.$$
$$\mathcal{T}[\textbf{new } t1, ..., tn](\sigma, \sigma', A, \mu) = (\sigma.O \cap \{t1, ..., tn\} = \emptyset) \wedge (\sigma'.O = \sigma.O \cup \{t1, ..., tn\}).$$

Example

```
create = proc () returns (s: set)
  uses SetOfInt
       pre true
       post s↓ = empty ∧ new s ∧ returns
       end
```

This procedure specification specifies that the *create* procedure when invoked returns a new, initially empty *set* object. The previous examples can be strengthened by adding a **new** ∅ assertion to their post-conditions.

2.2.2.4 Mutation

A procedure can mutate objects as well as return them. We add an assertion that specifies that no objects are allowed to be mutated and an assertion that specifies what objects a procedure is allowed to mutate.

Syntax

Assn ::= ... | **mutates** Ø | **mutates** *Term* + ,

Checking

A **mutates** assertion can appear only in a post-condition. Let *a* be an assertion of the form **mutates** *t1, ..., tn*, where *t1, ..., tn* are in *Term*. Subterms of *a* are the subterms of each term in the list *t1, ..., tn*. We check that for the assertion a:

1. Each subterm of each term in the list *t1, ..., tn* sort checks.

2. Each term in the list *t1, ..., tn* denotes an object.

Meaning

We extend the truth function *T* as follows:

$$T[\textbf{mutates } \emptyset](\sigma, \sigma', A, \mu) = T[\forall y{:}T_obj\ (y \in \sigma.O \Rightarrow y\downarrow = y\uparrow)](\sigma, \sigma', A, \mu)$$

$$T[\textbf{mutates } t1, ..., tn](\sigma, \sigma', A, \mu) = T[\forall y{:}T_obj\ ((y \in \sigma.O \wedge \sim(y = t1) \wedge ... \wedge \sim(y = tn)) \Rightarrow (y\downarrow = y\uparrow))](\sigma, \sigma', A, \mu)$$

Example

```
intersect = proc (s1, s2: set)
  uses SetOfInt
       pre true
       post ∀i:Int [has(s2↓,i) = has(s1↑,i) ∧ has(s2↑,i)]
            ∧ mutates s2 ∧ returns
       end
```

This procedure specification specifies that *intersect* may change only the value of the second input argument. Since *s1* and *s2* might denote the same input actual and *s2* might be mutated, we cannot guarantee that *s1* is not mutated; the final value of *s1* is not necessarily equal to its initial value. The previous examples can be strengthened by adding the **mutates** Ø assertion to the post-conditions.

2.2.2.5 Procedures as Objects

In CLU, procedures are also considered as objects that can be passed to or returned from procedures. For example, an input procedure argument, arg, to a procedure, pr, can be applied to other input arguments of pr.

Syntax

The type of a procedure object is given by its procedure heading. We add to the syntax of the interface language:

TypeSpec :: = ... | *ProcHead*

We add to the syntax of the assertion language:

Assn :: = ... | *Assn* {*Term*} *Assn*

We call this new kind of assertion a "procedure object assertion (poa)."[2]

Checking

Let *a* be a poa, P{τ}Q, where P and Q are assertions and τ is a term. Subterms of *a* are subterms of P, Q, and τ. We check that the procedure specification,

τ
pre P
post Q

is syntactically well-formed. We also check that the subterms of τ sort-check.

---

2. Poa's should not be confused with partial or total correctness assertions that deal with procedure invocations. Poa's deal with procedure objects.

Meaning

Recall that the meaning of a procedure object is a pair consisting of a relation and an algebra. The meaning of a poa, i.e., an assertion that refers to a procedure object is given in terms of the relation of the procedure object. We extend the truth function $\mathcal{T}$ as follows:

$$\mathcal{T}[P\{\tau\}Q](\sigma, \sigma', A, \mu) = \mathcal{V}[\tau](\sigma, \sigma', A, \mu) \models \langle P, Q \rangle$$

where $\models$ was defined in Section 2.2.2.1.

Example

Suppose we specify a procedure that copies the elements of an array using the *copyElem* procedure as an input argument. If we wish to place a restriction on the *copyElem* procedure object, we would write it in the pre-condition of *copyArray*. The *ArrayOfElemObj* trait, which uses the *Array* trait, is given in Figure 8.

```
copyArray = proc (a1: array[elem], copyElem: proc (e1: elem) returns (e2: elem))
                returns (a2: array[elem])
  uses ArrayOfElemObj
       pre true{copyElem}(e1↑ = e2↓ ∧ new e2 ∧ mutates ∅ ∧ returns)
       post new a2 ∧ length(a1↑) = length(a2↓) ∧ low(a1↑) = low(a2↓)
            ∧ (∀j:Int low(a1↑)≤j≤high(a1↑)
                 [fetch(a1↑,j) = fetch(a2↓,j) ∧ new fetch(a2↓,j)]
            ∧ mutates ∅ ∧ returns
       end
```

We are not able in our specification language to specify the invocation of another procedure. That is, we are not able to make an assertion in the procedure specification, Pr1, about the application of a procedure, Pr2, to a list of arguments, ArgList, such as:

apply(Pr2, ArgList)

The reason is that we cannot know in which states to evaluate (i.e., apply $\mathcal{V}$) the objects in ArgList. To specify the effect we would want, because Pr2 may have side effects, we would

```
ArrayOfElemObj: trait
  includes Array with [AOE for A, elem_obj for E]

Array: trait
  includes Integer, Elem
  introduces
        create: Int → A
        addh: A, E → A
        remh: A → A
        low: A → Int
        high: A → Int
        fetch: A, Int → E
        store: A, Int, E → A
        size: A → Bool
  closes A over [create, addh]
  constrains [A] so that for all [i,i1,i2: Int, e,e1,e2: E, a: A]
        remh(create(i)) exempt
        remh(addh(a,e)) = a
        low(create(i)) = i
        low(addh(a,e)) = low(a)
        high(a) = low(a) + size(a) - 1
        fetch(create(i1),i2) exempt
        fetch(addh(a,e),i) = if i .eq (low(a) + size(a)) then e else fetch(a,i)
        store(create(i1),i2,e) exempt
        store(addh(a,e1),i,e2) = if i .eq (low(a) + size(a)) then addh(a,e2)
              else addh(store(a,i,e2),e1)
        size(create(i)) = 0
        size(addh(a,e)) = size(a) + 1
```

Figure 8. ArrayOfElemObj Trait

---

want to evaluate ArgList with respect to pairs of intermediate states of the invocation of Pr1, and not the initial and final states.

The *copyArray* example illustrates this failure of expressive power in our specification language. We would like to be able to specify that any implementation of *copyArray* must invoke the *copyElem* procedure such that the effects of executing the copyArray procedure include the effects of executing the *copyElem* procedure. We specified in *copyArray*'s post-condition, what the behavior of *copyArray* would be as if *copyElem* were invoked from *copyArray*. Nowhere, however, do we actually state in the post-condition that *copyElem* must

be used--it is as if the *copyElem* argument were ignored. Hence, a procedure whose behavior is the same as specified above, but is implemented without using the *copyElem* procedure argument, would satisfy the procedure specification. In order to rule out such procedures, we would need to be able to make an assertion such as:

$$\forall j{:}\text{Int}\ \text{low}(a1\uparrow) \leq j \leq \text{high}(a1\uparrow)\ \textbf{apply}(\text{copyElem}, \text{fetch}(a1\uparrow, j)).$$

2.2.3 Cluster Specifications

A model of a cluster specification is an abstract data type. A cluster specification includes a type identifier, a list of procedure specification identifiers, a link, and a body. The link includes the name of a trait and a mapping from the type identifier to a sort identifier. The body includes a set of procedure specifications.

Syntax

ClusSpec :: = *TypeId* = **cluster is** *ProcId* +, *ClusLink ClusBody* **end**
ClusLink :: = *Link ClusMap*
ClusMap :: = **provides** *MutFlag TypeId* **from** *SortId*
ClusBody :: = *ProcSpec* +
MutFlag :: = **mutable | immutable**

Def: The type identifier named by a cluster specification is called the *defined type*.

Def: The trait named in the **uses** clause of a cluster specification, cl, is called the *used trait* of cl.

Def: A procedure specification defined within a cluster specification is called a *bound procedure specification*. A procedure specification defined outside of all cluster specifications is called a *free procedure specification*.

Checking

We check that:

1. All procedure specifications whose identifiers appear in the heading of a cluster specification are defined in the body of the cluster specification, and all identifiers of procedure specifications in the body of the cluster specification appear in the heading.

2. The type identifier found in the type-to-sort mapping is the same as the type identifier that names the cluster specification.

3. The sort identifier in the type-to-sort mapping is the name of a sort provided by the used trait.

4. If the "flag" (in *MutFlag*) is **mutable**, some **mutates** *t1, ..., tn* assertion must appear in a procedure specification in the cluster specification where the defined type of the cluster specification is the type of the object denoted by some term in *t1, ..., tn*. If the "flag" is **immutable**, none of the objects denoted by terms in **mutates** assertions in any of the procedure specifications can be of the defined type.

5. Each procedure specification is well-formed.

Meaning

A model of a cluster specification is an abstract data type, which consists of a pair of a set of objects and a set of operations. Let Cl be a cluster specification; Prs, the set of procedure specifications of Cl; Tr, the used trait of Cl.

Def: For an abstract data type, T = <Obs, Ops>, *T is a model of Cl*, i.e., T $\models$ Cl, if and only if:

1. Obs = {o | o$\in$*Obj* $\wedge$ the sort of o is T_obj},
2. $\forall$pr$\in$Prs $\exists$op$\in$Ops, op $\models$ pr,
3. $\forall op_i = \langle R_i, A_i \rangle \in Ops$, A = A_i, where A is a model of Tr.

The type-to-sort mapping of the form, "**provides** (...) T **from** S," of the cluster specification tells us that the value of TtoS for type T is S.

Example

The *set* cluster specification (Figure 9) defines a mutable set abstract data type. *Singleton* and *union* return new nonempty set objects. *Delete* might mutate its input set argument, if doing so does not empty it; otherwise, it terminates exceptionally, signaling *emptiesSet*. From the theory (Chapter 3) associated with this cluster specification, we can show that no set object can be empty. *Size* returns the cardinality of its input set argument.

```
set = cluster is singleton, union, delete, size
  uses SetOfInt
  provides mutable set from SI

        singleton = proc (i: int) returns (s: set)
          uses SetOfInt
                pre true
                post s↓ = add(empty, i↑) ∧ new s ∧ mutates Ø ∧ returns
                end

        union = proc (s1, s2: set) returns (s3: set)
          uses SetOfInt
                pre true
                post ∀i:Int [has(s3↓,i) = has(s1↑,i) ∨ has(s2↑,i)]
                        ∧ new s3 ∧ mutates Ø ∧ returns
                end

        delete = proc (s: set, i: int) signals (emptiesSet)
          uses SetOfInt
                pre true
                post [((card(s↑) ≥ 2) ∨ ~has(s↑,i↑)) ⇒
                                (s↓ = remove(s↑,i↑) ∧ mutates s ∧ returns)] ∧
                     [((card(s↑) .eq 1) ∧ has(s↑,i↑)) ⇒
                                mutates Ø ∧ signals emptiesSet] ∧
                     new Ø
                end

        size = proc (s: set) returns (i: int)
          uses SetOfInt
                pre true
                post i↓ = card(s↑) ∧ new Ø ∧ mutates Ø ∧ returns
                end
end
```

Figure 9. Set Cluster Specification (SetClusSpec)

The *set* cluster specification example illustrates a clear distinction between a (value) sort identifier and a type identifier. Although the trait *SetOfInt* defines an "empty" value of sort *SI*, no object of *set* type will ever have such a value since operations on objects of *set* type construct only nonempty set objects. One could have specified a more conventional *set* type with operations *create* and *insert*, so that a possible value for a set object would be "empty."

We will be returning to this somewhat contrived example in later chapters. We henceforth refer to the specification of Figure 9 as *SetClusSpec* and repeat it in Appendix I for future reference.

2.3 Summary

In this chapter we described models of specifications and implementations, and we described a kernel interface language. Models of traits are many-sorted *algebras*; models of procedures and procedure specifications are *operations*, each of which is a pair consisting of a relation on *states*, and an algebra; models of clusters and cluster specifications are *abstract data types*, each of which is a pair consisting of a set of objects and a set of operations.

The kernel interface language contains procedure specifications and cluster specifications. Interface assertions constitute the body of a procedure specification; procedure specifications constitute the body of a cluster specification. The language of interface assertions is built from the language of Larch assertions. We added notation (↑ and ↓) to be able to refer to the initial and final values of objects, since interface assertions are interpreted with respect to two states. A procedure specification basically consists of a used trait and a pair of assertions. We introduced special assertions to handle multiple termination conditions, creation of new objects, mutation of existing objects, and procedure objects as arguments. A cluster specification basically consists of a type name, a used trait, a type-to-sort mapping, and a set of procedure specifications. In the next chapter we see how to map a specification into the set of well-formed formulae of the theory it denotes.

3. Theories

In this chapter we switch to the syntactic viewpoint of specifications and implementations. The two main objectives of this chapter are (1) to define when an implementation *satisfies* a specification, and (2) to define precisely the theories denoted by specifications and implementations.

Section 3.1 contains some definitions dealing with first-order theories. From these basic definitions, in Section 3.2 we define the satisfaction relation between implementations and specifications. Section 3.3 and 3.4 define the theory of a specification and the theory of an implementation, respectively. Their definitions depend on the definition of a *type induction principle*, which we defer defining to Section 3.5. Section 3.5 builds up to defining this principle, which is complicated because of the possibility of "exposing the rep" in CLU.

3.1 Definitions

The following definitions dealing with theories and formal systems are provided as a review of basic concepts in logic. We borrow from three introductory logic texts [Shoenfield67, Mendelson64, Enderton72].

Theory and Formal System

A theory is specified by giving a *formal system*, which has three parts:

> 1. Its *language*. To specify a language, we specify its set of *symbols*, and its set of *well-formed formulae* (wff's). We denote the language of a formal system F by L(F).
>
> 2. Its *axioms*. Each axiom must be a well-formed formula of the language of the formal system.
>
> 3. Its *rules of inference*, which we sometimes call *rules*. Each rule of inference states that under certain conditions, one formula, called the *conclusion* of the rule, can be *inferred* from certain other formulae, called the *hypotheses* of the rule. Each rule is an

inference relation among wff's.

A *proof* in F is a finite sequence of wff's, each of which is either an axiom or is the conclusion of a rule whose hypotheses precede that wff in the proof. A *theorem* of F is a wff, A, such that there is a proof whose last wff is A. Such a proof is called a *proof of A*. The *theory* specified by a formal system F is the smallest set of formulae reflexively and transitively closed over the set of axioms under the rules of F.

The *logical symbols* of a first-order language are the usual connectives, quantifiers, and possibly an equality symbol, =. All other symbols, e.g., function symbols, are called *nonlogical*. A first-order language L' is an *extension* of the first-order language L if every nonlogical symbol of L is a nonlogical symbol of L'. Let F and F' denote formal systems that respectively specify the first-order theories T and T'. T' is an *extension* of T if L(F') is an extension of L(F) and every theorem of T is a theorem of T'. A *conservative extension* of T is an extension T' of T such that every formula of F which is a theorem of T' is also a theorem of T.

Used and Imported Types

The following definitions are based on the interface language.

A *used type* of a procedure specification is a type whose identifier appears in its heading. The type of any object that is an input or an output argument of that procedure is a used type. A *used type* of a cluster specification is a used type of each of its procedure specifications.

For a used type, T, the sort, TtoS(T), is called the *used sort*. For a rep type, T, the sort, TtoS(T), is called the *rep sort*. For an abstract type, T, the sort, TtoS(T), is called the *abstract sort*.

Recall from Chapter 2, a *bound procedure specification* is a procedure specification that is defined within a cluster specification. A *free procedure specification* is a procedure specification that is defined outside all cluster specifications.

An *imported type of a cluster specification* is a used type of a cluster specification that is not the defined type. An *imported type of a bound procedure specification* is a used type of the procedure specification that is not the defined type of the cluster specification. So that we can use the same terminology for free and bound procedure specifications, we define an *imported type of a free procedure specification* as a used type of the procedure specification.

Syntactic Conventions

For a predicate, P, of n arguments, we write P[X] to denote P(x1, ..., xn). For a predicate P of 1 argument, and a list, X = x1, ..., xn, we write $\bigwedge_n$P(X) to denote P(x1) $\wedge$ ... $\wedge$ P(xn). For two lists of equal length, X = x1, ..., xn, and A = a1, ..., an, we write X = A for x1 = a1 $\wedge$ ... $\wedge$ xn = an. We write "Pr.pre" and "Pr.post" to denote the pre-condition and the post-condition of the procedure specification Pr.

3.2 Satisfaction

We define satisfaction of an implementation with respect to a specification in terms of theories so we need not directly refer to states. This point of view of couching definitions in terms of theories will lead to subsequent definitions of properties of specifications given in Chapter 5. We choose to use the term "satisfaction" instead of "correctness" because it better suggests that a relation exists between an implementation and a specification, and because in terms of theories, the notion of a "correct" theory seems strange.

Def: A procedure, ProcImp, *satisfies* the procedure specification, Pr, if and only if Th(Pr) $\subseteq$ Th(ProcImp).

Def: A cluster, ClusImp, *satisfies* the cluster specification, Cl, if there exists a homomorphism, A, from terms of the rep sort to terms of the abstract sort such that Th(Cl) $\subseteq$ Th(ClusImp) $[T/R]_A$.

$[T/R]_A$ (read "T for R under A") means that T, the identifier denoting the abstract type, is substituted for every occurrence of R, the identifier denoting the rep type, and A(r) is substituted for every occurrence of a term of rep sort denoted by r.

We discuss how one would prove that an implementation satisfies a specification after we have formally defined the theories of specifications and implementation. In Section 3.4.1 we discuss this for procedures; in 3.4.2, for clusters.

3.3 Theory of a Specification

We are very careful to separate the trait language from the interface language, and the interface language from the programming language. We must similarly be careful to distinguish among the theory of a trait, the theories of procedure and cluster specifications, and the theory of an implementation. In this section we begin with a formal definition of the theory of a trait and then define the theories of procedure and cluster specifications.

3.3.1 Theory of a Trait

Let Th(tr) denote the theory of the trait tr. Th(tr) is a conservative extension of first-order many-sorted predicate calculus with equality. It is an extension by the addition of the function identifiers of tr, the axioms of tr, and two rules of inference. The formal system is as follows:

Symbols

Logical symbols: $\sim$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, $\exists$, $=$; the set of variable identifiers, *VarId*; true, false;

Nonlogical symbols: the set of function identifiers, *OpId*; the punctuation marks: comma, colon, and parentheses.

Wff's

Wff :: = *Assn*

Assn :: = true | false | ~*Assn* | *Assn* ∧ *Assn* | *Assn* ∨ *Assn*

| *Assn* ⇒ *Assn* | *Assn* ⇔ *Assn* | (*Assn*)

| ∀ *VarId*: *SortId Assn* | ∃ *VarId*: *SortId Assn*

| *Term* = *Term*

Term :: = *VarId* | *OpId*<(*Term* + ,)>

The precedence of the operators and quantifiers from highest to lowest is ~, ∀, ∃, ∧, ∨, ⇒, ⇔. When one connective is used repeatedly, the expression is grouped to the right.

Axioms

1. All logical axioms of first-order predicate calculus with equality.

   a. All propositional axioms. E.g., ~P ∨ P.

   b. Substitution axiom: ∀x:S (P) ⇒ (P[t/x]), where term t is substitutable for variable identifier x in P (defined precisely below), and t and x are of sort S.

   c. Identity axiom: t = t.

   d. Equality axiom: s1 = t1 ∧ ... ∧ sn = tn ⇒ f(s1, ..., sn) = f(t1, ..., tn).

2. All equations of the form t1 = t2 in tr.

3. ~(true = false). All other inequations in Th(tr) are derivable from this one and the meaning of =.

Rules of Inference

1. Rules for first-order predicate calculus with equality:

a. Modus ponens

$$\frac{P,\ P \Rightarrow Q}{Q}$$

b. Generalization

$$\frac{P}{\forall x{:}S\ P}$$

Here $\forall x{:}S$ stands for universal quantification over all sorted variables x_i in P with corresponding sorts S_i.

2. Sort Induction

If "**closes** S **over** [op1, ..., opn]" appears in tr, the following is the corresponding sort induction rule for predicate P(t) with free variable t of sort S.

$$\frac{\begin{array}{c} P(x_1) \wedge \ldots \wedge P(x_{k1}) \Rightarrow P(op1(x_1, \ldots, x_{k1})) \\ \ldots \\ P(x_1) \wedge \ldots \wedge P(x_{kn}) \Rightarrow P(opn(x_1, \ldots, x_{kn})) \end{array}}{\forall t{:}S\ P(t)}$$

where ki is the arity of opi, $P(x_i)$ = true if x_i is not of sort S.

3. Sort Reduction[3]

If "**reduces** S **over** [op1, ..., opn]" appears in tr, the following is the corresponding sort reduction rule.

$$\frac{\begin{array}{c} op1(x_1, \ldots, x_{j-1}, t1, \ldots, x_k) = op1(x_1, \ldots, x_{j-1}, t2, \ldots, x_k) \\ \ldots \\ opn(x_1, \ldots, x_{j-1}, t1, \ldots, x_k) = opn(x_1, \ldots, x_{j-1}, t2, \ldots, x_k) \end{array}}{t1 = t2}$$

---

3. Although in Chapter 1 we did not discuss sort reduction because we do not need it for our example traits, we include it here for completeness.

where t1 and t2 are terms of sort S, and the x_i's do not occur in t1 or t2, and the ti's appear in all argument positions of sort S.

Substitution

In the substitution axiom we used the phrase "a term that is substitutable for a variable in a predicate," which we now define.

Def: An occurrence of x in a formula P is *bound* if it occurs in a part of P of the form ∀x:S *Assn* or ∃x:S *Assn*; otherwise, it is *free* in P.

Def: A *term*, τ, is *substitutable for x in P* if for each variable identifier y occurring in τ, no part of P of the form "∀y:S B" or "∃y:S B" contains an occurrence of x that is free in B.

We write "P[τ/x]" (read "substitute τ for x in P") to denote the formula P obtained from the substitution of τ for free occurrences of x in P, restricted to the cases where τ is substitutable for x in P. We extend this notation for lists (of equal length) of terms and identifiers, A and X, so that P[A/X] stands for the formula obtained from P by respectively replacing all occurrences of x1, ..., xn by terms a1, ..., an, where each term ai is substitutable for xi in P.

3.3.2 Theory of a Procedure Specification

Let Th(Pr) denote the theory of the procedure specification Pr. Th(Pr) is a conservative extension of the theory of the used trait of Pr. We extend the theory of the used trait of Pr by adding to the formal system:

Symbols

The identifier, Pr; terminal symbols of *Assn*'s; the set of object identifiers, *ObjId*; curly braces, ↑ and ↓.

Wff's

Wff :: = *Assn* | *Assn* {*ProcId*} *Assn*

Assn :: = % as in Section 3.3.1

| **returns** | **signals** *SigId*

| **new** Ø | **new** *Term* +,

| **mutates** Ø | **mutates** *Term* +,

| *Assn* {*Term*} *Assn*

Term :: = % as in Section 3.3.1

| *ObjId* | *Term*↑ | *Term*↓

Axiom

$$\text{Pr.pre[X]}\ \{\text{Pr}\}\ \text{Pr.post[X,Y]}$$

where X is the list of input formals of Pr; Y, the list of output formals.

Rules of Inference

1. Rule of Consequence

$$\frac{P \Rightarrow P1,\ P1\{Pr\}Q1,\ Q1 \Rightarrow Q}{P\{Pr\}Q}$$

where P, P1, Q, and Q1 are assertions. Recall that the validity of the assertions of the hypotheses of this rule is with respect to two states. In particular, Q1 can refer to initial values of objects referred to in P1.

2. Simplified Invocation Rule

$$\frac{X = A \wedge Y = B,\ \text{Pr.pre[X]}\ \{\text{Pr}\}\ \text{Pr.post[X,Y]}}{\text{Pr.pre[A/X]}\ \{\text{Pr}\}\ \text{Pr.post[A/X, B/Y]}}$$

X is the list of input formals of Pr; Y, the list of output formals; A is the list of terms denoting objects that are input arguments; B, the list of output arguments. This is a simplified case of

the CLU procedure invocation rule (see [Schaffert81]).[4]

3. All type induction rules of each imported type. We define this set of type induction rules in Section 3.5.2.

Th(Pr) contains the theories of all of Pr's imported types. We intentionally excluded the defined type from the set of imported types of a bound procedure specification so that its theory would not include the theory of its defined type. This is done to avoid a circular definition of the theory of a cluster specification (Section 3.3.3).

Example

Recall the choose procedure specification:

```
choose = proc (s: set) returns (i: int)
  uses SetOfInt
       pre ~isEmpty(s↑)
       post has(s↑,i↓) ∧ new ∅ ∧ mutates ∅ ∧ returns
       end
```

Th(*choose*) includes the trait theory, Th(*SetOfInt*), which contains some axioms, e.g., isEmpty(empty) = true, and ∀x:SI e:E [isEmpty(add(x,e)) = false]; and the sort induction rule with the hypotheses P(empty) and P(x) ⇒ P(add(x,e)), and the conclusion ∀t:SI P(t). An example theorem that is derivable from the axioms and the rules in Th(*SetOfInt*) is ∀t:S card(s) ≥ 0. Since the *Integer* trait is imported in the *SetOfInt* trait, Th(*choose*) includes all theorems on terms of *Int* sort.

An additional theorem in Th(*choose*) is ~isEmpty(s↑){choose}(has(s↑,i↓) ∧ **new** ∅ ∧ **mutates** ∅ ∧ **returns**). Given the simplified invocation rule, and the rule of consequence, we derive theorems from this axiom. For example, the formula

---

4. We do not need the part of the rule that handles recursive invocations.

~isEmpty(add(empty,1))
{choose}
has(add(empty,1),1) ∧ **new** ∅ ∧ **mutates** ∅ ∧ **returns**

is in Th(*choose*).

3.3.3 Theory of a Cluster Specification

Let Th(Cl) denote the theory of the cluster specification Cl. Th(Cl) is the union of the theories of its procedure specifications closed under the following:

Rules of Inference

1. All type induction rules of the defined type, T. See Section 3.5.2.

Sometimes it is useful to include the theory of the defined type of the cluster specification with the theory of a bound procedure specification. We denote this theory by "Th(Pr +)." For notational convenience, if Pr is a free procedure, let Th(Pr +) be Th(Pr).

3.4 Theory of an Implementation

3.4.1 Theory of a Procedure

Let ProcImp be a procedure and Th(ProcImp) denote the theory of the procedure ProcImp. The formal system that specifies Th(ProcImp) is as follows:

Symbols

Identifiers that appear in the procedure body; keywords of CLU and *Assn*'s; curly braces, ↑ and ↓; ProcImp (the name of the procedure), if the body of ProcImp contains a recursive invocation.

Wff's

Wff :: = *Assn* | *Assn* {*Stmt*} *Assn*

Stmt :: = CLU statements or expressions in the body of ProcImp

Assn :: = % as in Section 3.3.2

Axioms

All valid formulae of the form *Assn* {*Stmt*} *Assn*; in particular, consequences of the simplified invocation rule for the procedure specifications that specify the behavior of the procedures called from within the body of the procedure, ProcImp.

Rules of Inference

1. Rule of Consequence

2. All proof rules of CLU [Schaffert81], including those for sequential, iterative, and conditional statements.

3. All type induction rules of each imported type of ProcImp.

If ProcImp is defined within a cluster we also add:

4. All type induction rules for the rep type of the cluster.

From the proof rules of CLU and the rule of consequence, given the body of a procedure, we derive the set of formulae involving the body of the procedure that are valid in all models of ProcImp. These formulae comprise Th(ProcImp).

Proving Satisfaction

In order to show that a procedure (implementation), ProcImp, satisfies a procedure specification, Pr, we need to show that each theorem in Th(Pr) is in Th(ProcImp). Let Pr be:

```
Pr = proc (x1, ..., xn) returns (y1, ..., ym) signals (..)
  pre P
  pre Q
  end
```

and an implementation of Pr be:

```
ProcImp = proc (x1, ..., xn) returns (...) signals (..)
  BODY
  end
```

Let A and B be lists of terms denoting input and output objects, and X and Y be the lists of input and output formals. Assume P[A/X] {Pr} Q[A/X, B/Y] is a theorem in Th(Pr). We must show that P[A/X] {Pr} Q[A/X, B/Y] $\in$ Th(ProcImp). To show this, we use the following (non-recursive) procedure definition CLU proof rule,

$$\frac{x1 = a1 \wedge \ldots \wedge xn = an \wedge P1 \{BODY\} Q1}{P1 \{Pr\} Q1}$$

where P1 and Q1 are assertions, ai are terms denoting objects, and the procedure's local (not own) variables must not occur free in P1 or Q1. Notice that $\forall i[xi = ai] \Rightarrow \forall i[xi\uparrow = ai\uparrow]$. Any local variables are freshly created on each invocation of the procedure, and are discarded when it returns, so P1 and Q1 must not refer to them.

The conclusion of the procedure definition rule produces a specification of Pr. Typically, we must then show that (1) P[A/X] $\Rightarrow$ P1, and (2) Q1 $\Rightarrow$ Q[A/X, B/Y]. Then from the rule of consequence, we have:

$$\frac{P[A/X] \Rightarrow P1,\ P1 \{Pr\} Q1,\ Q1 \Rightarrow Q[A/X, B/Y]}{P[A/X] \{Pr\} Q[A/X, B/Y]}$$

which gives us that P[A/X] {Pr} Q[A/X, B/Y] $\in$ Th(ProcImp).

3.4.2 Theory of a Cluster

Let Th(ClusImp) denote the theory of the cluster ClusImp. (ClusImp) is the union of the theories of its procedures closed under the CLU proof rules. There are no type induction rules associated with a cluster.

Proving Satisfaction

Carrying out the following steps is sufficient to show that a cluster satisfies a cluster specification.

1. Define a homomorphism A that maps terms of the rep sort to terms of the abstract sort.

2. Define a rep invariant on terms of the rep sort used to help prove satisfaction of each procedure.

3. For each procedure, show it satisfies its corresponding procedure specification under A and that the rep invariant is maintained.

These steps are no different from those used in usual proofs of satisfaction, where A is called an *abstraction function* [Hoare72, Guttag78, Guttag80a]. For our purposes, however, the abstraction function is defined on (sorted) terms and not on (typed) objects. We give an example of a proof of satisfaction between a cluster and a cluster specification in Appendix II.2.

3.5 Type Induction

In the definitions of the formal systems that specify the theories of specifications and implementations, we referred to the "type induction rules" of a type. We derive each rule syntactically from cluster specifications. We argue that each rule is sound, however, because it is derivable from the computational induction rule for CLU, which we assume is sound. In Section 3.5.1, we define this computational induction rule. In Section 3.5.2, we define how to

derive syntactically a set of type induction rules for a cluster specification.

3.5.1 Computational Induction

Recall that our model of computation is an alternating sequence of states and statements starting in some initial state, σ_0. For the states, σ_i, and the statements, S_i, $1 \leq i \leq n$, let a computation sequence be:

$$\sigma_0 \, S_1 \, \sigma_1, \ldots, \sigma_{n-1} \, S_n \, \sigma_n$$

Informally, if some predicate P is true for each successive pair of states in the computation, then P is true of a computation. P is essentially an invariant over the computation sequence. We need to introduce a function, *flip*, on assertions because we want P to be true for all successive pairs of states in the computation, where the final state of one pair becomes the initial state of the next pair. Since assertions are interpreted with respect to two states, in order to use the same truth function *T*, which we defined in Chapter 2, we need to ignore one of the two states in which an invariant is interpreted. Hence, we use *flip* to make all the arrows in an assertion point in the same direction.

Formally, we state the computational rule as follows. For some predicate P:

$$\frac{\begin{array}{c} \text{true } \{S_1\} \; \mathit{flip}(P) \\ P \; \{S_2\} \; \mathit{flip}(P) \\ \ldots \\ P \; \{S_n\} \; \mathit{flip}(P) \end{array}}{\text{true } \{S\} \; \mathit{flip}(P)}$$

for all statements S of the computation.

flip(P) is P with all occurrences of ↑ replaced by ↓, with a restriction on the form of P to which *flip* is applicable, and a restriction on the flipping of arrows in a procedure object assertion (poa):

1. Only assertions whose value depends on a single state can appear in P. Specifically, no **returns**, **signals**, **new**, or **mutates** assertions are allowed in P. Otherwise, we could not properly ignore one of the two states in which an assertion is interpreted.

2. If P contains an assertion about a procedure object of the form P1{τ}Q1, where P1 and Q1 are assertions and τ is a term denoting a procedure object, we do not replace $\uparrow$ by $\downarrow$ in P1 or Q1. This is because P1 and Q1 are not interpreted with respect to the same state as that for P1{τ}Q1.[5]

We emphasize that the first restriction is only for the computational induction rule and *not* on all assertions. For example, formulae of the form P {Pr} Q where Q has **returns**, **signals**, **new**, or **mutates** assertions are still well-formed, as in the axiom of Th(Pr), Pr.pre {Pr} Pr.post.

Henceforth, we write P^f for *flip*(P). Notice we must also be careful when using the usual Hoare proof rules for statements like sequential composition, conditional, and loops. For example, the sequential composition rule should be:

$$\frac{P\ \{S1\}\ Q^f,\ Q\ \{S2\}\ R^f}{P\ \{S1;S2\}\ R^f}$$

Similar syntactic transformations must be performed on all other proof rules so that they can be applied appropriately in proofs.

3.5.2 Type Induction Principle

A cluster specification is ideally more than just a syntactic way of grouping together a set of procedure specifications. It gives us a way of localizing the specifications of the behaviors (input-output relations) of all operations on objects of the defined type. This modularization should give a means of localizing the proof of invariant properties of all

---

5. Recall that the truth of such a poa is defined to be true if the value of τ, i.e., some relation-algebra pair, satisfies the pair of assertions <P1, Q1> (Section 2.2.2.5).

objects of the defined type. We would like to associate with a cluster specification a type induction rule and assert that it is a sound rule in any cluster that satisfies the cluster specification. This rule would allow us to infer that some property is true of all objects of type T by considering only a subset of the procedures that create and mutate objects of type T. In this section we see that defining such an induction rule is not quite so straightforward because of situations that arise in implementations that "expose the rep."

In Section 3.5.2.1 we show how to derive this desired type induction rule for a cluster specification and give an example of a derivation. In Section 3.5.2.2, we explain the problem of exposing the rep that can invalidate this type induction rule, and so in Section 3.5.2.3 we extend the derivation procedure to allow for some implementations that expose the rep.

3.5.2.1 A Type Induction Rule

We first state how to derive the type induction rule for a type T, then explain the rule, then justify it.

For a procedure specification, let T1 be the sublist of its input formals that are of type T; T2, the sublist of output formals that are of type T. (Recall by our definitions in Chapter 2, formals in a **signals** clause are included as output formals of a procedure header.) T1 and T2 are sublists because some input and output formals may not be of type T. Let i and j be the lengths of the lists T1 and T2, respectively.

Method: Derivation of a type induction rule for predicate, P(t), with free variable t of type T.

Hypotheses: The hypotheses are named HB, HP, and HM for basic, producing, and mutating constructors (to be defined), respectively.

1. For each bc∈BC(T), add an HB hypothesis of the form:

$$\text{true } \{bc\} \bigwedge_j P^f(T2)$$

2. For each pc∈PC(T), add an HP hypothesis of the form:

$$\bigwedge_i P(T1) \ \{pc\} \ \bigwedge_j P^f(T2)$$

3. For each mc∈MC(T), add an HM hypothesis of the form:

$$\bigwedge_i P(T1) \ \{mc\} \ \bigwedge_i P^f(T1) \wedge \bigwedge_j P^f(T2)$$

where P is restricted as for the computational induction rule (Section 3.5.1). $\bigwedge_i P^f(T1)$ can be conjoined to $\bigwedge_j P^f(T2)$ to the right of the braces in the first two kinds of hypotheses, but by the definitions of basic and producing constructors (defined below), it would be vacuously true.

Conclusion: true {S} $\forall t{:}T\ P^f(t)$ for all statements S.

(end of Method)∎

The sets, BC(T), PC(T), and MC(T), represent the sets of specifications of procedures that can create and mutate objects of type T. These sets are not necessarily disjoint since a procedure might do both. Roughly speaking, the differences among the three are whether any input arguments are of type T, whether any output arguments are of type T, and whether any objects of type T are mutated. BC(T) is the set of basic constructors of type T. A *basic constructor of type T* is a procedure specification that has no input arguments of type T; whose pre-condition contains no explicit assertions about objects of type T; and whose post-condition specifies the return of a new object of type T. For example, *singleton* of *SetClusSpec* (Appendix I, Figure 9) is a basic constructor of type set. PC(T) is the set of producing constructors of type T. A *producing constructor of type T* is a procedure specification that has both input and output formals of type T; whose post-condition specifies

the return of a new object of type T; and for all assertions in its post-condition of the form **mutates** t1,..., tn, none of the types of the objects denoted by the terms in the list t1, ..., tn is T. For example, *union* of *SetClusSpec* is a producing constructor of type set. MC(T) is the set of mutating constructors of type T. A *mutating constructor of type T* is a procedure specification that has an assertion in its post-condition of the form **mutates** t1, ..., tn, and T is the type of the object denoted by some term in the list t1,..., tn. For example, *delete* of *SetClusSpec* is a mutating constructor of type set.

To justify the rule, consider the computational induction rule given a predicate, P(t), on objects of type T. We need be concerned only with invocations of procedures that create and manipulate objects of type T. We reduce the number of hypotheses of the computational induction rule to obtain a type induction rule by retaining only those relevant hypotheses. Notice we have available, however, only the procedure specifications and not their implementations. Hence, the hypotheses we select from the computational induction rule can be based solely on the specification of the procedures, and not their implementations.

Example 1

Consider our simple example, *SetClusSpec*. Following the method given, we have instances of each of the three kinds of hypotheses, HB, HP, and HM, to obtain the following type induction rule:

$$\frac{\begin{array}{c}\text{true } \{\text{singleton}\}\ P^f(s) \\ P(s1) \wedge P(s2)\ \{\text{union}\}\ P^f(s3) \\ P(s)\ \{\text{delete}\}\ P^f(s)\end{array}}{\text{true } \{S\}\ \forall t{:}\text{set}\ P^f(t)}$$

Suppose P(t) is card(t↑) > 0. The hypotheses are:

HB true {singleton} card(s↓) > 0
HP card(s1↑) > 0 ∧ card(s2↑) > 0 {union} card(s3↓) > 0
HM card(s↑) > 0 {delete} card(s↓) > 0

The conclusion is true {S} ∀t:set[int] card(t↓) > 0 for all statements S.

We use the axiom of the theory of the procedure specification and the rule of consequence to show the validity of each of these hypotheses. For example, to show the validity of HP above, we have:

1. Assume [card(s1↑) > 0 ∧ card(s2↑) > 0].
2. From the above assumption and the sort induction rule associated with Th(*SetOfInt*),
   ∀i:Int [has(s3↓,i) = has(s1↑,i) ∨ has(s2↑,i)] ⇒ card(s3↓) > 0
3. Th(*union*) contains the axiom,
   true {union} [**new** s3 ∧ **mutates** Ø ∧ **returns**
   ∧ ∀i:Int [has(s3↓,i) = has(s1↑,i) ∨ has(s2↑,i)].
4. So, by the rule of consequence (union.post ⇒ 2) we have:
   HP: card(s1↑) > 0 ∧ card(s2↑) > 0 {union} card(s3↓) > 0

Similar reasoning is used to show the validity of HB and HM for *singleton* and *delete*. Therefore, we can conclude that the size of all objects of type set is greater than zero. Notice that this is a very different theorem from that in Th(*SetOfInt*), ∀x:SI card(x) ≥ 0.

3.5.2.2 Exposing the Rep

We have defined an object to belong to only one type. In CLU, however, this property of objects does not always hold since one can write programs where an object belongs to more than one type, e.g., both the abstract and the rep type. CLU type checking does not prevent this situation from arising because it cannot detect it syntactically. Since operations of both types might possibly mutate such an object, the desired locality principle of a cluster can be violated; our single type induction rule might be invalid.

When some operations besides those specified in the cluster specification defining T can mutate objects of type T (by means other than invoking procedures of the cluster), we say that "the rep is exposed." There are two ways in which such a situation may arise. Both involve sharing of objects of mutable type.[6] One way is when the rep type object and the abstract type object are the same object. We call this "exposing the whole rep." Any mutating operation of the rep type can then mutate an object of the abstract type, and vice versa. A simple example of this in shown in Figure 10. Exposing the whole rep can (and most of the time should) be avoided. In the *queue* example, the *make* procedure should copy the array before returning the queue to avoid exposing the rep. Since it does not, a mutating array operation, e.g., *addh*, that changes the original input array object also changes the returned queue object since they are the same object.

A second way an object of type T can be mutated by an operation other than those specified in the cluster specification defining T is by establishing sharing with an object of type T1 whose value is incorporated in the value of the rep of type T. We call this "exposing the subrep." Whether or not an implementation exposes its subrep is relative to a specification. For example, the *read* procedure in Figure 11 would be exposing the subrep if the specification of *read* were to require that the top of the input stack returned be a new

---

```
queue = cluster is ..., make, ...
        rep = array[elem]
        ...
        make = proc (r: rep) returns (cvt)
                return(r)
                end make
        ...
        end queue
```

Figure 10. Exposing the Whole Rep for Queues

---

6. If we had only immutable types or if we eliminated sharing in CLU, the problem of exposing the rep would not exist.

object. Since *read* returns the top of the input stack argument, without copying, then any changes made to that set would appear to change the value of the stack. Again, to avoid this sharing, a copy of the top of the sequence should be made before returning it or pushing it.

One could argue that implementations that expose the rep (of any kind) should be banned. There are two reasons why such a restriction is too severe. The first is that in practice, one sometimes intentionally wants such sharing among objects, perhaps for

---

```
stack = cluster is empty, grow, read, ...
        rep = sequence[set]

        empty = proc () returns (cvt)
                return (rep$new())
                end new

        % grow will only push on the input stack a set whose size is less than 64
        grow = proc (s1: cvt, s: set) returns (cvt)
                if set$size(s) ≥ 64 then return (s1)
                seq: rep := rep$new()
                for e: set in rep$elements(s1)
                        seq := rep$addh(seq, e)
                        end
                return (seq)
                end grow

        read = proc (t: cvt) returns (set) signals (bounds)
                return (rep$top(t)) resignal (bounds)
                end read
        ...
        end stack

set = cluster is ..., delete, ...
        rep = array[int]
        ...
        % delete mutates s if i is in s
        delete = proc (s: cvt, i: int)
                ...
                end delete

        end set
```

Figure 11. Exposing the Subrep for Stacks

efficiency reasons, and cleverly exploits it. The second is that there is no reasonable way to ban such sharing, i.e., to detect it syntactically. Before we proceed with the definitions of these induction rules, we point out that CLU, which cannot completely enforce a restriction against exposing the rep type, can still be used to construct "true" abstract types. The programmer need only follow a programming discipline that ensures that reps are not exposed or that sharing of mutable objects is not abused.

3.5.2.3 Type Induction Rule Revisited

If we were to associate a type induction rule as thus far defined with each cluster specification then an implementation that exposes the rep might violate this rule and not necessarily satisfy the cluster specification. In deciding whether an implementation satisfies a specification, we could either be very restrictive and outlaw any implementations that expose the rep or be less demanding. We choose to be less demanding and allow for some implementations that expose their subrep. In doing so, we choose not to associate a single type induction rule with a cluster specification, but rather a set of rules. We call this set of rules, the *type induction principle* of the cluster specification. Each rule is dependent on the form of a predicate, P(t), which we would like to assert holds true for all objects of type T between all pairs of successive states in any computation. In essence, the predicate is shown to be an invariant for the cluster specification. Since there is one rule per predicate, one could take an alternative viewpoint that we are associating a set of invariants with a cluster specification, where each invariant is a predicate corresponding to a rule.

Notice that hypotheses (1), (2), and (3) of the derivation method (Section 3.5.2.1) are independent of the form of the predicate P(t). However, an object of type T might contain objects of mutable type, M, and for any predicate containing a term that refers to *values* of these subobjects, the truth of the predicate depends on the behavior of all procedures that possibly change the values of objects of type M. We need to show that the predicate P(t) remains invariant for each mutating constructor of type M, and hence include a hypothesis for

each mc∈MC(M).

Thus, we add the following rule to the derivation of a type induction rule.

Method (continued): *Derivation of a Type Induction Rule*

4. For each subterm, τ, in P(t) that denotes an object of mutable type M ($\neq$ T) if τ is followed by $\uparrow$ or $\downarrow$, add a *τ-instance* (defined below) of HM for each mc∈MC(M).

(end of Method)∎

Def: Let P(t) be a predicate with t a free variable in P. Let τ be a subterm of P, and t be a subterm of τ, where τ denotes an object of type M. A *τ-instance* of HM for Pr and predicate P(t) is:

$$x_1 = \tau[v_1/t] \wedge \ldots \wedge x_n = \tau[v_n/t] \wedge [P[v_1/t] \wedge \ldots \wedge P[v_n/t]]$$
$$\{Pr\}$$
$$[P^f[v_1/t] \wedge \ldots \wedge P^f[v_n/t] \wedge P^f[v_{n+1}/t] \wedge \ldots \wedge P^f[v_{n+m}/t]]$$

where

1. Each v_i in $P[v_i/t]$ or $P^f[v_i/t]$ is a fresh variable. There is a v_i for each of Pr's input and output formals x_i of type M. We need these fresh variables because Pr might have more than one argument of type M.
2. $P^f[v_i/t]$ is $(P[v_i/t])^f$. I.e., substitute v_i for t; then flip.

Example 2

Suppose we specify the type stack of small sets, where sets are mutable, and that the identities of set objects are pushed onto the stack, not just their values. Figures 12 and 13[7] give the cluster specification for the type stack of small sets and for the trait it uses. The implementation of Figure 11 satisfies the cluster specification of Figure 12, even though the implementation exposes its subrep. An implementation that does not expose its rep, e.g., one in which the *read* procedure returns a copy of the top of the stack, would also satisfy the specification since the post-condition of the *read* procedure specification specifies only that

7. These two figures with minor variations are repeated in Appendix I for future reference.

```
stack = cluster is empty, grow, read
  uses StackOfSS
  provides immutable stack from SSS

        empty = proc () returns (st: stack)
          pre true
          post st↓ = null ∧ new st ∧ mutates Ø ∧ returns
          end

        grow = proc (s1: stack, s: set) returns (s2: stack)
          pre card(s1↑) < 64
          post s2↓ = push(s1↑, s) ∧ new s2 ∧ mutates Ø ∧ returns
          end

        read = proc (t: stack) returns (s: set)
          pre ~isNull(t↑)
          post s↓ = top(t↑)↑ ∧ mutates Ø ∧ returns
          end

  end stack
```

Figure 12. Stack Cluster Specification

---

the value of the set object returned be the same as the value of the top of the input stack object.

Suppose instead we specified in *read*'s post-condition:

$$s\downarrow = top(t\uparrow)\uparrow \wedge \underline{new\ s} \wedge \textbf{mutates}\ \emptyset \wedge \textbf{returns}$$

i.e., that not only the value of the set object returned be the same as the value of the top of the stack, but also that the set object be **new**, then the implementation of Figure 11 would not satisfy the specification.

Returning to the specification of Figure 12, for any predicate, P, involving the values of sets as well as the values of stacks, it would be incorrect to assume we could prove P without considering the cluster specification for sets--we must include hypotheses for all procedure specifications that mutate set objects.

```
StackOfSS: trait
  includes      SetOfInt,
                StackOfE with [SSS for C, set[int]_obj for E]

StackOfE: trait
  includes Integer
  introduces
        null: → C
        push: C, E → C
        top: C → E
        pop: C → C
        isNull: C → Bool
        isIn: C, E → Bool
        size: C → Int
  closes C over [null, push]
  constrains [C] so that for all [s: C, e: E]
        top(null) exempt
        top(push(s,e)) = e
        pop(null) exempt
        pop(push(s,e)) = s
        isNull(null) = true
        isNull(push(s,e)) = false
        isIn(null,e) = false
        isIn(push(s,e),e1) = if e .eq e1 then true else isIn(s,e1)
        size(null) = 0
        size(push(s,e)) = size(s) + 1
```

Figure 13. Traits for Stacks

Hence, our induction rule must include a hypothesis for the *delete* procedure specification of sets. For example, suppose we want to prove ~isNull(t↑) ⇒ [card(top(t↑)↑) < 64] for t of type stack. We have instances of HB and HP for *empty* and *grow* as follows:

HB true {empty} ~isNull(st↓) ⇒ [card(top(st↓)↓) < 64]
HP ~isNull(s1↑) ⇒ [card(top(s1↑)↑) < 64] {grow}
                ~isNull(s2↓) ⇒ [card(top(s2↓)↓) < 64]

We also need to add τ-instances of HM for the term, τ = top(t↑), since top(t↑) denotes an object of *mutable* type set and top(t↑) is followed by an ↑ in P. The *delete* procedure specification is the only mutating constructor of type set so we have a top(t↑)-instance of HM with the fresh variable, v1, substituted in for t in top(t↑).

HM s = top(v1↑) ∧ ~isNull(v1↑) ⇒ [card(top(v1↑)↑) < 64] {delete}
~isNull(v1↓) ⇒ [card(top(v1↓)↓) < 64]

The conclusion of this rule is true {S} ∀t:stack[set] ~isNull(t↓) ⇒ card(top(t↓)↓) < 64 for all statements S. We show the validity of the hypotheses of this rule in Appendix II.1.

If we do not include the hypotheses for mutating constructors of type set, we could possibly prove a statement that is not true. For example, suppose *SetClusSpec* has a procedure that mutates its input set argument by inserting integers into it. If called, this procedure could possibly change the value of a set pushed on the stack and we could not ensure that the size of all sets in the stack would be less than 64. If we had not included the hypothesis for this add procedure, we could have proved a false statement--that the size of the top of all stacks is less than 64.

3.6 Summary

In this chapter we gave a precise definition of when an implementation satisfies a specification in terms of their theories. We defined theories of specifications and implementations by precisely defining their formal systems. We also described in detail the derivation of a type induction principle associated with a cluster specification and gave examples of its use.

4. Extended Interface Language for CLU

In this chapter we describe some extensions to the kernel interface language that make it easier to read and write specifications, and some that make it easier to specify certain features particular to CLU. The design objectives in extending the kernel interface language were:

1. To enhance the readability of specifications,
2. To encourage a stylized form of writing specifications,
3. To be applicable to interface languages for other programming languages.

Section 4.1 presents four simple syntactic extensions. The prime motivation for introducing them is to enhance the readability of specifications. The meaning of each new construct is given a translation into the kernel language. For each extension we also give any necessary additions to the syntax and checking of specifications. Section 4.2 discusses extensions to both the syntax and semantics of the interface language to handle three features particular to CLU: own variables, iterators, and parameterization.

4.1 Simple Extensions

The assertions in the pre- and post-conditions of a procedure specification tend to be unwieldy and long. In order to streamline the appearance of each of these assertions and to highlight the significant ones (e.g., **mutates**), we introduce the following four changes to the kernel language: a default used trait, a separate **mutates** clause, a default termination condition value, and multiple pre- and post-conditions.

4.1.1 Default Used Trait

Naming the used trait in a procedure specification becomes optional. For a free procedure specification, since the theory of the used trait must include the theories of each of the used traits of the cluster specifications that define the used types of the procedure specification, we can always introduce a new trait that includes (in the Larch sense) the used traits associated with the used types. For bound procedure specifications, if the name of the used trait does not explicitly appear, we define the default used trait to be the used trait of the cluster specification to which the procedure specification is bound.

Syntax

ProcSpec ::= *ProcId* = *ProcHead* <*Link*> *ProcBody* **end**

Translation

For the following *free* procedure specification,

```
Pr = proc (...) returns (...) signals (...)
        pre P
        post Q
        end
```

let $\{Tr_1, ..., Tr_n\}$ be the set of used traits of the used types of the input and output arguments to Pr. The above translates to:

```
Pr = proc (...) returns (...) signals (...)
  uses Tr
        pre P
        post Q
        end
```

where Tr is the trait:

```
Tr: trait
  includes Tr1, ..., Trn
```

A *bound* procedure specification, Pr, appearing in a cluster specification, Cl,

```
Cl = cluster is ..., Pr, ...
  uses Tr
        ...
        Pr = proc (...) returns (...) signals (...)
                pre P
                post Q
                end
        ...
  end
```

translates to:

```
Cl = cluster is ..., Pr, ...
  uses Tr
        ...
        Pr = proc (...) returns (...) signals (...)
          uses Tr
                pre P
                post Q
                end
        ...
  end
```

4.1.2 Mutates Clause

We highlight a procedure's potential effect of mutation of objects by lifting from the post-condition a **mutates** assertion of the form **mutates** t1, ..., tn and setting it off as a clause on its own. If no explicit **mutates** clause appears, we conjoin the **mutates** Ø assertion to the post-condition.

Syntax

We modify the syntax to allow for a **mutates** clause:

ProcBody :: = *Triple*
Triple :: = *PreC* <*Muts*> *PostC*
Muts :: = **mutates** *Term* + ,

Recall that a procedure object assertion is of the form "P{Pr}Q" where P and Q are assertions; hence the syntax must still allow **mutates** assertions to appear in post-conditions.

Translation

A triple of the form:

pre P
post Q

where Q has no **mutates** assertion, translates to:

pre P
post Q ∧ **mutates** Ø

A triple of the form:

pre P
mutates *Term* + ,
post Q

where Q has no **mutates** assertion, translates to:

pre P
post Q ∧ **mutates** *Term* + ,

4.1.3 Default Termination Condition Value

We choose **normal** to be the default value for the **terminates** object of a procedure specification. If no **returns** or **signals** assertion appears in a post-condition, then there is an implicit **returns** assertion in that post-condition.

Translation

A procedure specification of the form:

```
Pr = proc (...) returns (...) signals (...)
        pre P
        post Q
        end
```

where Q has neither a **returns** nor a **signals** assertion translates to:

```
Pr = proc (...) returns (...) signals (...)
        pre P
        post Q ∧ returns
        end
```

Example

```
intersect = proc (s1: set, s2: set)
        pre true
        mutates s2
        post ∀i:Int [has(s2↓,i) = has(s1↑,i) ∧ has(s2↑,i)]
        end
```

This specification has an implicit used trait, a separate **mutates** clause, and an implicit termination condition value (i.e., **normal**). The reader should compare the above *intersect* procedure specification with that in Section 2.2.2.4.

4.1.4 Multiple Pre- and Post- Conditions

The behavior of a procedure can often be broken down into several cases depending on the input state. Demarcating these individual cases enhances the readability of the specification and also disciplines the specifier to consider all possible cases in a stylized way. We introduce the use of multiple pre- and post-conditions.

Syntax

We modify the syntax as follows:

ProcBody ::= *Triple* +

Translation

A procedure specification, Pr, of the form:

```
Pr = proc (...) returns (...) signals (...)
        pre P1
        post Q1

        ...

        pre Pn
        post Qn
        end
```

translates to:

```
Pr = proc (...) returns (...) signals (...)
        pre P1 ∨ ... ∨ Pn
        post (P1 ⇒ Q1) ∧ ... ∧ (Pn ⇒ Qn)
        end
```

We do not require that the pre-conditions cover all cases nor that they be disjoint.

Example

```
absVal = proc (i: int) returns (j: int)
         pre i↑ ≥ 0
         post j↓ = i↑

         pre i↑ < 0
         post j↓ = -i↑
         end
```

Multiple pre- and post-conditions are most useful in distinguishing among the various termination conditions of a procedure and in conjunction with an implicit **returns** assertion. Typically, one pre- and post-condition pair is written for each distinct termination condition.

Example

```
choose = proc (s: set) returns (i: int) signals (isEmpty)
         pre ~isEmpty(s↑)
         post has(s↑,i↓)

         pre isEmpty(s↑)
         post signals isEmpty
         end
```

The reader should compare the above *choose* procedure specification with that in Section 2.2.2.2.

4.2 Handling Other CLU Features

We have so far ignored the following three features of CLU: own variables, iterators, and parameterization. We discuss an own variable as a particular kind of "memory object" in Section 4.2.1, and the other two features in the subsequent two sections. We add some extensions to CLU computation sequences and to procedure invocations to handle memory and iterators, and we add a semantic check for one kind of restriction on type parameters of parameterized specifications.

4.2.1 Memory Objects

A procedure's behavior may depend on the values of objects in the input state not explicitly bound to the formals. We call these "memory objects." In CLU, for example, an own variable is an object whose value is "remembered" from invocation to invocation. In other programming languages, a global variable is an example of another kind of memory object accessible from all procedures.

We need to specify the behavior of a procedure with memory, which we cannot do in the framework presented so far. Hence, we extend the syntax and semantics of procedure and cluster specifications. We use CLU own variables to model these extensions.[8]

Specifying memory raises two problems. The first is that unlike for input and output formals, we need to be able to specify the possibility of changing the bindings of memory object identifiers. Thus far, we did not need to specify this because the effect of changing bindings of formals does not affect the bindings of the actuals. That is, except for own variables, bindings from CLU program variables to objects can be changed only through CLU assignment and not through procedure invocation. Hence, analogous to a **mutates** assertion for stating a possible change to the store component of a state, we introduce a **changes** assertion for stating a possible change to the environment component. One subtle difference between **changes** and **mutates** is that whereas only terms denoting mutable objects can follow the **mutates** keyword, identifiers for both immutable and mutable objects can follow the **changes** keyword.

---

8. As a matter of programming style, the use of own variables in CLU is discouraged because they add semantic complexity. Their use can always be avoided by retaining state information in a "dummy" cluster; however, own variables are often used to save overhead in extra procedure calls.

The second problem deals with keeping track of whether a memory object has been initialized. In CLU, initialization of a procedure's memory occurs at (possibly) the procedure's first invocation. It may not occur if the initialization code within the procedure is not executed (e.g., because of a conditional), in which case memory is left uninitialized. Hence, we associate with each memory object, x, an implicit memory boolean object that is initially false and denoted by the identifier x$init. If x$init is false, x is uninitialized; if true, x is initialized.

Syntax

We modify the syntax as follows:

ClusBody :: = ⟨*Rmbr*⟩ *ProcSpec* +
ProcBody :: = ⟨*Rmbr*⟩ *Quad* +
Rmbr :: = **remembers** *RemDecl* +
RemDecl :: = *ObjId*: *TypeSpec*
Quad :: = *PreC* ⟨*Chgs*⟩ ⟨*Muts*⟩ *PostC*
Chgs :: = **changes** *ObjId* + ,

The **remembers** clause simply allows the user to introduce object identifiers for memory. We emphasize that the declaration of memory objects in a specification does not imply the use of memory (e.g., own variables) in a corresponding implementation. As with a **mutates** assertion, we make a **changes** assertion a separate clause in the body of a procedure specification.

We add to the syntax of the assertion language,

Assn :: = ... | **changes** *ObjId* + ,

with truth value:

$$T[\textbf{changes } x1, ..., xn](\sigma, \sigma', A, \mu) =$$
$$\forall y\, [\sim(y = x1) \wedge ... \wedge \sim(y = xn) \Rightarrow (\sigma.e(y) = \sigma'.e(y))]$$

Checking

We check that

1. Object identifiers appearing in a **remembers** clause of a procedure specification, Pr, are disjoint from Pr's input and output formals.

2. Object identifiers appearing in a **remembers** clause of a cluster specification, Cl, are disjoint from the sets of input formals, output formals, and memory object identifiers of all of Cl's procedure specifications.

3. Only memory object identifiers can appear after the **changes** keyword.

Meaning

We treat memory objects as implicit input and output arguments to a procedure. We modify the structure of an operation (a relation-algebra pair) so that the domain and range of the environment components of the input and output states of the relation includes memory (compare with Section 2.2.2.1) and their corresponding "init" objects. Let *MemId* be the set {x | x is a memory object identifier} ∪ {x$init | x is a memory object identifier}, and let *MemObj* be the set of objects denoted by identifiers in *MemId*.

1. $dom(R) = \{\langle D, e, s\rangle \mid dom(e) = \text{set of input formals} \cup MemId \wedge ran(e) = \text{set of input arguments} \cup MemObj\}$

2. $ran(R) = \{\langle D, e, s\rangle \mid dom(e) = \text{set of output formals} \cup MemId \wedge ran(e) = \text{set of output arguments} \cup MemObj\}$

The first equation states that the environment of each input state includes the bindings from memory object identifiers to memory objects and the bindings for the corresponding "init" objects as well as the set of bindings from input formals (object identifiers) to input arguments (objects). The second equation states a similar property for the environment of each output state.

We add the following two properties to the initial state of a computation, σ_0, for all memory objects, x,

1. $\{x, x\$init\} \subseteq \sigma_0.O$
2. $\sigma_0.s(\sigma_0.e(x\$init)) = FALSE$

The first property states that all memory objects and their associated boolean "init" object are in the set of existing objects of the initial state. The second property states that the "init" objects are initialized to the boolean value false. Notice that since x$init denotes an immutable boolean object, it makes sense to change x$init, but not to mutate it.

Example

```
increment = proc () returns (j: int)
  uses Integer
  remembers ctr: int
        pre ctr$init↑ = false
        changes ctr, ctr$init
        post ctr↓ = 1 ∧ j↓ = 1 ∧ ctr$init↓ = true

        pre ctr$init↑ = true
        changes ctr
        post ctr↓ = ctr↑ + 1 ∧ j↓ = ctr↓
  end
```

The first time the *increment* procedure is called, the value of the integer object, *ctr*, is initialized to 1 and returned. Subsequent invocations will return successive integers.

4.2.2 Iterators

An iterator computes a sequence of *items* of objects, one item at a time, where an item is a set of zero or more objects. We amend our model of a computation sequence to include iterator invocations, which we treat similarly to procedure invocations. The only way an iterator can be invoked is by use of a **for** statement. The execution of the **for** statement includes one or more invocations of the iterator and is terminated when the iterator terminates.

```
elements = iter (a: array[int]) yields (int)
  next: int := array[int]$low(a)              %1
  while true do                               %2
        yield (a[next])                       %3
        next := next + 1                      %4
        end                                   %5
        except when bounds: return            %6
          end                                 %7
  end elements

flip_sign = proc (a: array[int]) returns (array[int])
  b := array[int]$create(array[int]$low(a))
  for i: int in elements(a) do
        addh(b, -i)
        end
  return (b)
  end flip_sign
```

Figure 14. Elements Iterator, Implementation and Use

An example of an *elements* iterator and its use are given in Figure 14. *Elements* computes a sequence of integers. The *flip_sign* procedure creates a new array with the same low bound as *a*, the input array, and returns an array with the signs of all the integers of *a* reversed. The first time *elements* is invoked, the integer at the low bound of *a* is yielded (statement 3). A subsequent invocation of *elements* yields the next integer of *a*. This process continues until a bounds exception is raised, in which case *elements* terminates (statement 6).

We need to distinguish between two kinds of termination for iterators. The first is when an iterator yields an item following an invocation from a **for** statement, e.g. statement 3 of *elements*. An alternate view of this situation is that the iterator does not "terminate," but is just in a "suspended" state. The additional piece of semantics we need for the specification of an iterator is a special termination condition. We reserve the identifier, **suspend** $\in$ *TermCond*, for the value of this termination condition, and we add a corresponding **suspends** assertion to the assertion language. The second kind of termination is when the

iterator returns, causing the **for** statement to terminate, e.g., statement 6 of *elements*. As with procedure specifications, we use the termination condition **normal** for this kind of termination.

Syntax

The syntax for an iterator specification is as follows:

IterSpec :: = *IterId* = *IterHead* ⟨*Link*⟩ *IterBody* **end**
IterHead :: = **iter** *Args* ⟨*Yields*⟩ ⟨*Sigs*⟩
IterBody :: = ⟨*Rmbr*⟩ *Quad* +
Yields :: = **yields** *Args*

As with a *Rets* clause in procedure specifications, an object identifier in a *Yields* clause is an *output formal*; the object it denotes is an *output argument*.

Recall that we list in the header of a cluster specification the identifiers of procedure specifications that are specified in the body. We also include iterator specifications in a cluster specification. We modify the syntax as follows:

ClusSpec :: = *TypeId* = **cluster is** *RoutId* + , *ClusLink ClusBody* **end**
ClusBody :: = *RoutSpec* +
RoutId :: = *ProcId* | *IterId*
RoutSpec :: = *ProcSpec* | *IterSpec*

A routine specification is either a procedure or iterator specification. *Bound* and *free* routine specifications are defined in a similar way to bound and free procedure specifications.

We add to the syntax of the assertion language:

Assn :: = ... | **suspends**

with truth value:

$\mathcal{T}$[**suspends**](σ, σ', A, μ) = σ'.s(**terminates**) = **suspend**

Checking

The syntax-checking of the body of an iterator specification is as defined for procedure specifications. A **suspends** assertion can appear in only post-conditions. We also allow the use of all syntactic amenities introduced in Section 4.1 for iterator specifications.

Translation

An iterator specification of the form:

```
It = iter (x1: S1, ..., xm: Sm) yields (y1: T1, ..., yn: Tn) signals (e1, ..., ep)
  uses Tr
       pre P
       post Q
  end
```

translates to:

```
It = proc (x1: S1, ..., xm: Sm) signals (suspend (y1: T1, ..., yn: Tn), e1, ..., ep)
  uses Tr
       pre P
       post Q
  end
```

Example

```
tokens = iter (s: stream) yields (t: token)
  uses StreamTrait
       pre ~isEmpty(s↑)
       mutates s
       post t↓ = head(s↑) ∧ s↓ = rest(s↑) ∧ suspends

       pre isEmpty(s↑)
       post returns
  end
```

Each time the iterator is invoked with a nonempty input *stream* object, *tokens* mutates the stream and yields a token from it. The specification does not forbid the possibility that *s* be changed in the body of a **for** statement. Recall that a **returns** assertion in the second

post-condition is equivalent to the assertion **terminates↓ = normal**.

Memory Used With Iterators

The specification of memory objects in iterator specifications requires making additions to our model of CLU computations. Because we are modeling each individual invocation of an iterator, and not each **for** statement that invokes an iterator, we need to be careful about specifying the effect of an iterator on its memory. In particular, initialization of memory for an iterator is done at the first invocation of that iterator in the first **for** statement of the computation that invokes it. Subsequent **for** statements that invoke it do not "reinitialize" memory.

We distinguish a *use* from an *invocation* of an iterator, Iter. Each **for** statement that invokes Iter is a *use* of it. Each iteration within a **for** statement that uses Iter is an *invocation* of it. For example, in Figure 14, flip_sign uses elements once but invokes it (possibly) many times.

Meaning

Let **first** denote a special memory object that enables us to distinguish the first invocation of an iterator from subsequent invocations in a **for** statement. We view **first** as a "global" or "ghost" variable accessible in all states in a computation. At the first invocation of each use of an iterator, **first** is true; otherwise, it is false. Therefore, at the first invocation of an iterator of each of its uses, **first** is true; at each intermediate invocation of each use, **first** is false. Immediately before each use **first** is true.

To achieve the desired effect of **first** being true before each use of an iterator, we associate an implicit assignment statement "**first** := true" before the (syntactic) appearance of each **for** statement in the program text. This ensures that if a statement, S_i, in a computation is the first invocation of an iterator the value of **first** is true in the state preceding

S_i. For a computation sequence,

$$\sigma_0 S_1 \sigma_1, ..., \sigma_{n-1} S_n \sigma_n$$

we have:

1. **first**$\in\sigma_0$.O
2. For all $i \geq 1$, if S_i is a first invocation of an iterator, σ_{i-1}.s(σ_{i-1}.e(**first**)) = *TRUE*;
   otherwise, σ_{i-1}.s(σ_{i-1}.e(**first**)) = *FALSE*;

We extend the domain and range of the relations of all iterators to include **first** as we did for other memory objects.

Syntax

Since we often need to check whether or not we are at the first invocation of an iterator, we add to the assertion language:

Assn :: = ... | **firstInv**

with truth value

$$T[\textbf{firstInv}](\sigma, \sigma', A, \mu) = \sigma.s(\sigma.e(\textbf{first})) = TRUE$$

We do not provide an assertion to check whether we are at the first use of an iterator for the same reason we do not provide an assertion to check whether we are at the first invocation of a procedure. The only reason we might (incorrectly) think we would need the ability to make these distinctions is because of the initialization of memory. Recall, however, that initialization of memory objects is not necessarily done at the first use of an iterator or at the first invocation of a procedure. It is necessary only to distinguish between whether memory has been initialized, which we can do using the "init" boolean object associated with each memory object.

We do provide two implicit assertions with iterator specifications. First, note that after the first invocation of any use of an iterator, the final value of **first** should be false, and after subsequent invocations, its value can remain false. Hence, we implicitly append the assertion **first↓** = false to each post-condition of a quadruple of an iterator specification.

Second, since one of the possible effects of an iterator invocation is to change the binding of **first**, we implicitly append **first** to the list of object identifiers of each **changes** clause in each quadruple of an iterator specification. If a **changes** clause does not explicitly appear, we implicitly include one in each quadruple.

Translation

A body of the form:

pre P
mutates M
post Q

where Q has no **changes** assertion, translates to:

pre P
changes first
mutates M
post Q ∧ **first↓** = false

A body of the form:

pre P
changes C
mutates M
post Q

translates to:

pre P
changes C, **first**
mutates M
post Q ∧ **first**↓ = false

Example

One use of memory with iterators is to specify that the initial value of an argument to the iterator is the same as the final value from the previous invocation.

elements = **iter** (s: set) **yields** (e: elem)
uses SetOfElem
remembers myset: set

pre ~isEmpty(s↑) ∧ [**firstInv** ∨ s↑ = myset↑]
mutates myset, s
post has(s↑,e↓) ∧ s↓ = remove(s↑,e↓) ∧ myset↓ = s↓ ∧ **suspends**

pre isEmpty(s↑) ∧ [**firstInv** ∨ s↑ = myset↑]
post returns
end

In the above *elements* specification, *myset* is a set object used to remember the value of the set object from invocation to invocation. The s↑ = myset↑ conjunct that appears in both pre-conditions requires that the initial value of the set object at each invocation be the same as the "remembered" value from the previous invocation. The first triple handles the cases when the set argument is not empty and either (1) it is the first invocation of *elements*, or (2) it is not the first invocation and the initial value of *s* is the same as the remembered value. The second triple handles the cases when *s* is either initially empty, i.e., at its first use, or becomes empty from the previous invocation of any of its uses.

4.2.3 Parameterized Specifications

Procedures, iterators, and clusters may all be parameterized in two ways: over certain types of objects and over type identifiers. We call a parameter of the first kind an *object parameter*; the second, a *type parameter*. An integer object parameter, n, for example, can be

used in a procedure that computes the average of a list of numbers, where n is the length of the list. Type parameters are far more common in CLU than object parameters. A *list* cluster, for example, can be parameterized over a type parameter, T, to stand for a set of clusters, each defining a list[A] type for some actual type identifier, A. Type parameters can also have restrictions. In Section 4.2.3.1 we discuss parameterized specifications without restrictions; in Section 4.2.3.2 we describe the kinds of restrictions that we can impose on type parameters.

4.2.3.1 Parameterization Without Restrictions

Syntax

We modify the syntax as follows:

ProcHead :: = **proc** ⟨*Parms*⟩ *Args* ⟨*Rets*⟩ ⟨*Sigs*⟩
IterHead :: = **iter** ⟨*Parms*⟩ *Args* ⟨*Yields*⟩ ⟨*Sigs*⟩
ClusSpec :: = *TypeId* = **cluster** ⟨*Parms*⟩ **is** *RoutId* + , *ClusLink ClusBody* **end**
ClusMap :: = **provides** *MutFlag TypeId* **from** *SortId*

Parms :: = [*ParmDecl* + ,]
ParmDecl :: = *ObjId*: *TypeSpec* | *Idn*: **type**
Where :: = **where** *Restriction* + ,

Object parameters are of the form *ObjId*: *TypeSpec*; type parameters are of the form *Idn*: **type**. Parameters of a procedure or iterator specification should not be confused with the input and output *formals* (object identifiers) of the specification, nor with objects bound to the formals.

Checking

1. Object parameters are of only the following types: null, bool, int, real, char, and string.

2. The body of a parameterized specification sort checks. For a term, τ, denoting an object of type T, where T is a type parameter, the sort of τ is T_obj. The sort of the terms, $\tau\uparrow$ and $\tau\downarrow$, is TtoS(T). As usual, the names of these sorts must appear in the used trait.

Meaning

A model of a parameterized procedure specification is a set of operations (relation-algebra pairs). Each operation in the set is a model of an *instantiated* specification, obtained by textually substituting a list of actual parameters, A, for the list of (object and type) parameters, F, of the parameterized procedure specification. For the following parameterized procedure specification,

```
Pr = proc [F] (InList) returns (OutList) signals (SigList)
  uses Tr
       pre P
       post Q
       end
```

an instantiated specification is of the form:

```
Pr[A] = proc (InList [A/F]) returns (OutList [A/F])
                                    signals (SigList [A/F])
  uses Tr'
       pre P [A_obj/F_obj, TtoS(A)/TtoS(F)]
       post Q [A_obj/F_obj, TtoS(A)/TtoS(F)]
       end
```

where Tr' is the trait,

```
Tr': trait
       includes Tr with [A_obj for F_obj, TtoS(A) for TtoS(F)]
```

We adopt the convention of naming each of these instantiations "Pr[A]." We do the renamings in the pre- and post-conditions because sort identifiers can appear in quantified

expressions in the assertions. The first list of renamings handles *obj sort* identifiers; the second, *value sort* identifiers.

A model of a parameterized cluster specification is a set of abstract data types (recall that an abstract data type is a pair consisting of a set of objects and a set of operations). Each abstract data type is a model of an instantiated cluster specification. For the parameterized cluster specification (MutFlag is either the keyword **mutable** or **immutable**),

```
C = cluster [F] is RoutIdList
  uses Tr
       provides MutFlag C from S
       RoutSpecs
  end
```

each instantiation is of the form:

```
C[A] = cluster is RoutIdList
  uses Tr'
       provides MutFlag C[A] from S
       RoutSpecs [A/F, A_obj/F_obj, TtoS(A)/TtoS(F)]
  end
```

where again Tr' is the trait,

```
Tr': trait
       includes Tr with [A_obj for F_obj, TtoS(A) for TtoS(F)]
```

The first list of renamings for RoutSpecs (A/F) is used to rename type identifiers in the headers; the second and third lists are used to rename the sort identifiers in the pre- and post-conditions of each of the routine specifications. We adopt the convention of naming each of these cluster specifications "C[A]." Notice that each type, *C[A]*, maps to the same sort identifier, *S*.

Example

The following is a parameterized set cluster specification:

```
set = cluster [T: type] is ..., insert, ...
  uses SetOfT
  provides mutable set from ST

      ...
      insert = proc (s: set[T], t: T)
               pre true
               mutates s
               post s↓ = add(s↑,t)
        end
      ...
      end
```

where the *SetOfT* trait is given below using the *SetOfE* trait of previous chapters.

```
SetOfT: trait
  includes SetOfE with [ST for C, T_obj for E]
```

An instantiation of the above parameterized cluster specification is as follows, where the actual type identifier is *int*, and *SetOfT'* is the *SetOfT* trait with *int_obj* substituted for *T_obj*.

```
set[int] = cluster is ..., insert, ...
  uses SetOfT'
       provides mutable set[int] from ST

      ...
      insert = proc (s: set[int], t: int)
               pre true
               mutates s
               post s↓ = add(s↑,t)
        end
      ...
      end
```

4.2.3.2 Parameterization With Restrictions

We often find it useful to place restrictions on type parameters. These restrictions play a similar role to that of the assumptions of a trait in Larch. We write these restrictions in a *Where* clause. We modify the syntax:

Syntax

ProcHead :: = **proc** ⟨*Parms*⟩ *Args* ⟨*Rets*⟩ ⟨*Sigs*⟩ ⟨*Where*⟩
IterHead :: = **iter** ⟨*Parms*⟩ *Args* ⟨*Yields*⟩ ⟨*Sigs*⟩ ⟨*Where*⟩
ClusSpec :: = *TypeId* = **cluster** ⟨*Parms*⟩ **is** *RoutId* + ,
⟨*Where*⟩ *ClusLink ClusBody* **end**

Where :: = **where** *Restriction* + ,
Restriction :: = *BasicRestriction* | *TypeId* **in** *TypeSet*
BasicRestriction :: = *TypeId* **immutable** | *TypeId* **has** *RoutHead*
| *TypeId* **has** *RoutSpec*
TypeSet :: = {*TypeId* | *BasicRestriction* + ,}
RoutHead :: = *ProcHead* | *IterHead*

The **where** clause is removed upon an instantiation of a parameterized specification. The "|" symbol[9] in the *TypeSet* production should not be confused with the "|" symbol used as an alternative separator in the grammar.

Checking

We check that the actuals substituted for type parameters satisfy the restrictions in the **where** clause. There are four kinds of restrictions on a type parameter. Three are "basic" restrictions, two of which require only syntax checks; the third requires a semantic check. The fourth kind of restriction is built up from these basic restrictions and hence, may also require semantic checks. In the following discussion on these four restrictions, let T be a type parameter, A be a type, and Cl_A be the cluster specification defining A.

---

9. It is a reserved symbol in CLU.

The first kind of restriction is of the form, T **immutable**. To check that A satisfies this restriction, we check that the type flag of CI_A is **immutable**. It is not a kind of restriction that can be placed on type parameters in CLU, but we include it in the specification language because proofs (e.g., those that use the type induction principle of A) may depend on a type being immutable.

The second kind of restriction is of the form, T **has** R = Sig, where R is in *RoutId* and Sig is in *RoutHead*. To check that A satisfies this restriction, we check that CI_A contains a routine named R with the signature Sig.

The third kind of restriction, stricter than the second, is of the form, T **has** R, where R is in *RoutSpec* (R includes a signature and a body). To check that A satisfies this restriction, we check that the theory of R is a subset of the theory of A. This restriction is not present in CLU because it involves semantic checking. The second kind of restriction is a special case of the third where the pre- and post-conditions are both identically true.

The fourth kind of restriction is included for completeness since it is allowed, but rarely used, in CLU. It is of the form, T **in** {X | X **has** r1, ..., rn}, where r1, ..., rn are restrictions of the three forms just described. To check that A satisfies this restriction, we check that A satisfies all the restrictions, r1, ..., rn.

Examples

```
set = cluster [T] is ...
  where T has
        equal = proc (t1, t2: T) returns (b: bool)
                pre true
                post b↓ = (t1 = t2)
          end

  uses SetOfT
        provides mutable set from ST
        ...
  end
```

The implementations that satisfy this specification would differ from those that would a specification in which the post-condition of *equal* was replaced by

```
post b↓ = (t1↑ = t2↑)
```

The difference is that the first specifies that the elements to the equal procedure be the same objects whereas the second specifies only that the elements have the same value. There would be fewer implementations satisfying each of these two specifications than those satisfying a specification in which we do not specify the behavior of *equal* at all.

5. Evaluating Specifications

In the incremental development of a large specification, providing useful feedback to a specifier can increase his confidence that his specification is on the right track. For example, a specifier may wish to know if his specification is in some sense "correct," i.e., that it captures his intuition of what he is trying to specify, or that it is in some sense "good," i.e., that it satisfies a set of desired objective and possibly subjective properties.

We distinguish a *specification* from what it specifies, i.e., from the *specificand set* of a specification [Guttag82]. Providing feedback to a specifier may help him better understand both the specification and its specificand set, and consequently may cause him to modify or improve the specification. Depending on how informative the feedback is, it may even point to a place in the specification where an improvement can be made.

One way of providing such feedback is to provide the specifier ways of evaluating a specification. In this chapter, we consider two forms of evaluation: checking specifications for various properties, and comparing specifications with respect to various qualities. For example, we might like to check if a specification is *consistent* or compare the *strength* of two specifications.

Checking is performed on a single specification; in Section 5.1 we discuss checking for the following four properties: consistency, full-coverage, determinism, and protection. Comparing is performed on two specifications; in Section 5.2 we discuss comparing two specifications with respect to the quality strength. In Section 5.3, we discuss checking a specification for a property, essentiality, with respect to a theory. All definitions are in terms of theories.

We do not give an extensive enumeration of properties and qualities, but just a sample to suggest the usefulness of evaluating specifications and to illustrate our approach. We leave for future work the tasks of identifying and defining additional properties and qualities,

analyzing the tradeoffs among them, and finding other methods of evaluating specifications.

5.1 Properties of Specifications

Following our specification approach, we put together pieces of existing specifications to create a larger specification targeted for a particular problem or problem domain. As the specification grows incrementally, we might invoke a "checker" to test for a property of the specification. In the process of tuning a specification, we would probably invoke such a checker many times. If a specification does not have a property, we can choose either to modify the specification so that it does, or accept the fact that it does not--a checker is used only to provide information, not to inhibit the progress of writing the specification. Checking for a property might also necessitate a clarification in the client's problem statement. For example, discovering that a specification is inconsistent may point to a contradiction in the problem statement--the specification merely reflected the mistake. The signatures of the properties we will discuss are shown in Figure 15.

Two properties of a specification that might be of interest are consistency and completeness. The ability to check for consistency is probably of more use than the ability to check for completeness. Knowing a specification is inconsistent informs the specifier that no

---

consistent: trait → boolean
consistent: procedure specification → boolean
consistent: cluster specification → boolean

fully-covering: procedure specification → boolean
fully-covering: cluster specification → boolean

deterministic: procedure specification → boolean
deterministic: cluster specification → boolean

protective: procedure specification → boolean
protective: cluster specification → boolean

Figure 15. Signatures of Properties

implementation could be written to satisfy the specification. We define consistency in Section 5.1.1.

We do not define completeness because we expect most specifications to be incomplete in the logical sense[10] as well as in the practical sense--in the development of a large specification, we may have no intention of ever "finishing" it. We usually want to know when we have said "enough" as opposed to "everything." In Sections 5.1.2-5.1.4 we define three properties: full-coverage, determinism, and protection. Each gets at a different notion of sufficiency as a different kind of approximation to completeness.

For each property, we first motivate it, then define it, and then discuss specifications with that property. When we define each property we also motivate our definition.

5.1.1 Consistency

5.1.1.1 Definition

The usual notion of consistency of a formal system refers to the inability to derive an explicit contradiction. For a given first-order predicate logic formal system, a set of formulae, φ, is *inconsistent* if and only if for some A, both A and ~A are theorems in φ. Equivalently, φ is *inconsistent* if and only if false is in φ. We will use the second definition to build the notion of an inconsistent specification.

Def: A trait, Tr, is *inconsistent* if and only if the formula (true = false) or the formula false is in Th(Tr).

Def: A procedure specification, Pr, is *inconsistent* if and only if (1) there exists a satisfiable formula P such that the formula P{Pr}false is in Th(Pr), or (2) Pr's used trait is inconsistent.

Def: A cluster specification, Cl, is *inconsistent* if and only if (1) true{S}false is in Th(Cl), or (2) for any of Cl's procedure specifications, Pr, there exists a satisfiable formula P such that the formula P{Pr}false is in Th(Cl), where P is satisfiable, or (3) Cl's used trait is inconsistent.

---

10. Given a formal system, its theory is *complete* if for all formulae, F, we can determine whether F or ~F is in the theory.

Def: A specification is *consistent* if and only if it is not inconsistent.

Checking for consistency is in general undecidable since first-order logic is undecidable. Under certain conditions, however, we may be able to show that a specification is consistent or inconsistent. For example, for equational theories, on which trait theories are based, a semi-decision procedure exists that checks for inconsistency by generating the contradiction true = false (and checks for consistency by generating true) for some sets of equations when treated as sets of rewrite rules [Knuth69, Musser77].

From the way we construct procedure and cluster specifications, it would be useful to know under what conditions putting smaller consistent pieces together results in a specification that is guaranteed to be consistent, or, on the other hand, to know when inconsistencies may be introduced.

A procedure or cluster specification cannot add formulae that would be inconsistent with a consistent used trait. The theory of a procedure specification is a conservative extension of the theory of its used trait; it adds formulae only of the form P{Pr}Q, and none of the form t1 = t2 or ∀x:S P(x). Therefore, the procedure specification cannot add the formula true = false or false, either of which would be inconsistent with a consistent trait.

To check a procedure specification for consistency, if the used trait is consistent, we need to check only that no formula P{Pr}false, where P is a satisfiable predicate, is in Th(Pr). Notice also we define inconsistency of a procedure specification in terms of Th(Pr) and not Th(Pr+) so as not to include the theory of the defined type when Pr is a bound procedure specification. Since the theory of a cluster specification is defined in terms of the theories of its procedure specifications, we avoid a circular definition.

To check a cluster specification for consistency, if the used trait is consistent, we need to check that each bound procedure specification is consistent and that their union is consistent (both cases covered by clause 2 of the definition of an inconsistent cluster

specification), and that the addition of the type induction principle for the defined type does not introduce any inconsistencies (covered by clause 1). This matches our intuition since even if the theories of the procedure specifications are individually consistent, their union may not be; moreover, an additional rule of inference may be used to introduce an inconsistency.

5.1.1.2 Consistent Specifications

Consistency is a desirable property of all specifications. Inconsistent specifications are more common than one might imagine, as the following example illustrates.

```
intersect = proc (s1, s2: set) returns (s3: set)
  uses SetOfInt
       pre true
       post ∀i:int [has(s3↓,i) = has(s1↑,i) ∧ has(s2↑,i)]
       end
```

Suppose *intersect* is a free procedure specification. We show that Th(*intersect*) is inconsistent, given the set cluster specification is *SetClusSpec*. It is inconsistent because there is no set object that can be returned as the intersection of disjoint input arguments. Notice that step 5 uses the theorem, true {intersect} ∀s:set card(s↓) > 0, from Th(*set*) derivable from the type induction principle for sets.

1. Let s1↑ = add(empty,1) ∧ s2↑ = add(empty,2).
2. true {intersect} ∀i [has(s3↓,i) = has(s1↑,i) ∧ has(s2↑,i)]
   --axiom of Th(*intersect*)
3. true {intersect} ∀i [has(s3↓,i) = has(add(empty,1),i) ∧ has(add(empty,2),i)]
   --simplified invocation rule with the substitution as indicated
4. true {intersect} card(s3↓) = 0
   --∀x:SI [∀i:Int has(x,i) = false ⇒ card(x) = 0] ∈ Th(*SetOfInt*)
5. true {intersect} ∀s:set card(s↓) > 0
   --Induction rule from Th(*set*)
6. true {intersect} ∀:s card(s↓) > 0 ∧ card(s3↓) = 0
   --conjunction of two post-conditions (Hoare proof rule)
7. true {intersect} false
   --Let s = s3.

Notice that if *intersect* were bound, it would be consistent because the theorem of step 5 would no longer hold. Th(*set*) would be different (e.g., we could construct an empty set object) because it would include Th(*intersect*) and so *set*'s type induction principle would have a weaker form.

5.1.2 Full-Coverage

In this section and the next two, we will define three properties that are related to the "completeness" property of a specification. These three represent examples of the kinds of approximations to completeness a specifier might want to check of his specification.

A common error in programming is forgetting to cover all the cases. As a result, a program may behave in an erroneous or surprising manner on some inputs. We would like to be able to prevent the occurrence of these errors before coding begins, i.e., in the design phase, by making sure our specification covers all the cases that can arise. For example, the following specification,

```
search = proc (a: array, e: elem) returns (index: int)
  uses ArrayOfElem
       pre isSorted(a↑)
       post e↑ = fetch(a↑, index↓)
       end
```

is not fully-covering because the case for the unsorted array is not covered. A checker for full-coverage invoked on *search* might prompt us to add another pre/post pair to handle the unsorted array.

Unlike consistency, however, full-coverage is not always desired. We may intentionally want to leave some cases unspecified because we know they will never arise or because we want to let the programmer decide how to handle them. In the example above, we may decide not to add another pre/post pair if we expect *search* to be invoked always with a sorted array.

5.1.2.1 Definition

We want the definition of full-coverage to capture the notion that the behavior of a procedure is specified for all "reachable" input states. In terms of models, a procedure is *fully-covering* if the domain of the input-output relation of any operation modeling a procedure is the entire set of states, $\Sigma(Val)$. One way of capturing the notion of full-coverage of a procedure specification in terms of theories is that if the pre-condition of the procedure specification is equivalent to true, then the relation is defined for all input states, and so the procedure specification is fully-covering. That is,

Def: A procedure specification, Pr, is *fully-covering* if and only if true {Pr} Pr.post is in Th(Pr+).

Def: A cluster specification is *fully-covering* if and only if all its procedure specifications are fully-covering.

5.1.2.2 Fully-Covering Specifications

A specification may not appear to be fully-covering when it is. Consider *SetClusSpec*, in which each of its procedure specifications, in particular, *delete*, is fully-covering. Although the disjunction[11] of *delete*'s pre-conditions is not identically true, it is provably true from the Th(set), which is contained in Th(delete+). The proof that *delete* is fully-covering would use the theorem, true {S} ∀x:set card(x↓) > 0, which comes from the type induction principle for *SetClusSpec*.

In practice, writing a procedure specification that is fully-covering is similar to generating sufficient test cases for a program [Goodenough75, McMullin82]. A helpful guideline to follow is for the specifier to use in a stylized manner, multiple pre/changes/mutates/post quadruples in conjunction with **signals** assertions (for multiple

---

11. Recall from Chapter 4 that the appearance of multiple pre-conditions translates to the disjunction of all the pre-conditions.

termination conditions) to cover all the cases. If one pre-condition places a restriction on the input state, then other pre-conditions should cover the cases for which the restriction does not hold. For each separate case, there is typically a different termination condition. As a result, the behavior of the procedure is "fully" specified.

5.1.3 Determinism

In specifying a program, it is not always easy to separate decisions that should be made at design time from those that should be delayed to implementation time. A specification should impose as few constraints as possible to avoid unnecessarily overspecifying the behavior of the program. An intentional lack of constraint can be regarded as an intentional incompleteness.

Nondeterminism gets at the notion of introducing an intentional incompleteness in a specification. It says that the values of input and output objects of a procedure specification are not predictable in the final state. A nondeterministic specification allows the implementor the freedom to choose the most convenient (e.g., efficient to implement) values. For example, in implementing a *choose* procedure for sets, returning the last integer inserted may be more efficient than returning the largest integer.

In contrast, *determinism* requires that the final values of the input and output objects be predictable. Whereas the fully-covering property deals with the "completeness" of a specification with respect to input states, *determinism* deals with it with respect to output states.

5.1.3.1 Definition

A specification is deterministic if for each state that satisfies the pre-condition, only one set of final values for the input and output objects satisfies the post-condition. We define this property in terms of theories, analogously to the usual definition for a function. A relation, f,

on X × Y is a partial function if for all x∈X, y1,y2∈Y [(⟨x, y1⟩∈f ∧ ⟨x, y2⟩∈f) ⇒ y1 = y2]. For determinism, we require the relation between the values of input and output objects defined by a procedure specification to be a partial function.

Let X be the list of input formals and Y be the list of output formals for the procedure specification Pr. To simplify the following discussion and definitions, we will treat memory objects as (implicit) input objects and require that all memory object identifiers be included in X. All formals in the **signals** clauses are included in Y (by definition). Let Pr.pre(X↑) be the pre-condition on the initial values of input objects, and Pr.post(X↑, X↓, Y↓) be the post-condition on the initial and final values of input and output objects.

Def: A procedure specification, Pr, is *deterministic* if and only if Th(Pr+) contains the following formula:

∀ A, A1, A2: T-in, B1, B2: T-out
    Pr.pre(A↑) ⇒
        [Pr.post(A↑, A1↓, B1↓) ∧ Pr.post(A↑, A2↓, B2↓)] ⇒
            A1↓ = A2↓ ∧ B1↓ = B2↓.

where T-in is the list of types of the input objects and T-out is the list of types of the output objects.

Def: A cluster specification is *deterministic* if and only if all of its procedure specifications are deterministic.

Def: A specification is *nondeterministic* if it is not deterministic.

Recall that a state consists of not only a store (mapping from objects to values), but also a set of (existing) objects, and an environment (mapping from object identifiers to objects). The definition of deterministic places no constraints on the set of objects or the environment of the final states. A more restrictive definition could require that for each input state in which the pre-condition is satisfied, there exists a unique output state in which the post-condition is satisfied--restricting the set of output states satisfying a post-condition to be a singleton set. We see no reason, however, to rule out a procedure that may, for example, create in the process of execution new objects that may be inaccessible upon termination of the

procedure. Similarly, we should not rule out a procedure that may change the bindings of its formals since those changes are not observable outside the procedure. In these cases, the sets of objects or the environments of the possible output states satisfying the post-condition may differ.

5.1.3.2 Deterministic Specifications

A specifier may intend a specification to be deterministic or not. A procedure specification may turn out to be nondeterministic because of an unintentional oversight on the part of the specifier. The following procedure specification,

```
choose1 = proc (s: stack) returns (i: int)
  uses StackOfInt
       pre ~isNull(s↑)
       mutates s
       post i↓ = top(s↑)
       end
```

is nondeterministic--the final value of *s* is indeterminate because of the presence of the **mutates** clause. To make *choose1* deterministic, the specifier could add the conjunct s↓ = pop(s↑) to the post-condition, or remove the **mutates** clause. On the other hand, the specifier may have intended to let the implementer decide whether or not to pop the stack, and therefore may have intended *choose1* to be nondeterministic.

Checking for determinism requires showing that a formula is in a theory; checking for nondeterminism, that it is not. A specifier could show the latter by assuming the formula is in the theory and finding a contradiction to show otherwise. For example, the following procedure specification,

```
choose2 = proc (s: stack) returns (i: int)
  uses StackOfInt
       pre ~isNull(s↑)
       post isIn(s↑, i↓)
       end
```

is nondeterministic. Suppose

$$\forall s{:}stack,\ i1,i2{:}int \\ \sim isNull(s\uparrow) \Rightarrow \\ [isIn(s\uparrow,i1\downarrow) \wedge isIn(s\uparrow,i2\downarrow) \wedge \textbf{mutates } \emptyset] \Rightarrow \\ [i1\downarrow = i2\downarrow]$$

is in Th(*choose2* +). Then let s↑ be push(push(null, 5), 7), i1↓ be 5, and i2↓ be 7 to derive a contradiction.

5.1.4 Protection

By partitioning a specification into two tiers, we can avoid at the top tier an incompleteness at the bottom tier. In particular, a procedure specification should be able to use a trait even if the trait is not sufficiently-complete [Guttag75]. It is the procedure specification's responsibility to *protect* any of its users from the incompletenesses of the trait by ensuring that the meaning of the procedure specification is independent of those incompletenesses.

Axioms of the form "τ **exempt**" are included in a trait to inform the specifier of an intentional incompleteness. We would like to ensure such incompletenesses do not show through to the interface level. For example, since the axiom *top(null)* **exempt** is in the *StackOfInt* trait, the following procedure specification is not protective.

```
read1 = proc (st: stack) returns (i: int)
  uses StackOfInt
       pre true
       post i↓ = top(st↑)
       end
```

If the initial value of st were null, then the incompleteness of the stack trait would show

through to the interface level because the value of the integer returned would be denoted by the **exempt** term *top(null)*.

Factoring a specification into two tiers allows us to factor our checks as well. If upon checking a trait for sufficient-completeness, we discover it is not sufficiently-complete, we may be inclined to invoke our checker for protection. For example, invoking a checker for protection on *read1* might cause us to modify it to be:

```
read2 = proc (st: stack) returns (i: int)
  uses StackOfInt
       pre ~isNull(st↑)
       post i↓ = top(st↑)
       end
```

Read2's pre-condition is sufficiently strong so that the value of the returned integer object would never be denoted by the term *top(null)*; hence, the incompleteness at the trait level would not show through to the interface level.

5.1.4.1 Definition

We say that a procedure specification is *protective* if it is independent of the set of **exempt** terms of its used trait. We build up to the definition of protection by first characterizing the set, E(Tr), of **exempt** terms of a trait, Tr, and then defining "independent of a set of terms."

Def: For a trait, Tr, the *set, E(Tr), of exempt terms of Tr* **is**

E(Tr) = {t | ∃t'∃u such that (t' = u)∈Th(Tr), where t' is a subterm of t,
and u is an instantiation of a term appearing **exempt** in Tr}

E(Tr) includes all terms that have a subterm that is provably equal to an instantiation of an **exempt** term. For example, for the *StackOfE* trait (Appendix I, Figure 13), E(*StackOfE*) = {top(null), pop(null), size(top(null)), top(pop(push(null,e))), ...}. E(Tr) does not include terms about which the trait does not say anything. For example, if the last equation in *StackOfE* were removed, it then would not constrain the term *size(push(s,e))*. The reason we do not

include these kinds of terms in E(Tr) is that given a set of axioms in a trait, we cannot, in general, generate all the terms that are "intentionally" and "implicitly" not constrained. It is easy, however, to know what terms are explicitly **exempt**.

We now give the definition of "independent of a set of terms." Intuitively, it captures the notion of never having to deal with certain terms. We follow it with the definition of protection.

Def: Let S be a set of terms. An assertion, A, appearing in Pr is *independent of S*, if
1. No subterm of A is in S, or
2. $\exists B$ ([A $\Leftrightarrow$ B] $\in$ Th(Pr)), and B is independent of S.

Def: Pr is *protective* if
1. Pr.pre is independent of E(Tr), and
2. Pr.pre $\Rightarrow$ Pr.post is independent of E(Tr).

Def: A cluster specification is *protective* if each of its procedure specifications is protective.

5.1.4.2 Protective Specifications

Protection is a desirable property of an interface specification. The specification should not be dependent on properties of the values denoted by **exempt** terms, and in reasoning about it the specifier does not want to be "stuck" with terms that are **exempt**. If upon checking to see if a specification is protective, we find that it is not, we may be able to find the dependency in the specification and then fix the specification to remove it.

Checking may require some cleverness on the specifier's part. It may involve finding an assertion equivalent to the one being shown independent of a set of **exempt** terms. Checking that the pre-condition is protective is usually easy because pre-conditions are usually simple. Checking the post-condition, however, is likely to be more difficult. Consider again the following example:

```
read2 = proc (st: stack) returns (i: int)
  uses StackOfInt
       pre ~isNull(st↑)
       post i↓ = top(st↑)
       end
```

To show that *read2* is protective, we show that it is independent of the set of terms E(*StackOfInt*).

1. Show ~isNull(st↑) is independent of E(*StackOfInt*). Trivial.

2. Show ~isNull(st↑) ⟹ i↓ = top(st↑) is independent of E(*StackOfInt*). Referring to part (2) of the definition of when an assertion is independent of a set of terms, let B be [isNull(st↑) ∨ ∃s1:SI, i1:Int [st↑ = push(s1,i1) ∧ i↓ = i1]].

In practice, writing a protective procedure specification is straightforward provided that the trait is actually strong enough to specify the desired properties. Strong enough pre-conditions are written to make sure that even if a post-condition alone is not independent of an exempt term, the assertion "Pre ⟹ Post" is. Often enriching the set of functions of the used trait makes it easier to read and write pre-conditions to handle these cases. For example, the function *isNull* is included in the *StackOfInt* trait instead of writing in the pre-condition the equivalent assertion, ~(st↑ = null).

5.2 Comparing Specifications

In the context of developing a large specification, one kind of evaluation we intend to perform is to compare specifications. For example, we might want to compare specifications with respect to their restrictivity, concision, elegance, or lucidity. (Judging a specification for some of these qualities is purely subjective, e.g., elegance and lucidity, and so we would not attempt to define these qualities formally.) We might invoke a "comparator" to compare specifications with respect to these qualities. As with checkers, we would invoke a comparator many times during the development of the specification. Comparators can be used to help us decide between two specifications. For example, we often want to choose the less restrictive (constraining) of two specifications. Comparators can also be used to check

whether a change we make to a specification had some expected or unexpected effect on one of its qualities. For example, if we add something to a specification, we might like to know whether we have made it more restrictive or left its restrictivity unchanged.

We discuss comparing specifications with respect to one quality, *strength*, of which *restrictivity* is a special case. Figure 16 gives the signatures of the corresponding comparators. In Section 5.2.1 we motivate comparing the relative strength between specifications. In Section 5.2.2 we define strength. In Section 5.2.3 we discuss the effect certain modifications to a specification has on its strength.

5.2.1 Comparing Strength

Intuitively, the stronger or more restrictive a specification, the fewer the number of implementations that satisfy it. In writing a specification, we may want to know whether one specification is *as strong as* or *stronger* than another. We may discover that after modifying a specification the new one is *incomparable* to the original.

There are at least two situations in which it is useful to know when a specification is as strong as another. One is where we modify a specification but want to ensure its strength is unchanged. For example, if we rename identifiers of a specification in order to have mnemonic names, we would want to make sure we have made only a syntactic and not a semantic change. A second situation is in determining if it is permissible to replace a specification with another without affecting any of its users. If one specification is as strong as another, then under certain circumstances we should be able to substitute one for the

---

as strong as: specification, specification → boolean
stronger: specification, specification → boolean
restrictive: specification, specification → boolean

Figure 16. Signatures of Comparators

other. Comparing the strengths of the two specifications can help determine legality of replacement. This situation is addressed in [Bloom83] in the context of distributed programs.

Sometimes, we may want a stronger specification. We might realize the specification is not strong enough in trying to prove a property of the specification or its specificand set. We could choose to either weaken the statement of what we were trying to prove or strengthen the specification. If we were to decide to strengthen the specification, we might want to compare the new and original specifications to make sure we did not make them incomparable. For example, if we were unsuccessfully trying to prove a cardinality property about sets based on a specification for bags, we might realize that either our axioms are not sufficient to prove it or that they are wrong. We might choose to strengthen the specification for bags to obtain one for sets that allows us to prove the desired cardinality property. When we discuss the essentiality of a specification in Section 5.3, we rely on the notion of strength in determining whether a specification is strong enough to prove some property.

5.2.2 Definition of Strength

The intuition we want to capture formally is that the stronger the specification, the fewer the number of implementations that satisfy it. We borrow the analogous concept from logic that the stronger a theory, the fewer the number of models that satisfy it, and define a *strength* relation between specifications in terms of strength between their theories. For example, the theory of $\langle Z, +, -\rangle$ is as strong as $\langle N, 0, succ\rangle$, but not vice versa, where Z is the set of all integers, and N is the set of all natural numbers.

We could define a theory, Th1, to be as strong as or stronger than another theory, Th2, if the two theories are in the same language and Th2 $\subseteq$ Th1. Theory containment, however, is not sufficient to capture the notion of relative strength between two theories for two reasons. The first is that the two theories may be in different languages; thus, they may be disjoint, but still be as strong as each other. The second is that even if the two theories are in the same

language, a formula that is in Th1, but not in Th2, may be translatable to one in Th2; thus Th1, although larger, may not be stronger than Th2.

In general, even if the theories are in different languages, there may exist a way of translating from one language to the other such that theorems of Th1 are translations of theorems of Th2. One reasonable way of translating from one language, L1, to another, L2 is to map symbols of L1 to those of L2. Mapping symbols is not sufficient because in some cases we could then show that one theory is stronger than another when they really are as strong as each other. For example, adding a new function symbol to L1 to obtain L2 may not strengthen Th1 because the new function symbol can be defined in terms of symbols of L1. We will give an example of this situation in the next subsection.

Therefore, more generally, determining when one theory is as strong as another depends on finding an *interpretation* that translates *formulae* of one theory into those of another. Most of the following definitions are adapted from [Enderton72]. Notice that an interpretation is a generalization of the notion of theory morphisms from algebraic theories [Burstall80, Burstall81] to theories in full first-order logic with equality.

Let Th1 be a theory in a language L1 and Th2 be a theory in a (possibly different) language L2.[12] Let π be a mapping from L1 into L2.

Def: If $\forall \sigma \in L1\ [\sigma \in Th1 \Rightarrow \pi(\sigma) \in Th2]$, then π is an *interpretation of Th1 into Th2*.

Def: Th1 is *as strong as* Th2 if there exists an interpretation of Th2 into Th1.

Def: Th1 is *stronger* than Th2 if Th1 is as strong as Th2 and Th2 is not as strong as Th1.

Def: Th1 and Th2 are *incomparable* if Th1 is not as strong as Th2 and Th2 is not as strong as Th1.

Def: If Th1 and Th2 are in the same language, Th1 is *more restrictive* than Th2 if Th1 is stronger than Th2.

---

12. L2 must include equality for technical reasons.

We extend the last four definitions to two specifications in the obvious way. For example, given two specifications, Spec1 and Spec2, Spec1 is as strong as Spec2 if Th(Spec1) is as strong as Th(Spec2).

Showing that Th1 is as strong as Th2 requires showing the existence of an interpretation from L2 into L1. Showing that Th1 is stronger than Th2 is much harder; it requires showing not only the existence of an interpretation from L2 into L1, but also that there does not exist any interpretation from L1 into L2. Notice that showing that Th1 is not stronger than Th2 is easier than showing Th1 is stronger than Th2 since for the former it suffices to show the existence of an interpretation from L1 into L2.

Finding an interpretation or showing the nonexistence of one is difficult in general. If we were to base our definition of strength on the simpler, but more restricted, definition of an interpretation that is defined to map symbols of one language into those of another, then it would be easier to find an interpretation or show the nonexistence of one when comparing relative strengths of specifications. As previously mentioned, the alternate definition may be simpler, but it does not capture the strength relation we want.

Finally, showing that two theories are incomparable requires showing the nonexistence of interpretations between the two languages in both directions. In some cases, however, to convince ourselves of incomparability, it suffices to show that there is a formula in $L1 \cap L2$ that is in Th1 and not in Th2, and a formula in $L1 \cap L2$ that is in Th2 and not in Th1. For interface specifications, the language of a shared trait can often be used as a basis for $L1 \cap L2$. We give an example of this situation in the next section.

5.2.3 Modifying a Specification With Respect To Strength

It would be useful to characterize changes we can make to a specification by their effect on the strength of the original specification. Adding equations, **reduces** clauses, or **closes** clauses can strengthen a trait. Selecting a stronger used trait, or changing its pre- or post-condition can strengthen a procedure specification.

To strengthen a cluster specification, we could select a stronger used trait or add a procedure specification. Adding a procedure specification does not necessarily strengthen a cluster specification. Doing so might leave the strength of the cluster specification unchanged or weaken it. It might even make the original and new cluster specifications incomparable because type induction rules of the original cluster specification might become invalid. We later give examples of each of these cases.

The kind of procedure specification that is added to a cluster specification can restrict the possible effects on its strength. If T is the type defined by the cluster specification, a procedure specification can be classified according to whether it specifies a procedure to *construct* or to *observe* objects of type T. A constructor returns or mutates objects of type T while an observer returns or mutates objects of type other than T. Using the terminology from Chapter 3, we can further classify constructors into *basic*, *producing*, and *mutating* constructors. In general, a procedure specification might both construct and observe objects of type T, as well as do combinations of all three kinds of construction. For the present discussion, we only consider the "pure" cases in which a procedure specification specifies either the construction or observation of objects of type T, but not both. For example, a "pure observer" specifies that a procedure takes in objects of type T, does not mutate any objects, and only returns objects other than type T. Figure 17 shows the possible effect adding a pure constructor or observer has on the strength of a cluster specification.

	stronger	as strong as	incomparable	weaker
constructor	?	yes	yes	yes
observer	yes	yes	no	no

Figure 17. Effect of Adding a Constructor or Observer on Strength

---

Adding any kind of "pure constructor" has the possible effect of leaving the original specification unchanged, making it incomparable to the new, or weakening it. We conjecture that adding a constructor cannot strengthen a cluster specification because adding a constructor adds a hypothesis to each of the type induction rules. Adding a hypothesis to a rule might leave unchanged, weaken, or invalidate an existing rule; it cannot allow us to conclude a stronger invariant. We leave the proof of our conjecture as an open problem.

We now give some examples. Let Spec1 be *SetClusSpec* and Spec2 be the result of adding a constructor to Spec1. As an example of adding a constructor that leaves a specification's strength unchanged, consider adding a *pair* procedure specification that takes in two (possibly equal) integers, i and j, and returns a set that is the union of {i} and {j}. Since formulae involving *pair* can be expressed in terms of *singleton* and *union*, no theorems of Th(Spec1) are invalidated and no new theorems are added. If, however, we had chosen our alternate definition that defines an interpretation to map between symbols, then adding the identifier, "*pair*," would strengthen *SetClusSpec* because *pair* could not be mapped to any identifier, *id*, in *SetClusSpec* such that formulae involving *pair* in Spec2 could be translated into formulae in Spec1 with *id* substituted in for *pair*. This example motivated our choosing the definition of strength as given since we intuitively believe that adding a constructor that does not change the invariant of a type should not strengthen the cluster specification.

Adding to Spec1 a *create* procedure specification that takes in no arguments and returns an empty set makes Spec1 and Spec2 incomparable. One might think that by the addition of *create* Th(Spec2) would be strictly larger than Th(Spec1) and so Th(Spec2) would

be stronger than Th(Spec1). This is not true, however, since the formula, true{S}∀s:set card(s↓) > 0, which is in Th(Spec1), is not in Th(Spec2) and the formula, true{S}∃s:set card(s↓) = 0, which is in Th(Spec2), is not in Th(Spec1). This example illustrates a perhaps surprising consequence of our definition. Intuitively, we would think that adding a constructor that increases the value set of a type should strictly strengthen the cluster specification. Strength, however, is defined in terms of theories, i.e., what is derivable from specifications, and not in terms of the "expressive" power of specifications.[13]

As an example of adding a constructor that weakens the strength of a specification, consider a *stack[elem]* cluster specification, Spec1, that has a *pop* procedure specification that returns a new stack whose value is that of the input stack with the top element removed. Let an invariant of Spec1 be that no stack object is mutated. Adding a mutating constructor, *shrink*, that mutates the input stack by removing the top element invalidates that invariant.

Adding a "pure observer," can strengthen a cluster specification or leave it unchanged. It cannot weaken the original cluster specification nor make the original and new specifications incomparable. Adding an observer can at most add formulae of the form P{Pr}Q to the theory of a cluster specification. Since hypotheses of type induction rules deal with only constructors, adding an observer has no effect on the type induction rules of the cluster specification. Hence, the addition of a (pure) observer cannot weaken or invalidate any of the rules.

As an example of strengthening with an observer, consider adding a *size* procedure specification to a *stack[elem]* cluster specification that has only constructors. Doing so adds theorems about integers to the Th(*stack[elem]*). As an example of leaving the strength unchanged, suppose *stack[elem]* has *null*, *push*, and *top*, where *top* mutates its stack

---

13. This observation suggests pursuing the definition of a different property of specifications that might be related to "expressive-completeness" [Kapur80b].

argument. Adding a *read* procedure specification that is like *top* except that it does not mutate its stack argument, does not change the strength of the original specification.

5.3 Essentiality

In the construction of a specification, we often want it to be "minimal" in a given context. That is, we would like to able to pare down a specification to just the "essential part" necessary for a desired set of properties to hold. Removing parts that have been shown to be inessential gives us a way of paring down a specification.

A part, P, of a specification, Spec, is *inessential* for a theory, T, if Spec with P removed can still be used to deduce the theorems in T. We say "P is an inessential part of Spec for T." Identifying a part of a specification that is inessential to prove a property means that we can freely remove or alter that part of the specification and still be ensured that the desired property holds. On the other hand, if we were to change some part that is essential then we might have to reverify that the property holds.

Whereas checking for properties defined in Section 5.1 is performed on a single specification, checking essentiality and inessentiality is performed on two specifications and a theory, where the second specification is defined to be a "part" of the second. The signatures for checkers for essentiality and inessentiality are as follows:

essential: specification, specification, theory → boolean
inessential: specification, specification, theory → boolean

In Section 5.3.1 we define essentiality and inessentiality by first defining what we mean by a part of a specification. In Section 5.3.2 we give some situations for when we might want to determine inessential parts of a specification.

5.3.1 Definitions

In the following discussion we treat a specification as a formal system, which is a set of symbols, a set of wff's, a set of axioms, and a set of rules. (See Chapter 3 for the correspondence between a specification and its formal system.) Thus, it makes sense to talk about the language (set of symbols and set of wff's), axioms, and rules of a specification. For a specification, Spec = ⟨L, A, R⟩, L is its language, A is its set of axioms and R is its set of rules.

Def: A *part* of Spec is a specification with a language, L'⊆L, a set of axioms, A'⊆A, and a set of rules, R'⊆R.

Examples of parts of a specification are the used trait of a procedure or cluster specification, and each of the bound procedure specifications of a cluster specification. Notice also that the type induction principle is a part of a cluster specification. Let two parts of Spec be P1 = ⟨L1, A1, R1⟩ and P2 = ⟨L2, A2, R2⟩.

> *Equal*: P1 = P2 if and only if L1 = L2, A1 = A2, and R1 = R2.
>
> *Subset*: P1⊆P2 if and only if L1⊆L2, A1⊆A2, and R1⊆R2.
>
> *Proper subset*: P1⊂P2 if and only if P1⊆P2 but P1 ≠ P2.
>
> *Difference*: (Spec - P1) is the specification whose language is (L - L1), whose set of axioms is (A - A1), and whose set of rules is (R - R1).

We require that subsets of sets of axioms and sets of rules are well-formed. For example, if L1 ⊂ L2, all axioms in A2 and all hypotheses and conclusions of rules in R2 are restricted to be in L2. Notice that P1 ⊂ P2 does not imply Th(P1) ⊂ Th(P2).

Let P be a part of a specification, Spec. Let T be a theory such that each formula in T is deducible from Spec. We write this "Spec ⊢ T."

Def: *P is an inessential part of Spec for T* if and only if (Spec - P) ⊢ T.

Def: An inessential part P of Spec for T is *maximal* if no part properly containing P is inessential.

Notice that there can be more than one maximal inessential part of a specification for a given theory.

Def: *P is an essential part of Spec for T* if and only if (Spec - P) is a maximal inessential part of Spec for T.

Checking for essentiality or inessentiality must be done with respect to a theory since a part of a specification that is essential for one theory might be inessential for a different theory. Given a theory, T, if a part, P, of a specification, Spec, is purported to be inessential for T, then one method for checking the inessentiality of P would be to remove P from Spec and check if the remaining specification is strong enough to prove each theorem in T. If each theorem in T is provable from (Spec - P), then P is inessential. If there is some theorem in T such that it is not provable from (Spec - P), then some subset of P is essential for T.

5.3.2 Situations for Determining Inessentiality

Here are three situations in which it would be useful to determine whether a part of a specification is inessential. One situation is to check if some part of a specification is inessential to prove some property of the specification itself. For example, we might want to know what part of a specification is inessential to proving it is fully-covering or deterministic. We might want to make a specification weaker, but ensure that it is still fully-covering or deterministic.

A second situation is to check if some part of a specification is inessential to prove particular properties of its specificand set. For example, suppose we want to determine if some part of our trait for sets is inessential for proving the property, has(delete(s,i),j) = ~(i .eq j) $\wedge$ has(s,j). We see, in particular, that the axioms about *card* are inessential to prove it. Another example of this second situation is to determine what part of a trait is inessential to establishing one of the hypotheses of a type induction rule associated with a cluster specification. For example, in Chapter 3 when we showed the property that the size of all set objects is greater than zero (for sets as defined by *SetClusSpec*), we used the property from

the SetOfInt trait that the cardinality of values of set objects is greater than or equal to zero. In this case, sort induction is essential, but, for instance, axioms about *delete* are not.

A third situation is to determine what part of a specification is inessential in the proof of satisfaction between an implementation, Imp, and a specification, Spec. Let T be {Imp *satisfies* Spec}. Suppose in showing T we use a specification S, whose theory is a subset of Th(Imp). We might be interested in knowing what an inessential part of S is that is not needed to prove T. In knowing what part of S is inessential to the proof of satisfaction, we can change that part of S and be guaranteed that Imp still satisfies Spec.

6. Conclusions, Contributions, and Further Work

6.1 Summary of Conclusions and Contributions

In Chapter 1 we observed that at present formal specifications are difficult to write and to apply in the design of software. We believe that the two-tiered approach presented in this thesis is one step toward a solution to this problem.

Our presentation included an approach to writing specifications, a specification language, and some ways to evaluate specifications. The approach separates the specification of state transformations and target programming language dependencies from the specification of underlying abstractions. The language supports this approach and was designed with the programmer in mind. The ways to evaluate specifications, i.e., checking and comparing, give a specifier means of convincing himself that his specification reflects his understanding of the problem statement. The distinguishing aspects of our solution are (1) the separation of concerns in the specification approach, (2) the incorporation of programming language dependencies in the specification language, and (3) a theory-oriented framework that provides a basis to reason about specifications independently of their underlying models.

The four main contributions of this thesis are:

1. The rigorous semantics for the two-tiered approach,

2. The design of a CLU interface language,

3. A framework for reasoning about two-tiered specifications and what they specify, and

4. Exploiting the framework for evaluating specifications.

The complex part of doing the semantics was in carefully fitting the two tiers together, and at the same time, keeping the separation clean. Mathematical entities such as algebras and relations serve as a basis for defining our model-oriented semantics. Although the models chosen are motivated by CLU, they can be used to model the semantics of interface languages for other programming languages. The models are relatively independent of Larch.

The key contribution behind the design of the interface specification language for CLU was isolating programming language dependencies into one component of a specification. In doing so, we shed light on what aspects of a programming language should show through to an interface specification language, and on what aspects were complex to handle (e.g., own variables). Another related contribution is the factorization of the presentation of the interface language into a kernel part and an extended part. Although we presented a design targeted for a particular programming language, we believe it is general enough to be adapted for others.

We also defined a proof-theoretic framework for reasoning about specifications. This reflected the same clean separation between the two tiers as the model-oriented semantics. It was designed to allow one to reason about what is being specified completely in terms of the text of the specifications. This advantage is especially significant if one has appropriate machine support, e.g., a theorem prover.

In exploring the utility of this framework, we defined some sample properties of specifications and ways to compare them. In making these definitions, we illustrated how to state their definitions within the proof-theoretic framework. Identifying these properties is of concern to a specifier who wants to know if some developing specification is getting "better." Experimentation is needed to see if we have focused on the right properties, but we have provided here at least some of the properties that might be of use to a specifier, and an indication of how to define them.

6.2 Directions for Further Work

We first discuss two areas of "basic" research: developing other interface languages and evaluating collections of specifications. We then discuss two areas of "experimental" research: building machine support and applying the two-tiered approach to examples.

6.2.1 Development of Interface Languages

One test of our two-tiered approach is to develop interface languages for other programming languages, both sequential and concurrent. We have not discussed concurrency at all in this thesis, and would be interested to see how easily the kernel interface language can be extended to handle concurrent programming issues. A first step to take is to extend our model to concurrent programming and then add syntactic extensions to the kernel language. Stark [Stark83] defines a model of the behavior of concurrent systems, which could serve as a reasonable basis for such a specification language. Jones extends his own work for sequential programs to concurrent ones [Jones81].

Development of interface languages for other sequential programming languages is currently being done for Cedar Mesa [Horning83]. Its design borrows directly from the kernel language we defined in Chapter 2.

Finally, we mention with hindsight a change we might make to the CLU interface language. Instead of giving two assertions in a procedure specification, since they are both interpreted with respect to two states, we could give only one assertion [Horning83, Yelick83]. Hence, instead of writing a pair, <pre, post>, in the body of a procedure specification, we write a single assertion. We also mention an obvious extension to the language. Instead of listing a single used trait in a **uses** clause of a procedure or cluster specification, we can list a set of used traits. Furthermore, we can perform operations on each of the traits in the list, e.g., renaming and inclusion. This extension does not change the semantics of a procedure or cluster specification because a single trait can be defined to include (i.e., **includes** in Larch)

each trait in a set of traits.

6.2.2 Evaluating Collections of Specification

In Chapter 5, we concentrated on individual specifications, and not at all on collections of specifications. As a collection of specifications grows, the issue of evaluating it becomes just as important as, and probably harder than, evaluating each of its individual components. We briefly mention some relations among specifications that are easily derived from the formalism we have described for the interface language.

A specifier usually has in mind some structure among the mass of specifications written. Depicting this structure is good practice in the design of a large specification as well as good documentation for the reader. For example, we define *uses* to be a relation on a collection of specifications, where a specification, Spec, *uses* a trait, Tr, if Tr is Spec's used trait. Similarly, we define *imports* to be a relation on a collection of specifications, where a specification, Spec, *imports* a cluster specification, Clus, if Spec imports the type defined by Clus.

These relations indicate global, or interconnection complexity, as opposed to the local complexity that can be seen in individual specification units. Evaluating the complexity of each of these kinds of relations can give the reader and writer of specifications an idea of the complexity of the specification. We might treat the relation associated with each of these kinds of specifications as a graph and then analyze the complexity of the specification in terms of properties of the graph. Some properties to check of a graph are whether it is acyclic, whether it is hierarchical (no sharing), or whether it is a tree (one root, no sharing). Whether a property is desirable or not would depend on the use of the specification. For example, one can argue that in writing a good specification one should have a *uses* relation that has a lot of sharing of the used traits to avoid repetition and to reuse work already done. On the other hand, care must be taken when changes are made to a shared trait; a specification with a hierarchical *uses* relation might be easier to modify.

6.2.3 Machine Support

The limited experience we have had in writing specifications makes evident the need for machine-support. Without machine-support, we have no hope of expecting either specifiers to write or programmers to use specifications, except as an academic exercise.

Minimally, machine-support should provide ways to manage the text of specifications; ideally, it should provide ways to reason about their meaning as well. Our list of tools includes (see [Guttag82]):

1. *A syntax checker.*

2. *A library.* Both traits and interface specifications, and both problem independent and dependent specifications should be included. Traits should be included for possible reuse; interfaces, primarily to provide examples.

3. *An editor.* A syntax-directed, interactive editor should supply templates, generate redundant information, and keep track of missing information.

4. *A semantic checker.* Theorem proving technology can be applied to the manipulation of specifications for checking properties of both specifications and what they specify. Much work remains in finding algorithms and heuristics that check for these properties.

The Larch project at M.I.T. has started on the development of these tools as part of a specification environment. Included in this development effort are implementations of a syntax and static semantics checker [Kownacki83] and a semantic checker that can manipulate equations in traits [Lescanne83, Forgaard83], and designs of a library [Atreya82] and a syntax-directed editor [Zachary83].

6.2.4 Experimentation

The two-tiered approach needs to be tested on realistic examples of substantial size. We can test the utility of the formal framework we set up only by trying it out. In doing so, we can then evaluate whether the two level partitioning is good, whether it makes it easier to read and write specifications, and whether it leads to better specifications. We can also see whether the separation of concerns leads to a better understanding of the specificands.

We may discover that we need to make changes to the design of the interface language. Identifying the language constructs that are used frequently, those that are rarely used, and those that would be nice to have in order to enhance expressibility can help in the designs of future interface languages.

We also need to discover other ways to evaluate a specification, other properties and qualities, and ways to analyze tradeoffs among them. We should test whether the properties we have discussed or variations of them are of any use or interest to a specifier. We should see under what circumstances a specifier tends to perform evaluation and classify what kinds of changes to a specification are made as a result of evaluation.

Finally, with more experimentation, we hope to show the utility of using formal specifications; in particular, to demonstrate that forcing precision in the design process has a beneficial effect on the overall programming process.

References

[Abrial80] Abrial, J.R., "The Specification Language Z: Syntax and Semantics," Programming Research Group, Oxford University, 1980.

[Ada79] Preliminary Ada Reference Manual, *SIGPLAN Notices*, Vol. 14, No. 6, Part A, June 1979.

[Apt81] Apt, K.R., "Ten Years of Hoare's Logic: A Survey--Part I," *Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October 1981, pp. 431-483.

[Atreya82] Atreya, S.K., "Formal Specification of a Specification Library," S.M. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1982.

[Berzins79] Berzins, V.A., "Abstract Model Specifications for Data Abstractions," MIT Laboratory for Computer Science, TR-221, Cambridge, MA, July 1979.

[Birkhoff70] Birkhoff, G., and J.D. Lipson, "Heterogeneous Algebras," *Journal of Combinatorial Theory*, Vol. 8, 1970, pp. 115-133.

[Bjorner78] Bjorner, D., and C.B. Jones (eds.), *The Vienna Development Method: the Meta-language*, Springer-Verlag, Lecture Notes in Computer Science 61, Berlin-Heidelberg-New York, 1978.

[Bloom83] Bloom, T., "Dynamic Module Replacement in a Distributed Programming System," Ph.D. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 1983.

[Boyer75] Boyer, R.S., and J.S. Moore, "Proving Theorems About Lisp Functions," *Journal of the ACM*, Vol. 22, January 1975, pp. 1975.

[Burstall72] Burstall, R.M., "Some Techniques for Proving Correctness of Programs Which Alter Data Structures," *Machine Intelligence 7*, Halstead Press, 1972, pp. 23-50.

[Burstall77] Burstall, R.M., and J.A. Goguen, "Putting Theories Together To Make Specifications," Invited Paper at the *Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA, August 1977, pp. 1045-1058.

[Burstall80] Burstall, R.M., and J.A. Goguen, "The Semantics of CLEAR, a Specification Language," *Proceedings of 1979 Copenhagen Winter School on Abstract Software Specifications*, Springer-Verlag, ed. Bjorner, 1980.

[Burstall81] Burstall, R.M., and J.A. Goguen, "An Informal Introduction to Specifications Using CLEAR," *The Correctness Problem in Computer Science*, eds. Boyer and Moore, Academic Press, 1981.

[Caine75] Caine, S.H., and E.K. Gordon, "PDL--A Tool for Software Design," *Proceedings of the 1975 National Computer Conference*, Vol. 44, Montvale, NJ., AFIPS Press, 1975, pp. 271-276.

[Chang73] Chang, C.C., and H.J. Keisler, *Model Theory*, North-Holland Publishing Company, 1973.

[deBakker80] deBakker, J., *Mathematical Theory of Program Correctness*, Prentice/Hall International, Englewood Cliffs, 1980.

[Deutsch73] Deutsch, L.P., "An Interactive Program Verifier," Ph.D. Thesis, University of California, Berkeley, 1973.

[Dijkstra76] Dijkstra E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[Ehrich78] Ehrich, H.-D., "Extensions and Implementations of Abstract Data Type Specifications," *Mathematical Foundations of Computer Science 1978 Proceedings*, Lecture Notes in Computer Science 64, 7th Symposium, Springer-Verlag, Poland, 1978, pp. 155-164.

[Ehrig80] Ehrig, H., H.-J. Kreowski, J. Thatcher, E. Wagner, and J. Wright, "Parameterized Data Types in Algebraic Specification Languages," *Automata, Languages, and Programming*, Lecture Notes in Computer Science 85, 7th Colloquium, Springer-Verlag, Noordwijkerhout, July 1980, pp. 157-168.

[Enderton72] Enderton, H.B., *A Mathematical Introduction to Logic*, Academic Press, New York, 1972.

[Floyd67] Floyd, R.W., "Assigning Meanings to Programs," *Proceedings of Symposium in Applied Mathematics"*, Vol. 19, American Mathematical Society, 1967, pp. 19-32.

[Forgaard83] Forgaard, R., "A Program for Generating and Analyzing Term Rewriting Systems," S.M. Thesis, MIT Department of Electrical Engineering and Computer Science, 1983 (forthcoming).

[Goguen75] Goguen, J.A., J.W. Thatcher, E.G Wagner, and J.B. Wright, "Abstract Data-Types as Initial Algebras and Correctness of Data Representations," *Proceedings from the Conference of Computer Graphics, Pattern Recognition and Data Structures*, May 1975, pp. 89-93.

[Goguen77] Goguen, J.A., "Abstract Errors for Abstract Data Types," *Proceedings of the IFIP Working Conference on Formal Basis of Programming Concepts*, Vol. 2, August 1977, pp. 21.1-21.32.

[Goguen78] Goguen, J.A., J.W., Thatcher, and E.G. Wagner, "Inital Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," *Current Trends in Programming Methodology*, Vol. IV, Data Structuring, ed. R.T. Yeh, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Goguen81] Goguen, J.A., and K. Parsaye-Ghomi, "Algebraic Denotational Semantics Using Parameterized Abstract Modules," Stanford Research Institute, TR CSL-119, Stanford, CA, February 1981.

[Good75] Good, D.I., R.L. London, and W.W. Bledsoe, "An Interactive Program Verification System," *IEEE Transactions on Software Engineering*, Vol. 1, No. 1, 1975, pp. 59-67.

[Good78] Good, D.I., R.M. Cohen, C.G. Hoch, L.W. Hunter, and D.F. Hare, "Report on the the Language Gypsy, Version 2.0," Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September 1978.

[Goodenough75] Goodenough, J.B. and S.L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, Vol. 1, No. 2, June 1975, pp. 156-173.

[Guttag75] Guttag, J.V., "The Specification and Application to Programming of Abstract Data Types," Ph.D. Thesis, University of Toronto, Toronto, Canada, September 1975.

[Guttag78] Guttag, J.V., E. Horowitz, and D.R. Musser, "Abstract Data Types and Software Validation," *Communications of the ACM*, Vol. 21, No. 12, December 1978, pp. 1048-1064.

[Guttag80a] Guttag, J.V., "Notes on Type Abstraction (Version 2)," *IEEE Transactions on Software Engineering*, Vol. 6, No. 1, January 1980, pp. 13-23.

[Guttag80b] Guttag, J.V., and J.J. Horning, "Formal Specification As a Design Tool," *Proceedings on the Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, Nevada, January 1980, pp. 251-261.

[Guttag82] Guttag, J.V., J.J. Horning, and J.M. Wing, "Some Notes on Putting Formal Specifications to Productive Use," *Science of Computer Programming*, Vol. 2, No. 1, October 1982, pp. 53-68.

[Guttag83a] Guttag, J.V., and J.J. Horning, *An Introduction to the Larch Shared Language*, IFIP 83, Paris, France, September 1983 (forthcoming).

[Guttag83b] Guttag, J.V., and J.J. Horning, *Preliminary Report on the Larch Shared Language*, Xerox PARC Technical Report, 1983 (forthcoming).

[Hoare69] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, Vol. 12, No. 10, October 1969, pp. 576-580.

[Hoare72] Hoare, C.A.R., "Proof of Correctness of Data Representations," *Acta Informatica*, Vol. 1, No. 1, 1972, pp. 271-281.

[Horning83] Horning, J.J., private communication.

[Jackson75] Jackson, M.A., *Principles of Program Design*, London, Academic Press, 1975.

[Jones80] Jones, C.B., *Software Development: A Rigorous Approach*, Prentice-Hall, 1980.

[Jones81] Jones, C.B., "Development Methods for Computer Programs Including a Notion of Interference," Ph.D. Thesis, Oxford University, England, June 1981.

[Kamin83] Kamin, S., "Final Data Types and Their Specification," *Transactions on Programming Languages and Systems*, Vol. 5, No. 1, January 1983, pp. 97-121.

[Kapur80a] Kapur, D., "Towards a Theory for Abstract Data Types," MIT Laboratory for Computer Science, TR-237, Cambridge, MA, May 1980.

[Kapur80b] Kapur, D., and S. Mandayam, "Expressiveness of the Operation Set of a Data Abstraction," *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages"*, Las Vegas, Nevada, January 1980, pp. 139-153.

[Katzan76] Katzan, H., Jr., *Systems Design and Documentation: An Introduction to the HIPO Method*, New York, Van Nostrand Reinhold, 1976.

[King69] King, J.C., "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1969.

[Knuth69] Knuth, D.E., and P.B. Bendix, "Simple Word Problems in Universal Algebras," *Computational Problems in Abstract Algebra*, Pergamon Press, Oxford, ed. J. Leech, 1969.

[Kownacki83] Kownacki, R.W., "A Tool for Partial Semantic Analysis of Formal Specifications," S.M. Thesis, MIT Department of Electrical Engineering and Computer Science 1983 (forthcoming).

[Lescanne83] Lescanne, P., "Computer Experiments with The REVE Term Rewriting System Generator," *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages"*, Austin, TX, January 1983, pp. 99-108.

[Liskov77] Liskov, B.H., A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *Communications of the ACM*, Vol. 20, No. 8, August 1977, pp. 564-576.

[Liskov79] Liskov, B.H., and Berzins, V., "An Appraisal of Program Specifications," *Research Directions in Software Technology*, MIT Press, Cambridge, MA, 1979.

[Liskov81] Liskov, B.H., et al., *CLU Reference Manual*, Lecture Notes in Computer Science 114, Springer-Verlag, 1981.

[London75] London, R.L., "A View of Program Verification," *Proceedings of the International Conference on Reliable Software*, April 1975, pp. 534-545.

[Luckham76] Luckham, D., and N. Suzuki, "Automatic Program Verification V: Verification-Oriented Proof Rules for Arrays, Records, and Pointers," Stanford University, AIM-278, Stanford, CA, March 1976.

[McMullin82] McMullin, P.R., "DAISTS: A System for Using Specifications to Test Implementations," University of Maryland, Ph.D. thesis, 1982.

[Mendelson64] Mendelson, E., *Introduction to Mathematical Logic*, D. Van Nostrand Co., New York, 1964.

[Mitchell78] Mitchell, J.G., W. Maybury, and R. Sweet, *Mesa Language Manual*, Xerox Palo Alto Research Center, CSL-78-1, Palo Alto, CA, February 1978.

[Musser77] Musser, D.R., "A Data Type Verification System Based on Rewrite Rules," *Proceedings of the Sixth Texas Conference on Computing Systems*, Austin, TX, November 1977, pp. 1A-22-1A-31.

[Musser80] Musser, D.R., "Abstract Data Type Specification in the Affirm System," *IEEE Transactions on Software Engineering*, Vol. 6, No. 1, January 1980, pp. 24-32.

[Myers75] Myers, G.J., *Reliable Software Through Composite Design*, Petrocelli/Charter, New York, 1975.

[Nakajima80] Nakajima, R., M. Honda, and H. Nakahara, "Hierarchical Program Specification and Verification--A Many-sorted Logical Approach," *Acta Informatica*, Vol. 14, 1980, pp. 135-155.

[Oppen75] Oppen, D.C., "On Logic and Program Verification," University of Toronto, TR 82, Toronto, Canada, April 1975.

[Parnas72a] Parnas, D.L., "A Technique for Software Module Specification with Examples," *Communications of the ACM*, Vol 15., No. 5, May 1972, pp. 330-336.

[Parnas72b] Parnas, D.L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*", Vol 15, No. 12, December 1972, pp. 1053-1058.

[Parnas77] Parnas, D.L., "The Use of Precise Specifications in the Development of Software," Information Processing 77, ed. B. Gilchrist, North-Holland Publishing Company, 1977, pp. 861-867.

[Reynolds77] Reynolds, J.C., "Reasoning About Arrays," University of Edinburgh, CSR-6-77, July 1977.

[Robinson77] Robinson, L., and O. Roubine, "SPECIAL--A Specification and Assertion Language," Stanford Research Institute, Stanford, CA, TR CSL-46, January 1977.

[Schaffert81] Schaffert, J.S., "Specification and Verification of Programs using Data Abstraction and Sharing," Ph.D. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 1981.

[Shoenfield67] Shoenfield, J.R., *Mathematical Logic*, Addison-Wesley, 1967.

[Standish73] Standish, T.A., "Data Structures: An Axiomatic Approach," Bolt, Beranek and Newman, Inc., Report #2639, Cambridge, MA, August 1973.

[Stark83] Stark, E.W., "Foundations of a Theory of Specification for Distributed Systems," Ph.D. Thesis, MIT Department of Electrical Engineering and Computer Science, 1983 (forthcoming).

[Suzuki75] Suzuki, N., "Verifying Programs by Algebraic and Logical Reduction," *Proceedings International Conference on Reliable Software*, 1975.

[Suzuki76] Suzuki, N., "Automatic Verification of Programs with Complex Data Structures," Stanford University, AIM-279, Stanford, CA, February 1976.

[Thatcher78] Thatcher, J.W., E.G. Wagner, and J.B. Wright, "Data Type Specification: Parameterization and the Power of Specification Techniques," *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, May 1978, pp. 119-132.

[vonHenke75] von Henke, F.W., and D.C. Luckham, "A Methodology for Verifying Programs," *Proceedings International Conference on Reliable Software*, 1975.

[Wand79] Wand, M., "Final Algebra Semantics and Data Type Extensions," *Journal of Computer and System Sciences*, Vol. 19, No. 1, August 1979, pp. 27-44.

[Wegbreit76] Wegbreit, B., and J.M. Spitzen, "Proving Properties of Complex Data Structures," *Journal of the ACM*, Vol. 23, No. 2, April 1976, pp. 389-396.

[Yelick83] Yelick, K.A, "The CLU Interface Language Reference Manual," MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 1983 (forthcoming).

[Yonezawa77] Yonezawa, A., "Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics," MIT Laboratory for Computer Science, TR-191, Cambridge, MA, December 1977.

[Yourdon78] Yourdon, E., and L.L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Programs and Systems Design*, 2nd ed., Yourdon Press, New York, 1978.

[Zachary83] Zachary, J.L., "A Syntax-Directed Tool for Constructing Specifications," S.M. Thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, March 1983.

[Zilles75] Zilles, S.N., "Abstract Specifications for Data Types," IBM Research Laboratory, San Jose, 1975.

Appendix I - Interface and Trait Specifications

```
Equivalence: trait
  introduces
        eq: E, E → Bool
  constrains [eq] so that for all [x, y, z: E]
        eq(x,x) = true
        eq(x,y) = eq(y,x)
        ((eq(x,y) ∧ eq(y,z)) ⇒ eq(x,z)) = true
```

Figure 3. Equivalence Trait

```
SetOfE: trait
  includes Integer, Equivalence
  introduces
        empty: → C
        add: C, E → C
        remove: C, E → C
        has: C, E → Bool
        isEmpty: C → Bool
        card: C → Int
  closes C over [empty, add]
  constrains [C] so that for all [s: C, e, e1: E]
        remove(empty, e) = empty
        remove(add(s,e), e1) = if eq(e,e1) then remove(s,e1) else add(remove(s,e1),e)
        has(empty, e) = false
        has(add(s,e), e1) = if eq(e,e1) then true else has(s,e1)
        isEmpty(empty) = true
        isEmpty(add(s,e)) = false
        card(empty) = 0
        card(add(s,e)) = if has(s,e) then card(s) else 1 + card(s)

SetOfInt: trait
  includes SetOfE with [SI for C, Int for E]
```

Figure 4. SetOfE and SetOfInt Traits

```
set = cluster is singleton, union, delete, size
  uses SetOfInt
  provides mutable set from SI

        singleton = proc (i: int) returns (s: set)
          uses SetOfInt
                pre true
                post s↓ = add(empty, i↑) ∧ new s ∧ mutates Ø ∧ returns
                end

        union = proc (s1, s2: set) returns (s3: set)
          uses SetOfInt
                pre true
                post ∀i:Int [has(s3↓,i) = has(s1↑,i) ∨ has(s2↑,i)]
                        ∧ new s3 ∧ mutates Ø ∧ returns
                end

        delete = proc (s: set, i: int) signals (emptiesSet)
          uses SetOfInt
                pre true
                post [((card(s↑) ≥ 2) ∨ ~has(s↑,i↑)) ⇒
                                (s↓ = remove(s↑,i↑) ∧ mutates s ∧ returns)] ∧
                     [((card(s↑) .eq 1) ∧ has(s↑,i↑)) ⇒
                                mutates Ø ∧ signals emptiesSet] ∧
                     new Ø
                end

        size = proc (s: set) returns (i: int)
          uses SetOfInt
                pre true
                post i↓ = card(s↑) ∧ new Ø ∧ mutates Ø ∧ returns
                end
end
```

Figure 9. Set Cluster Specification (SetClusSpec)

```
stack = cluster is empty, grow, read
  uses StackOfInt
  provides mutable stack from Stkl

        empty = proc () returns (st: stack)
          pre true
          post st↓ = null ∧ new st
          end

        grow = proc (st: stack, i: int)
          pre true
          mutates st
          post st↓ = push(st↑, i↑)
          end

        read = proc (st: stack) returns (i: int)
          pre ~isNull(st↑)
          post i↓ = top(st↑)
          end

  end stack
```

Figure 12. Stack Cluster Specification

```
StackOfInt: trait
  includes StackOfE with [Stkl for C, Integer for E]

StackOfE: trait
  includes Integer
  introduces
        null: → C
        push: C, E → C
        top: C → E
        pop: C → C
        isNull: C → Bool
        isIn: C, E → Bool
        size: C → Int
  closes C over [null, push]
  constrains [C] so that for all [s: C, e,e1: E]
        top(null) exempt
        top(push(s,e)) = e
        pop(null) exempt
        pop(push(s,e)) = s
        isNull(null) = true
        isNull(push(s,e)) = false
        isIn(null,e) = false
        isIn(push(s,e),e1) = if e .eq e1 then true else isIn(s,e1)
        size(null) = 0
        size(push(s,e)) = size(s) + 1
```

Figure 13. Traits for Stacks

Appendix II - Proofs

II.1. Validity of a Type Induction Rule

For the predicate,

$$P(t) = \sim isNull(t\uparrow) \Rightarrow card(top(t\uparrow)\uparrow) < 64.$$

we show the validity of the hypotheses of the following type induction rule.

Hypotheses:

HB true {empty} ~isNull(st↓) ⇒ [card(top(st↓)↓) < 64]

HP ~isNull(s1↑) ⇒ [card(top(s1↑)↑) < 64] {grow}
~isNull(s2↓) ⇒ [card(top(s2↓)↓) < 64]

HM s = top(v1↑) ∧ ~isNull(v1↑) ⇒ [card(top(v1↑)↑) < 64] {delete}
~isNull(v1↓) ⇒ [card(top(v1↓)↓) < 64]

Conclusion: true {S} ∀t:stack[set] ~isNull(t↓) ⇒ card(top(t↓)↓) < 64 for all

Proof:

1. HB: true {empty} ~isNull(st↓) ⇒ [card(top(st↓)↓) < 64]
Th(*empty*) gives the axiom true {empty} empty.post(st)
where empty.post(st) ≡ st↓ = null ∧ **new** st ∧ **mutates** Ø ∧ **returns**

empty.post(st) ⇒ P[st/t] is valid because
st↓ = null ⇒ [~isEmpty(st↓) ⇒ card(top(st↓)↓) < 64],
which is true since ~isEmpty(st↓) is false.

HB is valid by the rule of consequence.

2. HP: ~isNull(s1↑) ⇒ [card(top(s1↑)↑) < 64] {grow}
~isNull(s2↓) ⇒ [card(top(s2↓)↓) < 64]

Assume ~isNull(s1↑) ⇒ card(top(s1↑)↑) < 64
We have the axiom, card(s↑) < 64 {grow} grow.post(s1, s2, s)
where grow.post(s1, s2, s) ≡
s2↓ = push(s1↑,s) ∧ **new** s2 ∧ **mutates** Ø ∧ **returns**

We have that card(s↑) < 64
⇒ card(s↓) < 64, from **mutates** Ø
⇒ card(top(push(s1↑,s))↓) < 64, from Th(*StackOfSS*)

$\Rightarrow$ card(top(s2↓)↓) < 64, from substitution for s2↓ from grow.post(s1, s2, s)
$\Rightarrow$ [~isNull(s2↓) $\Rightarrow$ [card(top(s2↓)↓) < 64]] (a weaker assertion)

HP is valid by the rule of consequence.

3. HM: s = top(v1↑) $\wedge$ ~isNull(v1↑) $\Rightarrow$ [card(top(v1↑)↑) < 64] {delete}
~isNull(v1↓) $\Rightarrow$ [card(top(v1↓)↓) < 64]

Assume ~isNull(v1↑) $\Rightarrow$ [card(top(v1↑)↑) < 64]. The post-condition of delete is:
[((card(s↑) $\geq$ 2) $\vee$ ~has(s↑,i↑)) $\Rightarrow$
(s↓ = remove(s↑,i↑) $\wedge$ **mutates** s $\wedge$ **returns**)] $\wedge$
[((card(s↑) .eq 1) $\wedge$ has(s↑,i↑)) $\Rightarrow$
mutates Ø $\wedge$ **signals** emptiesSet] $\wedge$
new Ø

Assume ~isNull(v1↑). With the term top(v1↑) substituted in for s, we have:

(a) ((card(top(v1↑)↑) $\geq$ 2) $\vee$ ~has(top(v1↑)↑,i↑)) $\Rightarrow$
[top(v1↑)↓ = remove(top(v1↑)↑,i↑) $\wedge$ **mutates** top(v1↑) $\wedge$ **returns**]

Since card(top(v1↑)↑) < 64 (from the assumptions),
card(remove(top(v1↑)↑,i↑)) < 64 by Th(*SetOfInt*)
card(top(v1↑)↓) < 64 by substitution,
card(top(v1↓)↓) < 64 since the object v1 is not mutated.
(Only top(v1↑) is possibly mutated.)

(b) ((card(top(v1↑)↑) .eq 1) $\wedge$ has(top(v1↑)↑,i↑)) $\Rightarrow$
$\wedge$ card(top(v1↑)↑) .eq 1 $\wedge$ **mutates** Ø $\wedge$ **signals** emptiesSet

Since card(top(v1↑)↑) < 64 (again, from the assumptions),
card(top(v1↓)↓) < 64, from **mutates** Ø.

HM is valid by the rule of consequence. ∎

II.2. Proof of Satisfaction

We now give an example of a cluster that satisfies a cluster specification. Figure 18 gives a set cluster specification. Figure 19 gives an implementation of this cluster specification. The implementation uses the rep type, array[int], for which a cluster specification is given in Figure 20. The ArrayOfInt trait used to define the array[int] type is given in Figure 21.

```
set = cluster is create, insert, size, member
  uses SetOfInt
  provides mutable set from SI

        create = proc () returns (s: set)
          pre true
          post s↓ = empty ∧ new s ∧ mutates Ø ∧ returns
          end

        insert = proc (s: set, i: int)
          pre true
          post s↓ = add(s↑,i) ∧ new Ø ∧ mutates s ∧ returns
          end

        size = proc (s: set) returns (i: int)
          pre true
          post i↓ = card(s↑) ∧ new Ø ∧ mutates Ø ∧ returns
          end

        member = proc (s: set, i: int) returns (b: bool)
          pre true
          post has(s↑, i) = b↓ ∧ new Ø ∧ mutates Ø ∧ returns
          end member

        end
```

Figure 18. Set Cluster Specification

We sketch the proof of satisfaction below. We prefix procedure names by "T$" to distinguish them from trait function names. We expect machine tools to aid the implementor in performing much of the symbol manipulation found in these kinds of proofs [Boyer79, Good75, Good78, Musser77, Musser80].

1. Let the abstraction function be:

$$A: TtoS(array[int]) \rightarrow TtoS(set)$$

$$A(\alpha) = \text{if } size(\alpha) = 0 \text{ then } empty \text{ else if } size(\alpha) > 0 \text{ } add(A(remh(\alpha)), top(\alpha))$$

2. The rep invariant, $RI(\alpha)$, is:

$$\forall \alpha{:}AI\ [low(\alpha) = 1 \wedge size(\alpha) \geq 0 \wedge NoDups(\alpha)],$$
$$\text{where } NoDups(\alpha) = \forall i,j\ [fetch(\alpha,i) = fetch(\alpha,j) \Rightarrow i = j].$$

```
set = cluster is empty, insert, size, member

        rep = array[int]

        create = proc () returns (cvt)
                 return (rep$create(1))
                 end create

        insert = proc (c: cvt, i: int)
                 if ~member(up(c), i) then rep$addh(c,i) end
                 end insert

        size = proc (c: cvt) returns (int)
                 return(rep$size(c))
                 end size

        member = proc (c: cvt, i: int) returns (bool)
                 k: int := rep$low(c)
                 while k ≤ rep$high(c) do
                         if i = rep$fetch(c,k) then
                           return(true) end
                         k := k+1
                         end
                 return(false)
                 end member

        end set
```

Figure 19. Implementation of the Set Cluster Specification

---

3. For each procedure in the set cluster we must show it satisfies its corresponding procedure specification in the set cluster specification under A. For our simple example, in most cases this reduces to showing that the post-condition of some procedure specification of the array[int] cluster specification implies the post-condition of the corresponding procedure specification of the set cluster specification. We also need to show that the rep invariant holds for each procedure of the set cluster implementation.

3.1. set$create: Let c↓ = create(1) from array[int]'s create.post. Show that s↓ = empty.

s↓ = A(c↓)
= A(create(1)) by substitution
= empty by the definition of A, since size(create(1)) = 0.

```
array[int] = cluster is create, addh, size, low, high, fetch
  uses ArrayOfInt
  provides mutable array[int] from AI

        create = proc (i: int) returns (a: array[int])
          pre true
          post a↓ = create(1) ∧ new a ∧ mutates ∅ ∧ returns
          end

        addh = proc (a: array[int], i: int)
          pre true
          post a↓ = addh(a↑,i) ∧ new ∅ ∧ mutates a ∧ returns
          end

        size = proc (a: array[int]) returns (i: int)
          pre true
          post i↓ = size(s↑) ∧ new ∅ ∧ mutates ∅ ∧ returns
          end

        low = proc (a: array[int]) returns (i: int)
          pre true
          post i↓ = low(s↑) ∧ new ∅ ∧ mutates ∅ ∧ returns
          end

        high = proc (a: array[int]) returns (i: int)
          pre true
          post i↓ = high(s↑) ∧ new ∅ ∧ mutates ∅ ∧ returns
          end

        fetch = proc (a: array[int], i: int) returns (j: int) signals (bounds)
          pre true
          post [low(a↑)≤i≤high(a↑) ⇒ (j↓ = fetch(a↑,i) ∧ mutates ∅ ∧ returns] ∧
               [(i<low(a↑) ∨ i>high(a↑)) ⇒ (signals bounds ∧ mutates ∅)]
               ∧ new ∅

        end array[int]
```

Figure 20. Array Cluster Specification

We know that s is new since rep$create returns a new object, i.e., new c ⇒ new s. Since rep$create does not mutate any object, the mutates ∅ assertion is true. Thus, the post-condition of create is satisfied. We show that the rep invariant, $RI^f(c\downarrow)$, is established:

low(c↓) = low(create(1)) = 1, from Th(*ArrayOfInt*).

size(c↓) = size(create(1)) = 0 from Th(*ArrayOfInt*).

NoDups(c↓) = NoDups(create(1)) = ∀i,j:Int [fetch(c↓,i) = fetch(c↓,j) ⇒ i = j],
In Th(*ArrayOfInt*), fetch(create(x),y) is defined, but **exempt**.
Let v = fetch(create(1),i) and w = fetch(create(1),j).

```
ArrayOfInt trait
  includes Array [AI for A, int_obj for E]
  introduces
        empty: AI → Bool
        size: AI → Int
        isin: AI, int_obj → Bool
  constrains [AI] so that for ∀ [k: Int, i,j: int_obj, a: AI]
        empty(create(k)) = true
        empty(addh(a,i)) = false
        size(create(k)) = 0
        size(addh(a,i)) = size(a) + 1
        isin(create(k)) = false
        isin(addh(a,i),j) = if i .eq j then true else isin(a,j)

Array: trait
  includes Integer, Elem
  introduces
        create: Int → A
        addh: A, E → A
        remh: A → A
        low: A → Int
        high: A → Int
        fetch: A, Int → E
        store: A, Int, E → A
        size: A → Bool
  closes A over [create, addh]
  constrains [A] so that for all [i,i1,i2: Int, e,e1,e2: E, a: A]
        remh(create(i)) exempt
        remh(addh(a,e)) = a
        low(create(i)) = i
        low(addh(a,e)) = low(a)
        high(a) = low(a) + size(a) - 1
        fetch(create(i1),i2) exempt
        fetch(addh(a,e),i) = if i .eq (low(a) + size(a)) then e else fetch(a,i)
        store(create(i1),i2,e) exempt
        store(addh(a,e1),i,e2) = if i .eq (low(a) + size(a)) then addh(a,e2)
              else addh(store(a,i,e2),e1)
        size(create(i)) = 0
        size(addh(a,e)) = size(a) + 1
```

Figure 21. ArrayOfInt and Array Traits

Then v = w ⇒ i = j, and so NoDups(c↓) holds.

3.2. set$insert: Let s↑ = A(c↑). Show that s↓ = add(s↑, i).

Case 1: ~member(s↑, i)

Let c↓ = addh(c↑,i) from addh.post.

s↓ = A(c↓)
= add(A(remh(c↓),top(c↓)))
= add(A(remh(addh(c↑,i))), top(addh(c↑,i)))
= add(A(c↑), i)
= add(s↑, i)

Case 2: member(s↑, i)
⇒ has(s↑, i)
⇒ add(s↑, i) = s↑ from Th(*SetOfInt*)
s↓ = A(c↓)
= A(c↑) since c↑ = c↓
= s↑
= add(s↑, i)

Since set$member (see 3.4 below) and rep$addh do not create new objects, the **new** Ø assertion of insert's post-condition is true. The **mutates** assertion is true since the value of the input set object, s, might be changed. Thus, the post-condition of insert is satisfied. We show that the rep invariant is maintained:

low(c↓) = low(addh(c↑,i)) = low(c↑) = 1
size(c↓) = size(addh(c↑,i)) = 1 + size(c↑), which is true since size(c↑) ≥ 0.
NoDups(c↓) = NoDups(addh(c↑,i))
∀j,k:Int [fetch(addh(c↑,i),j) = fetch(addh(c↑,i),k)]
= ∀j,k:Int [(**if** j = low(c↑) + size(c↑) **then** i **else** fetch(c↑,j)) =
(**if** k = low(c↑) + size(c↑) **then** i **else** fetch(c↑,k))]
⇒ j = k since NoDups(c↑).

3.3. set$size: Let s↑ = A(c↑). Show that size(c↑) = card(s↑). We prove this by induction.

Case 1: c↑ = create(i).
size(c↑) = 0
= card(empty)
= card(A(c↑))
= card(s↑)

Case 2: c↑ = addh(x,y). The induction hypothesis (IH) is size(x) = card(A(x)).
From NoDups, we know that ~isin(x,y).
From Lemma (below) ~isin(x,y) ⇒ ~has(A(x),y)
Show size(c↑) = card(s↑).
size(c↑) = 1 + size(x)
= 1 + card(A(x)), by IH
= card(add(A(x),y)) since ~has(A(x),y)
= card(add(A(remh(addh(x,y))),top(addh(x,y))))
= card(A(addh(x,y)))
= card(A(c↑))
= card(s↑)

Since rep$size neither creates new objects nor mutates existing ones, the **new** Ø and **mutates** Ø assertions of size's post-condition are both true. Thus, the post-condition of size is satisfied. We show that the rep invariant is maintained. Since rep$size mutates nothing, c↓ = c↑.

low(c↓) = low(c↑) = 1,
size(c↓) = size(c↑) ≥ 0,

NoDups(c↓) = NoDups(c↑).

3.4. set$member: Let s↑ = A(c↑) and let b be the boolean returned by member. Show that has(s↑,i) = b↓.

Case 1: empty(c↑) ⇒ (isin(c↑,i) = false) ⇒
(has(A(c↑),i) = false), by Lemma below.

Case 2: The loop invariant is inbounds(k) and ∀d:Int low(c↑)≤d<k [fetch(c↑,d) ≠ i]
where inbounds(k) = low(c↑)≤k↑≤high(c↑)

Case 2.1: i = j
At the return(true) statement we know
that b↓ = true ∧ isin(c↑,i) = b↓.
isin(c↑,i) ⇒ has(A(c↑),i) ⇒ has(s↑,i), by Lemma below.

Case 2.2: i ≠ j
We increment k and go to the start of the loop.

At termination of loop, k↑ = high(c↑) + 1 ∧
∀d:Int low(c↑)≤d<high(c↑) + 1 [fetch(c↑,d) ≠ i]
⇒ ∀d:Int low(c↑)≤d≤high(c↑) [fetch(c↑,d) ≠ i]
⇒ (isin(c↑,i) = false)
⇒ (has(A(c↑),i) = false), by Lemma below.

Since replow, rephigh, rep$fetch, and int$add do not create new objects nor mutate existing ones, the new ∅ and mutates ∅ assertions of member's post-condition are both true. Thus, the post-condition of member is satisfied. The rep invariant is maintained because replow, rephigh, rep$fetch do not mutate any objects, and so c↓ = c↑, as in the case for set$size.

Lemma: ∀x:AI [isin(x,i) ⇒ has(A(x),i)]
Pf: By sort induction.

Case 1: Let x = create(k)
isin(x,i) = false
has(A(create(k)),i) = has(empty,i) = false

Case 2: Let x = addh(y,k)
isin(x,i)
= isin(addh(y,k),i)
= if i = k then true else isin(y,i)

has(A(addh(y,k),i)
= has(add(y,k),i)
= if i = k then true else has(y,i)
True, by induction.

(Proof of lemma)∎
(Proof of set)∎

This blank page was inserted to preserve pagination.

CS-TR Scanning Project
Document Control Form

Date : 7/14/95

Report # LCS-TR-299

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 163(178-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☒ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP (1) UN#'ED TITLE PAGES	
(2-163) PAGES #'ED 2-	
(164-169) SCANCONTROL, COVER, SPINE, PRINTER'S NOTES, DOD(2)	
(170-172) TRGTS (3)	

Scanning Agent Signoff:
Date Received: 7/14/95 Date Scanned: 7/20/95 Date Returned: 7/20/95

Scanning Agent Signature: Michael W. Cook

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: MIT/LCS/TR-299	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: A Two-Tiered Approach to Specifying Programs	5. TYPE OF REPORT & PERIOD COVERED: Ph.D. Thesis, June '83
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS
7. AUTHOR(s): Jeannette Marie Wing	8. CONTRACT OR GRANT NUMBER(s): DARPA/ONR N0014-83-K-0125 NSF MCS-8119846
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT Laboratory for Computer Science 545 Technology Square Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA/Dept. of Defense Information Processing Techniuqes Office 1400 Wilson Boulevard Arlington, VA 22209	12. REPORT DATE: June 1983 13. NUMBER OF PAGES: 163
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Office of Naval Research/Dept. of Navy Information Systems Program Arlington, VA 22217	15. SECURITY CLASS. *(of this report)*: Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Formal Specifications, Program Design, Specification Languages, Specification Analysis, Algebraic Specifications, Abstract Data Types, Programming Methodology

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

Current research in specifications is beginning to emphasize the practical use of formal specifications in program design. This thesis presents a specification approach, a specification language that supports that approach, and some ways to evaluate specifications written in that language.

The two-tiered approach separates the specification of underlying abstractions from the specification of state transformations. In

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. .. continued

this approach, state transformations and target programming language dependencies are isolated into an interface language component. All interface specifications are built upon shared language specifications that describe the underlying abstractions. This thesis presents an interface specification language for the CLU programming language and presumes the Larch shared language.

This thesis also suggests a number of kinds of analyses that one might want to perform on two-tiered specifications. These are related to the consistency, completeness, and strength of specifications, and are all presented in terms of the theories associated with specifications.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 7/20/1995

M.I.T. Libraries
Document Services