SPECWARE USER'S GUIDE

Specware 2.0.3

March 1998

Kestrel Institute
3260 Hillview Avenue
2nd Floor
Palo Alto, CA 94304

Kestrel Development Corporation
3260 Hillview Avenue
2nd Floor
Palo Alto, CA 94304

Phone: (650) 493-6871
Fax: (650) 424-1807
Email: specware-request@kestrel.edu

Table of Contents

Preface

About This Manual

This section describes how the *User's Guide* is organized and how to use it, and the conventions used in the manual.

Organization of This Manual

Chapter 1, "Introduction to Specware" gives an overview of the SPECWARE system, and introduces the basic terminology and design process.

Chapter 2, "Essential Specware Concepts" introduces the basic concepts and constructs of the Slang language in some detail.

Chapter 3, "Composition in Specware" describes the process of composition by which you refine a specification and generate code.

Chapter 4, "The Specware User Interface" introduces the windows and basic techniques you use to manipulate SPECWARE objects interactively. This chapter describes how to start and exit the SPECWARE system.

Chapter 5, "Building Specifications" describes the techniques you use to build specifications in Slang and to view the structures you build in the graphical user interface. This chapter describes how to create and parse Slang files.

Chapter 6, "Building Refinements" describes the techniques you use to refine your specification using the graphical user interface. This chapter describes how to generate code from your specification.

Appendix A, "Customizing Specware" describes how to set behavioral and display preferences for SPECWARE.

How to Use This Manual

The first half of this manual provides an introduction to the Slang language and the process of building an application, using both the Slang language and the SPECWARE user interface. It introduces the terminology, much of the syntax, and most of the techniques you will need to use SPECWARE. It is recommended that you read this section to familiarize yourself with the product, process, and terminology, before you begin to design an application.

The second half of the manual focuses on specific techniques and provides step-by-step how-to information for all of the basic processes. It is recommended that you read this section in conjunction with experimental use of the product, to familiarize yourself with the procedures you will use in the course of building an application.

Additional Documentation

The ***Specware Tutorial*** is a companion document to the *User's Guide*. It walks you through the process of building and refining the specification of a small, real-world problem. When you have finished the tutorial, you will have built a specification of a bank account, and produced and tested an implementation of the specification in Lisp.

The ***Specware Language Manual*** is a complete reference for the Slang specification-building language and the SPECWARE system concepts. This document is available in printed form, and also online as part of the SpecWeb help system.

The **SpecWeb** help system, available from the SPECWARE main menu bar, provides access to an online version of the *Specware Language Manual*, a tutorial introduction to SPECWARE, release notes, and access to the source code for the SPECWARE specification library.

Prerequisites

This manual assumes that you are familiar with the emacs editor and Lisp. Familiarity with mathematical computational theory, particularly set theory and category theory, is helpful in understanding SPECWARE concepts, terminology, and many of the examples.

Suggested Reading

The following books are helpful introductions to the mathematics behind SPECWARE.

- Lawvere, Bill and Schanuel, Steven: *Conceptual Mathematics*, Cambridge University Press, 1997.
- Pierce, Benjamin: *Basic Category Theory for Computer Scientists*, MIT Press, 1991.
- Barr, Michael and Wells, Charles: *Category Theory for Computing Science*, Prentice Hall, 1990.

Conventions Used in This Manual

The following font conventions are used in this manual:

- The `Courier` font is used to indicate Slang language keywords and literals, code, and the names of specifications, sorts, operations, diagrams, and other SPECWARE constructs.
- *`Courier italics`* are used to indicate variables in code.
- *Italics* are used to indicate emphasis, and to introduce new terms.
- The Helvetica font is used for menu items.

The following conventions are used to refer to emacs commands:

- `C-x` indicates the key combination Control-x.
- `M-x` indicates the key combination Meta-x.

Chapter 1

Introduction to Specware

This chapter describes the SPECWARE system and what you can use it to accomplish. The chapter introduces the basic terminology and design process.

What Is Specware?

SPECWARE is a tool for building and manipulating a collection of related specifications. SPECWARE can be considered:

- a *design tool*, because it can represent and manipulate designs for complex systems, software or otherwise
- a *logic*, because it can describe concepts in a formal language with rules of deduction
- a *programming language*, because it can express programs and their properties
- a *database*, because it can store and manipulate collections of concepts, facts, and relationships

Specifications are the primary objects in SPECWARE. A specification, or theory, describes a concept to some degree of detail. To add properties and extend definitions, you create new specifications that import or combine earlier specifications. Within a specification, you can reason about objects and their relationships. You declare *sorts* (data types) and *operations* (functions), *axioms* that state properties of operations, and *theorems* that follow logically from axioms.

A *morphism* is a relationship between specifications that describes how the properties of one map to the properties of another. Morphisms describe both part-of and is-a relationships. You can propagate theorems from one specification to another using morphisms; for example, if the QEII is a ship, and ships cannot fly, then the QEII cannot fly.

SPECWARE extends the notion of morphisms to *interpretations*. Interpretations map one theory to another by extending the target theory to define the concepts of the source theory. When you create an interpretation, you maintain the truth of theorems as you translate them to the target theory. By combining interpretations, you can translate your abstract specification all the way to an implementation in code. The process guarantees that the code correctly implements the theory.

What Is Specware For?

SPECWARE is a general-purpose tool that you can use to develop specifications for any system or realm of knowledge. You can do this as an abstract process, with no reference to computer programming; or you can produce a computer program that is known to be a correct implementation of a specification; or you can use the process to redesign an existing program.

You can use SPECWARE to:

- Develop domain theories

  You can use SPECWARE to do “ontological engineering”—that is, to describe a real-world domain of knowledge in explicit or rigorous terms. You might wish to develop a domain theory in abstract terms that are not necessarily intended to become a computer program. You can use the inference engine to test the internal logic of your theory, derive conclusions, and propose theorems.

  You can use specifications and morphisms to represent abstract knowledge, with no refinement to any kind of concrete implementation.

  More commonly, you would use SPECWARE to capture expert knowledge of engineering design. In this case you would refine your theoretical specifications and morphisms to more concrete levels.

- Develop code from specifications

You can use SPECWARE to develop computer programs from specifications. One advantage of using SPECWARE for this task is that you can prove that the generated code does implement the specification correctly. Another advantage is that you can develop and compare different implementations of the same specification.

- Develop specifications from code

You can use SPECWARE for reverse engineering—that is, to help you derive a specification from existing code. To do this, you must examine the code to determine what problems are being solved by it, then use Slang to express the problems as specifications. In addition to providing a notation tool for this process, SPECWARE allows you to operate on the derived specification. Once you have derived a specification from the original code, you can determine the specification's correctness and completeness, and also generate different implementations for it.

The Design Process in Specware

To solve real problems, programs typically combine domain theories about the physical world with problem solving theories about the computational world. Your domain theory is an abstract representation of a real-world problem domain. To implement it, you must transform the domain theory to a concrete computational model. The built-in specification library describes mathematical and computational concepts, which are building blocks for an implementation. Your specifications combine real-world knowledge with this built-in computational knowledge to generate program code that solves real-world problems in a rigorous and provable way.

You interpret designs relative to an initial universe of models. In software design, for example, the models are programs, while in engineering design, they are circuits or pieces of metal. To design an object is to choose it from among the universe of possible models. You make this choice by beginning with an initial description and augmenting it until it uniquely describes the model you desire. In SPECWARE, this process is called *refinement*.

Composition and refinement are the basic techniques of application building in SPECWARE. You compose simpler specifications into more complex ones, and refine more abstract specifications into more concrete ones. When you refine a specification, you create a more specific case of it; that is, you reduce the number of possible models of it.

The process of refinement is also one of composition. To begin the refinement, you construct primitive interpretations that show how to implement an abstract concept in terms of a concrete concept. You then compose interpretations to deepen and widen the refinement.

SPECWARE provides two types of composition for interpretations; horizontal (or parallel), and vertical (or sequential).

- When you compose interpretations horizontally, you increase the scope of what is refined. In the same way you compose specifications to create a more complex specification from simpler parts, you compose refinements horizontally to create a complex refinement from simpler parts.
- When you compose interpretations vertically, you increase the degree of refinement. You compose interpretations sequentially, in a simple, linear progression, to create a deeper refinement from a shallower one.

For example, suppose you are designing a house. A wide but not deep view of the design specifies several rooms but gives no details. A deep but not wide view of the design specifies one room in complete detail. To complete the refinement, you must create a view that is both wide and deep; however, it makes no difference which view you create first.

The final refinement implements a complex, abstract specification by interpreting it to code.

Stages of Application Building

There are three major stages in producing a SPECWARE application.

- Building a specification
- Refining your specifications to library specifications
- Refining the library specifications to code

Building a Specification

You must build a specification that describes your domain theory in rigorous terms. To do this, you first create small specifications for basic, abstract concepts, then specialize and combine these to make them more concrete and complex.

To relate concepts to each other in SPECWARE, you use specification morphisms. A specification morphism shows how one concept is a specialization or part of another. For example, the concept “fast car” specializes both “car” and “fast thing”. The concept “room” is part of the concept “house”. You can specialize “room” in different ways, one for each room of the house.

You specialize in order to derive a more concrete specification from a more abstract specification. Because the specialization relation is transitive (if *A* specializes *B* and *B* specializes *C*, then *A* specializes *C* as well), you can combine a series of morphisms to achieve a step-wise refinement of abstract specifications into increasingly concrete ones.

You combine specifications in order to construct a more complex concept from a collection of simpler parts. In general, you increase the complexity of a specification by adding more structural detail.

SPECWARE helps you to handle complexity and scale by providing composition operators that take small specifications and combine them in a rigorous way to produce a complex specification that retains the logic of its parts. SPECWARE provides several techniques for combining specifications, that can be used in combination:

- The *import* operation allows you to include earlier specifications in a later one.
- The *translate* operation allows you to rename the parts of a specification.
- The *colimit* operation glues concepts together into a shared union along shared subconcepts.

A shared union specification combines specializations of a concept. For example, if you combine “red car” with “fast car” sharing the concept “car”, you obtain the shared union “fast, red car”. If you combine them sharing nothing, you obtain “red car and fast car”, which is two cars rather than one. Both choices are available.

The colimit, or shared union, is pervasively used for composing structures of various kinds in SPECWARE. It is the basic means of achieving parallel composition, for both specifications and interpretations.

Building a Refinement

SPECWARE provides refinement techniques and tools to help you create an implementation that is completely consistent with the abstract specification. To refine a specification, you build interpretations between your abstract domain theories and the SPECWARE library of computational theories.

Interpretations allow the system to translate axioms between different terminologies, while maintaining the provability of all theorems. Figure 1-1 shows the parts of an interpretation.

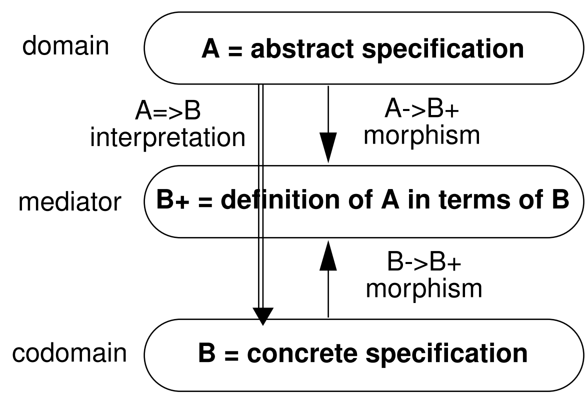

Figure 1-1 Parts of an interpretation

An interpretation requires a *mediator* between a more abstract source specification (the *domain*) and a more concrete target specification (the *codomain*). You choose a target that specifies a particular type of implementation. You create a mediating specification in which you define the source's sorts and operations in terms of the target's sorts and operations. You can then create the interpretation, which specifies exactly how the source and target specifications map to the mediator, and thus to one another. SPECWARE can, in many cases, establish the required morphisms automatically.

You combine interpretations to extend the refinement iteratively. The goal is to create an interpretation between the abstract specification of your problem domain and a concrete implementation of the problem solution in terms of sorts and operations that are defined in the SPECWARE library of mathematical and computational theories.

For example, suppose you want to create a specification for a card game. An abstract specification of a card game would include concepts such as *card*, *suit*, and *hand*. A refinement for this specification might map cards to natural numbers and hands to sets containing natural numbers.

The SPECWARE library contains specifications for various kinds of containers, including sets, and various kinds of numbers, including natural numbers.

To refine your abstract specification, you build an interpretation between the `HAND` specification and the `SET` specification. (Natural numbers are a built-in sort (`Nat`), and are automatically included in all other specifications.) To do this, you create a mediator that extends `SET` to define `Hands` of `Cards` as `Sets` of `Nats`. You would call the mediator `HAND-as-SET`, and the interpretation `HAND-to-SET`.

Refining to Code

SPECWARE contains Slang-to-Lisp and Slang-to-C++ interpretations for library specifications. Once you have refined your specification to a library specification (such as `INTEGER` or `SET`), you combine your interpretations with the Slang-to-*language* interpretations to generate code.

For example, once you have created the Slang-to-Slang interpretation `HAND-to-SET`, you can compose it with the Slang-to-Lisp interpretation `SET-to-LIST`, that translates sets into lists. As a result of the composition, SPECWARE builds a new Slang-to-Lisp interpretation that maps your original specification to a Lisp implementation.

Much of the refinement process is automatic and is accomplished using the graphical user interface. When it is completed, the system generates a code implementation of your specification in the target language, in a source-code file that you can save, test, and incorporate into an application.

Abstractness in Specware

SPECWARE allows you to work directly with abstract concepts independently of implementation decisions. You can put off making implementation decisions by describing the problem domain in general terms, specifying only those properties you need for the task at hand.

In most languages, you can declare either everything about a function or nothing about it. That is, you can declare only its type, or its complete definition. In SPECWARE you must declare the signature of an operation, but after that you have almost complete freedom in stating properties of the operation. You can declare nothing or anything about it. Any properties you have stated can be used for program transformation.

For example, you can describe how squaring distributes over addition:

```
axiom
  square(plus(a, b))
  = plus(plus (square(a), square(b)),
         times(2, times(a, b)))
```

This property is not a complete definition of the `square` operation, but it is true. The truth of this axiom must be preserved as you refine the operation. However, unless you are going to generate code for `square`, you do not need to supply a complete definition.

The completeness of your definitions determines whether you can create interpretations to code. An interpretation must completely define the operations of the source theory in terms of the target theory. This guarantees that, if the target is implementable, the source is also implementable. However, SPECWARE also allows you to construct *interpretation schemes*, which need not be definitional extensions. This allows you to make considerable progress in building up and refining an abstract specification in advance of providing complete definitions.

Logical Inference in Specware

SPECWARE performs inference using external theorem provers; the distribution includes the Kestrel Interactive Theorem Prover (KITP). External provers are connected to SPECWARE through *logic morphisms*, which relate logics to each other.

You can apply external reasoning agents to refinements in different ways (although only verification is fully implemented in the current release of SPECWARE).

- *Verification* tests the correctness of a refinement. For example, you can prove that quicksort is a correct refinement of the sorting specification.
- *Simplification* reduces the complexity of a refinement. For example, given appropriate axioms, in a function definition, you can rewrite:

  ```
  plus (times (3, a),times (3, b))
  ```

  to:

  ```
  times(3, plus(a, b))
  ```

- *Synthesis* derives a refinement for a given specification by combining the domain theory with computational theory. For example, you can derive quicksort semi-automatically from the sorting specification as a solution to a sorting problem, if you describe exactly how the problem is a sorting problem.

Different provers can provide different functionality. Additional provers are under consideration for integration with SPECWARE, including Stanford Research Institute's SNARK.

Chapter 2

Essential Specware Concepts

The SPECWARE specification language is called Slang and is a variant of higher-order logic. The *Specware Language Manual* describes Slang in detail. This chapter describes the basic concepts and terminology of the Slang language.

Specifications

Specifications are the primary objects in SPECWARE. A specification describes a concept or theory. In general, a specification is composed of:

- sorts
- sort axioms that describe the structure of sorts
- operations
- axioms, definitions, theorems, and constructors that describe the structure of operations

A specification need not include all of these elements. You can build specifications by defining them explicitly, importing or translating previously defined specifications, or combining previously defined specifications as a shared union.

The specification syntax begins with the keyword `spec` and ends with the keyword `end-spec`. By convention, the name of the specification is all upper-case with words separated by hyphens, the names of sorts have initial capitals, and the names of operations, axioms, definitions, and theorems are all lower-case. Variables are also lower-case. However, the SPECWARE system is not case-sensitive, and the printer frequently changes the case when you view an object through the user interface.

Comments can be introduced into a specification (or anywhere in a Slang source file) using one or more percent (`%`) characters. Any text on the same line following the percent character is considered a comment.

EXAMPLE The following specification describes the concept of partial order from abstract algebra.

```
spec PARTIAL-ORDER is
  %% declare a sort
  sorts E
  %% declare an operation's signature
  op leq: E, E -> Boolean
  %% describe the operation with axioms
  axiom transitivity-axiom is
      fa(z: E, y: E, x: E)
      leq (x, y) & leq (y , z) => leq (x , z)
  axiom reflexivity-axiom is
      fa(x: E)
      leq(x, x)
  axiom anti-symmetry-axiom is
      fa(y: E, x: E)
      leq(x, y) & leq(y, x) => x = y
 end-spec
```

Sorts and Sort Axioms

Sorts are like data types, except that they are uninterpreted; you need not describe their internal structure. This allows you to create and use abstract sorts, defining their construction later in the development process.

Certain sorts are part of Slang Base. That is, they are built into the system and can be used without declaration. The built-in sorts are `Nat`, `Char`, `String`, and `Boolean`. These are said to be defined in the empty specification, named `EMPTY`. Other sorts are defined in library specifications; for example, `INTEGER`, `SET`, and `LIST`. To use these, you must import the defining specification. (See "Composing Specifications" on page 3-2.)

You can declare new sorts in a specification. Use the `sort` keyword to declare a sort, followed by the sort name. By convention, sort names are capitalized. You can declare multiple sorts using the `sorts` keyword, separating the sort names with commas.

To describe the internal structure of a sort, use the `sort-axiom` keyword. You can construct sorts using products, sums, function spaces, subsorts, and quotients.

Product Sorts

The product sort combines two or more sorts into a tuple structure. Product sorts are used to pass or return multiple values. Elements of a product sort are constructed using tuple notation (as in the example below), and deconstructed using the `project` operation. The `project` operation extracts elements from the tuple. (This terminology is taken from the mathematical notion of projecting one space onto another—in this case, the tuple onto one of its components.)

EXAMPLE This specification describes a specific implementation of a simple bank account using a tuple composed of a string and an integer.

```
spec ACCOUNT-IMPL is
  import INTEGER
  sort Account
%% declare product sort
  sort-axiom Account = (String, Integer)

  op owner : Account -> String
  definition of owner is
   axiom fa (a : Account)
      owner(a)= project(1)(a) %% deconstruct product sort
  end-definition

  op balance : Account -> Integer
  definition of balance is
      axiom balance(a)= project(2)(a)
  end-definition

  op make-account : String, Integer-> Account
  definition of make-account is
      axiom make-account(s, i)
      = (s, i) %% construct product sort
  end-definition
```

```
end-spec
```

Sum Sorts

The sum sort (also known as a coproduct) combines two or more sorts into a disjoint union. An element of the sum sort can come from any of the sorts that compose the sum. Elements of a sum sort are constructed using the `embed` operation, and deconstructed using pattern matching (as in the example below). The `embed` operation maps an element of one of the component sorts into the sum sort.

EXAMPLE This specification defines a sum sort and a predicate operation that determines which of the sorts has been passed to it. An element of the sum sort `Integer + Error` can be either an `Integer` or an `Error`.

```
spec ERROR-VALUE is
  sorts Integer, Error, ReturnValue
%% declare sum sort
  sort-axiom ReturnValue = Integer + Error

  op error? : ReturnValue -> Boolean
  definition of error? is
      axiom error?(embed(2)(i)) = true
      axiom error?(embed(1)(i)) = false
  end-definition
 end-spec
```

For a complete specification of an `Error` sort that uses the sum sort, see the library spec `PARTIAL-as-TRIV-Defn-Extn`, which can be found in the distribution directory `library/slang-specs/partial`.

Function Sorts

A function sort allows you to treat a function as a value. It is used, for example, to pass a function to another function as a parameter. Elements of a function sort (functions) are constructed using `lambda`, and deconstructed by application (as in the example below).

EXAMPLE This operation describes the `map` function for sequences, which accepts a function and a sequence, and returns a sequence in which the function has been applied to each element of the original sequence. A function of type `E->E` is the first parameter.

```
op map : (E -> E), Seq -> Seq
%% map of identity is identity
```

```
axiom map(lambda(e : E) e, s) = s
%% f(e) applies f to e.
definition of map is
  axiom map(f, empty-seq) = empty-seq
  axiom map(f, prepend(e, L))
      = prepend(f(e), map(f, L))
end-definition
```

Subsorts

The subsort constructor builds the sort of elements satisfying a predicate. Elements of a subsort are constructed using pattern matching (as in the example below), and deconstructed using the `relax` operation. The `relax` operation includes the subsort into the parent sort (also called the supersort).

EXAMPLE This specification uses a subsort to describe the even natural numbers.

```
spec EVEN is
  sort Even
  sort-axiom Even = Nat | even?
  op even-to-nat : Even -> Nat
  definition of even-to-nat is
      axiom
          even-to-nat(e) = relax(even?)(e)
  end-definition
  op double : Nat -> Even
  definition of double is
      axiom
          relax(even?)(double(n)) = plus(n, n)
  end-definition
 end-spec
```

Quotient Sorts

The quotient constructor builds the sort of equivalence classes under an equivalence relation. An equivalence relation is one that is reflexive, transitive, and symmetric. Elements of a quotient sort are constructed using the `quotient` operation, and deconstructed using pattern matching (as in the example below). The `quotient` operation projects the parent sort onto the quotient sort.

EXAMPLE This specification uses a quotient sort to describe the integers modulo five.

```
spec INT-MOD-FIVE is
  import INTEGER
  sort Int-Mod-Five
  sort-axiom Int-Mod-Five = Integer / equal-mod-five?
  op equal-mod-five? : Integer, Integer -> Boolean
  axiom equal-mod-five?(n1, n2)
      <=> exists (k : Integer) times(5, k)
          = minus(n1, n2)
  op int-to-int-mod-five : Integer -> Int-Mod-Five
  axiom int-to-int-mod-five(n)
      = quotient(equal-mod-five?)(n)
  op plus : Int-Mod-Five, Int-Mod-Five -> Int-Mod-Five
  axiom plus(quotient(equal-mod-five?)(n1),
                quotient(equal-mod-five?)(n2))
          = quotient(equal-mod-five?)(plus(n1, n2))
end-spec
```

Operations

As in any typed language, each operation has a signature, which declares the sorts of elements that are passed to it and the sort of what it returns. Like sorts, operations are uninterpreted. You can choose to say only part of or nothing about how an operation works when it is declared, and describe it partially or fully later in the development process. When you want to describe the internal structure of an operation, you write axioms that state properties of the operation.

Two operations can have the same name if they have different signatures. This is called overloading. If there is any doubt about which operation is meant, you can specify the complete signature anywhere you can use the name alone. For example, where `plus` could refer to either plus-for-naturals or plus-for-integers, you can specify `plus : Nat, Nat -> Nat` in place of `plus`. Sometimes SPECWARE can resolve ambiguities, but not always.

In Slang, every operation is total and deterministic, meaning that it maps every element of its domain to a unique element of its codomain (range). If an operation is not normally total, you can:

- reduce its domain using a subsort
- extend its codomain using a sum

For example, division is not total because it is not defined for zero. The following example uses both methods to make division total.

EXAMPLE This specification defines division for real numbers.

```
spec DIVISION is
  sorts Real, Nonzero-Real, Real-with-Error, Error
  sort-axiom Nonzero-Real = Real | Non-Zero
  sort-axiom Real-with-Error = Real + Error
  sort-axiom Error = () %% empty product
  op div-with-subsort : Real, Nonzero-Real -> Real
  op div-with-sum : Real, Real -> Real-with-Error
 end-spec
```

An operation whose signature is a single sort is called a constant. You can declare a constant using either the `op` keyword or the `const` keyword. Constants are not semantically different from other operations; the `const` keyword is provided for purposes of readability.

EXAMPLE This declares and sets the value of a numeric constant.

```
const max : Nat
axiom max = 100
```

Axioms

Axioms state properties that operations must satisfy. Axioms are written in higher-order logic with implicit universal quantification.

You declare an axiom using the `axiom` keyword. The axiom can be named, and the implicit universal quantification can be made explicit with the construct `fa` (*for all*). You can specify existential quantification with the construct `ex` (*there exists...such that*).

EXAMPLE This is a named axiom in which the quantification is made explicit:

```
axiom reflexivity-axiom is
   fa(x: E) %% for all x of sort E
   leq(x, x)
```

EXAMPLE Here are some axioms characterizing order on a line.

```
%% between?(p, p1, p2) means that p is between
%% p1 and p2. p1 and p2 are unordered.

axiom between?(a, b, c)
      or between?(b, a, c)
```

```
      or between?(c, a, b)
axiom  ex(p : Pt) %% there exists p of sort Pt such that
       between?(p, p1, p2)
```

Definitions

A definition is a collection of axioms that is expected to completely define an operation. A theorem prover could verify that the axioms in a definition do indeed define a total and deterministic relation (and therefore a function).

The code generator translates definitions into Lisp or C++. It does not translate axioms or theorems that are not part of a definition.

One specification can be a *definitional extension* of another. A definitional extension defines new concepts in terms of the sorts and operations of the specification it extends. A specification `A+` is a definitional extension of `A` if:

- it introduces no new axioms for the sorts and operations of `A`
- any sorts or operations that it adds have definitions

Definitional extensions are used to create interpretations. The mediator of an interpretation must be a definitional extension of its target. It provides a definition for each sort and operation in the source, in terms of the sorts and operations in the interpretation's target. (See "Interpretations" on page 2-12.)

When SPECWARE can determine that one specification in a diagram is a definitional extension of another, the morphism between them is labeled with a **d**. (See "Diagrams" on page 2-19.)

You declare a definition for an operation using the `definition` keyword. One or more axioms are contained in the definition, which ends with the `end-definition` keyword.

EXAMPLE This specification contains a definition for the factorial operation. Notice that in this example a theorem prover would have to do considerable work to verify the definition, because of the recursion.

```
spec FACT is
  op fact : Nat -> Nat
  definition of fact is
   axiom n = 0 => fact(n) = 1
   axiom ~(n = 0) =>
         fact(n) = times(n, fact(minus(n, 1)))
```

```
  end-definition
end-spec
```

Theorems

Theorems are like axioms, except that they are expected to follow logically from the axioms of a specification. You can use a theorem prover to verify that they actually do. When you verify a specification, the theorem prover attempts to prove all theorems in the specification.

You declare a theorem using the `theorem` keyword. As for axioms, the theorem can be named, and you can use the constructs `fa` (*for all*) and `ex` (*there exists...such that*).

EXAMPLE This theorem states that there are infinitely many primes:

```
theorem
  fa(n : Nat) %% for all n of sort Nat
  ex(p : Nat) %% there exists p of sort Nat such that
     gt(p, n) & prime?(p)
```

Constructors

A constructor set is a set of operations that have a common codomain (range), and whose image is all of the codomain. Declaring a constructor set is equivalent to writing an induction axiom for the constructors.

You declare a constructor set using the `constructors` keyword. The names of the operations are given in braces, and the `construct` keyword is followed by the sort that is constructed.

EXAMPLE This is the standard constructor set for natural numbers.

```
constructors {zero, succ} construct Nat
```

Morphisms

Specification morphisms relate specifications to each other. A morphism maps each sort and operation in the source to a sort or operation in the target. Using this map, any sentence in the source can be translated into a sentence in the language of the target. Additionally, morphisms must map elements in such a way that the axioms in the source translate into theorems in the target.

Morphisms are directional; they are represented graphically by an arrow between specification, as shown in Figure 2-1.

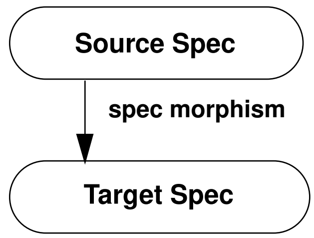

Figure 2-1 A specification morphism

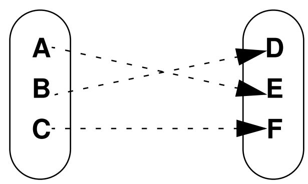

Figure 2-2 The morphism maps sorts and operations from the source to the target

Morphisms can be defined with the `morphism` keyword, or can occur as part of a diagram or interpretation definition.

EXAMPLE This morphism maps an abstract specification of an amount to an explicit specification of an amount in US dollars.

```
morphism AMT-to-DOL-morphism : AMOUNT -> DOLLAR-AMOUNT
   =   { Amount ->  Dollar-Amount,
          (plus: Amount,Amount -> Amount)    ->  d-plus,
          (minus: Amount, Amount -> Amount)  ->  d-minus,
          sign   ->  sign
        }
```

Notice that in this example, an operation name that could be ambiguous (`plus`) is made unambiguous by including its signature. This distinguishes it from, for example, the `plus` operation defined for natural numbers.

The single notion of specification morphism plays many intuitively different roles. If there is a morphism from specification A to specification B, you can say that:

- A is a part of B
- A is a view of B
- B is an implementation of A
- B is a specialization of A
- B is a refinement of A

Using specification morphisms you can:

- Transfer derived properties between specifications.
- Compose specifications from parts.
- Decompose specifications into parts.

The specification construction operations automatically create morphisms between the constructed specification and its defining parts. (See "Composing Specifications" on page 3-2.) Table 2-1 shows the keywords you can use to refer to these morphisms.

Table 2-1 Morphism Keywords

Keyword	Description
`identity-morphism`	Describes how spec A is related to itself
`import-morphism`	Describes how spec A is related to spec B that it imports
`translation-morphism`	Describes how spec A is related to spec B that it translates
`cocone-morphism`	Describes how a spec A that is a shared union (colimit) is related to those specs from which it is derived

When elements have the same name in both the source and the target specifications, you need not explicitly state their correspondence. For example, the morphism {A -> A, B ->B, C ->D} can be stated as {C->D}. When this is the case for all elements, you can indicate the default morphism by empty braces `{ }` in the morphism definition. You must explicitly state any other relationships between the braces when a morphism is defined.

Interpretations

Interpretations generalize morphisms. A morphism maps the source directly to the target; an interpretation maps the source to an *extension* of the target, which defines the source's sorts and operations in terms of the target's sorts and operations. Where a morphism maps operations in the source to pre-existing operations in the target, an interpretation maps operations to newly constructed operations.

An interpretation describes an implementation of a more abstract concept in terms of a more concrete concept. The source is the more abstract specification, and the target is the more concrete specification, as shown inFigure 2-3.

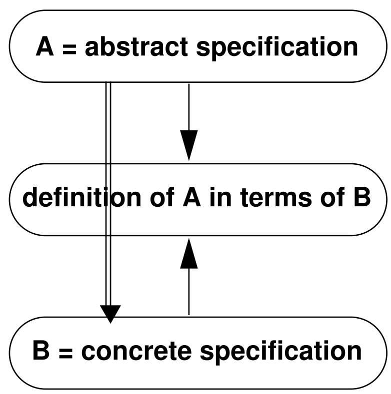

Figure 2-3 An interpretation is an implementation

Morphisms are not sufficient to create implementations, because the target specification does not always (or usually) define all the concepts needed by the source. When you create an interpretation, you extend the target specification to supply the definitions you need. For example, you can implement sets as sequences. To create an interpretation between set and sequence, you take `SET` as the source and `SEQ` as the target. You must define each set operation in terms of sequences. Since the `SEQ` specification does not already define all the set operations, you create an extension of it, `SET-as-SEQ`.

An interpretation maps a *domain*, or source specification, to a *codomain*, or target specification, through a *mediator* which extends the target. The interpretation is constructed from the mediator and two morphisms. The interpretation is directional, as shown in Figure 2-4.

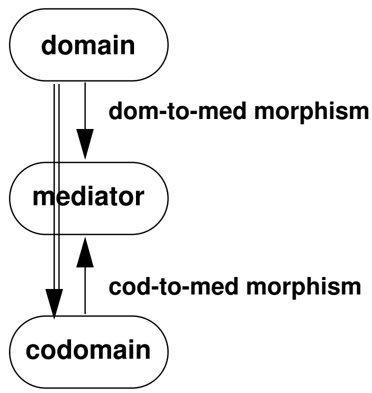

Figure 2-4 An interpretation

The mediator provides a definition for each of the sorts and operations in the source specification in terms of the target. The domain-to-mediator morphism maps the sorts and operations to their definitions.

The codomain-to-mediator morphism shows how the codomain is part of the mediator. You generally construct a mediator by importing the target specification, then defining each of the sorts and operations of the source. Thus, the codomain-to-mediator morphism is often an import morphism.

The mediator is required to be a *definitional extension* of the interpretation's target specification. Any new sorts or operations that it adds to the target must be fully defined. This ensures that if the target can be implemented in code, the source can also be implemented.

To interpret a concept in terms of built-in sorts and operations (such as `Nat` and `plus` for `Nat`), create a mediator between the source specification and the empty specification, named `EMPTY`. To implement the concept in terms of sorts and operations from a library specification (or a specification of your own), create a mediator between the source and the specification where the sorts and operations are defined.

You can create interpretations interactively using the Refinement Editor, or in Slang text, using the `interpretation` keyword. Specify the domain, codomain, mediator, and morphisms from the domain and codomain to the mediator. By convention, the mediator specification is named `SOURCE-SPEC-as-TARGET-SPEC`, and the interpretation is named `SOURCE-SPEC-to-TARGET-SPEC`.

EXAMPLE This interpretation maps sets to lists:

```
interpretation SET-to-LIST :
  SET => LIST is
  mediator SET-as-LIST
  domain-to-mediator {Set       -> Set-As-List,
                      empty-set -> set-empty,
                      empty?    -> set-empty?,
                      nonempty? -> set-nonempty?,
                      singleton -> set-singleton,
                      insert    -> set-insert,
                      in        -> set-in,
                      union     -> set-union,
                      delete    -> set-delete,
                      size      -> set-size,
                      subset    -> set-subset,
                      exists    -> set-exists,
                      forall    -> set-forall,
                      reduce    -> set-reduce,
                      image     -> set-image,
                      filter    -> set-filter
                      }
  codomain-to-mediator { }
```

Methods for Creating Interpretations

There are a number of methods available by which you can construct an interpretation either in Slang text or in the graphical user interface. When you create an interpretation in text, you typically construct a mediator by importing the target specification, then defining each of the sorts and operations of the source. However, there are other ways of constructing interpretations. For example:

- If a morphism already exists between the source and the target you can convert the morphism into an interpretation by using the morphism target as both the target and the mediator of the interpretation. In this case, the domain-to-mediator morphism is the existing morphism, and the codomain-to-mediator morphism is an identity morphism.
- You can create an interpretation where the source, target, and mediator are all the same specification. In this case, all of the morphisms are identity morphisms.
- You can create an interpretation where the source is a translation of the target, and the target is used as the mediator. In this case, the domain-to-mediator morphism is a translation morphism, and the codomain-to-mediator morphism is an identity morphism.

You might want to create interpretations in these ways in order to combine them with other interpretations. Combining interpretations is another important way to create new interpretations; see "Composing Interpretations" on page 3-7.

SPECWARE can use these methods to create interpretations automatically, to some extent. When you create a new interpretation through the SPECWARE menus, the Select/Construct Interpretation menu gives you a choice of methods. The menu only offers those methods that apply in context. The possible methods are shown in Table 2-2.

Table 2-2 Methods for Creating Interpretations

Method (Menu Item)	Description
Declaration	Declare in Slang. A=>B through mediator B+, a definitional extension of B.
Selection	Select an existing interpretation from menu.
Identity (identity)	A=>A. A is source, target, and mediator.

Table 2-2 Methods for Creating Interpretations

Method (Menu Item)	Description
Convert Spec Morphism (by lifting a spec morphism)	A => B where a morphism exists from A to B. B is target and mediator.
Translation (via translation)	B=> A where B is a translation of A. A is target and mediator.
Definitional Extension (via definitional extension)	A+ => A where A+ is a definitional extension of A. A+ is source and mediator.
Sequential Composition (via sequential composition)	Build A=>C from A=>B and B=>C
Parallel Composition (via colimit)	Build A+C=>B+D from A=>B and C=->D

Notice that, except for the composition methods, these techniques could all be used to create interpretations in Slang text as well. There is no specific syntax for them; simply define the source, mediator, target and morphisms appropriately.

The via translation and via definitional extension menu options are provided as variations of sequential composition, where the first interpretation in the sequence is automatically derived by the chosen method. To complete the composition, you must specify an additional interpretation, which is combined sequentially with the constructed interpretation, as shown in Figure 2-5. (See “Composing Interpretations Sequentially” on page 3-9.)

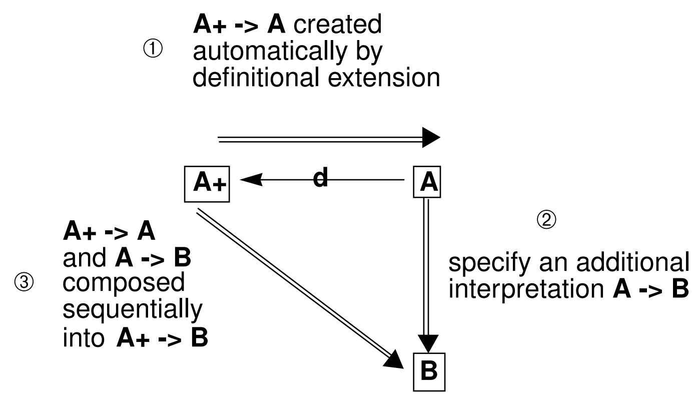

Figure 2-5 The definitional extension method through menus requires an extra step

Interpretation Schemes

An interpretation scheme is a partially constructed interpretation. It differs from an interpretation in that the mediator need not be a definitional extension of the target. This looser kind of interpretation allows you to construct a mediator using abstract sorts or operations that are not yet completely defined.

You define an interpretation scheme using the keyword `ip-scheme`. The syntax is otherwise identical to that of an interpretation.

It is a feature of SPECWARE that you can use abstract concepts to build up refinements in advance of the detailed work of providing complete definitions. However, when you build a refinement using concepts that are not completely defined, it cannot necessarily be implemented as code. Any code that you do produce with it is incomplete in those areas that are not fully defined.

You can combine interpretation schemes to produce a strict interpretation. However, the definition of each operation must be completely contained in a single interpretation scheme. Between them, the component schemes must define all of the operations of the source specification. See “Composing Interpretation Schemes” on page 3-16.

Interpretation Morphisms

An interpretation morphism maps one interpretation to another, in the same way that a specification morphism maps one specification to another. It shows how one interpretation is part of another, or is compatible with it. For example, an interpretation that refines a single abstract sort to `Nat` is part of any other interpretation that refines a sort to `Nat`. Interpretation morphisms specify the shared part of two interpretations, so that they can be combined into a shared union. (See "Parallel Composition" on page 3-9.)

An interpretation morphism is composed of three morphisms, as shown in Figure 2-6. There is one morphism between the domains, one between the codomains, and one between the mediators of the source and target interpretations.

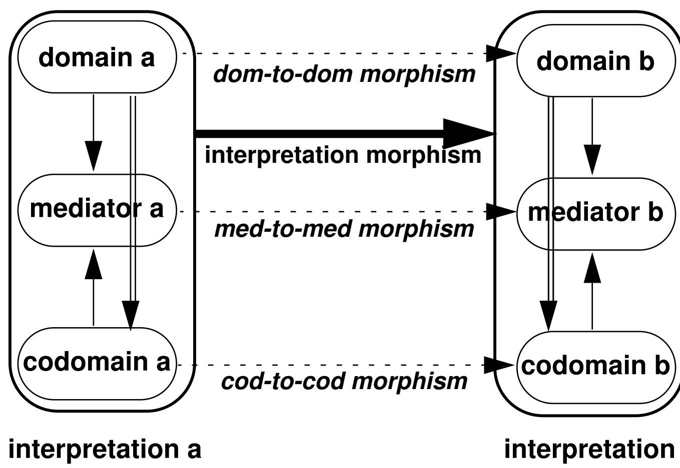

Figure 2-6 Parts of an interpretation morphism

You can create interpretation morphisms automatically as part of the process of combining interpretations through the Refinement Editor in the SPECWARE user interface. See "Interpretation Diagrams" on page 2-21 and "Parallel Composition" on page 3-9.

You can also create interpretation morphisms explicitly in Slang text, using the `ip-scheme-morphism` keyword. (Note that you use this syntax regardless of whether the morphism is between interpretations or interpretation schemes.) Specify the source and target interpretations, and the morphisms between the domains, mediators, and codomains of the two interpretations.

EXAMPLE The `SIMPLE-ACCOUNT-to-EMPTY` and `ACCOUNT-with-WITHDRAW-to-EMPTY` interpretations refine parts of a checking account toward an implementation using the built-in sorts `Nat` and `String`. This code creates an interpretation morphism between them, so that the interpretations can be combined.

```
ip-scheme-morphism ACCOUNT-with-WITHDRAW-ipsm :
  SIMPLE-ACCOUNT-to-EMPTY ->
     ACCOUNT-with-WITHDRAW-to-EMPTY
  is
  domain-sm import-morphism
  mediator-sm import-morphism
  codomain-sm identity-morphism
```

Diagrams

A diagram is a graphical representation of a collection of related objects. The nodes of the diagram show the objects and arcs between them show their relationships. Diagrams are used primarily to create sets of objects and specify their shared parts, so that they can be combined using parallel composition. (See Chapter 3, “Composition in Specware”.)

The SPECWARE system uses two basic types of diagrams, specification diagrams and interpretation diagrams.

Specification Diagrams

A specification diagram, also known as a spec/morphism diagram, defines a set of specifications related by morphisms. Specifications are attached to nodes and morphisms between specifications are attached to arcs between the nodes. It is necessary to organize specifications into diagrams in order to combine them into a shared union using the `colimit` operation. See “Composing Specifications” on page 3-2.

Arcs in diagrams are often labeled to indicate some property of the attached morphism. Table 2-3 shows the types of morphisms and their identifying labels.

Table 2-3 Arc Labels in Diagrams

Arc Label	Morphism Type
i	import morphism
d	definitional extension
t	translation morphism
c	cocone morphism
id	identity morphism

Specification diagrams can be created and displayed graphically, as shown in Figure 2-7.

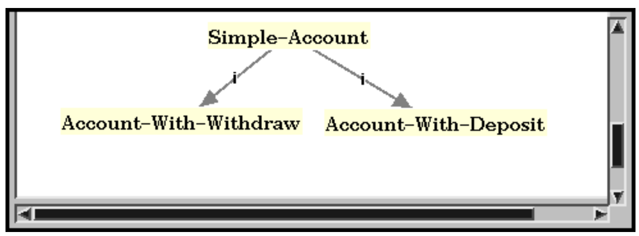

Figure 2-7 A spec/morphism diagram

Specification diagrams can also be created in Slang text. Use the `diagram` keyword to create a spec/morphism diagram, either at the top level of the Slang file or as part of a `colimit` expression. Specify the nodes (specifications) and arcs (morphisms), separated by commas.

EXAMPLE This diagram includes the partial specifications of a checking account, which will later be combined using a colimit.

```
diagram ACCOUNT-DIAGRAM is
 nodes ACCOUNT-with-DEPOSIT,
       ACCOUNT-with-WITHDRAW,
       SIMPLE-ACCOUNT
 arcs
   SIMPLE-ACCOUNT -> ACCOUNT-with-WITHDRAW :
      import-morphism,
   SIMPLE-ACCOUNT -> ACCOUNT-with-DEPOSIT :
      import-morphism
end-diagram
```

Interpretation Diagrams

An interpretation diagram represents interpretations related by interpretation morphisms. It is necessary to organize interpretations into diagrams in order to combine them into a shared union using the via colimit menu item on the Select/Create Interpretation menu. See "Parallel Composition" on page 3-9.

You can create and operate on interpretation diagrams only through the Refinement Editor in the SPECWARE user interface.

When you are creating a new interpretation, dotted lines indicate unresolved arcs and nodes; that is, those to which specific objects are not yet attached, as shown in Figure 2-8. Notice that before the interpretation is attached, the target node is also unresolved. Each pair of nodes connected by a blue line represents one interpretation.

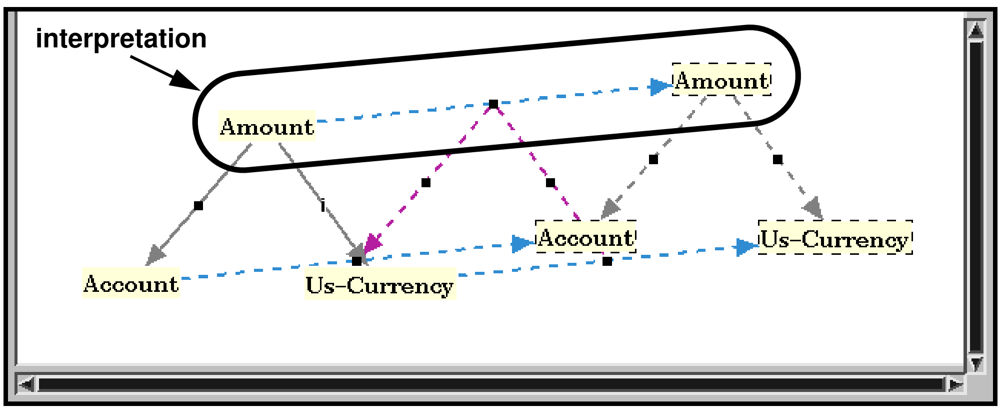

Figure 2-8 An interpretation diagram

When you attach an interpretation to an arc, it becomes a solid line in the appropriate color, and the target node becomes attached to the target of the chosen interpretation. Figure 2-9 shows a resolved diagram. This diagram also shows the interpretation that is created when you combine the defining interpretations via colimit. (See "Parallel Composition" on page 3-9.)

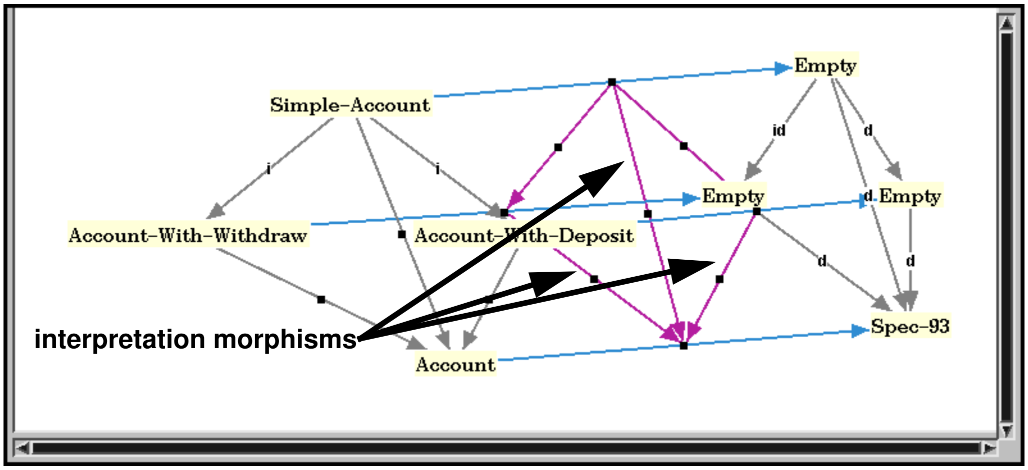

Figure 2-9 An interpretation diagram with colimit

The color of an arc in the diagram indicates what kind of object can be attached to it. (The colors are different for language-specific interpretations; click Color Legend on the menu bar to see what colors are used.)

- Each interpretation in the diagram appears as two nodes connected by a blue arc. The interpretation is attached to the arc. The source and target specifications of the interpretation are attached to the nodes.
- A box at the center of the blue arc graphically represents the mediator, and is also the selection point for the arc and its attached interpretation.
- Gray arcs between nodes are attached to morphisms between source and target specifications of the interpretations.
- The purple arcs between the mediator points are attached to interpretation morphisms. They also graphically represent the morphisms between the mediators of the interpretations.

Inference

SPECWARE contains an inference engine called the Kestrel Interactive Theorem Prover (KITP). As its name implies, this engine is used to prove that theorems follow from the stated axioms.

You can use the theorem prover to verify a specification, meaning that any explicitly defined theorems in that specification are provable from the stated axioms in that specification.

Additionally, you can use the theorem prover to verify morphisms. When you create a morphism, a theorem is automatically created in the target to correspond to each axiom in the source. (These theorems are created internally, and are not actually added to the target specification.) When you verify a morphism, the prover verifies that each of these theorems does follow from the axioms in the target.

You can set environment preferences that control the behavior of the various parts of the theorem prover. (See Appendix A, "Customizing Specware".) KITP contains facilities that can be used to maintain consistency and improve the quality of generated code, although these are not currently accessible. Only the verifier is currently available to you through the Verify option on the Spec and Spec Morphism menus.

Chapter 3

Composition in Specware

When you compose specifications, you build up a more complex overall specification from simpler parts. When you compose morphisms and interpretations, you build up a more complex overall interpretation from simpler parts.

- The first stage of building an application is to define and compose specifications that describe your abstract domain theory. You do this almost entirely in text.
- The second stage of building an application is to build a refinement of the specification created in the first stage. You do this by creating and composing interpretations. You can perform some operations in text, but this process takes place largely in the graphical user interface. In the second stage, you build and compose interpretations that map your abstract domain specifications to the concrete computational specifications of Slang Base and the SPECWARE library.
- The third stage of building an application is to further refine your specifications to code in a specific computer language. You combine the Slang-to-Slang interpretations of the second stage with Slang-to-code interpretations, thus creating a Slang-to-code interpretation that maps your abstract domain specifications to Lisp or C++ code.

Composing Specifications

SPECWARE provides three methods for composing specifications:

- Importation
- Translation
- Shared Union

Importation

The `import` operation includes one specification in another. Importing allows you to build up larger specifications out of smaller ones by extension. Importing a specification is essentially the same as including the entire definition of the imported specification inline.

EXAMPLE This specification imports the library specification `INTEGER` in order to extend it.

```
spec EXTENDED-INTEGER is
 import INTEGER

 op nat-to-integer: Nat -> Integer
 definition of nat-to-integer is
  axiom n = zero =>
        nat-to-integer(n) = zero
  axiom ~(n = zero) =>
        nat-to-integer(n) = succ(nat-to-integer(pred(n)))
 end-definition

 op integer-to-nat: Integer -> Nat
 definition of integer-to-nat is
  axiom gt(n, zero) =>
        integer-to-nat(n) = succ(integer-to-nat(pred(n)))
  axiom n = zero =>
        integer-to-nat(n) = integer-to-nat(abs(n))
  axiom lt(n, zero) =>
        integer-to-nat(n) = integer-to-nat(abs(n))
 end-definition

  end-spec
```

SPECWARE automatically creates a morphism from a specification A to a specification B that imports A. You can refer to this morphism using the `import-morphism` keyword. The morphism maps each sort and operation in A to its copy in B.

Translation

The `translate` operation allows you to impose a new terminology by renaming the sorts and operations of a specification. Translation creates a copy of the original specification. In the copy, only the new names can be used.

EXAMPLE This specification imports a translation of the `ACCOUNT` specification in order to extend it.

```
spec CHECKING-ACCOUNT is
  import translate ACCOUNT by
  { Account -> Checking-Account }
  op account-type : Checking-Account -> String
  axiom fa (a : Checking-Account)
      account-type(a) = "checking"
 end-spec
```

SPECWARE automatically creates a morphism from a specification A to a specification B that translates A. You can refer to this morphism using the `translation-morphism` keyword. The morphism maps each sort and operation in A to its translation in B.

Shared Union

The `colimit` operation creates a more complex specification from simpler ones contained in a diagram. (The term colimit is taken from category theory.) The combined specification is called a colimit or a shared union.

EXAMPLE This specification is the shared union of a diagram that combines two aspects of a bank account.

```
spec ACCOUNT is
  colimit of diagram
      nodes ACCOUNT-with-DEPOSIT,
           ACCOUNT-with-WITHDRAW,
           SIMPLE-ACCOUNT
      arcs
         SIMPLE-ACCOUNT -> ACCOUNT-with-WITHDRAW :
```

```
            import-morphism,
        SIMPLE-ACCOUNT -> ACCOUNT-with-DEPOSIT :
            import-morphism
end-diagram
```

You can include the defining diagram inline, as in this example, or you can specify an existing diagram by name. When a specification is created as a colimit, you do not use the `end-spec` keyword.

SPECWARE automatically creates a morphism from each specification in the defining diagram to the colimit. You can refer to these morphisms using the `cocone-morphism` keyword.

The shared union is the primary means for composing complex specifications. When you create a shared union you indicate explicitly which concepts to share. For example, you can combine “red car” with “fast car”, sharing the concept “car” to obtain “fast, red car” rather than “red car and fast car” (which would be two cars rather than one).

The shared union can be represented by a graph. There are two graphical views; the *defining diagram*, which shows only the spec/morphism diagram from which the shared union is formed; and the *cocone diagram*, which also shows the colimit and the cocone morphisms. A cocone diagram is shown in Figure 3-1.

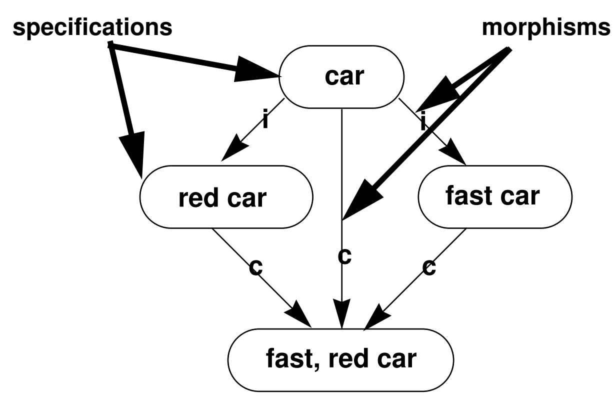

Figure 3-1 Using colimit to combine specifications

The defining diagram for a shared union is not limited to this three-node structure. You can take a colimit of any diagram. An example of a different diagram shape is shown in Figure 3-2.

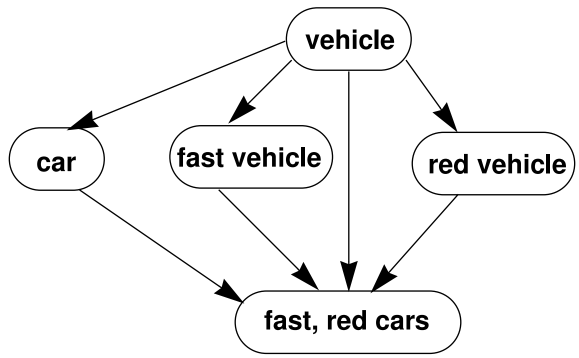

Figure 3-2 Using colimit on a more complex diagram

When you take a colimit, any sorts related by morphisms in the diagram are mapped to the same sort in the shared union. Conversely, any unrelated sorts are mapped to different sorts in the shared union. The same is true of operations.

Naming in Composed Specifications

When you compose specifications, sorts or operations that have the same names in different component specifications might be mapped to different result operations. For example, suppose specifications **A** and B are combined to form C. Both A and B have operations named `concat`, but the operations do not work the same way, and need to be differentiated in C.

In this case, SPECWARE generates unambiguous names in the shared union specification. To differentiate the names, the system prefixes the node name from the defining diagram, and postfixes a number. For example, the resulting operations in C are given the generated names `A-concat-1` and `B-concat-2`.

Similarly, sorts and operations that have different names in the component specifications can be mapped to a single element in the composed specification. For example, the operation `concat` in spec A and `add` in spec B might both be mapped to a single concatenation operation in the colimit spec C.

In this case, the resulting element has both names; you can use either of the component names to refer to it. When you view a composed specification, the printer displays only one name for each element, which it chooses arbitrarily.

There is no syntax for defining a sort or operation with multiple names. Multiple naming occurs only through composition.

When you translate a composed specification that contains multiple names for single elements, the translation collapses the names. The name to which you translate an element becomes its only name. Any additional names are discarded, and can no longer be used to refer to the element.

Parameterized Specifications

A parameterized specification (*pspec*) is a specification that can be instantiated on a parameter. For example, `LIST` is a parameterized specification; the `LIST-of-REALS` specification instantiates `LIST` on `REAL` to define lists that contain real numbers.

NOTE `LIST` and `ARRAY` are the only parameterized specifications currently available. Future releases will allow you to define new parameterized specifications. ❖

To instantiate a parameterized specification, you create a shared union with the specification that defines the parameter; for example, the sort that will really be contained in a collection.

The `LIST` and `ARRAY` specifications define `List` and `Array` sorts that contain elements of sort `E`. An abstract sort `E` is defined in a specification called `TRIV`. You use `TRIV` to combine `LIST` or `ARRAY` with another specification in a shared union.

For example, the `LIST-of-REALS` specification is created as a colimit of `LIST`, `TRIV`, and `REAL`. Figure 3-16 shows a defining diagram for `LIST-of-REALS`.

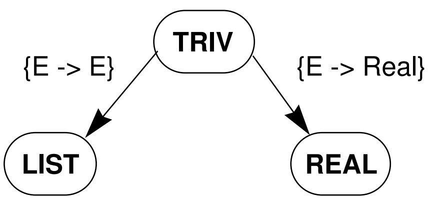

Figure 3-3 Defining diagram for `LIST-of-REALS`

Composing Morphisms

A morphism between specifications shows how the source is a part of the target. This relationship is transitive: if A is part of B and B is part of C, then A is part of C. Thus, you can compose morphisms A->B and B->C to show how specification A is part of specification C.

It is possible to compose morphisms explicitly, although it is not generally necessary. Morphisms are automatically combined by SPECWARE when you perform other operations. To compose morphisms explicitly, choose via sequential composition from the Create/Select Morphism menu. (See "Composing Morphisms" on page 6-1.) You would compose morphisms explicitly in order to create new morphisms that you need to compose specifications, or in order to propagate the theorems in one specification directly to another that is related to it through a chain of morphisms.

Composing Interpretations

You compose interpretations in order to build more complex interpretations from simpler ones. You can compose interpretations sequentially or in parallel.

- Sequential (or vertical) composition combines layers of refinement, building deep refinements from shallow ones. You build a sequential refinement to take each part of your specification from the abstract to the concrete.

- Parallel (or horizontal) composition builds refinements of larger specifications from refinements of smaller ones. You build a parallel refinement to combine the parts into a more complex structure.

In creating an application you generally combine both techniques of composition to build refinements that are both wide and deep. You use parallel composition to increase the scope of what is specified, and you use sequential composition to increase the degree to which the specification is refined.

Although you must do both, you can approach the tasks in any order. That is, you can refine all the parts sequentially, then combine them in parallel, or you can create the complex structure first by parallel composition, then refine the entire structure sequentially. In a complex application, you will probably alternate methods more than once.

Sequential Composition

Sequential composition combines layers of refinements. Each interpretation maps a more abstract specification to a more concrete one. When you combine interpretations sequentially, you create a new interpretation from the most abstract specification to the most concrete one. No interpretation morphisms are required for this type of composition.

Sequential composition can only be applied to vertically compatible interpretations. Two interpretations are vertically compatible if the first ends where the second begins; that is, the target specification of the first interpretation is identical to the source specification of the second interpretation (A=>B + B=>C -> A=>C). The interpretations in Figure 3-4 are vertically compatible because the first ends with 'ship', and the second begins with 'ship'.

You use sequential composition to interpret an abstract specification in increasingly concrete terms. To create an implementation, you continue this process until you have reached the level of code. For a complete discussion, see "Generating Code" on page 3-18.

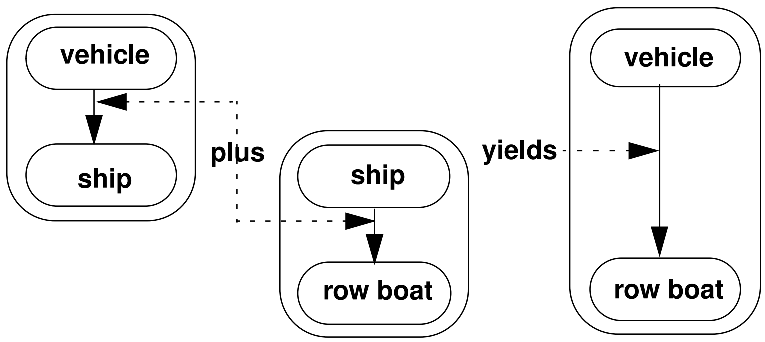

Figure 3-4 Example of sequential composition

Composing Interpretations Sequentially

To combine interpretations sequentially, focus on the source specification of the first interpretation in the sequence. Choose Create new interpretation from, then choose the method via sequential composition. Choose each interpretation in the sequence in turn, until the sequence is complete. At each point, the menu offers only those interpretations that are vertically compatible with the previous interpretation in the sequence.

The methods via translation and via definitional extension are variations of sequential composition, where the first interpretation in the sequence is constructed by the chosen method.

Parallel Composition

Parallel composition of interpretations builds refinements of larger specifications from refinements of smaller ones. You build the complexity, or width, of the refinement by composing interpretations in parallel, or horizontally.

Parallel composition can only be applied to horizontally compatible interpretations. Two interpretations are horizontally compatible if they refine shared concepts identically. You can, for example, compose “car => fast Porsche” with “car => red Porsche” as shown in Figure 3-5. These interpretations are compatible because each refines the shared concept “car” to “Porsche”. The interpretations “car => fast Porsche” and “car => red Chevrolet” are incompatible, because each one refines the shared part “car” in a different way.

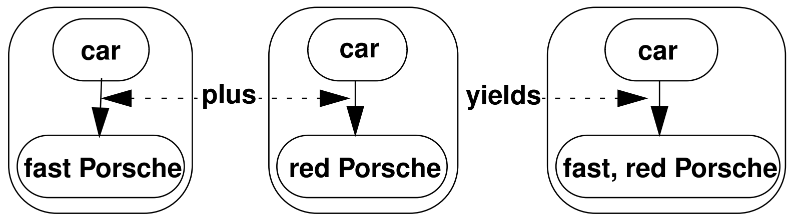

Figure 3-5 Parallel composition with shared part

The interpretations in Figure 3-6 are compatible because there is no shared part.

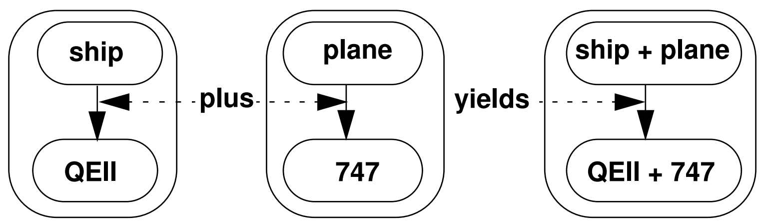

Figure 3-6 Parallel composition with no shared part

Because there is no shared part, ships and planes do not interact in the “ship+plane” specification (or in the “QEII+747” specification). However, you can use this new composed interpretation to refine an extension of the combined “ship+plane” specification. For example, suppose you extend “ship+plane” to describe the FAA regulations for seaplanes. You could then extend this interpretation to describe how seaplane regulations apply to a 747.

Parallel composition combines interpretations in a shared union. This is exactly the same as combining specifications with the `colimit` operation, except that the nodes of the defining diagram are interpretations, rather than specifications. Just as a specification diagram describes the shared parts of a colimit specification, an interpretation diagram describes the shared parts of a colimit interpretation. Interpretation morphisms define how the component interpretations are mapped into the shared union. Figure 3-7 shows a parallel composition of interpretations.

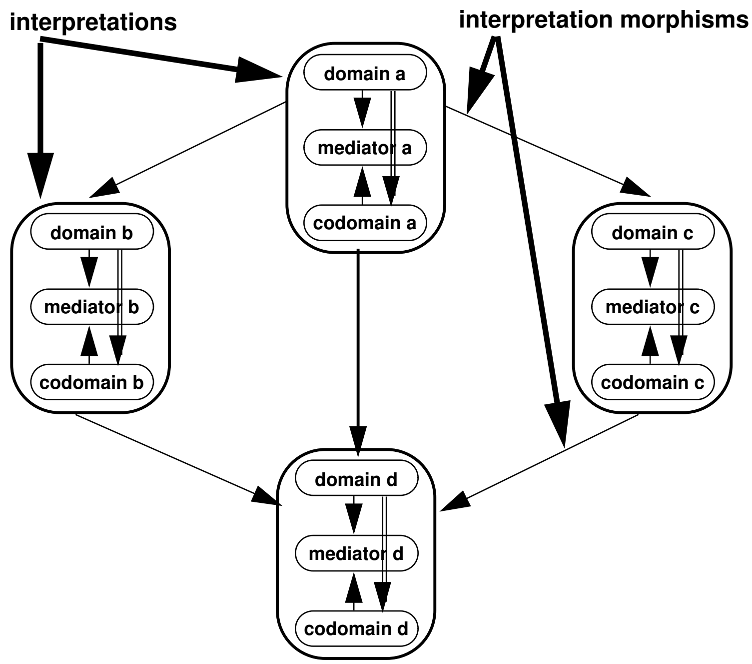

Figure 3-7 Interpretation morphisms relate interpretations in an interpretation diagram

Composing Interpretations in Parallel

You create a refinement of a combined specification by combining the refinements of its parts. For example, suppose that specification A is the colimit of a diagram, and you have created a corresponding diagram with more concrete versions of each specification. All of the specifications in the defining diagrams are linked by interpretations, as shown in Figure 3-8.

You can combine these interpretations in parallel to create an interpretation of spec A. To start this process, focus on specification A and choose Create new interpretation from. From the Create/Select

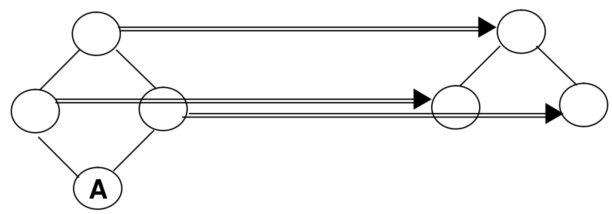

Figure 3-8 Parallel interpretations

Interpretation menu, choose the method via colimit. When you combine the interpretations via colimit, the system automatically creates the colimit specification B to be the target of the new interpretation, as shown in Figure 3-9.

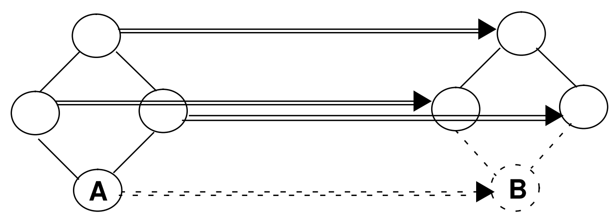

Figure 3-9 A new interpretation composes parallel interpretations

To create the new interpretation the system displays the defining diagram of A in the Refinement Editor. You must select the interpretations from each node. As you select interpretations, their targets become attached to the nodes of the defining diagram for B.

When you have specified all of the component interpretations, the Refinement Editor lets you select or create the interpretation morphisms between them, as shown in Figure 3-10. Notice how the structure of the composed interpretations echoes the structure of the composed specifications.

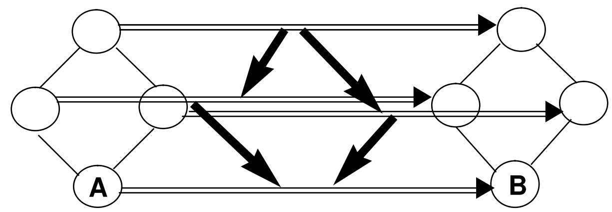

Figure 3-10 Composition of interpretations with interpretation morphisms

Finally, you must install the new interpretation in SPECWARE. Middle-click the background of the diagram and choose Finish Edit from the menu. When this process is finished (it can take some time), a menu appears that allows you to rename or view the new interpretation.

You can do the same thing with a more complex diagram, as shown in the following figures. Once again, the structure of the composed interpretations echoes the structure of the composed specifications.

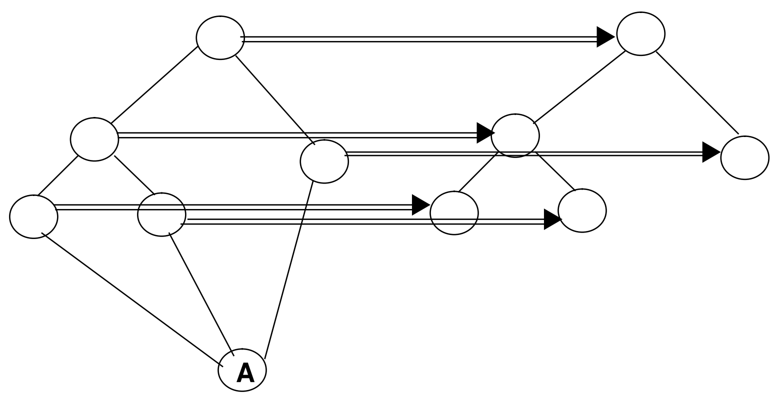

Figure 3-11 Parallel interpretations

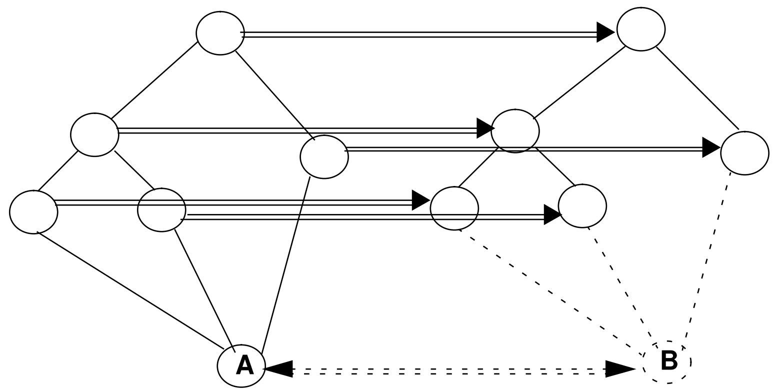

Figure 3-12 A new interpretation composes parallel interpretations

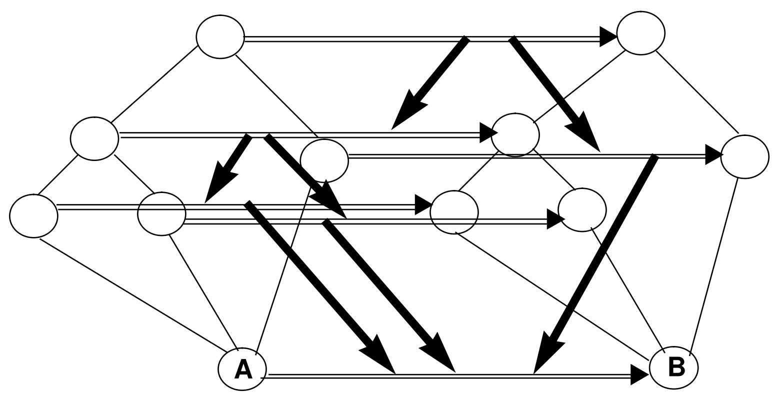

Figure 3-13 Composition of interpretations with interpretation morphisms

Refining Both Vertically and Horizontally

To make your refinement both adequately complex to specify the problem domain and adequately deep to generate code, you must use both parallel and sequential composition. However, the order in which you use the techniques is not important.

You can create a complex specification using parallel refinement, then combine these composed interpretations sequentially, as shown in Figure 3-14. In this figure, the fine arrows indicate an existing structure of specifications and interpretations, the bolder arrows indicate the first parallel refinements, and the boldest arrow indicates the final sequential refinement, that combines the earlier interpretations.

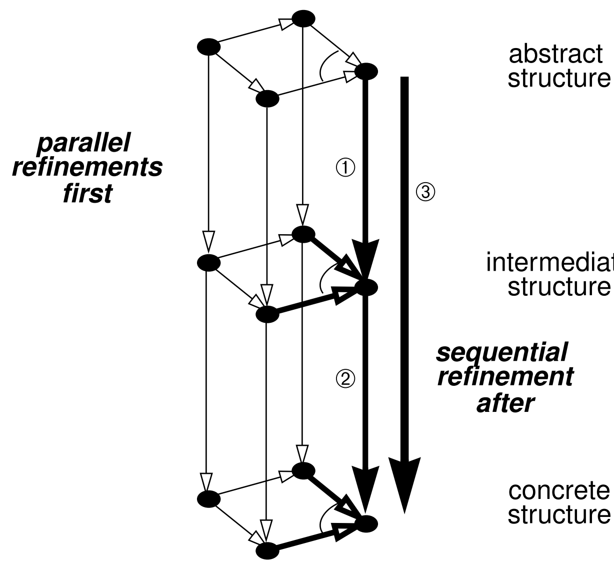

Figure 3-14   Parallel followed by sequential refinement

Alternatively, you can refine each part of the specification vertically, then combine these composed interpretations horizontally, as shown in Figure 3-15. In this figure, the fine arrows indicate existing structures, the bolder arrows indicate the first sequential refinements, and the boldest arrow indicates the final parallel refinement, that combines the vertically combined interpretations.

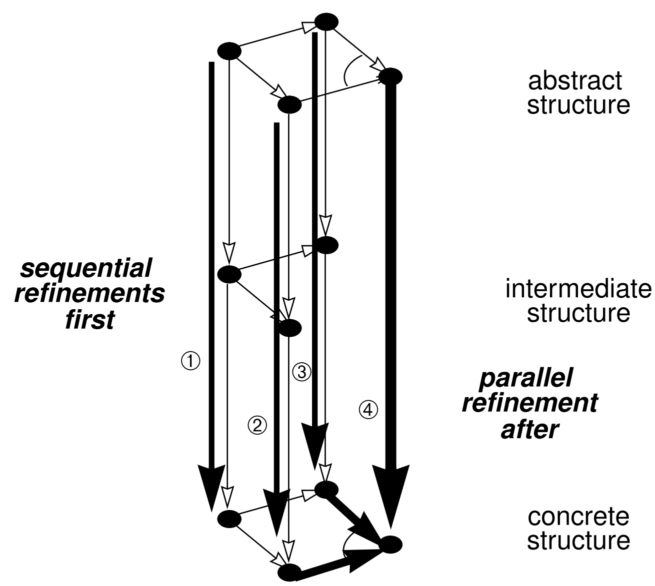

Figure 3-15 Sequential followed by parallel refinement

The interpretation that results from the final refinement is the same, regardless of how you construct it.

Composing Interpretation Schemes

Although each interpretation scheme need not completely define all the operations of its source, you can combine schemes in such a way that the result is a strict interpretation, whose target morphism is a definitional extension.

To compose schemes into an interpretation:

- Each operation must be completely defined in one scheme, although that scheme need not provide definitions for all operations.
- All of the operations must be defined in at least one of the schemes being combined.

You can use interpretation schemes to work with abstract sorts and operations, which you define in a different branch of development. You combine the interpretations and schemes from the branches to form a strict interpretation that fully defines the abstract sorts and operations. The example application in the *Specware Tutorial* uses interpretation schemes in this way.

Interpretation schemes are also useful for mutually recursive definitions. For example, the following specifications introduce the concept of evenness and oddness in natural numbers.

```
spec EVEN is
  op even? : Nat -> Boolean
 end-spec

spec ODD is
  op odd? : Nat -> Boolean
 end-spec

spec EVEN-and-ODD is
  colimit of diagram
  nodes EVEN, ODD, EMPTY %% EMPTY needed later
  arcs EMPTY -> EVEN : { },
       EMPTY -> ODD : { }
  end-diagram
```

To refine the `EVEN-and-ODD` spec you need an interpretation to `EMPTY`. You can create a mediator to `EMPTY` for each of the component specs, as follows:

```
spec EVEN-as-EMPTY is
  import EVEN-and-ODD
  definition of even? is
  axiom n=0 => even?(n)
  axiom gt(n, 0) => even?(n) <=> odd?(minus(n, 1))
  end-definition
 end-spec

spec ODD-as-EMPTY is
  import EVEN-and-ODD
  definition of odd? is
  axiom n=0 => ~odd?(n)
  axiom gt(n, 0) => odd?(n) <=> even?(minus(n, 1))
  end-definition
 end-spec
```

Notice that the definitions are mutually recursive: the axioms that define the operation `even?` make use of the operation `odd?`, which is not defined in the spec, and vice versa. Together, the axioms completely define the even-and-odd concept, but neither is complete in itself. You cannot use either `EVEN-as-EMPTY` or `ODD-as-EMPTY` as a mediator to form a strict interpretation of `EVEN-and-ODD`. You can, however, make interpretation schemes:

```
ip-scheme EVEN-to-EMPTY : EVEN => EMPTY is
  mediator EVEN-as-EMPTY
  dom-to-med { }
  cod-to-med { }

ip-scheme ODD-to-EMPTY : ODD => EMPTY is
  mediator ODD-as-EMPTY
  dom-to-med { }
  cod-to-med { }
```

You can combine the schemes to provide a complete definition for both operations. In order to combine them, you need a third scheme that shows which concepts they share. The mediators share the `even?` and `odd?` operations; you need to explicitly state this sharing, or you would get two copies of each operation. Since no concepts are shared in the sources or targets, you can simply use `EMPTY` as the source and target of the scheme.

```
ip-scheme EMPTY-to-EMPTY-via-EVEN-and-ODD
  : EMPTY => EMPTY is
  mediator EVEN-and-ODD
  dom-to-med { }
  cod-to-med { }
```

You now have all the elements you need to combine the interpretation schemes. Use the Refinement Editor to create a new interpretation from `EVEN-and-ODD`, choosing the method via colimit. The schemes are combined to form a strict interpretation, `EVEN-and-ODD-to-EMPTY`, that fully defines the even-and-odd concept in terms of natural numbers.

Generating Code

SPECWARE includes specification languages that are subsets of Lisp and C++. These languages are mostly identical to Slang; specifically, they are functional subsets. The system uses them to generate Lisp or

C++ versions of Slang specifications, by way of Slang-to-Lisp and Slang-to-C++ interpretations. You can build your own Slang-to-Lisp and Slang-to-C++ interpretations, and the system also contains builtin Slang-to-Lisp and Slang-to-C++ interpretations from the library specifications, including `EMPTY`.

In order to generate code, you refine your specification to `EMPTY` (using sorts and operations that are built in, such as `Boolean` and `String`) or to a combination of implementable library specifications (such as `INTEGER` or `SET`). You then combine your interpretation sequentially with a Slang-to-*language* interpretation, generating a new Slang-to-*language* interpretation. SPECWARE constructs the mediator of the final interpretation by translating the mediator of the Slang-to-Slang interpretation into the target language.

The interpretations to code from `EMPTY` are named `EMPTY-to-SLANG-BASE`. The interpretations to code from other library specifications are named in the form *`SPECNAME`*`-to-`*`SPECNAME`*; for example, `INTEGER-to-INTEGER`. The two language versions reside in different namespaces. Only the appropriate interpretations are offered on menus.

Generating Code by Instantiation

A special technique, *instantiation*, is provided to take parameterized specifications to code. (`LIST` and `ARRAY` are the only parameterized specifications currently available.)

For example, suppose your specification requires lists that contain real numbers. Figure 3-16 shows a defining diagram for a specification (`LIST-of-REALS`) that defines such a list.

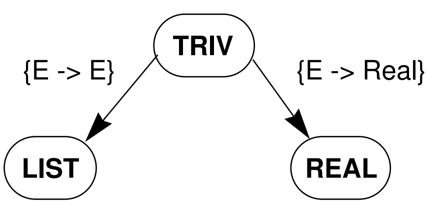

Figure 3-16 Defining diagram for `LIST-of-REALS`

When you refine an ordinary colimit to code, you must choose interpretations from each of the component specifications to code. This case is similar, except that you can interpret the collection itself completely independently of the interpretation of the contained sort.

When you create a Slang-to-Lisp or Slang-to-C++ interpretation for such a specification, the menu offers the method via instantiation. If this method is offered, you should always use it.

Understanding Generated Code

When you refine a specification to code, you construct a Slang-to-*language* interpretation, as shown in Figure 3-17. The mediator of this interpretation contains the generated code; the Refinement Editor allows you to save the mediator as a code file.

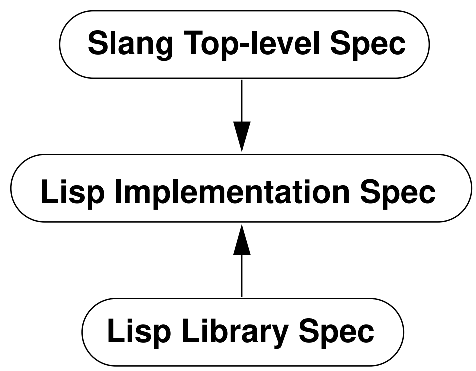

Figure 3-17 An interpretation to LIsp code

The final interpretation satisfies these properties:

- All of the operations in the top-level source specification are mapped to target-language functions in the mediator. If all morphisms are verified, all axioms and theorems in the top-level specification are true for the target language implementation.
- You construct the final interpretation by combining a set of intermediate interpretations, whose mediators contain definitions for operations. All of these definitions are translated to code in the mediator.

- The source specifications of intermediate interpretations are used only to establish the correctness of the morphism from the top-level Slang specification to the generated code in the mediator. Any axioms or theorems that appear in them are not translated directly to code.

The generated code file is source code for the operations that your top-level specification defines. You can embed the source code into an application in the chosen language that uses the operations.

Inspecting Generated Code

You should inspect the generated code file in an emacs buffer to see how your Slang operations have been translated. Because Slang differs from the target languages is some significant ways, the code is not always exactly what you would expect.

In C++, signatures are retained as function prototypes using appropriate data types. In Lisp, since there are no types, signature information is lost. You are responsible for passing appropriate arguments to functions; the functions do not check the types of their arguments.

The names of the functions in the generated file do not always match the operation names in the source specification. A section at the bottom of the generated code file shows any operations that have been renamed, with a map from the source operation name to the generated function name.

Because Slang uses a calling convention that is different from that of both Lisp and C++, the number of arguments taken by a function in code does not always match the number of arguments you specified for the equivalent operation in Slang. When a Slang operation takes a product sort as its sole argument, the corresponding function in the generated code takes each of the component sorts of that product sort as multiple arguments.

This occurs because Slang uses a calling convention where all functions take a single argument. What appear to be multiple arguments of operations are converted to product sorts internally, and passed as a single argument. Therefore, if you define sort `X` as the product `(A, B)`, in Slang the operations `f:X->Y` and `f:A,B->Y` are identical. When the Slang operation is converted to code, both signatures are translated as `A,B->Y`, resulting in a function with two arguments, not one. In Lisp, for example, the translation of `f:X->Y` is `(f A B)`. Since Lisp uses multiple arguments, this is not identical to `(f X)`.

For example, in the tutorial sample project, you define a sort `Account`, with a sort axiom that declares it to be the product sort `(String, Amount)`. The two parts represent the account's owner and balance. Accessor and constructor operations are defined as follows:

```
op owner : Account -> String
  definition of owner is
    axiom owner(a)= project(1)(a)
  end-definition

 op balance : Account -> Amount
  definition of balance is
   axiom balance(a)= project(2)(a)
 end-definition

 op make-account : String, Amount -> Account
 definition of make-account is
   axiom make-account(str, amt) = (str,amt)
 end-definition
```

When this definition is translated to Lisp, the corresponding accessor functions are defined as follows:

```
(defun balance (var::a_1 var::a_2) var::a_2)
(defun owner (var::a_1 var::a_2) var::a_1)
```

Notice that these functions take two arguments each, not one. The arguments are, in fact, expected to be the parts of the `Account` sort, the very things you were intending to access.

A test of these functions has these results:

```
(setf a (make-account "Mary" 100))
=> ("Mary" 100)
(owner a)
=> error: wrong number of arguments
(owner "Mary" 100)
=> "Mary"
```

Obviously, this is not very useful as an accessor. To convert this function back to the form intended in the Slang definition, you would have to edit the code file, adding a wrapper to convert the multiple arguments back to a single argument, as follows:

```
(defun new-owner (a)
  (owner (project 1 a)
         (project 2 a)))
```

Saving, Loading, and Testing Generated Code

When you have created the final refinement, the Refinement Editor allows you to write the generated code to a file using the Write code to file menu option. The code file is written to the current working directory, and named for the source specification of the final interpretation. For example, if the top-level specification is named `DOLLAR-ACCOUNT`, and the code is in Lisp, the file is named `dollar-account.lisp`. To specify a different name or location, choose the menu item Write code to file named.

Testing Lisp Code

SPECWARE itself runs in the Lisp environment, so you can load a generated `.lisp` file directly at the Lisp prompt, using the Lisp `load` function. For example:

```
(load "/home/specware-user/dollar-account.lisp")
```

Generated code loads into the `CODE` package. To change to this package, type the following command at the Lisp prompt:

```
:pa CODE
```

Functions defined in the file are immediately available for testing.

Testing C++ Code

To test C++ code, edit the generated source file to add a `main()` function that exercises all functions. Compile the file using the C++ compiler, then load and run the object code in a UNIX shell.

Currently, you must use the Gnu C++ compiler, because generated code uses multiple precision integers (bignums) and expression blocks (let).

CAUTION Small compilation errors can result in large numbers of error messages. Should this occur, check the generated code file for type errors. The problem is usually much simpler than the error messages would lead you to believe.

Chapter 4

The Specware User Interface

You use two basic methods to build SPECWARE applications:

- You write source code in the Slang language
- You display and manipulate SPECWARE objects graphically

You must use both methods, since some objects can only be specified in text, and some actions can only be initiated graphically. The SPECWARE user interface integrates these two worlds, so that you can move back and forth between them as necessary.

Starting and Quitting Specware

SPECWARE runs on a UNIX system and requires the xemacs version of the emacs editor.

How to: Start the Specware system

SPECWARE runs inside the REFINE system, which runs inside emacs. To start the system, you must first start emacs and REFINE, as follows:

1. Open an emacs window by typing `xemacs&` in a shell window. (Note that REFINE and SPECWARE run only in xemacs, not in GNU emacs.)
2. In emacs, type:

   `M-x run-a-refine`
   currentworld

   (for example, `specware-2-0-2-dist-970521`)
3. The REFINE system starts up, which takes a few minutes. You must answer a question about whether to load the initialization file. If

you have not set system preferences in a `home-init.lisp` file in your home directory, answer "`y`" to load the default initialization file. (See "Setting Preferences in an Initialization File" on page A-5.)

4. When the REFINE system is ready, a prompt appears. At the prompt, type:

   ```
   (start-specware)
   ```

Progress messages, warnings, error messages, and results of operations are displayed in the `*REFINE*` buffer of emacs. The Lisp prompt in the `*REFINE*` buffer is the command line for SPECWARE.

To exit the SPECWARE system, use the command:

```
M-x kill-refine-process
```

This kills the Lisp process, but does not kill emacs or any buffers.

The Specware Windows

The SPECWARE main window appears in the default configuration, with the `*REFINE*` emacs buffer on the right, a graphical window on the left, a menu bar across the top, and a status bar at the bottom. New graphical windows come up as you choose graphical operations. You can reconfigure the windows that appear by default and their placement by choosing from the Configuration menu. (See "Changing the System Window Layout" on page A-1.)

The `*REFINE*` buffer in emacs is the system console window, which shows error, warning, and status messages from the system. You can use emacs buffers to enter text in the Slang language, in order to define specifications, morphisms, and interpretations. You can also view source files for the library specs and morphisms in emacs buffers using Find File, although there are several other ways to view them.

The graphical windows show you graphical representations of objects, and allow you to perform some operations graphically. Graphical windows can be Diagram Editors, Morphism Editors, or Refinement Editors. Each type of editor displays different objects and has different context menus.

Whenever you choose to display an object graphically, you are given a choice of using an existing window or creating a new window. You can set a system preference to default one way or the other. (See "Setting Display Preferences" on page A-3.) When you use an existing window to display a diagram, the new diagram is displayed in that window together with the prevously displayed diagram, rather than replacing it.

Some operations can be performed through either text or graphics, but some operations are only text-based, and some are only graphics-based. Text-only operations include creating and viewing basic specifications. Graphics-only operations include composing interpretations and morphisms. Other operations, including the creation and manipulation of morphisms, interpretations, diagrams, and interpretation morphisms, can be accomplished by either method.

Invoking System Commands

When you specify operations in the text of a Slang file, you compile them into the system using the parsing commands that are available in the REFINE mode in emacs. For a list of these commands, see Table 5-1 in Chapter 5, "Building Specifications".

When you perform operations interactively through the graphical interface, commands are available through menus. The Focus command on the menu bar allows you to access different kinds of objects, through a series of menus that eventually bring up graphical windows. In the graphical windows, you access operations on their contents through pop-up menus.

Using the Menu Bar

Use the Focus command on the menu bar to begin working with any kind of object. When you have chosen the kind of object you wish to work with, further menus allow you to choose an operation or action appropriate to that object type, and bring up the appropriate graphical window.

Other menu bar commands allow you to set system configuration preferences (see Appendix A, "Customizing Specware"), and access online help (see "Invoking System Help" on page 4-4).

Using Pop-up Menus

Pop-up menus are associated with both right and middle mouse clicks in graphical windows. The commands on these menus vary with the context, and apply to the contents or allowable contents of the window.

- A right click in a diagram pops up a menu that allows you to manipulate the picture. Right click on a node or arc to select it or move it. Right click on the background to move the entire diagram.
- A middle click pops up a menu that allows you to manipulate the objects represented by the picture. You will use the middle-click menu most frequently to access operations and bring up new windows. When you middle click a node or arc, the menu items pertain to that node or arc. When you middle click the background, the menu items pertain to the entire diagram.

Invoking System Help

Click SpecWeb in the menu bar to invoke the web-based help system. A web browser is started (or an existing browser is reused) to display the home page of the SPECWARE help system. This system includes an online copy of the Slang language manual, a tutorial introduction to SPECWARE, release notes, and access to the source code for the object library.

Accessing the Specware Library

SPECWARE includes a library of predefined specifications, morphisms, and interpretations in the domain of computational mathematics. You will frequently use these library objects in building up your own specifications. For example, if you need to use any kind of numerical information, you will probably include the existing specs for integers or real numbers, rather than defining your own.

In addition to including library objects directly through `import` and `translate`, you might want to study the source code for library objects and copy code fragments into your own objects.

There are several ways to view library objects:

- View library objects from the Focus menu

All of the existing library objects are listed when you choose an object type, such as Spec, from the Focus menu. You can choose a library object from the list and choose Show Text (long) from the resulting menu. The library object is displayed in a read-only emacs buffer.

- View library source code through the SpecWeb help system

Click SpecWeb in the menu bar. A web browser appears showing the home page of the SpecWeb help system. The library source files are organized into Interpretations, Code (Generation), and Slang Specs. When you have navigated to the specifications of interest, the source file containing those specs is displayed in the web browser. You can use the browser's Find in Page facilities to search the source file.

- View source code directly

When you have become familiar with the names of the library source files, you can find them in the `library` directory of the SPECWARE distribution and bring them up in emacs.

Chapter 5

Building Specifications

This chapter describes how to create the elements of a SPECWARE application by writing source code in the Slang language, using an emacs editor. In the Slang file, you define the original abstract specifications and morphisms, as well as the mediators for their interpretations. You can begin building the refinement in Slang text by defining the initial interpretations between abstract and implementation specifications.

You can view the structures you have built and perform some operations through the graphical windows of the SPECWARE interface. To build and extend the refinement, you work primarily through the user interface, displaying interpretation diagrams graphically in the Refinement Editor, as described in Chapter 6, “Building Refinements”.

Creating a Slang Source File

Slang source files are identified by the `.re` extension. To create a new Slang file for a SPECWARE application, create a new file in emacs whose name has the extension `.re`. For example:

```
C-x C-f myfile.re
```

Edit the first lines of the file to specify the SPECWARE default package and grammar. (See “Specware Grammars” on page 5-3.) Begin the file with the lines:

```
!! in-package("SPEC")
!! in-grammar('ispec-grammar)
```

Parsing Slang Definitions

When you add an element to a source file, you must parse it to install it into the SPECWARE system. You do this using emacs REFINE-mode commands. The emacs REFINE mode is entered automatically when you load a `.re` file, or create a new file whose name has the extension `.re`. You can also enter REFINE mode explicitly using the command `M-x refine-mode`.

Table 5-1 shows the REFINE commands that are used in SPECWARE.

Table 5-1 Refine commands used in Specware

Command	Shortcut	Action
`M-x refine-parse-buffer`	`M-x r p b`	Parse entire buffer
`M-x refine-parse-region`	`C-c M-p` or `C-c CM-p`	Parse marked region
`M-x refine-parse-definition`	`C-c p` or `C-c C-p`	Parse definition containing cursor
`M-x refine-parse-file`		Parse `.re` file
`M-x refine-load-file`		Load `.re` file
`M-x refine-mode`		Start REFINE mode
`M-x kill-refine-process`		Exit SPECWARE, REFINE, and Lisp

- To parse the entire buffer, use the command
  `M-x refine-parse-buffer`
  (or the shortcut, `M-x r p b`)
- To parse a region, mark the region, then use the command
  `M-x refine-parse-region`
  (or one of the shortcuts, `C-c M-p` *or* `C-c C-M-p`)
- To parse a single definition, place the cursor inside the definition and use the command
  `M-x refine-parse-definition`
  (or one of the shortcuts, `C-c p` *or* `C-c C-p`)
- To parse a file other than the one shown in the current buffer, specify the file name:
  `M-x refine-parse-file` *`file`*

When you parse a definition, check the `*REFINE*` buffer for warnings or error messages. If necessary, fix the syntax and parse again, until the definition parses without errors.

Parsing does not save the buffer. You must save it explicitly using emacs commands such as `C-x C-s`.

Changes in the source file have no effect on SPECWARE until the changed specification is re-parsed. If a specification has the same name as an existing specification, the existing definition is overwritten when the new definition is parsed. A warning message to this effect appears at the Lisp prompt in the emacs `*REFINE*` buffer.

Specware Grammars

There are two SPECWARE grammars available. You select which one to use by specifying the grammar at the top of your Slang source file using `in-grammar`.

- The newer `ispec-grammar` uses infix notation.
- The older `spec-grammar` uses LISP-like prefix notation.

The SPECWARE library uses the prefix grammar. The tutorial example file and the examples in this manual use the infix grammar.

You can translate a spec from prefix to infix grammar at the Lisp prompt in the `*REFINE*` buffer of emacs, using the following function:

```
(print-convert-spec 'SPEC::specname)
```

By default, printing in the `*REFINE*` buffer uses the new infix grammar. To print in prefix grammar, enter the following command at the Lisp prompt or in your initialization file:

```
(setq sp::*infix-printing?* nil).
```

Specifications

The first stage of building an application with SPECWARE is to specify your domain theory. The specification object, or spec, is the basic unit with which you work. Higher-level (more complex) specs are constructed from lower-level (simpler) specs. When you create a spec, you must parse it to install it into the SPECWARE system. You can replace specs at any time in order to make additions or corrections. Whenever you alter an existing spec, you must re-parse both that spec and all specs, morphisms, interpretations, and so on, that are constructed from it.

Creating Specifications

You create a basic specification by explicitly writing out its elements in Slang using emacs, then parsing the definition. Specifications begin with the keyword `spec` and end with the keyword `end-spec`.

NOTE You must leave a blank space before the keyword `end-spec` in order for the parser to recognize it. ❖

Specifications contain declarations for sorts and operations, and can also include sort-axioms that define the structure of sorts, and axioms, theorems and definitions that define the structure of operations. For example:

```
spec ACCOUNT-with-DEPOSIT is
  import SIMPLE-ACCOUNT
  op deposit : Account, Amount -> Account
  axiom
     fa (a : Account)
     balance(deposit(a, x)) = plus(balance(a),x)
 end-spec
```

How to: Create a Specification in Text

1. Compose the specification text in the Slang language, using emacs.
2. Parse the definition using one of the REFINE parsing commands.
3. Check the `*REFINE*` buffer for warnings or error messages.
4. If necessary, fix the syntax and parse again, until the definition parses without errors.
5. Save the buffer.

Once you have parsed the specification into the system, it appears on lists of specifications in menus where applicable.

Viewing a Specification

When you have constructed a new spec using the construction operations, it is useful to view the result in order to see whether the construction was applied in the way you intended.

You can view a specification's definition either textually or graphically, depending on how the spec was constructed. There are *short* and *long* textual views of a spec, and *diagram* and *cocone* graphical views. The graphical views are only available for a spec constructed as a colimit of a diagram.

Both the textual and graphical views are always generated from the internal representation of the specification. Comments and layout information in the source file and graphical layout changes that you make to a diagram are not shown.

To view a specification, choose Spec from the Focus menu, and select the spec you wish to view. You can also invoke the Spec menu from a graphical window where the spec is attached to a diagram node. Middle click the node and choose Visit attached spec.

In the Spec menu, choose one of the Show commands:

> Show Text (long): Displays the basic form of the specification in a pop-up, read-only emacs buffer. The long form explicitly shows all of the spec's sorts, operations, axioms, and so on. This viewing option is always available.
>
> Show Text: Displays the defining expression for the specification in a pop-up, read-only emacs buffer. This short form shows any expression involving the specification-building operations `import`, `translate`, or `colimit`. The short textual view of a colimit spec is only available if the spec was constructed in a Slang source file. The text view of a colimit constructed using the graphical interface is always in long form.
>
> Show Defining Diagram (base): Displays the defining diagram of a spec constructed using the `colimit` operation. Note that this diagram does not contain the colimit spec itself.
>
> Show Defining Diagram (cocone): Displays the defining diagram plus the colimit spec and the morphisms from the component specs into the colimit spec.

Specification Diagrams

A specification diagram, also known as a spec/morphism diagram, defines a set of specifications related by morphisms. (See "Morphisms" on page 5-9.) Specs are attached to nodes and morphisms between specs are attached to arcs between the nodes. The Diagram Editor window displays diagrams graphically, and allows you to create new diagrams and to perform operations on existing diagrams.

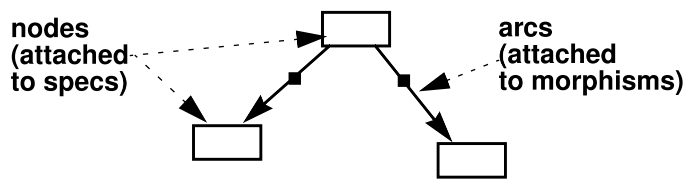

Figure 5-1 The structure of a spec/morphism diagram

Creating a New Spec/Morphism Diagram

You can define new spec/morphism diagrams in Slang text, or by using SPECWARE's graphical Diagram Editor.

Begin the diagram specification with the `diagram` keyword, and end it with the `end-diagram` keyword. Use the `nodes` keyword to specify specs to be attached to nodes, and the `arcs` keyword to specify morphisms to be attached to arcs between specified nodes.

For example, the following code creates a defining diagram for a shared union, `ACCOUNT`. Both `ACCOUNT-with-DEPOSIT` and `ACCOUNT-with-WITHDRAW` are specializations of `SIMPLE-ACCOUNT`; both import `SIMPLE-ACCOUNT`.

```
diagram ACCOUNT-DIAGRAM is
 nodes ACCOUNT-with-DEPOSIT,
       ACCOUNT-with-WITHDRAW,
       SIMPLE-ACCOUNT
 arcs
   SIMPLE-ACCOUNT -> ACCOUNT-with-WITHDRAW :
       import-morphism,
   SIMPLE-ACCOUNT -> ACCOUNT-with-DEPOSIT :
       import-morphism
end-diagram
```

You can also define a diagram as part of a specification. For example, the following code defines the preceding diagram along with the `colimit` operation that combines the nodes into a shared union. Notice that this version of the diagram is not named, and that the `end-spec` keyword is not used after `colimit`.

```
spec ACCOUNT is
 colimit of diagram
 nodes ACCOUNT-with-DEPOSIT,
       ACCOUNT-with-WITHDRAW,
       SIMPLE-ACCOUNT
 arcs
  SIMPLE-ACCOUNT -> ACCOUNT-with-WITHDRAW :
      import-morphism,
  SIMPLE-ACCOUNT -> ACCOUNT-with-DEPOSIT :
      import-morphism
 end-diagram
```

How to: Create a Diagram in Text

1. Enter the diagram syntax into a Slang text file in an emacs buffer.
2. Parse the definition using one of the REFINE commands.

Once you have parsed the diagram into the system, it appears on lists of diagrams in menus where applicable.

How to: Display a Diagram in a Graphical Window:

1. From the Focus menu, choose Spec Diagram.
2. From the list of diagrams, choose the diagram by name.
3. Select an existing or new window in which to display the diagram picture.

The diagram is displayed graphically, as shown in Figure 5-2.

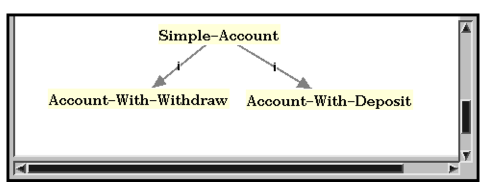

Figure 5-2 A specification diagram

Middle click the background of the diagram to see a menu of operations you can perform on the diagram, such as Rename. Middle click any of the specs or morphisms to visit it separately.

How to: Create a Diagram Using the Diagram Editor

1. From the Focus menu, choose Spec Diagram.
2. From the list of diagrams, select Make New (the next-to-last menu item).
3. Choose New Window to display the diagram picture. A Diagram Editor appears with a blank display.
4. Middle click the picture background to access the pop-up diagram menu.
5. Choose Add Node to create new nodes. You can give a new node a place-holder label, such as **A**, or allow it to remain anonymous.
6. Choose a spec to attach to the node. The place-holder label is automatically replaced with the name of the attached spec.
7. Create arcs in either of two ways:

   a) From the middle-click diagram menu, choose Add Arc. The cursor changes shape to indicate that you must choose source and target nodes for the arc.

   b) Click on the source node for the arc.

   c) Click on the target node for the arc.

   —or—

   a) Middle click the chosen source node.

   b) From the middle-click node menu, choose Add arc from. The cursor changes shape to indicate that you must choose a target node.

   c) Click on the desired target node.

   The new arc appears as a dotted line, to indicate that it is not yet attached to a morphism.

8. A list of applicable morphisms appears. Choose from the list to attach the new arc to a morphism, or choose Make New to create a new morphism. If you choose to create a new morphism:

a) A type-in window appears in which you can name the new morphism. Enter the name and click Do It, or click Abort to create a morphism with a generated name.

b) The Morphism Editor appears. Edit the morphism if necessary, then middle click the background and choose Finish Edit. (See "Creating New Morphisms" on page 5-10.)

The arc becomes a solid line to indicate that it is attached to a morphism.

9. When the diagram is complete, choose Finish Edit from the middle-click diagram menu.

10. To name the new diagram, middle click the background of the diagram picture and choose Rename from the diagram menu. Enter the new name in the type-in window and click Do It.

Alternatively, you can first create all the nodes and arcs in a diagram (its *shape*), and then attach specs to the nodes and morphisms to the arcs. To do this, choose Abort instead of selecting a spec or morphism to attach. However, before you can create an arc, both its source and target node must be present. Similarly, before you can attach a morphism to an arc, both the source and target nodes must have attached specs.

CAUTION If you attach a different spec to a node (Replace attached spec) you must update all the morphisms attached to arcs adjacent to the node. SPECWARE currently has no mechanism for tracking and supporting this activity. Diagram consistency is only checked when you choose the Finish Edit menu item. ❖

Morphisms

Morphisms map one spec to another by specifying exactly which parts correspond. SPECWARE can infer some classes of morphisms, identified by keywords. For example, the `identity-morphism` maps a spec to itself, and the `import-morphism` maps a spec A to spec B that imports A. SPECWARE can also automatically complete a partially specified morphism by mapping a sort or operation in the source corresponds to a sort or operation with the same name in the target.

Creating New Morphisms

You can create a morphism directly by defining it in Slang text. You can also create a morphism interactively by choosing from menus, and edit it using the graphical Morphism Editor. Finally, you can create new morphisms by composing previously existing morphisms. This operation can only be done graphically.

To create a morphism explicitly, use the `morphism` keyword. For example:

```
morphism AMT-to-DOL-morphism
   : AMOUNT -> DOLLAR-AMOUNT
   =  { Amount ->  Dollar-Amount,
        (plus: Amount,Amount -> Amount)   ->  d-plus,
        (minus: Amount, Amount -> Amount)  ->  d-minus,
       sign   ->  sign
      }
```

You can also describe morphisms inline in the course of specifying a diagram (defining arcs), interpretation (defining the maps to the mediator from the source and target), or translation (defining terminology correspondence). For example:

```
interpretation ACCOUNT-to-ACCOUNT-IMPL
   : ACCOUNT => ACCOUNT-IMPL is
   mediator ACCOUNT-IMPL
   dom-to-med { }
   cod-to-med identity-morphism
```

This example describes two morphisms in the last two lines. SPECWARE uses the description to create the morphisms automatically.

- The empty braces, `{ }`, identify a default morphism. No mappings are given between the braces, so SPECWARE creates a morphism by mapping elements with the same name to each other.
- The keyword `identity-morphism` identifies a standard type of morphism that SPECWARE can infer, in this case the identity.

For every sort or operation in the source whose corresponding sort or operation in the target has a different name, you must explicitly state the correspondence between the braces, using the arrow (->) syntax. For example:

```
{nonzero? -> nonzero-int?,
 pos? -> pos-int?}
```

SPECWARE automatically completes the morphism by mapping elements with the same name to each other.

How to: Create a Morphism in Text

1. Enter the morphism syntax into a Slang text file using emacs.
2. Parse the definition containing the morphism using one of the REFINE commands.

Once you have parsed the morphism into the system, it appears on lists of morphisms in menus where applicable.

How to: Create a Morphism Using the Morphism Editor

1. Start by choosing the domain, or source spec. You can do this in either of two ways:

   a) From the Focus menu, choose Spec.

   b) From the list of specs, choose the spec which will be the source of the new morphism.

   c) From the Spec menu for the chosen spec, choose Make New Spec Morphism from.

   —or—

   a) From the Focus menu, choose Spec Morphism.

   b) From the list of currently existing morphisms, select the next-to-last item, Make new.

   c) From the list of currently existing specs, select the domain spec for the new morphism.

2. From the Select/Construct menu for spec morphisms select the next-to-last item, via editor.
3. Select the codomain, or target spec for the new morphism from the list of currently existing specs.

The Morphism Editor appears, as shown in Figure 5-3. The editor displays all of the elements of the source (domain) spec on the left, and the elements of the target (codomain) spec on the right. The editor automatically creates correspondences where it can. The correspondences, called rules, are shown by arrows between corresponding elements.

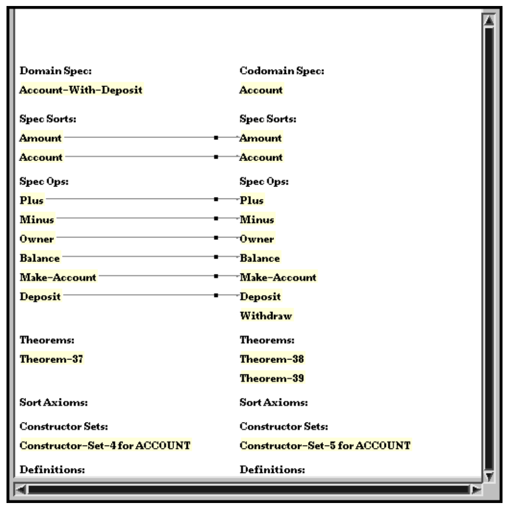

Figure 5-3 The Morphism Editor

Middle click any element or rule to see a menu of actions that pertain to it.

4. To delete a rule, middle click the center point of the rule and choose Delete rule.
5. To create a new rule:
   a) Middle click the source element and choose Select a target.
   b) Select from the menu of suitable target elements.
6. To map a source element to a different target:
   a) Middle click the center point of the current rule and choose Remove codomain of pair.
   b) Middle click the source element and choose Select a target.
   c) Select from the menu of suitable target elements.

7. Middle click the background for a menu of actions that pertain to the entire morphism. For example, a morphism is assigned a generated name when it is created; you can assign a new name by choosing Rename from the background middle-click menu.
8. To verify the morphism, middle click the background and choose Verify. Inspect the results of the verification in the emacs `*REFINE*` buffer.
9. To install the morphism and exit the editor, middle click the background and choose Finish Edit.

Combining Specifications

To compose specifications in text, use the specification construction operations `translate`, `import`, and `colimit`. The `translate` and `import` operations construct new specs by including and renaming earlier specs. Many of the specs you create begin by importing other specs, either from the SPECWARE spec library, or that you have already created.

The spec construction operations can be combined. For example, in a single spec declaration you can take the `colimit` of a diagram, rename the elements using `translate`, and then `import` the translated spec into the spec being constructed.

All three spec construction operations can be used in Slang text. Only shared union composition is accessible through the graphical user interface.

How to: Compose Specifications in Text

1. Enter the spec composition syntax into a Slang text file in an emacs buffer.
2. Parse the definition using one of the REFINE commands.

Once you have parsed the definition into the system, the new composed spec appears on lists of specs in menus, where applicable.

Creating Shared Unions

The most powerful tool for combining specifications is the shared union operation, called `colimit`. In order to combine specifications into a shared union, they must first be part of a diagram, as shown in Figure 5-4.

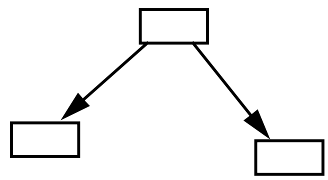

Figure 5-4 A defining diagram for a shared union

When you create a shared union, the system automatically creates morphisms to the new shared union spec from the specs in the defining diagram. The shared union spec is related to the defining diagram specs by cocone morphisms, as shown in Figure 5-5.

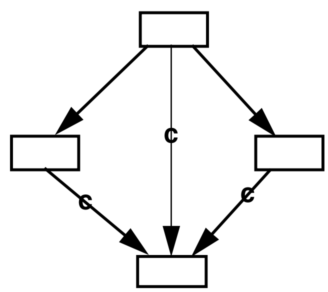

Figure 5-5 A cocone diagram for a shared union

You can create shared unions in Slang text, using the `colimit` keyword. You can also create or display the defining diagram in a graphical window and create the shared union interactively.

How to: Create a Shared Union in Text

1. Enter the `colimit` syntax into a Slang text file in an emacs buffer. For example:

```
spec ACCOUNT is
 colimit of ACCOUNT-DIAGRAM
```

NOTE Specs that are constructed in text as colimits do not use the `end-spec` keyword. ❖

2. Parse the definition using one of the REFINE commands.

Once you have parsed the definition into the system, the new colimit spec appears on lists of specs in menus, where applicable.

How to: Create a Shared Union Using the Graphics Editor

1. From the Focus menu, choose Spec Diagram.
2. From the list of diagrams, choose the diagram by name. (To construct a new diagram, see "Creating a New Spec/Morphism Diagram" on page 5-6.)
3. From the diagram menu, choose Edit as a Picture.
4. Select a window in which to display the diagram picture. The diagram is displayed graphically, as shown in Figure 5-6.

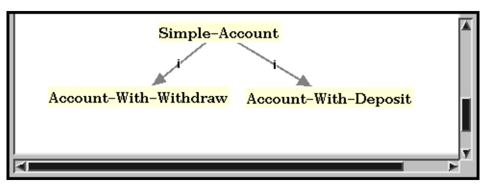

Figure 5-6 A specification diagram

5. Middle click the background and select Finish Edit from the Diagram menu.
6. Middle click the background again. Select Make Colimit from the Diagram menu.
7. A type-in window appears in which you can name the new colimit spec. Enter the name and click Do It.

8. The new colimit is created, but does not yet appear in the diagram. To see the diagram with the new colimit, choose Visit colimit from the Diagram menu, then choose Show Defining Diagram (cocone). A cocone diagram is displayed, as shown in Figure 5-7.

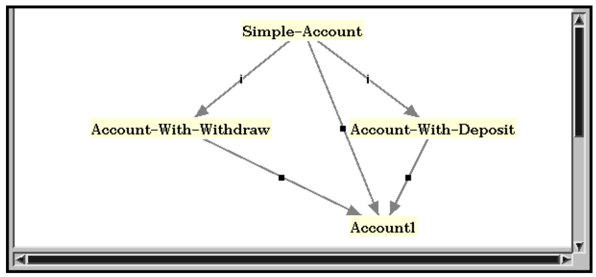

Figure 5-7 A cocone diagram for a shared union

Interpretations

An interpretation maps a domain, or source spec, to a codomain, or target spec, through a mediator which extends the target.

Use the `interpretation` keyword, and specify the domain (source spec), codomain (target spec), mediator spec, and morphisms from the domain and codomain to the mediator. For example:

```
interpretation ACCOUNT-to-EMPTY
  : ACCOUNT => EMPTY is
  mediator ACCOUNT-as-EMPTY
  dom-to-med { }
  cod-to-med { }
```

In this example, the mediator is constructed so that every sort and operation is named the same as in the domain. This is the default case for morphisms, as indicated by empty braces. SPECWARE infers the morphism using the default mapping. Since there is nothing in the EMPTY spec, the morphism from it to the mediator is also empty.

How to: Create an Interpretation in Text

1. Enter the interpretation syntax into a Slang text file in an emacs buffer.
2. Parse the definition using one of the REFINE commands.

Once you have parsed the interpretation into the system, it appears in lists of interpretations on menus where applicable.

How to: Display an Interpretation Graphically

1. Choose Interpretation from the Focus menu.
2. From the Select Interpretation menu, choose an existing interpretation.
3. From the Interpretation menu, choose Edit as a Picture.
4. Choose a new or existing window in which to display the diagram.

The selected interpretation is displayed in a graphical window, as shown in Figure 5-8.

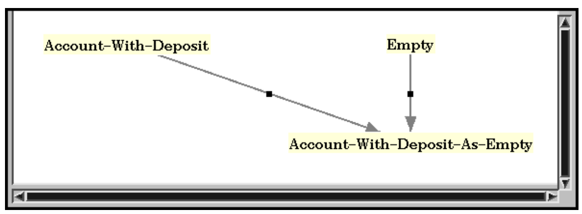

Figure 5-8 An interpretation

The diagram shows the source, mediator, and target specifications, and the morphisms between them. Middle click any element of the diagram to visit or verify that element. Middle click the background of the diagram to see a menu of actions (such as Rename) that you can perform on the interpretation.

Verifying Specs and Morphisms

You can choose to verify a specification or a morphism at any time.

- When you verify a spec, you are testing that the explicitly stated theorems (declared with the `theorem` keyword) follow from the axioms in that spec.
- When you verify a morphism, you are testing that the translations of all axioms in the source are provable theorems in the target.

When you create a morphism, the theorem prover can automatically verify it and print the results in the `*REFINE*` buffer of emacs. You can turn this behavior on or off by setting an environment preference. (See Appendix A, "Customizing Specware".)

When you create a new interpretation using the Refinement Editor, the theorem prover automatically verifies each of the morphisms that compose the interpretation. If a morphism cannot be verified, the interpretation is not created, and an error message box informs you of the failure.

You can access the KITP theorem prover directly through the Verify option on the Spec and Spec-Morphism menus.

How to: Verify a Specification

1. Choose Spec from the Focus menu.
2. Choose the spec you wish to verify.
3. From the Spec menu, choose Verify.
4. Inspect the results of the verification in the `*REFINE*` buffer of emacs.

How to: Verify a Morphism

If the morphism you wish to verify is named:

1. Choose Spec Morphism from the Focus menu.
2. Choose the morphism you wish to verify.
3. From the Spec Morphism menu, choose Verify.
4. Inspect the results of the verification in the `*REFINE*` buffer of emacs.

If the morphism is not named, you can start from the source spec of the morphism in order to find it.

1. Choose Spec from the Focus menu.
2. Choose the spec that is the source of the morphism you wish to verify.
3. From the Spec menu, choose Make new Spec Morphism from.

4. In the Select/Construct Spec Morphism menu, choose the morphism you wish to verify.
5. In the Spec-Morphism menu, choose Edit as a Picture.

   The morphism is displayed in a Morphism Editor.
6. Middle click the background of the Morphism Editor and choose Verify from the Spec-Morphism menu.
7. Inspect the results of the verification in the `*REFINE*` buffer of emacs.

Chapter 6

Building Refinements

You build refinements by composing morphisms and interpretations, using the graphical editors in the SPECWARE user interface. This is an iterative process; you compose intermediate interpretations to create increasingly concrete and complex interpretations.

Composing Morphisms

The sequential composition operation composes a sequence of morphisms, where the target of one morphism is the source of the next morphism in the sequence. You can either select a path through an existing diagram as the argument of the composition operation, or construct a new, linear diagram.

The sequential composition operation is accessible only through the graphical interface; there is currently no text syntax for this operation.

How to: Compose the Morphisms Along a Path

1. Display the diagram in a Diagram Editor, as described in "Specification Diagrams" on page 5-6.
2. Middle click the diagram picture background. From the diagram menu, choose Compose arcs from. A pop-up menu appears with the single command Done.
3. In the diagram picture, left click on the arcs of the path along which you want to compose the morphisms. The selected arcs turn red to indicate the selected path.
4. When you have finished selecting the path, click Done from the pop-up menu. The new composed morphism is created with a

generated name, and attached to a new arc that appears in the diagram.

How to: Create a Morphism Using Sequential Composition

1. Start by choosing the domain, or source spec. You can do this in either of two ways:

   a) From the Focus menu, choose Spec.

   b) From the list of specs, choose the spec that will be the source of the new morphism.

   c) From the Spec menu for the chosen spec, choose Make New Spec Morphism from.

   —or—

   a) From the Focus menu, choose Spec Morphism.

   b) From the list of currently existing morphisms, select the next-to-last item, Make new.

   c) From the list of currently existing specs, select the domain spec for the new morphism.

2. From the Select/Construct Spec Morphism menu, select via sequential composition.

3. If your system does not do so by default, select New Window.

   The Diagram Editor appears, showing the chosen source spec. A dotted gray arc indicates the morphism that needs to be determined. The Select/Construct Morphism menu also appears.

4. Use the Select/Construct Morphism menu to select the first morphism in the sequence.

   The diagram in the Diagram Editor updates to show the new morphism and its target spec. A pop-up menu appears, asking whether you wish to extend the refinement.

5. To extend the sequence of morphisms to be composed, click yes in the pop-up menu.

   A new dotted-line arc and node appear in the Diagram Editor, indicating the extension to be resolved, and the Select/Construct Spec Morphism menu appears again.

Each time you select or construct a new morphism in the sequence, it is added to the diagram and the Extend refinement? menu appears again. You can continue extending the morphism as long as necessary.

6. To finish the sequence of morphisms to be composed click no in the pop-up menu. SPECWARE creates a new morphism by composing the morphisms you have specified. The new morphism is attached to a new arc from the original source to the final target.
7. The Spec-Morphism menu for the new morphism appears. You can inspect the new composed morphism by choosing Show Text (long), or use the theorem prover to verify it by choosing Verify. (The results of the verification appear at the Lisp prompt in the `*REFINE*` buffer of emacs.)

Composing Interpretations

You use the Refinement Editor to compose interpretations in the second stage of building an application, when you have finished building the specifications in Slang. When you choose to create a new interpretation, you are given a choice of methods.

- To combine interpretations in parallel, choose via colimit as the method. You must then specify each of the interpretations to be combined, and create the required interpretation morphisms.
- To combine interpretations sequentially, choose via sequential composition as the method. No interpretation morphisms are required in this case; you need only choose the interpretations to be combined.

When you combine interpretations via colimit, an interpretation diagram is displayed in the Refinement Editor. In an interpretation diagram, arcs between nodes are blue, and represent interpretations between specs, while arcs between arcs are purple, and represent interpretation morphisms, or mappings between interpretations. (Click Color Legend on the menu bar to see the meanings of colors in the graphical editors.)

Composing Interpretations in Parallel

To compose interpretations in parallel, you create a new interpretation from a colimit spec, then use the Refinement Editor to specify each of the component interpretations. The diagram you see initially is like the one shown in Figure 6-1.

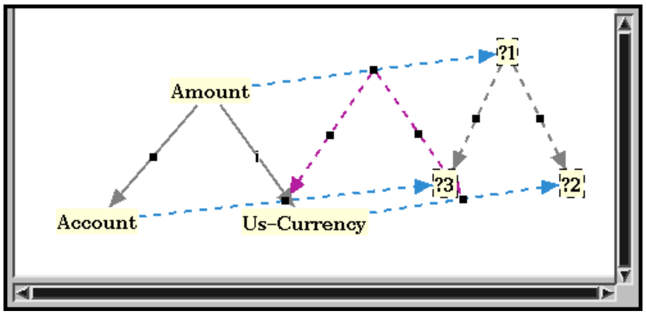

Figure 6-1 An unresolved interpretation diagram

How to: Create a Parallel Refinement in the Refinement Editor

1. From the Focus menu, choose Spec, then select the source spec for the interpretation you wish to refine. Choose a spec that was created as a colimit.
2. From the Spec menu, choose Make new Interpretation from.

   The Select/Construct Interpretation menu appears. This menu allows you to select a method for creating a new interpretation.
3. Select the method via colimit.
4. Choose New Window for display.

   The Refinement Editor appears and displays the defining diagram for the source of the new interpretation. For each component spec, a dotted blue arc indicates an unresolved interpretation. Dotted purple arcs between the blue arcs indicate interpretation morphisms between the interpretations.

   You must resolve all of the dotted arcs by selecting or creating specific interpretations and interpretation morphisms.

HINT In a complex or large diagram, nodes and arcs can overlap, making it difficult to click on the right object. To move a node, right-click on it and choose Move from the menu, then click on the new location for the node. The diagram is automatically redrawn. ❖

5. Starting with the blue interpretation arcs, middle click each dotted arc, choose Attach Interpretation, and select an interpretation. (If the source of the interpretation is itself the result of a colimit operation, you can construct the interpretation via colimit. In this case, a new interpretation diagram appears. See "Recursion in the Refinement Editor" on page 6-8.)

   The interpretation you choose is attached to the blue arc, which becomes solid. Notice that the target of the chosen interpretation is also attached to the target node.

6. Resolve the remaining interpretations in the same way.

7. When the interpretations have been resolved, middle click each dotted purple arc and choose Attach Interp Morphism. The interpretation morphisms are automatically created an attached to the purple arcs.

   Notice that, in the course of creating the interpretation morphism, SPECWARE also infers the morphism between the target specs, and attaches it to the gray arc between the target nodes.

8. When all arcs are resolved, middle click the diagram background and choose Finish Edit.

   SPECWARE creates the new interpretation by composing the interpretations you have specified. The new interpretation is attached to the blue arc from the source spec you focused on in Step 1.

   In the process of creating the new interpretation, SPECWARE also automatically creates its target spec. This is the colimit of the target specs of the component interpretations. The new colimit spec is given a generated name and attached to the target node of the new interpretation.

9. To rename the new interpretation, middle click the blue arc and choose Visit attached Interpretation. In the resulting Interpretation menu, choose Rename. Enter the new name in the type-in box and click Do It.

10. To rename the new target spec, middle click the target node and choose Visit attached Spec. In the resulting Spec menu, choose

Rename. Enter the new name in the type-in box, and click Do It. Left-click the node to redisplay it with the new name.

Composing Interpretations Sequentially

To compose interpretations sequentially, create a new interpretation from the first source spec and choose the method via sequential composition from the menu. You must specify each of the interpretations to be combined in sequence. The diagram you see initially is like the one shown in Figure 6-1.

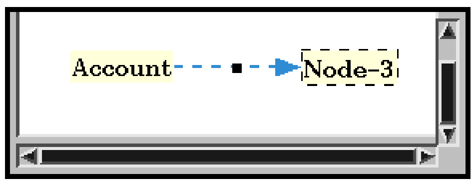

Figure 6-2 An unresolved sequential composition

The methods via definitional extension and via translation are variations of sequential composition and work similarly. They differ in that the interpretation offered as the first in the sequence is a definitional extension or translation. The diagram you see initially is like the one shown in Figure 6-1.

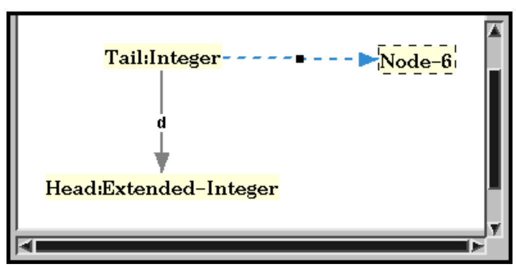

Figure 6-3 An unresolved composition via definitional extension

How to: Create a Sequential Refinement in the Refinement Editor

1. From the Focus menu, choose Spec, then select the source spec for the first interpretation in the sequence.
2. From the Spec menu, choose Make new Interpretation from.

   The Select/Construct Interpretation menu appears. This menu allows you to select a method for creating a new interpretation.
3. Select the method via sequential composition.
4. Choose New Window for display. The Refinement Editor appears, showing the source spec and a dotted blue arc to a node with a generated name. The dotted arc indicates the first interpretation in the sequence.
5. The Select/Construct Interpretation menu appears, offering a choice of all interpretations whose source is the spec you focused on. Choose the first interpretation in the sequence to be composed.

   The selected interpretation is attached to the blue arc, and its target spec is attached to the target node. A pop-up menu appears, asking whether you wish to extend the refinement.
6. To extend the sequence of interpretations to be composed, click yes in the pop-up menu.

   A new dotted arc and node appear in the Diagram Editor, indicating the extension to be resolved, and the Select/Construct Interpretation menu appears again. Only those interpretations that are vertically compatible are offered for selection; that is, those whose source spec is the target spec of the previous interpretation.

   Each time you select or construct a new interpretation in the sequence, it is added to the diagram and the Extend refinement? menu appears again. You can continue extending the interpretation as long as necessary.
7. To finish the sequence of interpretations to be composed click no in the pop-up menu.

   SPECWARE creates a new interpretation by composing the interpretations you have specified. The new interpretation is attached to a new arc from the original source to the final target.

The Interpretation menu appears for the new interpretation, allowing you to rename or inspect it.

Recursion in the Refinement Editor

In the course of resolving an interpretation diagram, you sometimes need to create new interpretations. Whenever you create an interpretation in a diagram by composition (that is, any method other than identity or direct selection), you must display a new interpretation diagram and resolve it. In the course of resolving the new diagram, you might again need to construct or infer an interpretation, which requires a new diagram. This process can continue recursively until you reach existing interpretations from which you can construct the needed interpretations.

When you have specified the needed interpretations at the lowest level of recursion, finish the edit of that diagram by choosing Finish Edit from the diagram's middle-click menu. You can then finish the edit of the next diagram up, and so on. By this means, you pop back out through the diagrams back to the highest level. The number of diagrams in the recursion depends on the complexity of your application.

HINT During recursions, you can choose to display each new diagram in the same window. This helps you keep track of the order in which the recursions appear. When you choose to display it in the same window, the new diagram is displayed above the previous diagram. ❖

Refining to Code

When you have finished building the specifications in Slang (the first stage of application building), and combined Slang-to-Slang interpretations to interpret your abstract domain theories in terms of the concrete computational theories in the SPECWARE library and Slang Base (the second stage of application building), you are ready to refine the specifications to code.

To refine a set of specifications all the way to code, you construct a Slang-to-code interpretation. You first build interpretations from your abstract specs to a set of library specs. You then compose these Slang-to-Slang interpretations sequentially with Slang-to-code interpretations, constructing a new Slang-to-code interpretation from your abstract spec to the target language.

If your top-level (most complex) abstract spec is interpreted in terms of a single library spec, you simply combine this interpretation sequentially with the Slang-to-code interpretation for that spec. In other cases, you must use one of the composition methods to construct a Slang-to-code interpretation to be combined with the spec-to-library interpretation.

- If it is offered, choose the via instantiation method. This option is only offered for interpreting instances of `ARRAY` or `LIST` (parameterized specs) to code.
- If your abstract specs are interpreted in terms of several library specs, choose the via colimit method to combine the library-to-code interpretations.

How to: Refine to Code Via Sequential Composition

Figure 6-4 shows a specification **A** that has been refined to an implementable library spec **B**.

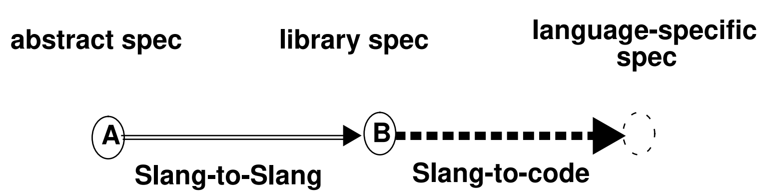

Figure 6-4 Constructing a Slang-to-code intepretation

To interpret spec **A** to code:

1. Focus on the abstract spec **A**.
2. From the Spec menu, choose Make new Spec-to-Lisp Interpretation from (or Make new Spec-to-C++ Interpretation from).
3. From the Select/Construct Interpretation menu, choose the method via sequential composition.

4. Choose the existing interpretation from **A** as the first in the sequence.
5. Choose the Spec-to-Lisp interpretation from **B** as the last in the sequence.
6. Middle click the background of the diagram and choose Finish Edit. The new Spec-to-Lisp interpretation is created, and Write code to file now appears on the diagram's middle-click menu.
7. Save and test the code, as described in "Saving and Testing Code" on page 6-13.

How to: Construct Slang-to-Code Via Colimit

In the course of creating the new Slang-to-code interpretation from an abstract spec **A** to code, you need a single Slang-to-code interpretation from library spec **B**. In practice, **A** is constructed from various component specs, which are interpreted to various library specs. Even if they are all interpreted to the same library spec, each interpretation is to a distinct copy. You must construct a new Slang-to-code spec that combines all of the component library-to-code interpretations.

To construct a Slang-to-code interpretation via colimit:

1. Focus on the highest-level abstract spec **A**.
2. From the Spec menu, choose Make new Spec-to-Lisp Interpretation from. (For purposes of this example, we assume a Lisp implementation. To create a C++ implementation, choose Make new Spec-to-C++ Interpretation from.)
3. The Select/Construct Interpretation menu appears. This menu allows you to select a method for creating a new interpretation. Select the method via colimit.
4. Choose New Window for display. The Refinement Editor appears, showing the first recursion into the structure that created the source spec. The dotted arcs indicates the interpretations and interpretation morphisms that need to be resolved.

   Note that the colors of the arcs are different from those that appeared when you created Slang-to-Slang interpretations. Click Color Legend in the menu bar to see the meaning of the arc colors.
5. Middle click the center point in a dotted interpretation arc.

6. From the menu, choose Attach Spec-to-Lisp Interpretation.
7. Choose the method via definitional extension. This allows you to combine the existing interpretation from the source spec with a Spec-to-Lisp interpretation from its target. You recurse into a new diagram to do this.
8. Select the existing interpretation as the first in the sequence. The menu offers the appropriate Spec-to-Lisp interpretation as the next in the sequence.
9. When you have selected the interpretations to be composed, middle-click the background of the diagram and choose Finish Edit. The new composed interpretation is created.
10. In the original diagram, continue resolving all interpretations in this way.
11. When all interpretations have been resolved, use the same method to resolve all interpretation morphisms, choosing the Spec-to-Lisp or Spec-to-C++ options.
12. When the interpretation morphisms are resolved, middle click the background of the diagram and choose Finish Edit. The new Spec-to-Lisp interpretation is created, and Write code to file now appears on the diagram's middle-click menu.
13. Save and test the code, as described in "Saving and Testing Code" on page 6-13.

How to: Construct Slang-to-Code Via Instantiation

When you create a Slang-to-code interpretation for an instance of a parameterized spec, the menu offers the method via instantiation. When you choose this method, the Refinement Editor shows a graph like that shown in Figure 6-5.

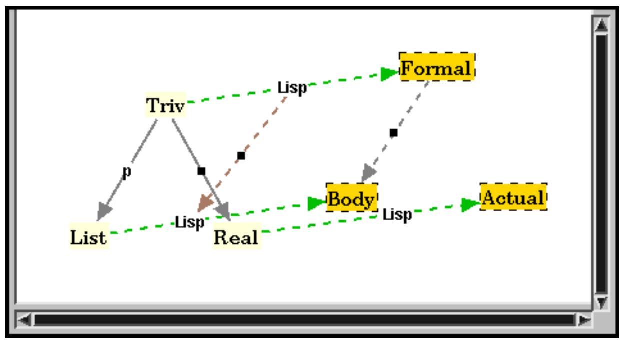

Figure 6-5 Refining `LIST-of-REALS` to Lisp

You must resolve two arcs in this diagram:

- the single interpretation morphism, represented by the dotted arc between the Lisp (or C++) mediators
- the interpretation from the element specification (`REAL`, in the example) to code

1. Middle-click the center point of the interpretation morphism arc.
2. Choose the existing morphism that is offered on the menu. SPECWARE infers the remaining relationships.
3. Middle-click the center point of the remaining interpretation.
4. Choose Attach Spec-to-Lisp (or Spec-to-C++) interpretation.
5. Choose or construct an interpretation.
6. Middle click the background of the diagram and choose Finish Edit. Write code to file now appears on the diagram's middle-click menu.

Saving and Testing Code

When you have finished building a refinement to code, the mediator of the final interpretation is a translation of your definitions to the target language. This translation can be written out as a code file, which you can embed into a larger application.

How to: Save and Test Code

1. When you have chosen Finish Edit to create your final Slang-to-code interpretation, the interpretation menu appears. Write code to file and Write code to file named now appear in the interpretation menu. (Once the final interpetation has been created, you can middle click it at any time to access these commands.)
2. Choose Write code to file to write the generated code to a file in the current working directory. The file is automatically named for the spec which is the source of the final interpretation. Normally, this is the highest level spec in your application. A Lisp file has the `.lisp` extension; a C++ source file has the `.cc` extension.

   If you wish to use a different name or location, choose Write code to file named, enter the path and file name in the type-in box, and click Do It.
3. Display the source file in an emacs buffer to inspect it.
   - Look for the renaming map at the end of the file to see if the generated function names differ from your operation names.
   - Check that functions take the number of arguments you expect.
4. If you have refined to Lisp, you can load the Lisp file and test it in the `*REFINE*` buffer in emacs.

   a) Load the file. At the Lisp prompt, enter the command:

   ```
   (load "file-name.lisp")
   ```

   b) Change to the `CODE` package. At the Lisp prompt enter the command:

   ```
   :pa CODE
   ```

   c) At the Lisp prompt, execute the functions defined in your Lisp code file to test them.

5. If you have refined to C++, you must compile the code to test it.

   a) Edit the source file to add a `main()` function that exercises each of the functions.

   b) Compile the source code file using a gnu C++ compiler.

   c) Load and run the object code in a shell window.

Appendix A

Customizing Specware

The menu bar provides access to behavioral and display parameters that control the way SPECWARE works. You can:

- Change the initial window layout using the Configure menu.
- Set behavior or display preferences for SPECWARE in general, using the Customize menu.
- Set behavior or display preferences for different parts of the KITP theorem prover, using the Prover Options menu.

You can also change the default behavior or appearance by setting these parameters in an initialization file.

Changing the System Window Layout

To change the standard window layout of SPECWARE, use the Configure menu. You can choose from the following standard configurations:

Spec diagram development, emacs panes

Large spec diagram development window

Large spec diagram development, emacs panes

Emacs pane

A recommended configuration is Emacs pane only, since new graphics windows are created as needed for operations.

You can right click on the background of a graphical window to get a menu of window display operations. You can also reshape and move windows using your window manager's standard techniques.

However, it is best not to close graphical windows from the window manager, as the Lisp system is not always aware that they have been closed.

Setting Behavior Preferences

To specify your preferences for system behavior, click Customize in the menu bar. In the resulting dialog box, specify the behaviors you prefer. To establish the new preferences you have selected, click Do It. To disregard your changes and dismiss the dialog box, click Abort.

The following table shows the system behavior parameters and their default values.

Table 1-1 System Behavior Parameters

Parameter	Default Value
When parsing formulas, report suspiciously named variables?	No
When parsing formulas, report variables that occur just once?	Yes
Verify parsed theorems (i.e. run prover when parsing specs)?	No
Verify parsed axiom translations (i.e. run prover when parsing spec-morphisms and interpretations)?	No
Assume imported conjectures are theorems?	Yes
Assume preceding local conjectures are theorems when proving a conjecture?	Yes
Ignore following local theorems when proving a conjecture?	Yes
Select conjectures to verify?	No
Specify proof directives for: Every-Batch, Every-Proof, or None	None
When prover fails, query user to accept theorems anyway?	Yes
For diagram refinements, always create a new codomain diagram?	Yes
For diagram refinements, always copy domain shape to codomain?	Yes

Table 1-1 System Behavior Parameters

Parameter	Default Value
In parser, ask before replacing an object?	No

Setting Display Preferences

To specify your preferences for display behavior, click Customize in the menu bar. In the resulting dialog box, specify the behaviors you prefer. To establish the new preferences you have selected, click Do It. To disregard your changes and dismiss the dialog box, click Abort.

The following table show parameters that you can set which affect display behavior, and their default values.

Table 1-2 System Display Parameters

Parameter	Default Value
Always use current window to display new picture?	No
Always use new window to display new picture?	No
Display picture borders?	No
Suppress node names?	Yes

Setting Prover Preferences

To specify your preferences for prover behavior, click Prover Options in the menu bar and select the part of the prover for which you wish to set preferences.You can choose to use default values for all options, to set the directory where proof information is written, or to set behavioral or display preferences.

When you choose to set behavioral or display preferences, use the resulting dialog box to specify the behaviors you prefer. To establish the new preferences you have selected, click Do It. To disregard your changes and dismiss the dialog box, click Abort.

The following table shows the parameters that affect prover behavior, and their default values.

Table 1-3 KITP Behavior Parameters

Parameter	Default Value
Conjectures-to-Focus (All, 1st, None, Ask)	All
Search-Strategy (O-F, D-F, B-F)	O-F
Use-Typed-Inference?	Yes
Assume-Hypothesis-Consistent?	No
Include-Sort-Axioms?	Yes
Use-Free-Paramodulation?	Yes
Maximum-Input-Clauses	1000
Generic-Forward-Inference?	No
Forward-Depth-Limit	2
Forward-Derivation-Limit	100
Backward-Depth-Limit	5
Backward-Derivation-Limit	6000
Forward-Fnsnest-Limit	1
Maximum-New-Variables	1
Maximum-Witnesses-Deduced	12
Return-Value-when-Fail (False, True, Ask)	False
Choice-of-Dump-Proofs (None, Failed, All)	None
Ask-when-Dump-Proof?	No
Enforce-Strong-Dependency?	Yes

The following table shows parameters that affect prover display, and their default values. .

Table 1-4 KITP Display Parameters

Parameter	Default Value
Suppress-Kitp-Message?	Yes
Display-Proof-Object? (No, Obj, Cnj, Both)	Cnj
Select-Witnesses-by (User, Prover)	Prover
Display-Input-Clauses?	No
Forward-Interact?	No
Display-Forward-Derivation?	No
Backward-Interact?	No
Display-Backward-Derivation?	No

Table 1-4 KITP Display Parameters

Parameter	Default Value
Display-Proof-Tree?	Yes
Display-Type-Constraints?	No
Ignore-Data-Types?	No
Ask-On-Failure	No

Setting Preferences in an Initialization File

When SPECWARE starts up, the system looks for an initialization file in the `~/specware/home-init` directory. During installation, the default initalization file is copied to this location. The name of the expected initialization file is a concatenation of the distribution version with the string "`-home-init.lisp`"-- for example:

```
~/specware/home-init/specware-2-0-2-home-init.lisp
```

If such a file is found, the system asks for confirmation to load and execute it. If no such file is found, the system asks for confirmation to load and execute the default initialization file:

```
install-dir/specware/2-0-2/code/post-code/home-init-for-
specware-2-0-2.lisp
```

If you wish to set any system behavior or display preferences automatically at startup, edit the initialization file in your home directory to set the desired variable to the desired value. The system variables are in the `SP` package, while the theorem prover variables are in the `KITP` package.

The tables below show the names of the variables that can be set in the initialization file, and the types of their possible values. These variables correspond to the choices on the preference menus.

Table 1-5 shows the system variables that can be set in the initialization file.

Table 1-5 System Preference Variables

System Variable	Allowed Values	Default Value
`*report-suspiciously-named-vars?*`	`boolean`	`nil`
`*report-solitary-vars?*`	`boolean`	`t`
`*verify-parsed-theorems?*`	`boolean`	`nil`
`*verify-parsed-axiom-translation?*`	`boolean`	`nil`
`*assume-imported-conjectures-are-theorems?*`	`boolean`	`t`
`*assume-preceding-conjectures-are-theorems?*`	`boolean`	`t`
`*ignore-following-theorems?*`	`boolean`	`t`
`*menu-select-conjectures-to-prove?*`	`boolean`	`nil`
`*menu-specify-proof-directives*`	`['every-batch, 'every-proof, 'none]`	`‘none`
`*query-after-prover-failure?*`	`boolean`	`t`
`*always-use-current-window?*`	`boolean`	`nil`
`*always-use-new-window?*`	`boolean`	`nil`
`*show-picture-borders?*`	`boolean`	`nil`
`*suppress-node-names?*`	`boolean`	`t`
`*always-make-new-dgref-codomain-diagram?*`	`boolean`	`t`
`*always-copy-shape-category-to-new-dgref-codomain-diagram?*`	`boolean`	`t`
`*ask-before-replacing-specware-object?*`	`boolean`	`nil`

Table 1-6 shows the behavior preference variables that can be set for the KITP theorem prover.

Table 1-6 KITP Behavior Preference Variables

System Variable	Allowed Values	Default Value
choice-of-proving-conjectures	["All", "1st", "None", "Ask"]	“All”
search-strategy	["O-F", "D-F", "B-F"]	“O-F”
typed-inference	boolean	t
assume-hypothesis-consistent	boolean	nil
use-sort-axioms	boolean	t
use-free-paramodulation	boolean	t
equality-maybe-used-for-rewriting	boolean	nil
max-input-clauses	integer	1000
generic-forward-inference	boolean	nil
forward-depth-limit	integer	2
forward-derivation-limit	integer	100
local-depth-limit	integer	5
backward-derivation-limit	integer	6000
default-nest-limit	integer	1
max-induced-vars	integer	1
max-witnesses	integer	12
select-witness-by	[“User”, “Prover”]	“Prover”
return-value-when-fail	["False", "True", "Ask"]	“False”
choice-of-dump-proofs	["None", "Failed", "All"]	“None”
ask-before-dump-proof	boolean	nil
check-types	boolean	t
enforced-strong-dependency	boolean	t

Table 1-7 shows the display preference variables that can be set for the KITP theorem prover.

Table 1-7 KITP Display Preference Variables

System Variable	Allowed Values	Default Value
`*suppress-output*`	`boolean`	`t`
`*display-proof-object*`	`["No", "Obj", "Cnj", "Both"]`	`“Cnj”`
`*show-input-clauses*`	`boolean`	`nil`
`*show-sort-axioms*`	`boolean`	`nil`
`*forward-interact*`	`boolean`	`nil`
`*manual-start-chd*`	`boolean`	`nil`
`*show-backward-derivation*`	`boolean`	`nil`
`*explain-chd*`	`boolean`	`nil`
`*show-proof-tree*`	`boolean`	`t`
`*explain-fd*`	`boolean`	`nil`
`*show-typing*`	`boolean`	`nil`
`*ask-before-abort-proof*`	`boolean`	`nil`

Default Initialization File

The default initialization file for SPECWARE 2.0.3 is reproduced here for reference.

```
(in-package "SPECWARE")
(defun HOME-SPECWARE-INIT ()
;;  Misc
(re::verbose-set
       '*abort-breaks-in-background-spec-processes?* t)
; Don't set to nil unless you understand how to debug
; in Allegro processes.

;;   Configure options
(re::verbose-set
   '*INITIAL-SPECWARE-CONFIGURATION-NAME* 'SD-E)
; single-choice "Choose among "
; SD-E SD LARGE-SD-E EMACS-ONLY
; 'sd-e        --  Spec diagram development, emacs panes
; 'sd          --  Large spec diagram development window
; 'large-sd-e  --  Large spec diagram development,
;                    emacs panes
; 'emacs-only  --  Emacs pane

;;  Xemacs window parameters
(re::verbose-set
   'RI::*LOWER-EMACS-WINDOW-TOP-BY-TITLE-HEIGHT?* 'NIL)
; boolean "Not in any menu -- see following notes."
; t --  always lower emacs window top by title height
; nil --  always not lower emacs window top by title ht
; *undefined* -- unpredictable; user mouse select
;         action each time the frame is first brought up
(re::verbose-set
   'RI::*EMACS-TITLE-BAR?* 'NIL)
; boolean "??"
(re::verbose-set
   'RI::*EMACS-WINDOW-BORDER* '3)
; expression "Integer"
(re::verbose-set
   'RI::*EMACS-WINDOW-TITLE-HEIGHT* '22)
; expression "Integer"
(re::verbose-set
   'RI::*RESHAPE-EMACS?* 'NIL)
; boolean "Reshape Emacs window whenever Specware
;      window changes?"
```

```
;;  Specware options
(re::verbose-set
   '*REPORT-SUSPICIOUSLY-NAMED-VARS?* 'NIL)
; boolean "When parsing formulas, report suspiciously
;  named variables?"
(re::verbose-set  '*REPORT-SOLITARY-VARS?* 'T)
; boolean "When parsing formulas, report variables that
;  occur just once?"
(re::verbose-set  '*VERIFY-PARSED-THEOREMS?* 'NIL)
; boolean "Verify parsed theorems
;  (i.e., run prover when parsing specs)? "
(re::verbose-set
   '*VERIFY-PARSED-AXIOM-TRANSLATION?* 'NIL)
; boolean "Verify parsed axiom translation
;  (i.e., run prover when parsing spec-morphisms and
;   interpretations)? "
(re::verbose-set
   '*ASSUME-IMPORTED-CONJECTURES-ARE-THEOREMS?* 'T)
; boolean "Assume imported conjectures are theorems? "
(re::verbose-set
   '*ASSUME-PRECEDING-CONJECTURES-ARE-THEOREMS?* 'T)
; boolean "Assume preceding local conjectures are
;  theorems when proving a conjecture? "
(re::verbose-set  '*IGNORE-FOLLOWING-THEOREMS?* 'T)
; boolean "Ignore following local theorems when proving
;  a conjecture? "
(re::verbose-set
   '*MENU-SELECT-CONJECTURES-TO-PROVE?* 'NIL)
; boolean "Select conjectures to verify? "
(re::verbose-set
   '*MENU-SPECIFY-PROOF-DIRECTIVES* 'NONE)
; single-choice "Specify proof directives for:
;   " EVERY-BATCH EVERY-PROOF NONE
(re::verbose-set  '*QUERY-AFTER-PROVER-FAILURE?* 'T)
; boolean "When prover fails, query user to accept
;  theorems anyway? "
(re::verbose-set  '*ALWAYS-USE-CURRENT-WINDOW?* 'NIL)
; boolean "Always use current window to display new
;  picture? "
(re::verbose-set  '*ALWAYS-USE-NEW-WINDOW?* 'NIL)
; boolean "Always use new window to display new pic? "
(re::verbose-set  '*SHOW-PICTURE-BORDERS?* 'NIL)
; boolean "Display picture borders? "
(re::verbose-set  '*SUPPRESS-NODE-NAMES?* 'T)
; boolean "Suppress node names? "
```

```
(re::verbose-set
   '*ALWAYS-MAKE-NEW-DGREF-CODOMAIN-DIAGRAM?* 'T)
; boolean "For diagram refinements, always create a new
;   codomain diagram? "
(re::verbose-set
   '*ALWAYS-COPY-SHAPE-CATEGORY-TO-NEW-DGREF-CODOMAIN-
DIAGRAM?* 'T)
; boolean "For diagram refinements, always copy domain
;   shape to codomain? "
(re::verbose-set
   '*ASK-BEFORE-REPLACING-SPECWARE-OBJECT?* 'NIL)
; boolean "In parser, ask before replacing an object? "
(re::verbose-set  'RI::*RESHAPE-EMACS?* 'NIL)
; boolean "Reshape emacs window whenever Specware
;   window changes? "

  ;;  KITP options
(re::verbose-set
   'KITP::*CHOICE-OF-PROVING-CONJECTURES* '"All")
; single-choice "Conjectures to Focus:"
;   "All" "1st" "None" "Ask"
(re::verbose-set  'KITP::*SEARCH-STRATEGY* '"O-F")
; single-choice "Search Strategy: " "O-F" "D-F" "B-F"
(re::verbose-set  'KITP::*TYPED-INFERENCE* 'T)
; boolean "Use Typed Inference? "
(re::verbose-set
   'KITP::*ASSUME-HYPOTHESIS-CONSISTENT* 'NIL)
; boolean "Assume Hypothesis Consistent? "
(re::verbose-set  'KITP::*USE-SORT-AXIOMS* 'T)
; boolean "Include Sort Axioms? "
(re::verbose-set  'KITP::*USE-FREE-PARAMODULATION* 'T)
; boolean "Use Free Paramodulation? "
(re::verbose-set
   'KITP::*EQUALITY-MAYBE-USED-FOR-REWRITING* 'NIL)
; boolean "Equality Maybe Used for Rewriting? "
(re::verbose-set  'KITP::*MAX-INPUT-CLAUSES* '1000)
; expression "Maximum Input Clauses: "
(re::verbose-set
   'KITP::*GENERIC-FORWARD-INFERENCE* 'NIL)
; boolean "Generic Forward Inference?"
(re::verbose-set  'KITP::*FORWARD-DEPTH-LIMIT* '2)
; expression "Forward Depth Limit: "
(re::verbose-set
   'KITP::*FORWARD-DERIVATION-LIMIT* '100)
; expression "Forward Derivation Limit:"
```

```
(re::verbose-set  'KITP::*LOCAL-DEPTH-LIMIT* '5)
; expression "Backward Depth Limit: "
(re::verbose-set
   'KITP::*BACKWARD-DERIVATION-LIMIT* '6000)
; expression "Backward Derivation Limit:"
(re::verbose-set  'KITP::*DEFAULT-NEST-LIMIT* '1)
; expression "Default Fnsnest Limit: "
(re::verbose-set  'KITP::*MAX-INDUCED-VARS* '1)
; expression "Maximum New Variables: "
(re::verbose-set  'KITP::*MIN-WITNESSES* '12)
; expression "Minimum Witnesses Deduced: "
(re::verbose-set  'KITP::*SELECT-WITNESS-BY* '"Prover")
; single-choice "Select Witnesses by: " "User" "Prover"
(re::verbose-set
   'KITP::*RETURN-VALUE-WHEN-FAIL* '"False")
; single-choice "Return Value when Fail: "
;   "False" "True" "Ask"
(re::verbose-set
   'KITP::*CHOICE-OF-DUMP-PROOFS* '"None")
; single-choice "Choice of Dump Proofs: "
;   "None" "Failed" "All"
(re::verbose-set  'KITP::*ASK-BEFORE-DUMP-PROOF* 'NIL)
; boolean "Ask when Dump Proof? "
(re::verbose-set  'KITP::*CHECK-TYPES* 'T)
; boolean "Check Data Types? "
(re::verbose-set
   'KITP::*ENFORCED-STRONG-DEPENDENCY* 'T)
; boolean "Enforce Strong Dependency? "

  ;;  KITP display options
(re::verbose-set  'KITP::*SUPPRESS-OUTPUT* 'T)
; boolean "Suppress Kitp Message? "
(re::verbose-set  'KITP::*DISPLAY-PROOF-OBJECT* '"Cnj")
; single-choice "Display Proof Object? "
;   "None" "Obj" "Cnj" "Both"
(re::verbose-set  'KITP::*SHOW-INPUT-CLAUSES* 'NIL)
; boolean "Display Input Clauses? "
(re::verbose-set  'KITP::*SHOW-SORT-AXIOMS* 'NIL)
; boolean "Display Sort Axioms? "
(re::verbose-set  'KITP::*FORWARD-INTERACT* 'NIL)
; boolean "Forward Interact? "
(re::verbose-set  'KITP::*MANUAL-START-CHD* 'NIL)
; boolean "Backward Interact? "
```

```
(re::verbose-set
   'KITP::*SHOW-BACKWARD-DERIVATION* 'NIL)
; boolean "Display Backward Derivations? "
(re::verbose-set  'KITP::*EXPLAIN-CHD* 'NIL)
; boolean "Explain Rejected Derivations? "
(re::verbose-set  'KITP::*SHOW-PROOF-TREE* 'T)
; boolean "Display Proof Tree? "
(re::verbose-set  'KITP::*EXPLAIN-FD* 'NIL)
; boolean "Explain Forward Derivation? "
(re::verbose-set  'KITP::*SHOW-TYPING* 'NIL)
; boolean "Display Type Constraints? "
(re::verbose-set  'KITP::*ASK-BEFORE-ABORT-PROOF* 'NIL)
; boolean "Ask On Failure? "

;;  KITP misc options
(re::verbose-set
   'KITP::*USER-PROBLEM-SET-DIRECTORY* 'RE:*UNDEFINED*)
; string "Set subdirectory for dumping proof objects"
)

(home-specware-init)
```

Index

D

E

K

L

M

S

T

U

V

W

X