Rosetta Strawman
Version 0.3

Perry Alexander and *Cindy Kong*
Information & Telecommunication Technology Center
The University of Kansas
{alex,ckong}@ittc.ku.edu

David Barton
EDAptive Computing, Inc.
dbarton@edaptive.com

Peter Ashenden
Ashenden Systems
peter@ashenden.com.au

Catherine Menon
Adelaide University
menon@cs.adelaide.edu.au

April 17, 2003

Contents

Chapter 1

Introduction

This document serves as a usage guide for the Rosetta specification language. It defines most of the base Rosetta semantics in an *ad hoc* fashion and provides general usage guidelines through examples.

The basic unit of Rosetta specification is a *facet*. Each facet is a parameterized collection of declarations and definitions specified using a domain theory. Facets are used to define: (i) system models; (ii) system components; (iii) architectures; (iv) libraries; and (v) semantic domains.

Although definitions within facets use many different semantic representations, the semantics of facet composition, inclusion and data types are shared among all facets. Collections of facets are composed using a collection of common operations that operate regardless of semantic domain. The basic facet definition provides an encapsulation, parameterization and naming convention for Rosetta systems.

This document describes the semantics of facets in an ad hoc fashion. Its intent is to provide an introduction to facets and their various uses without discussing any specific domain theory. In addition to facets themselves, this document also defines a type system shared among facet definitions. Basic types and operations available to Rosetta specifiers are identified and primitive definitions provided. Finally, the system construct used to describe hegerogeneous systems using facets is presented. The system construct supports definition of facets, assumptions, declarations, and verifications in support of a systems level design activity.

Chapter 2

Items, Variables, Values and Types

2.1 Items and Values

Rosetta's basic semantic unit is called an *item*. Item structures result when Rosetta descriptions are parsed prior to manipulation. Although most users will never deal directly with items, they present an effective way to describe the relationships between variables, values and types.

Informally, an item consists of a *label* naming the item, a *value* the item represents, and a *type* from which specific item values must be chosen. When any structure is defined in a Rosetta specification, an item is created with the specified label. Variables, constants, terms, even facets themselves are items in a Rosetta specification. When a label is referred to in a specification, it refers to the value of the item it is associated with. An item's set of potential values is delineated by its associated type. In a legal Rosetta specification, every item's value is an element of its associated type. A more complete description of items can be found in Chapter 8.

Value items, or simply values, represent items that can be used as values for other items. There are three general classes of values: (i) elements; (ii) composite items; and (iii) functions. Elemental values represent primitive, atomic values that are directly manipulated by Rosetta. Elemental values include such things as integers, naturals, characters, bits and boolean values. Traditional programming languages refer to elemental values as scalar. Composite values are constructed from other values. Composite values include such things as sequences, sets, and facets. Function values represent operations that by definition exhibit properties of mathematical functions. The name `universal` is used to refer to all possible Rosetta values.

All Rosetta types are *sets* where a set is simply a packaged collection of values. Functions and properties for sets are defined completely in Section 2.5.1. Throughout this document, the terms `set` and `subtype` are used interchangeably to refer to a subset of a set. The term `type` refers to any possible set.

By convention, we say that the concrete syntax `v::T` in a declarative region of a facet declares a *variable* item of type `T` whose value is an element of set `T`. No expression is included to constrain the value of `v`, thus its value is not known.

The concrete syntax `v::T is c` appearing in a declarative region of a facet, package, domain or component defines a new item `v` of type `T` whose constant value is `c`.

2.2 Elements

By definition, elements are values that are atomic and cannot be decomposed. Element types are sets of such values. Numbers such as 1, 5.32, and -32, characters such as 'a', 'B', and '1', and boolean values such as `true` and `false` represents such atomic values. In contrast, composite values such as sequences and sets

are not elemental in that each is defined by describing its contents. Element values are frequently called *scalars* in traditional programming languages.

The type `element` is comprised of the types `number` and `character`. The `element` type is largely a semantic construction with no common operations over all members of the type other than simple equality (=) and inequality (/=) operations.

2.3 Numbers

Numeric types include standard sets of values associated with traditional number systems. Predefined numeric types include `real`, `integer`, `natural`, `bit`, `imaginary`, `complex` and `boolean` and are listed in the following table:

Type	*Format*	*Subtype Of*
`complex`	`1+2*j, 3*e^(4*j)`	`number`
`real`	`-123.456, 123.456, 1.234e56`	`complex`
`posreal`	`123.456, 1.234e56`	`real`
`negreal`	`-123.456, -1.234e56`	`real`
`rational`	`123/456`	`real`
`integer`	`123,0,-123`	`rational`
`natural`	`0,123,`	`integer`
`posint`	`123`	`natural`
`negint`	`-123`	`integer`
`bit`	`1,0`	`natural`
`imaginary`	`j, 5*j`	`complex`
`boolean`	`true, false`	`number`
`number`	*Any element of the above types*	`element`

Predefined operators defined over `number` and its subtypes include: (Assuming `A` and `B` are numbers)

Operation	*Format*	*Valid For*
Negation	`- A`	`number`

2.3.1 Complex Numbers

Complex numbers form the most basic Rosetta number. All traditional number values are subtypes of `complex`. Traditional operations such as addition and subtraction are defined over complex numbers as anticipated. Projection functions extract real and imaginary values from complex values. `re` returns the magnitude of a number's real part while `im` returns the magnitude of the imaginary part. For any complex value `n` expressed in the cartesian form:

```
n = re(n)+im(n)*j
```

The polar form may also be used. For any complex value `n` expressed in the polar form:

```
n = mag(n)*e^(arg(n)*j)
```

The `conj` operator evaluates to the complex conjugate of its complex argument. The operator `mag` evaluates to the magnitude of the vector associated with a complex number in the complex plane. Additional predefined operators defined over `complex` and its subtypes include: (Assuming `A` and `B` are numbers)

Operation	*Format*	*Valid For*
Addition and Subtraction	`A+B,A-B`	`complex`
Multiplication and Division	`A*B,A/B`	`complex`
Power	`,A^ B`	`complex`
Square Root	`sqrt(A)`	`complex`
Imaginary and Real	`im(A), re(A)`	`complex`
Magnitude, Conjugate and Argument	`abs(A), conj(A), arg(A)`	`complex`
Trig functions	`sin(A), cos(A), tan(A)`	`complex`
	`arcsin(A), arccos(A), arctan(A)`	
	`sinh(A), cosh(A), tanh(A)`	
	`arcsinh(A), arccosh(A), arctanh(A)`	
Log and exponential	`exp(A), log(A), log10(A), log2(A)`	`complex`

Real Numbers

The type `real` can be defined as the subtype of `complex` such that `im(x)=0`. Formally:

```
real::subtype(complex) is sel(x::complex | im(x)=0);
```

The additional operations `min`, `max`, `floor`, `ceiling`, `round`, and `sgn` are defined over `real` as well as traditional ordering relationships:

Operation	*Format*	*Valid For*
Minimum and Maximum	`A min B, A max B`	`real`
Floor, Ceiling and Truncate	`floor(x), ceiling(x), trunc(x)`	`real`
Round	`round(x)`	`real`
Signum	`sgn(x)`	`real`
Ordering Relations	`A<B,A=<B,A>B,A>=B`	`real`

`min` and `max` evaluate to the minimum and maximum value of their arguments respectively.

`floor` evaluates to the greatest integer number less than or equal to its argument. Conversely, `ceiling` evaluates to the least integer number greater than or equal to its argument. `trunc` always truncates its value toward zero.

`round` evaluates to `ceiling` if the fractional part of its argument is greater than or equal to `0.5`. Otherwise, it evaluates to `floor`.

`sgn` is defined formally as:

```
sgn(x::real)::integer is if -(x=0) then x / abs(x) else 1 end if;
```

Classical ordering relationships are provided and defined the traditional manner.

The constant values `e` and `pi` are provided as `real` numbers:

Constant	*Format*	*Valid For*
Exponential	`e`	`number`
Pi	`pi`	`real`

Positive and Negative Real Numbers

The subtype `posreal` is defined as the subtype of `real` such that all values are greater than `0.0`. The subtype `negreal` is defined as the subtype of `real` such that all values are less than `0.0`. Formally:

```
posreal::subtype(real) is sel(x::real | x > 0);
negreal::subtype(real) is sel(x::real | x < 0);
```

No additional operators are defined over `posreal` or `negreal` beyond those defined for `real`.

Note that some operations such as negation are defined over `posreal` and `negreal`, but are not closed over those types.

Imaginary Numbers

The type `imaginary` is the subtype of complex such that its real part is 0. Formally:

```
imaginary::subtype(complex) is sel(x::complex | re(x)=0);
```

Imaginary numbers are formed by multiplying a real number by the predefined imaginary constant `j`. One can think of this as a conversion operation from real number types to imaginary numbers. Multiplying an imaginary number by the imaginary constant results in a non-imaginary value. Thus, `5*j*j` is equivalent to `-5` as expected.

The following operators are defined over `imaginary` beyond those defined for `complex`:

Operation	*Format*	*Valid For*
Minimum and Maximum	`A min B, A max B`	`imaginary`
Ordering Relations	`A<B,A=<B,A>B,A>=B`	`imaginary`

`min` and `max` evaluate to the minimum and maximum value of their arguments respectively.

Classical ordering relationships are provided and defined the traditional manner.

Rational Numbers

The type `rational` is the subtype of `real` such that each value is calculated as one integer value divided by another. Formally:

```
rational::subtype(real) is sel(x::real | exists(y,z::integer | x=y/z and z/=0));
```

No additional operators are defined over `rational` beyond those defined for `real`.

Integer Numbers

The subtype `integer` is a subtype of `rational` such that all values are discrete. Formally:

```
integer::subtype(rational) is sel(x::rational | floor(x)=ceiling(x));
```

The additional `mod`, `div` and `rem` functions are defined over `integer` in addition to operators defined over `rational`. All operators are defined in the traditional fashion.

Operation	*Format*	*Valid For*
Modulo Arithmetic	`x mod y`	`integer`
Integer Division	`x div y`	`integer`
Integer Remainder	`x rem y`	`integer`

`integer` is closed under `+`, `-`, `*`, but not under `/`, or trigonometric operations.

Natural Numbers

The subtype `natural` can be defined as the subtype of `integer` such that all values are greater than or equal to 0. Formally:

```
natural::subtype(integer) is sel(x::integer | x >= 0);
```

No additional operators are defined over `natural` beyond those defined for integer.

Note that some operations such as negation are defined over `natural` but are not closed over `natural`.

Positive and Negative Integer Numbers

The subtype `posint` is defined as the subtype of `integer` such that all values are greater than `0`. The subtype `negint` is defined as the subtype of `integer` such that all values are less than `0`. Note that `posint` is a subtype of `natural` while `negint` is not. Formally:

```
posint::subtype(natural) is sel(x::natural | x > 0);
negint::subtype(integer) is sel(x::integer | x < 0);
```

No additional operators are defined over `posint` or `negint` beyond those defined for `integer`.

Note that some operations such as negation are defined over `posint` and `negint` integers, but are not closed over those types.

Bits

The subtype `bit` can be defined as a subtype of `natural` consisting of the values `0` and `1`. Formally:

```
bit::subtype(natural) is {1,0};
```

Additional operators defined over bit include: (Assume the declarations `x,y::bit` and the definitions `a=% x`, and `b=% y`):

Operation	*Format*	*Definition*
`%`	`% 1, % true`	Converts bits and boolean.
Inverse	`not x`	`% -a`
Conjunction and Disjunction	`x and y, x or y`	`%(a and b), %(a or b)`
Negated Conjunction and Disjunction	`x nand y, x nor y`	`not(x and y), not(x or y)`
Exclusive or and nor	`x xor y, x xnor y`	`%(a xor b), %(a xnor b)`

The `%` operation translates between `bit` and `boolean` in such a way that `1` is isomorphic with `true` and `0` is isomorphic with `false`.

It should be noted that as a subtype of `natural`, `bit` is not closed under arithmetic operations such as plus and minus.

2.3.2 Lexical Structure of Number Constants

Numeric constants are represented by a strings of digits and optional sign, decimal point, exponential and radix indicators. Specifically:

- A number may be preceded by an optional "`-`" operator that inverts the sign of its argument. The number `-123` is equivalent to negative `123`.
- A single decimal point may be included in a number. The number `1.23` is interpreted in the traditional manner.
- A single exponent indicator may be included in a number. The number `1.234e7` is equivalent to `1.234` times `10` to the 7^{th} power.
- An optional radix may be included using the notation `R\N\eE` where `R` is the radix value (up to 16), `N` is a number, and `E` is an exponent. The radix value and the exponent value are always expressed in base 10 while the number value is specified in the indicated radix. The number `2\10001.1001\` is the base 2 representation of the binary real value `10001.1001`. Note that "\" is not a function, but a part of the number token itself.
- Imaginary numbers are formed by multiplying a `real` value by the complex root, `j`. The number `5.3*j` is interpreted in the traditional fashion.
- Complex numbers are formed by adding a real number to an imaginary number or using the polar form. The number `5+2*j` is interpreted as the complex number whose real part has magnitude `5` and imaginary part has magnitude `2`. The number `6*e^(pi*j)` is interpreted as the complex number whose magnitude is `6` and whose argument is `pi`.

2.3.3 Boolean

The Rosetta `boolean` type is defined by the two element set {`true`,`false`} and is a subtype of `number`. Although `boolean` is a number type, it is not a subtype of `complex` or `natural`.

The following operators are defined over `boolean` beyond those defined for `number`:

Operation	*Format*	*Valid For*
Minimum and Maximum	`A min B, A max B`	`boolean`
Ordering Relations	`A<B,A=<B,A>B,A>=B`	`boolean`
Bit/Boolean Conversion	`%A`	`boolean`

Logical Operations	`A and B, A or B, A xor B`
	`A nand B, A nor B, A xnor B`
	`not A`
		`boolean`

`min` and `max` evaluate to the minimum and maximum value of their arguments respectively. The maximum `boolean` value is `true` and the minimum `false`. Classical ordering relationships are provided and defined the traditional manner given the definitions of `true` and `false` as maximum and minimum values respectively.

The `%` operator converts between `boolean` and `bit` in the classical manner.

Classical logical operations including `and`, `or`, `xor`, `nand`, `nor` , `xnor`, and `not` are provided. `and` and `or` are synonyms for `min` and `max` while `not` is a synonym for the unary operation "`-`".

When treated as numeric values, `true` and `false` follow the following equivalence and ordering rules:

Property	*Meaning*
`false = -true`	`false` is equivalent to not `true`
`-false = true`	`true` is equivalent to not `false`
`forall(x::number | x /= true => x < true)`	`true` is the greatest number
`forall(x::number | x /= false => x > false)`	`false` is the least number

Boolean values obey ordering laws, but addition, subtraction, multiplication, division and other traditional operations are not defined. As numbers, `true` acts like positive infinity and `false` acts like negative infinity.

Consequences of this convention are numerous and useful. They include: (i) `max` is semantically the same as `or`; (ii) `min` is semantically the same as `and`; and (iii) `=<` is semantically the same as implication (`=>`) and `>=` is semantically the same as implied by (`<=`). The same laws apply to these operations in all cases, and the different signs are taken to be synonyms of each other, maintained here for the sake of historical recognition. Thus, we have the following table defining `min` and `max` over `true` and `false`:

A	*B*	*A min B*	*A max B*
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

This is identical to the truth table defining `and` and `or`. The negation operator, `-`, also follows directly from the numeric interpretation of `boolean`. The greatest positive number negated is the least negative number. Thus, `-true = false`. As negation is its own inverse, we know that `-(-x)) = x` for any boolean value `x`. Thus, `-false = true`. The resulting truth table has the form:

A	*-A*
false	true
true	false

Definitions for other logical operations follow directly. Of particular interest is the definition of implication as:

```
A => B == -A or B
```

By definition, this is equivalent to `-A max B`. Again, consider the truth table generated by the definition of `true` and `false` as numeric values:

A	*B*	*-A*	*-A max B*
false	false	true	true
false	true	true	true
true	false	false	false
true	true	false	true

This is semantically the same as the definition of implication. Reverse implication works similarly and the definition of equivalence (`A=B`) is consistent with the above definition. Further, when values are restricted to `boolean`, the following equivalences hold:

```
A => B == A =< B
A <= B == A >= B
A <= B and B <= A == A = B
```

It should be noted that Rosetta does not define logical equivalence, `iff`, separately from numerical equivalence. Given the mathematical definition of booleans, the normal equivalence operations are sufficient.

Example 1 (Number Constants) *Examples of defining number constants, including* `complex`*, its subtypes, and* `boolean` *include:*

Number	Interpretation
`12`	*The standard decimal constant* 12
`-12`	*The standard decimal constant* -12
`1.2`	*The standard decimal constant* 1.2
`1.23e4`	*The decimal constant* $1.23 * 10^4$
`1.23e-4`	*The decimal constant* $1.23 * 10^{-4}$
`16\E.1F\e5`	*The hexadecimal constant* $E.1F_{16} * 16^5$
`2\1101\e-7`	*The binary constant* $1101_2 * 2^{-7}$
`false`	*The boolean constant false*
`true`	*The boolean constant true*

2.4 Characters

The type `character` is a subtype of `element` and is defined as the collection of Unicode values.

Given `a::character`, `b::character`, and `n::natural` in the range of Unicode code values, operators on `character` include:

Operation	*Format*	*Definition*
Ord and character	`ord(a), char(n)`	Unicode value
Ordering Relations	`a<b, a=<b` ...	`ord(a)<ord(b), ord(a)=<ord(b)` ...
Raise and lower case	`uc(a), dc(a)`	Raise/lower case

2.4.1 ASCII Type

The type `ascii` is a subtype of `character` and is defined as the subset of characters that represent ASCII values. Formally:

```
ascii::subtype(character) is map(char,{0,..255});
```

No new functions are defined on ASCII other than those defined over `character` values.

```
%% Should add other character sets besides ascii.
```

Lexical Structure of Character Constants

Unicode literals are expressed using the standard notation `'Z'` where `Z` is a Unicode character of the form 'U+XXXX' where XXXX is a 4, 5, 6 or 8-digit, hexadecimal number. In the case of 5 and 6 digit numbers, leading zeros are not allowed. The enclosing ticks are significant and must be included. `character` values that have no printable form must be specified using their Unicode hex value.

Example 2 (Character Constants) *Examples of defining character constants:*

Character	Interpretation
'U+DD01'	*Unicode character associated with hex DD01*
'U+DD001'	*Unicode character associated with hex DD01*
'U+DD1001'	*Unicode character associated with hex DD01*
'U+DD01DD01'	*Unicode character associated with hex DD01*
'1'	*ASCII character 1*
'a'	*ASCII character a*

```
%% Need some decent examples of unicode constants
```

Summary: The following predefined elemental types are predefined for all Rosetta specifications:

- `element` — All atomic values including `number` and `character`.
- `number` — Subtype of `element` consisting of `complex` and `boolean` values.
- `complex` — Subtype of `number` and root of the numeric type tree. Formed as the the sum of any `imaginary` and any `real`. Thus, `7.0e2 + 2.1e4*j` is `complex`.

- `imaginary` — Subtype of `complex` where the real part is 0. Formed by any multiple of `j` and a `real`. Thus, `j` is `imaginary` as is `5e3*j`.
- `real` — Subtype of `complex` where the imaginary part is 0. Thus, `4`, `4.3e2`, and `-4.3e2` are all `real`.
- `posreal` — Subtype of `real` where all values are greater than 0. Thus, `4` and `4.3e2` are both `posreal`.
- `negreal` — Subtype of `real` where all values are less than 0. Thus, `-4` and `-4.3e2` are both `negreal`.
- `rational` — Subtype of `real` where values are fractions of integers. Formed by dividing one integer value by another. Thus, `5/4` is a `rational` constant.
- `integer` — Subtype of `rational` where values are integral numeric values. Formed when no decimal point or negative exponent are included in the number definition. Thus, `1`, `12`, `12e3` and `-12e3`are all `integer` constants.
- `natural` — Subtype of `integer` where all values are positive or zero. Thus, `0`, `5` and `12e3` are all `natural` constants.
- `posint` — Subtype of `natural` where all values are greater than zero. Thus, `5` and `12e3` are all `posint` constants.
- `negint` — Subtype of `integer` where all values are less than zero. Thus, `-5` and `-12e3` are `negint` constants.
- `bit` — Subtype of `natural` consisting of the values `0` and `1`. Operations on `bit` elements correspond to operations on the booleans in the canonical fashion.
- `boolean` — Subtype of `number` consisting of the named values `true` and `false`. `True` is the greatest `number` value while `false` is the smallest. The boolean operators `and`, `or` and `not` correspond to `min`, `max` and "`-`" respectively.
- `character` — Subtype of `element` consisting of all Unicode values. Formed using notation `'x'` where `x` is either a printable character, or a four digit, hexadecimal value. Thus, `'a'`, `'1'`, `'U+0024'` and `'U+EF37'` are `character` constants.
- `ascii` — Subtype of `character` consisting of all ascii values. Formed using the same notation as `character` values. Thus, `'a'` and `'1'` are `ascii` values.

2.5 Composite Types

Composite types make complex values by combining simpler values. There are two mechanisms for structuring: (i) containment and (ii) indexing. Containment groups items together into collections of items. Sets and sequences are both used as containers for multiple items of the same type. Sets provide a container for a specified type that is not indexed and does not contain duplicate items. Indexing establishes a function from the natural numbers (from zero to the size of the structure minus one) to the elements of the structure. Sequences effectively index sets allowing individual elements within the sequence to be accessed.

2.5.1 Sets

Rosetta sets are collections of items that exhibit properties traditionally defined in classical set theory. In the following, assume that `S` and `T` are sets and p is a predicate. The first table lists functions that form sets from items or other sets:

Operation	*Format*	*Definition*
Formation	`{1}, {1,2,3}`	*Forms a set from a collection of items*
Comprehension	`sel(x::T | p(x))`	$\{x \mid x \in T \text{ and } p(x)\}$
Filter	`filter(p,S)`	`sel(x::S | p(x)`
Union	`S+T`	$\{x \mid x \in T \vee x \in S\}$
Intersection	`S*T`	$\{x \mid x \in T \wedge x \in S\}$
Difference	`S-T`	$\{x \mid x \in S \wedge x \notin T\}$
Choose or eta	`choose(S)`	`choose(S) in S`
Power Set	`set(T)`	`(S in set(T)) == S=<T`
Integer Sequence	`{i,..j}`	`sel(x::integer | x >= i and x =< j)`
Image	`image(f,S)`	*f applied to each element of S*

The basic set former takes an arbitrary collection of items and forms a set by extension. Each argument to the set former is treated and evaluated as an expression. The `sel` operation provides a set comprehension capability where one set is filtered to form another. In the table, elements of `T` are filtered by the boolean predicate `p` to form a new set. The `filter` function is provided as a synonym for consistency with multisets and sequences. Operations for intersection, union, and difference are defined in the classical manner. The `set` function is equivalent to the set of all subsets of its argument and is typically used to define new set items. The sequence operation generates sets from sequences of integers. The notation `{1,..4}` generates the set `{1,2,3,4}`. The choose operation selects an arbitrary element of a set. The notation `choose({1,..4}` will return an arbitrary element of the set `{1,2,3,4}`. Finally, the `image` operation takes a function and applies it to all elements of a set. (`image` is synonymous with `ran` defined later.)

Classical relations between sets are defined and are listed in the following table:

Operation	*Format*	Definition
Equality	`S = T`	`S =< T and T =< S`
Inequality	`S /= T`	`not(S =< T) or not(T =< S)`
Subset	`S =< T, T >= S`	$\forall x : S \cdot x \in T$
Proper Subset	`S < T, T > S`	`S =< T and S /= T`
Element	`a in S`	$a \in S$
Size	`# S`	$\mid S \mid$
Empty Set	$\{\}$	$\forall x : universal \cdot not(x \in \{\})$

Equivalence is equivalence of contents. Subset and proper subset are defined in the classical manner. The `in` operation defines the set theoretic concept of "element of." Size returns the cardinality of the set while $\{\}$ names the empty set.

Defining items of a particular set type is achieved using the set type former or the power set former. The following notation defines `x` to be an element of the set of all possible subsets (the power set) of another set `S`:

```
x::set(S);
```

The declaration may intuitively be read as "`x` is an element of the power set `S`" or alternatively as "`x` is a subset of `S`." This is in contrast to the notation:

```
x::S;
```

that defines `x` to be a single element of the set `S`.

Like any Rosetta definition, it is possible to make a set valued item constant using an `is` clause to associate the item with a value. The following notation defines a set of integers that is equal to the set containing -1, 0 and 1:

```
trivalue::set(integer) is {−1, 0, 1};
```

Similarly, set comprehension can be used to define a set value:

```
natural::set(integer) is sel(x::integer | x >= 0);
```

In both cases, the type correctness restriction requires that the specified expression be an element of the type. In each of the above cases, the expressed value is a set of integers and is thus a legal value. The following expression:

```
trivalue::set(integer) is {−0.1, 0.0, 0.1}
```

Is not type correct because the specified set value is not a set of integers.

2.5.2 Multisets

Rosetta multisets are collections of items that exhibit properties traditionally defined for bags. In the following, assume that `M` and `N` are multisets. The first table lists functions that form multisets from items or other multisets:

Operation	*Format*	*Definition*
Formation	`{*1,1*}, {*1,1,2,3*},{*2:1,2,3*}`	*Forms a bag from a collection of items*
Union	`M+N`	$\{*n : x \mid n = count(x, N) + count(x, M)*\}$
Intersection	`M*N`	$\{*n : x \mid min(count(x, N), count(x, M)*\})$
Difference	`M-N`	$\{*n : x \mid n = count(x, M) - count(x, N)*\}$
Choose or eta	`choose(M)`	`choose(M) in M`
Integer Sequence	`{*i,..j*}`	`multiset_from({i,..j}`
Image	`image(f,M)`	*f applied to each element of M*
Number in	`count(x,M)`	Number of `x` instances in `M`
Multiset to Set	`~M`	Set of elements from `M`
Set to Multiset	`multiset_from(M)`	Multiset of elements from `M`

The basic multiset former takes an arbitrary collection of items and forms a bag by extension. The shorthand `n:i` includes `n` copies of `i` in the multiset. Each argument to the multiset former is treated and evaluated as an expression. Operations for intersection, union, and difference are defined in the classical manner. The `multiset` function is equivalent to the multiset of all sub-multisets of its argument and is typically used to define new multiset items. The sequence operation generates multisets from sequences of integers with one copy of each integer. The notation `{*1,..4*}` generates the multiset `{*1,2,3,4*}`. The choose operation selects an arbitrary element of a multiset. The notation `choose({*1,..4*}` will return an arbitrary element of the multiset `{*1,2,3,4*}`. The `image` operation takes a function and applies it to all elements of a multiset. The `count` function finds the number of occurrences of a value in a multiset. The ∼ and `multiset_from` operations convert to and from multisets and sets respectively. In the case of `multiset_from` one occurrence of each set element is present in the resulting multiset.

Classical relations between multisets are defined and are listed in the following table:

Operation	*Format*	Definition
Equality	`M = N`	`M =< N and N =< M`
Inequality	`M /= N`	`not(M =< N) or not(N =< M)`
Sub-multiset	`M =< N, N >= M`	$\forall x :\sim (M) \cdot count(x, M) =< count(x, M)$
Proper Sub-multiset	`M < N, N > M`	`M =< N and N /= M`
Element	`a in M`	$a \in M$
Size	`# M`	$\mid M \mid$
Empty multiset	`{**}`	$\forall x : universal \cdot count(x, \{\}) = 0))$

Equivalence is equivalence of contents. Sub-multiset and proper sub-multiset are defined in the classical manner. The `in` operation defines the concept of "element of." Size returns the cardinality of the multiset while `{**}` names the empty multiset.

Defining items of a particular multiset type is achieved using the multiset type former or the power multiset former. The following notation defines `x` to be an element of the set of all possible sub-multisets (the power multiset) of another multiset `S`:

```
x::multiset(S);
```

The declaration may intuitively be read as "`x` is an element of the power multiset of `S`."

Like any Rosetta definition, it is possible to make a multiset valued item constant using an `is` clause to associate the item with a value. The following notation defines a multiset of integers that is equal to the multiset containing two each of -1, 0 and 1:

```
trivalue::multiset(integer) is {*-1,-1,0,0,1,1*};
```

2.5.3 Sequences

Sequences are indexed collections of elements that combine the features of arrays and lists into a single, indexed container data structure. Sequences differ from sets in two important ways. First, they are indexed from 0 and allow random access of elements via their index. If `s=[1,2,1]` is a sequence, then `s(0)=1`, `s(1)=2` and so forth. Second, they allow multiple instances of the same value in the container. In the example `s=[1,2,1]` the value `1` appears in both the first and last position. The simplest sequence is `[]`, the empty sequence. If `S` and `T`, are sequences, `n` a natural number, `e` an element, and `I` a sequence of natural numbers, the following operations are defined to form sequences from items or other sequences:

Operation	*Format*	*Definition*
Formation	`[1,2,1,4]`	*Forms a sequence containing 1,2,1,4 in the specified order*
Subscription	`S sub I`	*Subsequence from S corresponding to integer sequence I*
Catenation	`S&T`	*Concatenation*
Integer Sequence	`[i,..j]`	*Sequence of integers from i to j*
Replacement	`n->e|S`	*Copy S with element n replaced by value e*
Empty Sequence	`[]`	*The empty sequence*
Head, tail and cons	`head(S),tail(S),cons(h,t)`	`cons(head(S),tail(S))==S`
Mapping	`map(f,S)`	`map(f,[s0,s1,...]==[f(s0),f(s1),...]`
Reduction	`reduce(f,S,i)`	`f(f(f(i,s(0)),s(1)),s(2))...`
Filtering	`filter(p,S)`	*Include only elements satisfying p*
Zipping	`zip(f,S,T)`	*Generate a sequence, R, where R(x)=f(S(x),T(x)*

The sequence former, `[]`, forms sequences by extension with ordering of elements in the sequence the same as the lexical ordering in the former.

Subscription is an extraction mechanism where elements from a sequence are extracted to form a new sequence. Given the sequence `S` and an integer sequence `I`, `S sub I` extracts the elements from `S` referenced by elements of `I` and forms a new sequence. For example:

```
[A,B,C,D] sub [0,2,1] == [A,C,B]
```

The catenation operator, `&`, concatenates two sequences.

The notation `[i,..j]` forms an integer sequence from `i` running to `j`. As an example, the functions `head`, `tail`, and `cons` can be defined using subscription and integer sequence as follows:

```
head(S) = S(0)
tail(S) = S sub [1,..(#S-1)]
cons(x,S) = [x]&S
```

`S(0)` returns the first element in the sequence. The integer sequence former `[1,..(#S-1)]` forms the integer sequence from 1 to the length of `S` minus `1`. Extracting elements of `S` associated with 1 through `#S-1` includes all elements except the first and thus defines tail in the canonical fashion.

The replace operation allows replacement of an element within a list. The notation `n->e | S` generates a new sequence with the element in position `n` replaced by `e`. For example:

```
2 -> 5 | [1,2,3,4,5] == [1,2,5,4,5]
```

The `map` and `filter` operators provide mechanisms for applying a function to each element of a sequence and filtering a sequence respectively. They correspond to `ran` and `sel` for functions and sets. The operation `map(f,S)` applies `f` to each element of `S` in order, generating a new sequence. Given the definition of an increment function:

```
inc(x::natural)::natural is x+1;
```

then:

```
map(inc,[0,1,2,3]) == [1,2,3,4]
```

`filter(p,S)` applies `p` to each element of `S` generating a new sequence of only those elements satisfying `p`. Given the definition of a greater than zero operation:

```
gtz(x::integer)::boolean is x>0;
```

then:

```
filter(gtz,[0,1,2,3]) == [1,2,3]
```

`reduce(f,S,i)` applies the binary function `f` recursively through the sequence `S`. The initialization value `i` is paired with `S(0)` to start the process. For example, the addition operator can be used to implement summation:

```
reduce(_+_,[1,2,3],0) == 6
```

The following operations define relationships between sequences and properties of sequences:

Operation	*Format*	*Definition*
Equality and inequality	`S = T, S /= T`	*Lexical equivalence*
Access	`S(n)`	*nth element of S from 0*
Ordering Relations	`S<T, S=<T, S>=T, S>T`	*Lexicographical ordering*
Size	`# S`	*Size*
Min and max	`S max T, S min T`	*Order defined on elements*
Contents	`~ S`	*Set of elements from the sequence*

Equal and not equal take their canonical meanings.

The ordering operations, `min` and `max` are lexicographic ordering relations. If `cons(x,S)<cons(y,T)`, then either `x<y` or `x=y` and `S<T`. Note that for any sequence, `S /= []` implies that `S > []`. `S=<T` is defined as `S<T or S=T`. The `S min T` and `S max T` operators return the minimum sequence of `S` and `T` and the maximum sequence respectively. Specifically, if `S=<T` then `S max T = T` and `S min T = S`.

The contents of a sequence can be extracted as a multiset using the notation `~ A`. For example:

`~[2,1,1,1] == {*1,2*}`

To define an item of type sequence containing only elements from type `B`, the following notation is used:

```
x::sequence(B);
```

To define an item of type `sequence`, the following notation is used:

```
x::sequence(universal);
```

where `x` is the new item and `sequence(universal)` refers to the set of all possible Rosetta sequences. Thus, `sequence(universal)` is any sequence while `sequence(B)` restricts possible sequences to the elements of `B`. Semantically, `sequence(B)` generates the set of all finite sequences created from `B` and thus the type containing all finite sequences of `B`.

Bitvectors

A special case of a sequence is the `bitvector` type. Formally, `bitvector` is defined as:

```
bitvector::type is sequence(bit);
```

The `sequence` operator generates set of all sequences containing `bit` values. Thus, the `bitvector` type contains all possible sequences of `bit`.

Operations over `bit` are generalized to `bitvector`'s of the same length by performing each operation on similarly indexed bits from the two bit vectors. Assuming that `op` ($\circ$) is any `bit` operation, the `bitvector`, `C`, result of applying the operation over arbitrary `bitvector` items `A` and `B` is defined by:

`forall(n::`$\{0,..(\#A-1)\}$ `| C(n) = A(n)` $\circ$ `B(n))`

If either `A` or `B` is longer, then the shorter bitvector is padded to the left with `0`s. to achieve the end result.

In addition, the following operations are defined over items of type `bitvector`: (Assume `A::bitvector` and `n::natural`)

Operation	*Format*	Definition
Bitwise Logic	`A or B,A and B, A xor B` `A nand B, A nor B, not A`	*Logical operators*
Conversion	`bv2n(A),n2bv(n)`	*Convert between bitvectors* and naturals
2's complement	`twos(A)`	*Generate two's complement*
Shift Operations	`ashr(A), ashl(A), lshl(A), lshr(A)`	*Logical and arithmetic shift*
Rotate	`rotr(A), rotl(A)`	*Rotate right and left*
Pad Operations	`padr(A,1,n), padl(A,0,n)`	*Pad with value to n bits.*

```
%% Add examples for the bitwise operations

%% Add examples for direct specification of bitvectors using strings
%% (or whatever we decide to use).
```

The operations `bv2n` and `n2bv` provide standard mechanisms for converting between binary and natural numbers. It is always true that `bv2n(n2bv(x))=x`.

The operation `twos` takes the two's complement of a binary value. The `lshr` and `lshl` operations provide logical shifts right and left while `ashr` and `ashl` provide arithmetic shifts right and left. The distinction being that logical shift operations shift in `0`s while arithmetic shift operations shift in `1`. The `rotr` and `rotl` operations provide rotation or circular shift. Note that none of the complement, shift, or rotate operations change the length of the bit vector.

The `padr` and `padl` operations pad or catenate a bit vector. Each takes three arguments: (i) the bitvector being manipulated; (ii) the pad value (`1` or `0`); and (iii) the resulting length. If the length value is less than the length of the argument vector, `padr` removes bits to the right and `padl` removes bits to the left resulting in a vector of length `n`. In this case, the pad value is ignored.

A special subtype of `bitvector` is defined to allow definition of bitvectors with specific lengths. The `wordtype` type former takes a single natural number argument and generates the type containing bitvectors of that length:

```
wordtype(n::integer)::set(bitvector) is sel(b::bitvector | #b = n);
```

The type definitions:

```
word::subtype(bitvector) is wordtype(16);
byte::subtype(bitvector) is wordtype(8);
nybble::subtype(bitvector) is wordtype(4);
```

defines new types called `word`, `nybble` and `byte` that consist of all bitvectors of lengths 16, 4 and 8 respectively. It should be noted that `wordtype` is simply a function that returns a set of bitvectors. Usage of functions in this way is defined in a subsequent chapter.

Strings

A special case of a sequence is the `string` type. Formally, `string` is defined as:

```
string::type is sequence(character);
```

The `sequence` operator generates set of all sequences containing `character` values. Thus, the `string` type contains all possible sequences of `character`.

A shorthand for forming strings is the classical notation embedding a sequence of characters in quotations. Specifically:

```
"ABcdEF" == ['A','B','c','d','E','F']
```

Functions defined over strings include those defined for general sequences. In particular, the notation `"abc" & "def"` is appropriate for concatenation of strings. It is important to note that the ordering operations for sequences provide lexicographical ordering for strings. No additional function definitions are required.

Summary: The following predefined composite types are available in a Rosetta specification:

- `set` — A set is a collection of items that obeys basic principles of set theory. Sets are formed by extension using the set former or by comprehension using `sel`. The notation `set(S)` refers to the power set of `S`. `subtype` is a synonym for `set` used in declarations. The notation `type` is a synonym for `set(universal)`, the power set of all possible Rosetta items.
- `multiset` — A multiset is a collection of items that allows multiple instances of the same element, but is not indexed. Multisets are formed by extension or comprehension using the `filter` function. The declaration `multiset(universal)` refers to any collection of Rosetta items.
- `sequence` — A `sequence` is an indexed collection of items. The notation `sequence(S)` refers to any sequence of Rosetta items formed from the elements of set `S`. Sequences are formed by extension using the sequence former or by filtering, mapping or folding sequences using `reduce` and `map`. The declaration `sequence(universal)` refers to any indexed collection of Rosetta items.
- `string` — A special sequence type defined as `string::type is sequence(character)`.
- `bitvector` — A special sequence type defined as `bitvector::type is sequence(bit)`. Operations from `bit` are defined as bitwise operations over `bitvector`.
- `wordtype(n::integer)` — The function `wordtype(n::natural)` is a special function that generates the set of bitvectors of length `n`.

2.6 Functions

Defining functions in Rosetta is a simple matter of defining mappings between types. Functions are extensive mappings between two different types, called the *domain* and *range* of the function. Functions are defined by defining their domain and stating an expression that transforms elements of the domain into elements of the range. This is achieved by introducing variables of the domain type whose scope is confined to the function definition and defining a result expression using that variable. The domain is given by an expression that describes the domain type. The range is given by an expression using the variable introduced by the function, called the result or result function.

2.6.1 Direct Definition

The direct definition mechanism for defining functions is to define the function's signature and an expression that relates domain values to a range value. Functions are typically defined by providing a signature and an optional expression. For example:

```
add(x,y::natural)::natural is x+y;
```

In this definition, `add` is the function name, `x,y::natural` defines the domain parameters, and `natural` is the return type. Together, these elements define the signature of `inc` as a function `add` that accepts two arguments of type `natural` and evaluates to a value of type `natural`.

Following the signature, the keyword `is` denotes the value of the return expression, in this case `x+y`. The return expression is a standard Rosetta expression defined over visible symbols whose type is the return type. Literally, what the function definition states is that anywhere in the defining scope `add(x,y)` can be replace by `x+y` for any arbitrary `natural` values. Whenever a function appears in an expression in a fully instantiated form, it can be replaced by the result of substituting formal parameters by actual parameters and evaluating the resulting expression. Specifically, if the function instantiation `add(3,4)` appears in an expression, it

can be replaced by the expression `3+4` and simplified to `7`. Any legal expression can be encapsulated into a function in this manner.

Like any Rosetta definition, `add` is an item with an associated type and value. In this case, the type of `add` is a function type defining a mapping from two naturals into the natural. The value of `add` is known and is a function encapsulating the expression `x+y`. The specific value resulting from function evaluation is determined by its associated expression.

The concrete syntax for definitions of this type has the form:

f`(`*params*`)::`*type* `is` *expression*`;`

where *f* is the function label, *params* is a list of declarations serving as parameters to the fucntion, *type* is the type associated with the return value, and *expression* is an expression defined over *params*.

A function signature can be defined separately by specifying its arguments and return type without an expression. The notation:

```
add(x,y::natural)::natural;
```

defines the signature of a function `add` that maps pairs of `natural` into `natural`. Because this `add` definition has no associated expression, it is referred to as a function signature. Allowing the definition of a signature without an associated expression supports flexibility in the definition style. In this case the expression associated with `add` can be defined directly using equality or indirectly by defining properties. Function signatures help dramatically in reducing over-specification in definitions.

2.6.2 Anonymous Functions and Function Types

Anonymous functions, frequently called lambda functions or function values, are defined by excluding the name and encapsulating the function definition in the function former `<* *>`. The definition:

```
<* (x,y::natural)::natural is x+y *>
```

defines an anonymous function identical to the function `add` above, except the anonymous function has no label. Such function definitions can be used as values and are evaluated in exactly the same manner as named functions. Specifically:

```
1. <* (x,y::natural)::natural is x+y *>(1,4)
2.    == <* ()::natural is 1+4 *> == 5
```

Anonymous function signatures can also be defined in a similar manner. The definition:

```
<* (x,y::natural)::natural *>
```

is equivalent to the function signature defined above without associating the signature with a name. We call this a function type because it defines a set of functions mapping pairs of natural numbers to natural numbers. Technically, it defines a set of functions that map two natural numbers to a third natural number. This is true because the function's expression is left unspecified. Because the definition represents a set of functions, it can be used as a type in formal definitions. The definition:

```
add::<* (x,y::natural)::natural *>
```

is semantically equivalent to the earlier `add` signature definition:

```
add(x,y::natural)::natural;
```

The definition says that `add` is a variable of type `<*(x,y::natural)::natural*>`. This implies that `add` is a function that maps two natural numbers to a third natural number. However, the actual mapping has not been specified. The earlier signature definition is a shorthand for this notation provided to make definitions easier to read and write.

The definition:

```
add::<* (x,y::natural)::natural *> is <* (x,y::natural)::natural is x+y *>
```

is equivalent to the first definition of `add` above and defines the semantics of the function definition shorthand. The `add` function is defined as an item of the function type. The `is` clause asserts that the `add` function is equivalent to the function value mapping two integers to their sum.

The notation `<* *>` is referred to as a *function former* because it encapsulates an expression with a collection of local symbols to form a function. The brackets form a scope for locally defined parameters. When no parameters are present, the function former brackets can be dropped as there is no need to define parameter scope.

Rosetta provides the special type `function` that contains all functions definable in a specification. Stating that `f::function` says that `f` is a function, but does not specify its `domain` or `range` values.

A function is an element of a function type if it is an element of the set of functions associated with the type. Given a function, `f`:

```
f(x::R)::T is <expression>;
```

and a function type `F`:

```
F::sybtype(function) is <*(x::M)::N *>;
```

then `f in F` if the following relationship holds:

```
f in F == R >= M and T =< N)
```

The domain of `F` must be a subset of the domain of `f` and applying `f` to each element of its domain must result in an element of `F`'s specified domain.

Because type memebership in Rosetta is set membership, if `f in F` is equivalent to `f::F` and `f` is of type `F`.

For example, given the standard definition of `inc`:

```
inc(x::integer)::integer is x+1;
```

`inc` is an element of the function type:

```
<* (x::integer)::integer *>
```

because it is one specific example of a function mapping `integer` to `integer`. Specifically, the domains of the function and function type are the same and applying `inc` to any integer results in an integer.

A function type is a subtype of another function type if a subset relationship holds between them. Specifically, given two function signatures defining function types:

```
F1::subtype(function) is <*(x::D1)::R1*>;
F2::subtype(function) is <*(x::D2)::R2*>;
```

Then `F1 =< F2` and `F1::subtype(F2)` if the following holds:

```
F1 =< F2 == D2 =< D1 and R1 =< R2;
```

The distinction between subtype and type inclusion is the return type of the function type is used rather than its range. This is because the largest possible range associated with an element of a type must be the return type associated with its function type.

For example, given the function types:

```
F1::subtype(function) is <*(x::real)::integer*>
F2::subtype(function) is <*(x::integer)::integer*>
```

we know that `F1 =< F2` because `integer =< real` and `integer =< integer` hold for the domains and return types respectively.

Functions defined over function types and anonymous functions include: (Assume that `f` and `g` are functions and `F` and `G` are a function types)

Operation	*Format*	Definition
Type Equivalence	`F=G, F/=G`	`F >= G and G >= G` `-(F >= G) or -(G >= F)`
Type Containment	`F=<G, F<G, F>G, F>=G`	Subset and proper subset relationships.
Containment	`f in F`	Type membership

Type containment is define as above. Proper containment, `F>G` occurs when `F>=G and -(F=G)`. Type equivalence, `F=G` occurs when the types have the same elements. A function is a member of a type, `f in F` if its domain is a subset of the type's domain and its range is a subset of the type's return type.

2.6.3 Property Based Definition

At times it is useful to define properties of a function without specifying an expression that evaluates to a function's value. For example, the definition of the `sqrt` function provided earlier simply specifies that `sqrt(x)` evaluates to a value that when squared is equal to `x`. Effectively:

```
forall(x::real | sqrt(x)*sqrt(x)=x);
```

Any function value that exhibits this behavior can be used the `sqrt` item's value. However, no expression can be given to evaluate because the result of evaluating `sqrt` is not fully constrained by the property. Specifically, both negative and positive roots satsify the requirement. Because it is undesirable to specify more behavior that necessary, this function is defined by specifying desired properties rather than equating it with an expression. Specifically, a function variable is defined and a term defined to constrain its value. The declaration appears in the declarative region while the property appears as a term in the term set of its enclosing facet. Specifically:

```
sqrt :: <*(x::real)::real*>;
```

or

```
sqrt(x::real)::real;
```

appears in the declarative region while:

```
t1: forall(x::real | f(x)*f(x)=x)
```

appears in the term declaration section.

Although this definitional form is effective, it is frequently desirable to colocate the constraint on `sqrt`'s value with the declaration of `sqrt`. This is particularly true in packages where no term declaration region is allowed. The following syntax provides a mechanism of combining the function declaration with its constraint:

```
sqrt(x::real)::real where
  forall(x::real | sqrt(x)*sqrt(x) = x);
```

This specification declares a function `sqrt` that is a mappiong from `real` to `real`. The `where` clause is much like the `is` clause in that it provides a constraint on the functions value. The distinction is that the `where` clause provides a general relation that must hold between `sqrt` and its return value while `is` provides an equivalence. Semantically, the `where` clause is identical to the combination of function variable and term discussed previously.

Function definitions of this kind have the following concrete syntax:

f`(`*params*`)::`*type* `where expression;`

where `expression` is a boolean valued expression defined over the function declaration, *f*. The semantics of this declaration are equivalent to declaring the function and adding the constraining predicate to the term set of the definitional construct.

It should be noted that any function definition using the `is` clause can be expressed using the `where` clause. For example, the definition of `inc` can be expressed as:

```
inc(x::natural)::posint where
  forall(x::natural | inc(x)=x+1);
```

The expression used to define the function's return value is equated with the function application. All such definitions using the `is` clasue can be defined using the `where` clause in this manner. However, the `is` clause should be used when possible. This definition follows directly from the the semantics of declarations using the `is` clause.

2.6.4 Function Evaluation

Evaluation of Rosetta functions follows the semantics of λ-calculus and allows for both currying and partial evaluation. All Rosetta functions are evaluated in a lazy fashion. Arguments can be instantiated and functions evaluated in any order. Consider the following use of `inc` and `add`:

```
inc(add(4,5))
```

Rather than evaluating in the traditional style, expand the function definitions Using the canonical definitions of `inc` and `add` results in the following anonymous function:

```
<* (z::natural)::natural is z+1 *>(<* (x,y::natural)::natural is x+y *>(4,5))
```

Instantiating the argument to what was the increment function results in the following definition:

```
<* ()::natural is <* (x,y::natural)::natural is x+y *>(4,5) + 1 *>
```

As the resulting function has no arguments, the outer function former can be dropped resulting in the new anonymous function:

```
<* (x,y::natural)::natural is x+y *>(4,5) + 1
```

Instantiating the `x` parameter of the new function by replacing the formal parameter with its associated actual parameter results in:

```
<* (y::natural)::natural is 4+y *>(5) + 1
```

Finally, instantiating the `y` parameter in the same manner results in:

```
<* ()::natural 4+5 *> + 1
```

When no arguments are defined in the scope of an anonymous function, the function former can be dropped resulting in the expected result:

```
4+5+1 == 10
```

In general, the notation `<* ()::T is e *>` is equivalent to simply stating `e`.

The same result occurs regardless of the order of instantiation. The following shows a different order resulting in the same result:

```
1.  <* (z::integer)::integer is z+1 *>(add(4,5))
2.     == <* ()::integer is add(4,5)+1 *>
3.     == <* ()::integer is <*(x,y::integer)::integer is x+y *>(4,5)+1 *>
4.     == <* ()::integer is <*(x::integer)::integer is x+5 *>(4) + 1 *>
5.     == <* (x::integer)::integer is x+5 *>(4) + 1
6.     == <* ()::integer is 4+5 *> + 1
7.     == 4+5+1
8.     == 10
```

2.6.5 Currying, Partial Evaluation, Function Composition and Selective Union

Thus far, all Rosetta functions have been defined by using an expression in the function definition. Rosetta provides three additional mechanisms for function construction: (i) curried functions and partial evaluation; (ii) function composition; and (iii) selective union. Partial evaluation generates new functions by substituting values for some parameters and simplifying. Function composition is simply an application of function definition techniques that allow a new function to be constructed from existing functions. Selective union allows functions to be specified by extension.

Currying Multi-Parameter Functions

Technically, evaluation of multi-parameter function is achieved by a process based on the concept of a curried function. This process provides the basis of the evaluation process used previously. All Rosetta functions can be expressed as functions of a single argument, or curried functions. Specifically, the function:

```
<* (x::R;y::S)::T is exp *>
```

can be expressed equivalently as:

```
<*(x::R)::<*(y::S)::T*> is <*(y::S)::T exp*>*>
```

The new function is expressed is unary function over items of type `R` that returns another unary function that maps items of type `S` to type `T`. Given a function `f(x::R; y::S)::T` and `r::R` and `s::S`, the following equivalence holds:

```
f(r,s) == f(r)(s)
```

Given the definition above, this equivalence generalizes to functions of arbitrarily many variables.

Consider again the definition of `add` defined over two integer numbers:

```
add(x,y::integer)::real is x+y;
```

Using the previous notation, `add` can be expressed in a curried fashion as:

```
add(x::integer)::<*(y::integer)::integer*> is
   <*(y::integer)::integer is x+y*>;
```

The use of `x` in the expression is perfectly legal as the second function definition is done in the scope of `x`.

The `add` function is now defined over a single parameter of type `integer`. Its return type is no longer an `integer` value, but a function that maps one `integer` onto another. Using this notation, adding two values `a` and `b` is achieved using `add(a)(b)` - exactly the notation discussed previously.

Now consider application of the `add` function using the curried function approach:

```
1. add(1,2)
2.  == add(1)(2)
3.  == <*(x::real)::<*(y::real)::real is x+y*>*>(1)(2)
4.  == <*(y::real)::real is 1+y*>(2)
5.  == <*()::real is 1+2*>
6.  == 3
```

The function is evaluated by substituting an actual parameter for a formal parameter in first the original add function and then in the unary function returned by add, exactly as it was done in the previous section.

It is particularly interesting to note the following equivalence:

```
1. add(1)
2.   == <*(x::real)::<*(y::real)::real is x+y*>*>(1)
3.   == <*(y::real)::real is 1+y*>
```

This process, called currying, defines the semantics of multi-parameter functions and is the basis for all function evaluation.

Partial Evaluation

Partial evaluation is the process of taking a function and instantiating only a subset of its parameters. The semantics of partial evaluation are defined using currying as described previously. Here, only the usage and application of partial evaluation are discussed.

Consider the following definition of `f` over real numbers:

```
f(x::real;y::real;z::real)::real is (x+y)/z;
```

Application of `f` follows the traditional rules of substituting actual parameters for formal parameters in the expression and substituting the expression for the function. Partial evaluation will perform the same function, but will not require instantiating all parameters. Consider a situation where `f` is applied knowing that in all cases the value of `z` will be fixed at 2 to perform an average. The following syntax partially evaluates `f` and assigns the resulting function to the new function name `avg`:

```
avg(x::real;y::real)::real;

avg=f(_,_,2);
```

In this definition, the "_" symbol is used as a placeholder for a parameter that will not be instantiated. To calculate the value of `f(_,_,2)`, we simply follow the instantiate and substitute rule:

```
f(_,_,2) == <*(x::real;y::real)::real is (x+y)/2*>;
```

The result is a 2-ary function that returns a real value. As noted, this function calculates the average of its two arguments. An alternate, more compact notation for the definition is:

```
avg::<*(x::real;y::real)::real*> is f(_,_,2);
```

This general approach is applicable to functions of arbitrarily many values.

Function Composition

Function composition is an application of function definition capabilities. Assume that two functions, `f` and `g` exist and that `ran(g)=<dom(f)`. We can define a new function `h` as the composition of `f` and `g` using the following definition style:

```
h(x::R)::T is (f && g)(x);
```

Alternatively, the same definition can be specified by defining a function item and assigning it a value directly:

```
h::<*(x::R)::T*> is (f && g);
```

The approach extends to other definition styles in addition to the direct definition style.

Consider the definition of `inc` and a function `sqr` defined as:

```
sqr(y::integer)::integer is y^2;
```

The definition of a function whose value is $(x+1)^2$ can be defined as:

```
<*(z::integer)::integer is (sqr && inc)(z)*>
```

Expanding the definitions of `sqr` and `inc` gives the following function:

```
<*(z::integer)::integer is
      <*(y::integer)::integer is
            y^2*>(<*(x::integer)::integer is x+1*>(z))*>
```

The only available simplification is to substitute `y`'s actual parameter in to the expression for `sqr` giving:

```
<*(z::integer)::integer is <* (<* (x::integer)::integer is x+1*>(z))^2 *> *>
```

Continuing to substitute, replacing formal parameters with actuals and eliminating function formers when parameters are replaced gives:

```
1. <*(z::integer)::integer is <* (<* z+1 *>)^2 *> *>
2.   == <*(z::integer)::integer is <* (z+1)^2 *> *>
3.   == <*(z::integer)::integer is (z+1)^2 *>
```

The result is a new function defined over `z` that gives the result of composing `inc` and `sqr`.

Selective Union

Selective union of two functions `f` and `g` is defined formally as:

```
(f||g)(x) = if x in dom(f)
             then f(x)
             else if x in dom(g)
                    then g(x)
                  end if;
           end if;
```

Using the `if` construct insures that the function associated with the first including domain will be called. In the above example, if `dom(f)=integer` and `dom(g)=real`, then an integer value will cause `f` to be selected while only a `real` value that is not an `integer` will cause `g` to be selected. If the domains are reverse, *i.e.* `dom(f)=real` and `dom(g)=integer`, `g` will never be selected because any element of `integer` is also in `real`. If the type of `x` is in none of the domains specified, then the result of evaluating the function is undefined.

The domain and range of `(f||g)` is defined as:

```
dom(f||g) == dom(f)+dom(g);
ran(f||g) == ran(f)+ran(g);
```

Selective union is highly useful for implementing a form of polymorphism. An example of a function defined by selective union is the simple `non_zero` function:

```
non_zero(n::number)::boolean is
  (<*(n::0) is true*> ||
   <*(n::sel(x::integer || x>0))::boolean is false*> ||
   <*(n::sel(x::integer || x<0))::boolean is false*>)
```

This is a rather pedestrian use of selective union and there are better definitions of the `non_zero` function. However, it does demonstrate how domain values can be used to select from among different function definitions.

2.6.6 Function Extension

Selective union provides a semantics for combining different function instances to form a single function. When these instances represent functions implementing the same general operation on different data types, a primitive form of overloading and polymorphism can be implemented. A special notation is provide to define a function that extends a named function already defined. Specifically:

```
overloading-function(parameters)::return-type
  extending overloaded-function is
  expression;
```

defines a new function, *overloading-function*, by taking the selective union of *overloaded-function* and the function:

```
<*(parameters)::return-type is expression*>
```

This mechanism is handy when defining a function that extends a function defined in an included package or in the containing scope. For example:

```
package example::logic is
  inc(x::real)::real is x+1.0;

  facet extension-example::logic is
    inc(x::character)::character extends example.inc is x;
  begin
    ...
  end facet extension-example;
```

In the facet `extension-example`, a new function `inc` is defined for characters that extends the increment function defined in the facet's scope. The new function, called `inc` in the facet, is defined as:

```
inc::<*(x::character+real)::character+real is
         <*(x::character)::character is x*> ||
         <*(x::real)::real is x+1*>;
```

The new function overloads the existing function because it comes first in the selective union. Within the facet `extension-example`, the function `inc` can be called legally with a parameter of either type `character` or `real` with the appropriate result. Note that the extension mechanism does not add semantics, only syntax to the existing definition.

It is also possible to define a function abstractly by excluding the expression. The following notation defines a new function that extends an existing function, but does not define the specifics of that extension. Such functions are useful when defining requirements where complete information is not available.

```
overloading-function(parameters)::return-type
  extending overloaded-function;
```

2.6.7 The If Expression

The Rosetta `if` expression is a polymorphic function that supports choice between options. The syntax of the `if` expression is:

```
if exp1 then exp2 else exp3 end if;
```

where `exp1` must be of type `boolean` while `exp2` and `exp3` may be of arbitrary types.

The rules for evaluating an `if` expression differ from the `if` statement in an imperative language. When `exp1` is `true`, the expression evaluates to `exp2`. When `exp1` is `false`, the expression evaluates to `exp3`. Specifically:

```
if true then a else b end if == a
if false then a else b end if == b
```

The `else` clause may be omitted:

```
if exp1 then exp2 else end if;
```

In this case, the `if` expression evaluates to *exp2* if `exp1` is true and is undefined otherwise. Specifically:

```
if true then a end if == a
if false then a end if == undefined
```

The domain of an `if` expression is simply `boolean` while the ranges is the union of the types of *exp2* and *exp3*.

For convenience, an `elsif` construct is provided to nest `if` statements. The notation:

```
if exp1 then exp2
  elsif exp3 then exp4
  elsif exp5 then exp6
  ...
  else expn
end if
```

is semantically equivalent to:

```
if exp1 then exp2
   else if exp3 then exp4
           else if exp5 then exp6
                ...
                else expn end if
        end if
end if
```

2.6.8 The Case Expression

The `case` expression supports selection from multiple options in a manner similar to using the `if` construct with the `elsif` extension.

The general form of a case statement is:

```
case exp0 is
  s1 -> exp1 |
  s2 -> exp2 |
  s3 -> exp3 |
  ...
  sn -> exprn
end case;
```

where *s1-sn* are sets and *exp0-expn* are expressions. The `case` statement evaluates to *expk* when `exp0 in sk` holds. If this relationship is satisfied by multiple sets, the `case` expression evaluates to the expression associated with the first such set. A default case can be achieved using `universal` as the set expression. The equivalence check performed in most traditional languages is performed by using singleton sets. For example:

```
case x is
  sel(x::integer | x > 0) -> false |
  {0} -> true |
  sel(x::integer | x < 0) -> false
end case;
```

implements a zero test on the `x`.

Please note that the case statement is semantically equivalent to application of a unary function defined using selective union. Specifically:

```
(<*(x::sel(i::integer | x > 0))::boolean is false *> |
 <*(x::0)::boolean is true *> |
 <*(x::sel(i::integer | x < 0))::boolean is false *> )
```

is identical to the previous `case` statement.

2.6.9 The Let Expression

Function application provides Rosetta with a mechanism for defining expressions over locally defined variables. An additional language construct, the `let` expression generalizes this providing a general purpose `let` construct. The Rosetta `let` is much like a Lisp `let` in that it allows definition of local variables with assigned expressions. The general form of a let expression is:

```
let (x::T be ex1) in ex2 end let;
```

This expression defines a local variable `x` of type `T` and associates expression *ex1* with it. The expression *ex2* is an arbitrary expression that references the variable `x`. Each reference to `x` is replaced by *ex1* in the expression when evaluated.

The `be` keyword is semantically similar to `is`. The distinction is that `is` clauses are evaluated when their associated variable declarations are evaluated. `be` clauses are evaluated when the `let` form is evaluated.

Unlike traditional variable declarations, `let` parameter definitions may not omit the `be` clause. All `let` parameters must have values when the `let` form is interpreted.

The syntax of the `let` expression is defined by transforming the expression into a function application. Specifically, the semantic equivalent of the previous `let` expression is:

```
<*(x::T)::universal is ex2*>(ex1)
```

When the function is applied, all occurrences of `x` in `ex2` are replaced by *ex1*. This process is identical to the application of any arbitrary function to an expression. Assume the declaration `i::integer` and consider the following `let` expression:

```
let (x::integer be i+1) in i'=x end let;
```

The semantics of this `let` expression is:

```
<*(x::integer)::universal is i'=x*>(i+1)
```

Evaluation of the function application gives:

```
i'=i+1
```

The usefulness of `let` becomes apparent when an expression is used repeatedly in a specification. Consider a facet with many terms that reference the same expression. The `let` construct dramatically simplifies such a specification.

`Let` expressions may be nested in the traditional fashion. In the following specification, the variable `x` of type `T` has the associated expression *ex1* while `y` of type `R` has the expression *ex2*. Both may be referenced in the expression *ex3*.

```
let (x::T be ex1) in
   let (y::R be ex2) in ex3 end let
end let;
```

This expression may also be written as:

```
let (x::T be ex1, y::R be ex2) in ex3 end let;
```

The semantics of this definition are obtained by applying the previously defined semantics of `let`:

```
<* (x::T)::universal is <*(y::R)::universal is ex3 *>*>(ex1)(ex2)
```

Consider the following specification assuming that `i` is of type integer and `fcn` is a two argument function that returns an integer:

```
let (x::integer be i+1, y::integer be i+2) in i'=fnc(x,y) end let;
```

When evaluated, the following function results:

```
<*(x::integer)::universal is <*(y::integer)::universal is i'=fnc(x,y)*>*>(i+1)(i+2)
```

The result of evaluating this function is:

```
<*(x::integer)::universal is <*y::integer is i'=fnc(x,y)*>*>(i+1)(i+2) ==
<*(y::integer)::universal is i'=fnc(i+1,y)*>(i+2) ==
i'=fnc(i+1,i+2)
```

In order for normal argument substitution to work, the expressions in the Rosetta `let` expression must not be mutually recursive. If recursion is necessary, the expressions must be represented as normal Rosetta rules or predicates.

Summary: A Rosetta function is defined by specifying a domain, range and an expression defining a relationship between domain and range elements. The notation:

```
f(d::domain)::range is exp ;
```

Defines a function mapping `domain` to `range` where `exp` is an expression defined over domain parameters and gives a value for the associated range element. The notation:

```
f(d::domain)::range ;
```

defines `f` as an element of the set of all functions relating *domain* to *range* without specifying the precise mapping function.

As an example, the increment function is defined over naturals using the notation:

```
inc(x::natural)::natural is x+1;
```

Applying a function is simply substitution of an actual parameter for a formal parameter. Evaluating `inc(2)` involves replacing `x` with `2` and applying the definition of a function. Specifically:

```
inc(2) = 2+1 =
inc(2) = 3
```

Function types are specified as anonymous functions using the notation:

```
<*(d::domain)::range*>
```

This function type specifies the set of all functions mapping *domain* to *range*. The definition:

```
f::<*(d::domain)::range*>
```

says that `f` is a function that maps *domain* to *range*. The function former (`<* *>`) defines the scope of the named parameter `x` and forms a function constant in the same manner as {} forms a set and `[]` forms a sequence.

An anonymous function is a function having no assigned label. It is treated like a lambda function in Lisp programming languages in that it can be evaluated like any other function, but has no name by which to reference it.

```
<*(d::domain)::range is exp*>
```

This anonymous function specifies the function mapping *domain* to *range* using the expression *exp*. It is semantically the same as `f(d::domain)::range`. The definition:

```
f::<*(d::domain)::range is exp*>
```

says that `f` is a function that maps *domain* to *range* using the expression `exp`. It is semantically the same as `f(d::domain)::range is exp`.

The `let` expression provides a mechanism for defining local variables and assigning expressions to them. This provides shorthand notations that can dramatically simplify complex specifications by reusing specification fragments. The syntax of the general `let` expression is:

```
let (v1::T1 be e1, v2::T2 be e2, ..., vn::Tn be en) in exp end let;
```

where `v1` through `vn` define variables, `T1` through `Tn` define the types associated with each variable, and `e1` through `en` define expressions for each variable.

Evaluating the `let` expression results in the expression *exp* with each variable replaced by its associated expression. The semantics of the let expression are defined using function semantics. It is sufficient to realize that the `let` provides local definitions for expressions.

The `if` expression provides a simple mechanism for expressing choice. The general form:

```
if expression then cond1 else cond2 end if;
```

evaluates to *cond1* if *expression* evaluates to `true` and *cond2* if *expression* evaluates to `false`.

The `case` expression provides a general purpose selection method used to choose from more that two, potentially non-exclusive options. The form of the case statement is:

The general form of a case statement is:

```
case exp0 is
  s1 -> exp1 |
  s2 -> exp2 |
  s3 -> exp3 |
  ...
  sn -> expn |
  otherwise -> exp
end case;
```

where exp_k are expressions and s_k are sets or expressions that result in sets. The `case` expression evaluates to the first exp_k such that exp_0 in s_k. The term `otherwise` is a synonym for `universal` and provides a default case. If the match condition holds for none of the `case` terms and an `otherwise` term is not included, the `case` statement is undefined.

2.7 Set Construction and Quantification

In Rosetta, all quantifiers are functions defined over other functions. A number of second order functions such as `min` and `max` are defined and will be presented here. Given that `F(x::universal)::universal` and `P(x::universal)::boolean`, the following quantifier and set constructor functions are defined:

Operation	*Format*	*Definition*
Function Former	`<* (rank)::return is exp *>`	*Forms a function value or type.*
Domain	`dom(F)`	*Function's actual domain*
Range	`ran(F)`	*Function's Range or Image*
Maximum and Minimum	`max(F),min(F)`	*Maximum and minimum value in range*
Selection	`sel(P)`	*Set Comprehension*
Exists	`exists(P)`	*Existential quantifier*
Forall	`forall(P)`	*Universal quantifier*

The signature for the `min` function is:

```
min(f::<*(x::universal)::universal*>)::universal
```

The `min` function accepts an arbitrary function and returns the minimum value associated with the range of the argument function. Recall that the range of the argument function is the result expression applied to each element of the domain. Consider the following function application:

```
min(<*(x::{1,2,3,4})::natural is x*2 *>)
```

The domain of the argument is the set {`1,2,3,4`}. Although it is unusual to define a set by extension in these circumstances, it is perfectly legal. The range of the argument function is the expression applied to each element of the domain. Specifically, {`2,4,6,8`}. The `min` function then returns the minimum value in {`2,4,6,8`} or `2`.

If the unaltered minimum value associated with the input set is desired, the `min` function can be applied using an identity function as in:

```
min(<*(x::{1,2,3,5})::natural is x *>)
```

The `max` function is defined similarly and operates in the same manner.

2.7.1 Domain and Range

The `ran` function takes a function and returns mathematical range of the function. Itis similar to a set mapping function and returns the image of a function with respect to its domain. It returns the set resulting from applying the parameter function's expression to each element of the domain.

Given the function value definition:

$$<*(\texttt{x} :: \tau_1)::\tau_2 \texttt{ is x} + 1*>$$

the definition of consistency in Chapter 5 konw that $ran(<*(\texttt{x} :: \tau_1)::\tau_2 \texttt{ is x} + 1*>) =< \tau_2$. However, it is not necessary for the range of a function to be equal to the return type specified in its definition. Consider the following example where `ran` is used to find the range of a simple increment function:

```
ran(<*(x::natural)::natural is x+1*>) == posint
```

`posnat` is precisely the application of `x+1` to each element of the domain set and does not include `0`. `posnat =< natural`, thus the definition is consistent, but the range is not equal to the declared return type.

The `dom` function is defined similarly to the range function but instead returns the domain associated with its function argument. For example:

```
dom(<*(x::T)::natural is x+1*>)
```

evaluates to the set `T`.

The domain and range functions present a greater challenge when dealing with functions of arity other than 1. The domain of a nullary function is defined as the empty type, {} while the range of a nullary function is the result of its evaluation:

```
dom(<* ()::natural is 3+2 *>) == {}
ran(<* ()::natural is 3+2 *>) == {5}
```

Using this identity, one can define evaluation of a fully instantiated function as taking the range of that function. Specifically, if all arguments to a function are known, then the range of that instantiated function is the same as evaluating the function.

Currying is applied when looking at the domain and range of functions with arity greater than one. Recall that any n-ary function can be treated as application of a series of unary functions. Thus, the domain of functions with arity greater than one is defined as the type of the first parameter. Thus, for the `add` function defined by:

```
add(x,y::natural)::natural is x+y;
```

The value of `dom(add)` is defined as:

```
dom(add) == natural;
```

or the set of values the curried form of `add` can be applied to.

The range of such a function is the set of functions that result from currying over all possible domain values. Literally, it is the set of values, albeit function values, that result from applying the curried function form to the domain values. Thus for the `add` function:

```
ran(add) == image(add,natural);
```

the range is defined as the set of functions that result from applying `add` to every domain element. The elements of the resulting set are functions of the form:

```
<* (x::natural)::natural is x+n *>
```

where `n` is any `natural`. Effectively, this is the set of fuctions:

```
{<*(x::natural)::natural is x+0*>,
  <*(x::natural)::natural is x+1*>,
  <*(x::natural)::natural is x+2*>,
  ...}
```

2.7.2 Quantifiers

As previously defined, the functions `min` and `max` provide minimum and maximum functions over function ranges. Over boolean valued functions, `min` and `max` provide quantification functions `forall` and `exists`. As noted earlier, `and` and `or` correspond to the binary relations `min` and `max` respectively. As `forall` and `exists` are commonly viewed as general purpose `and` and `or` operations, `forall` and `exists` should correspond to `min` and `max`.

Consider the following application of `forall` to determine if a set, `S`, contains only integers greater than zero:

```
forall(<* (x::S)::boolean is x>0 *>)
```

Here, the domain of the argument function is the set `S` and the result expression `x>0`. To determine the range of the argument function, `x>0` is applied to each element of `S`. Assume that `S={1,2,3}`. Substituting into the above expression results in:

```
forall(<*(x::{1,2,3})::boolean is x>0 *>)
```

Applying the result expression to each element of the domain, the range of the function becomes:

$\{true, true, true\}$ == $\{true\}$

As `true` is greater or equal to all boolean values, the minimum resulting value is `true` as expected. Assuming `S={-1,0,1}` demonstrates the opposite effect. Here, the range of the internal function becomes:

$\{false, false, true\}$ == $\{false, true\}$

As `false` is less than `true`, the minimum resulting value is `false`. Again, this is as expected.

2.7.3 Selection Operations

The function `sel` provides a comprehension operator over boolean functions. The signature for `sel` is defined as follows:

```
sel(<*(x::universal)::boolean*> is set(universal)*>)
```

Like `min` and `max`, `sel` observes the range of the input function. However, instead of returning a single value, `sel` returns a set of values from the domain that satisfy the result expression. Consider the following example where `sel` is used to filter out all elements of `S` that are not greater than `0`:

```
sel(<*(x::S)::boolean is x>0*>)
```

Assuming `S={1,2,3}`, `x>0` is true for each element. Thus, the above application of comprehension returns {`1,2,3`}. If `S={-1,0,1}` then `x>0` holds only for `1` and the instance of `sel` evaluates to {`1`}.

2.7.4 Shorthand Notation

A shorthand notation is provided to make specifying `forall`, `exists`, `sel`, `min` and `max` expressions simpler. Notationally, the following statement:

```
forall(x::S | x>0);
```

is equivalent to:

```
forall(<*(x::S)::boolean is x>0*>);
```

and returns `true` if every `x` selected from `S` is greater than `0`. The notation allows specification of the domain on the left side of the bar and the expression on the right. The domain of the expression is assumed to be boolean for `forall`, `exits`, and `sel`. For `min` and `max`, the domain is taken from the expression. This notation is substantially clearer and easier to read than the pure functional notation. Note that the original notation is still valid for specifying quantified functions.

The notation extends to n-ary functions by allowing parameter lists to appear before the "|" to represent parameter lists. The format of these lists is identical to the format of function parameter lists. Specific examples include:

```
forall(x,y::integer | x+y>0)
exists(x,y::integer | x+y>0)
sel(x,y::integer | x+y>0)
```

It is important to remember that like `forall` and `exists`, `sel` observes the function range and selects appropriately. Interpreting the notation in the standard way results in the definitions:

```
forall(<* (x,y:integer)::boolean is x+y>0 *>)
exists(<* (x,y:integer)::boolean is x+y>0 *>)
sel(<* (x,y:integer)::boolean is x+y>0 *>)

%% Need to update summary section
```

Summary: Quantifier functions operate on other functions. Each generates the range of their function argument and returns a specific value associated with that range. `min` and `max` return the minimum and maximum range values respectively and are synonymous with `forall` and `exists`. `sel` and `ran` provide comprehension and image functions respectively. `sel` applies a specific boolean expression to a function's range and returns a set of domain elements satisfying the expression. `dom` returns the domain of a function defined as the application of the result expression to every domain element.

2.7.5 Function Containment

Function containment, `f1 =< f2`, holds when a function is fully contained in another function. Assuming `f1(x::d1)::r1` and `f2(x::d2)::r2` where `d1`, `d2`, `r1` and `r2` are types representing domain and range respectively:

```
f1 =< f2 == d1 =< d2 and forall(x::d1 | f1(x) = f2(x))
```

`f1` is contained in `f2` if and only if the domain of `f1` is contained in the domain of `f2` and for every element of `f1`'s domain, `f1(x)` is equal to `f2(x)`.

Consider the case of determining if increment is contained in identity over natural numbers. In this case, the law should not hold:

```
1. inc =< id
2.   == <*(x::natural)::natural is x+1*> =< <*(x::natural)::natural is x*>
3.   == dom(inc) >= dom(id) and forall(x::natural | inc(x) = id(x))
4.   == natural in natural and forall(x::natural | x+1 = x)
5.   == true               and false
```

`false` is obtained from the second expression by the counter example provided by `x=0` as `0+1 /= 0`.

Assume that `f(x::df)::rf` and `g(x::dg)::rg` are functions. The following operations are defined over two functions:

Operation	*Format*	*Definition*
Equivalence	`f=g, f/=g`	`f=<g and g=<f, -(f=<g) or -(g=<f)`
Containment	`f=<g, g>=f`	`dom(f)=<dom(g) and forall(x::dom(f) | f(x)=g(x)`
Proper Containment	`f<g, g>f`	`f >= g and f /= g`

Functional equivalence checks to determine if every application of `f` and `g` to elements from the union of their domains results in the same value. Specifically, `f(x) = g(x)` for every `x` in either domain. Function inequality is defined as the negation of function equality.

Function containment, `f=<g`, occurs when `dom(f)<dom(g)` and `forall(x::dom(f) | f(x) = g(x)`. Proper containment occur when simple containment holds and the functions are not equal.

```
%% Working here...

%% Consider moving this section.  Technically, these are second
%% order funtions, but they are specific to real and complex valued
%% fuctions.  Don't know where to move them though...
```

2.7.6 Limits, Derivatives and Integrals

A special class of functions for defining limits, derivatives and integrals are provided for use with real valued functions. These functions exist to allow specification of differential equations (both ordinary and partial) over real valued functions. Given a real valued function `f(x::real)::real`, the following definitions are provided:

```
%% Need to add step and delta dirac functions.
```

Operation	*Format*	*Definition*
Limit	`lim(f,x,n)`	$\lim_{x \to n} f(x)$
Derivative	`deriv(f,x)`	$\frac{df}{dx}$
Indefinite Integral	`antideriv(f,x,c)`	$\int f(x)dx + c$
Definite Integral	`integ(f,x,u,l)`	$\int_l^u f(x)dx$

Note that although the limit, derivative, antiderivative and integral functions are well-formed for any real function, a solution may not exist.

The derivative of a function is defined using limit in the canonical fashion. The following axiom is defined for all real valued functions and real valued nonzero `delta`:

```
deriv(f,x) = lim((f(x+delta)-f(x))/(x+delta)-x,delta,0)
```

In the derivative function, `f` is the object function and `x` is the label of the parameter subject to the derivative. In the above function, the following holds:

`deriv(f,x)` = $\frac{df}{dx}$

The derivative function is generalizable to expressing partial derivatives. Assuming that `g` is defined over multiple parameters, such as `g(x::real;y::real;z::real)::real`, then:

`deriv(g,x)` = $\frac{\partial g}{\partial x}$

Antiderivative, or indefinite integral, is the inverse of derivative. The antiderivative of `f` with respect to `x` is expressed as:

`antideriv(f,x,c)` = $\int f(x)dx + c$

`f` being the function in question, `x` being the variable integrated over, and `c` being the constant of integration.

As antiderivative is the dual of derivative, the following axiom is defined for all real valued functions:

```
antideriv(deriv(f,x),x,0) == deriv(antideriv(f,x,0),x) == f
```

The definite integral of `f` with respect to `x` over the range `u` to `l` is expressed as:

`integ(f,x,l,u)` $== \int_l^u f(x)dx$

The definite integral is defined as the difference of the indefinite integral applied at the upper and lower bounds:

```
integ(f,x,l,u) == antideriv(f,x,0)(u) - antideriv(f,x,0)(l)
```

It is possible to express a definite integral over an infinite range using the notation:

`integ(f,x,false,true)` $= \int_{-\infty}^{\infty} f(x)dx$

It should be noted that limit, derivative, antiderivative and integral functions are defined over real valued functions only. Further, the functions provide a mechanism for expressing these operations and some semantic basis for them. Solution mechanisms are not provided.

2.8 Universal Type

The type `universal` is now introduced as the supertype of all Rosetta types. This includes `element`, `set`, `sequence`, constructed types, `function` and facet types. Declaring:

```
x :: universal;
```

results in an item `x` that can literally contain any Rosetta value. Declaring:

```
f(x::universal)::universal;
```

results in a function `f` that can accept any Rosetta value and may result in any Rosetta value. Declaring:

```
t :: subtype(universal);
```

results in a type variable, `t`, whose value may be any set of Rosetta values.

2.9 User Defined Types

2.9.1 Sets and Types

As noted earlier, all Rosetta types are sets and any Rosetta set can be used as a type. To support clarity in specifications, several notational shorthands are provided to support defining types and subtypes. The item declaration notation:

```
i::integer;
```

defines an item named `i` whose value is restricted to single elements from the set `integer`. When `integer` is viewed as a set, this restriction can be represented as:

```
i in integer;
```

Similarly, the notation:

```
natural::set(integer);
```

implies that `natural` is a set of elements from `integer`. This constraint can be expressed using `in` by equating `set(integer)` power set of integers:

```
natural in set(integer);
```

the set `natural` is contained in the power set of integers and is a subset of `integer`. The value of `natural` can be restricted to a single element of the power set that appropriately defines naturals using the notation:

```
natural::set(integer) is sel(x::integer | x >= 0);
```

Now the set `natural` is constrained to be equal to the set of elements from `integer` that are greater than or equal to `0`. Because the expression defines a set value by comprehension over `integer`, we know that the expression is contained in `set(integer)`.

In Rosetta, sets are first class items and any set can be used as a type. Thus, `natural` from the previous declaration can be used as a type in subsequent declarations. Thus, the declaration:

```
n::natural;
```

declares a new item named `n` that is an element of the set `natural` used as a type.

Subtype and Type

In Rosetta, one type is a subtype of another if all its elements are contained in the second type. Specifically, `S` is a subtype of `T` if `S=<T` holds. When defining a new set using the notation:

```
natural::set(integer);
```

it is known that `natural` is a subset of `integer` and thus that `natural` is a subtype of `integer`. Thus, Rosetta provides an operation, `subtype` that is semantically equivalent to `set`:

```
natural::subtype(integer);
```

The subtype notation is equivalent to the `set` notation. Both define a new set that includes possible subsets of `integer`. The `subtype` notation is simply syntactic sugar that provides a mechanism that a set will be used as a type.

It is also possible to define a new type that is not explicitly defined as a subtype of any existing type. Using the set notation, such a type is defined as:

```
T::set(universal);
```

or alternatively using `subtype`:

```
T::subtype(universal);
```

Both notations define a new set, `T`, whose elements are simply Rosetta items. No other type restriction is made. Such sets are frequently used when defining abstract types whose construction is not specified or known. Thus, Rosetta provides a keyword `type` that is equivalent to the definition `subtype(universal)`. Specifically:

```
T::type;
```

is equivalent to the previous notations and defines a new type that has no subtype relationships with other types.

Both `subtype` and `type` definitions can be used to define constants in a manner identical to that for any other Rosetta item. The `is` clause is included to provide a constant value for the symbol. The type `natural` is defined using this technique:

```
natural :: subtype(integer) is sel(n::integer | n >= 0);
```

The following example defines a type that includes sets of exactly four integers:

```
set4 :: subtype(set(integer)) is sel(x::set(integer | #x=4));
```

where `set4` is the set of subsets of `integer` that contain exactly four elements. In this case, `set4` is a set of integer sets, not simply a set of integers. Thus, the `set` operation is used to generate the power set and the new type `set4` is chosen from the power set of the power set. In other words, it is itself a set of integer sets. The `sel` operation uses the cardinality operator to choose integer subsets that contain 4 element sets only. The notation `z::set4` declares `z` to be an element of `set4`, or simply a subset of `integer` containing four elements.

The `type` notation can be used similarly to define the natural numbers:

```
natural::type is sel(x::integer | x >= 0)
```

Using this definition, `natural` is still a subset of `integer` and is thus a subtype of integer. However, this information must be inferred rather than directly found in the definition. The `type` declaration should be used carefully and only when defining or types not defined by filtering existing sets. If a subtype relationship exists, then it should be specified explicitly in the definition. Even if the type's value is not known, expressing a subtype relationship in the type definition aids automated analysis and readability.

In addition to constructing new types comprised of elements, the `subtype` construct can be used to define types comprised of composite values. The following definition:

```
bv::subtype(sequence(bit));
```

defines a new type named `bv` that is comprised of bitvectors. Similarly, it is possible to define types containing sets and constructed types.

2.9.2 Parameterized Type Formers

Any function returning a set can be used to define a Rosetta parameterized type. Consider the following function definition:

```
word(n::natural)::subtype(bitvector) is
  sel(b::bitvector | $b = n);
```

Remembering that `subtype` is a synonym for set, the function signature defines a mapping from natural numbers to a set of bitvectors. That set of bitvectors is defined by the `sel` operations to be those whose lengths are equal to the parameter `n`. Thus, `word` will return the set of bitvectors of length equal to its parameter. We can now use `wordtype` as as a type definition construct.

The notation:

```
reg::wordtype(8);
```

defines `reg` to be a bitvector of length 8.

The notation:

```
bv8::subtype(bitvector) is word(8);
```

defines `bv8` to be the set of all bitvectors of length 8.

```
%% Summary section needs to be updated.
```

Summary: User defined types are declared exactly as are other Rosetta variables and constants. While the notation `x::T` forces `x` to be a singleton element of `T`, the notation `x::subtype(T)` allows `x` to be a subset for `T`. Types can be formed from any element or composite type.

Uninterpreted types are defined as subtypes of the `universal` type.

Parameterized types are defined by using functions to return set as types.

2.10 Constructed Types

```
%% Working to determine if the type parameters from the function
%% should also be expressed as parameters to the data declaration.
```

2.10.1 Defining Constructed Types

Rosetta provides a general shorthand for defining types in a constructive fashion. Constructor, observer and recognizer functions are defined for the type and encapsulated in a single notation. These types are called *constructed types* and are created with the special `data` keyword and notation. As an example, consider a definition for a binary tree of integers:

```
Tree(a::type) :: type is data(a::type)
  empty::nullp |
  node(L::Tree(a),v::a,R::Tree(a))::nodep;
```

This definition provides two constructors for `Tree`: (i) the nullary function `empty`; and (ii) the ternary function `node`. The `empty` function creates an empty tree while the `node` function creates a node from a value and a left and right subtree. A tree of integers can be defined as:

```
IntTree :: Tree(integer);
```

A tree with one node whose value is `0` can be generated with the following function instantiation:

```
node(empty,0,empty);
```

A balanced tree with `0` as the root and `1` and `2` as the left and right nodes respectively can be generated:

```
node(node(empty,1,empty),0,node(empty,2,empty));
```

The recognizers `nullp` and `nodep` indicate the constructor used to generate a tree. Specifically, `nullp` is true if its argument is `empty` and `nodep` is true if its argument is an instantiation of the node function. Semantically, these functions are defined as follows:

```
nullp(x::tree(integer))::boolean is x=empty;

nodep(x::tree(integer))::boolean is
  exists(lt::tree(integer), v::integer, rt::tree(integer) | node(lt,v,rt)=x);
```

Finally, parameter names are used to generate observer functions that return actual parameters from constructor functions. Specifically, the following functions are generated from the integer tree definition:

```
lt(t::sel(x::tree(integer) | nodep(x)))::tree(integer)
rt(t::sel(x::tree(integer) | nodep(x)))::tree(integer)
v(t::sel(x::tree(integer) | nodep(x)))::integer
```

These functions return the actual parameter instantiation of their associated formal parameter. For example:

```
lt(node(empty,1,node(empty,2,empty))) == empty
v(node(empty,1,node(empty,2,empty))) == 1
v(rt(node(empty,1,node(empty,2,empty)))) = 2
```

The syntax for creating unparameterized constructed type definitions is:

```
T :: type is data
  f1(b11::T11,b12::T12 ... b1i::T1i)::r1 |
  f2(b21::T21,b22::T22 ... b2j::T2j)::r2 |
  ...
  fn(bn1::Tn1,bn2::Tn2 ... bnm::Tnm)::rn;
```

This data type definition defines n functions that create and recognize all elements of type `T`. Instantiating any of the f_k functions creates an element of type `T`. This set of functions are referred to as *constructors* of the type `T`.

Associated with each constructor, f_k, is a boolean *recognizer* function r_k that is true when its argument was created with the associated constructor function. Specifically, r_k will return true when its argument was created using f_k:

```
r(k)(t1, t2, t3,..., ti)==true;
```

Associated with each constructor function parameter is an *observer* function of the same name that observes the parameter. Given an instantiated constructor function, the observer associated with a parameter will return the actual parameter instantiating it:

```
b(k)(t1, t2, t3,..., ti)==tk;
```

Like any other function, constructor functions can be partially evaluated. If this is the case, then the results of applying observer functions associated with uninstantiated parameters are not defined.

The general expression above is equivalent to the following definitions and laws (where the definitions are in the definition section and the laws in the predicate section):

```
  T :: type;
  f1(b11::T11, b12::T12 ... b1i::T1i)::T;
  f2(b21::T21, b22::T22 ... b2j::T2j)::T;
  ...
  fn(bn1::Tn1, bn2::Tn2 ... bni::Tni)::T;

 r1(t::T)::boolean is exists(b11::T11, b12::T12 ... b1i::T1i |
                             f1(b11, b12 ... b1i)=t)
  r2(t::T)::boolean is exists(b21::T21, b22::T22 ... b2j::T2j |
                             f2(b21, b22 ... b2j)=t)
  ...
  rn(t::T)::boolean is exists(bn1::Tn1, bn2::Tn2 ... bni::Tnk |
                             fn(bn1, bn2 ... bni)=t)
  rn(t::T)::boolean is exists(bn1::Tn1, bn2::Tn2 ... bnm::Tnm |
                             fn(bn1, bn2 ... bnm)=t)

  b11(t::f1(x,_,_ ... _))::T11 is x;
  b12(t::f2(_,x,_ ... _))::T12 is x;
  ...

begin
  t1: forall(x::T | exists(x1::b11,...,xi::b1i | f1(x1,x2, ... xi) = x) or
                    exists(x1::b21,...,x2::b2j | f2(x1,x2, ... xi) = x) or
                    ...
                    exists(xn::bn1,...,xn::bnm | fn(x1,x2, ... xi) = x) or
```

The syntax for creating parameterized constructed type definitions adds a collection of type parameters to the definition:

```
F(p1::type,...,pn::type):: type is data (p1::type,...,pn::type)
  f1(b11::T11, b12::T12 ... b1i::T1i)::r1 |
  f2(b21::T21, b22::T22 ... b2j::T2j)::r2 |
  ...
  fn(bn1::Tn1, bn2::Tn2 ... bnk::Tnk)::rn ;
```

In this case, the result is a type definition function that can be used to create new subtypes of the new type `F`. Specifically, when instantiated `F(p1,...pn)` creates a new constructed type with constructed type variables instantiated.

The tree example is one such parameterized constructed type definition. The new type, `Tree(a)`, is parameterized over a single value that is used as a type in subsequent definitions:

```
Tree(a::type) :: type is data(a::type)
  empty::nullp |
  node(L::Tree(a),v::a,R::Tree(a))::nodep;
```

Thus, the definition:

```
IntTree :: type is Tree(integer);
```

This definition creates a new type called `IntTree` that is formed by instantiating the `Tree` constructed type with `integers`. Alternatively:

```
AnIntTree :: Tree(integer);
```

creates a single new integer tree named `AnIntTree` that is of the type created by the parameterized constructed type instantiation.

2.10.2 Enumerations

Enumerations provide a mechanism for declaring new elemental values and types by extension. The enumeration former translates directly into a constructed type definition. For example, the following notation:

```
enumeration(apple,orange,pear)
```

is semantically equivalent to:

```
data apple | orange | pear
```

`enumeration(apple,orange,pear)` == $\{apple,\ orange,\ pear\}$

Enumerations can be used to define new types using the canonical Rosetta notation:

```
fruit :: type is enumeration(apple,orange,pear);
```

This declaration creates a new item whose values are constructed using the constant constructors `apple`, `orange`, and `pear`. A variable defined as:

```
x :: fruit;
```

must take its value from the set {`apple,orange,pear`}.

2.10.3 Records

In Rosetta, no special syntax for defining records is defined as record structures follow directly from constructed types. A record type is a constructed type with a single constructor function that associates values with parameters used as field names. A typical record type will be defined with the following constructive technique:

```
record::type is data
  recordFormer(f0::T0 | f1::T1 | ... fn::Tn)::recordp;
```

where `recordFormer` is the single constructor, `f1`through `fn` are the names of the various fields and `T1` through `Tn` are the types associated with those fields. The recognizer `recordp` is also defined, but is largely unused. To define a specific record type that represents Cartesian coordinates, the following notation is used:

```
cartesian::type is data
  cartFormer(x::real, y::real, z::real)::cartp;
```

To define an item of this type, the standard Rosetta declaration syntax is used:

```
c :: cartesian;
```

Values an be associated with record items using the canonical `is` form:

```
origin :: cartesian is cartFormer(0,0,0);
```

Accessing individual fields of the record is achieved by applying one of the observer functions associated with a field name. To access field `y` in the record `c`, the following notation is used:

```
y(c)
```

Forming a record is achieved by calling the constructor function:

```
recordFormer(v1,v2,...,vn)
```

where `v1` through `vn` name the specific values for fields `f1` through `fn`. Defining a coordinate in Cartesian space using the definition above is achieved by:

```
cartFormer(1,0,0);
```

Accessing the result is achieved using the observer functions:

```
x(cartFormer(1,0,0))==1;
y(cartFormer(1,0,0))==0;
z(cartFormer(1,0,0))==0;
```

Using the "_" notation, it is possible to create records whose specific field values are not known. The following creates a cartesian coordinate whose `x` and `y` values are known, but whose `z` value is not specified:

```
cartFormer(1,0,_);
```

Should the function `z` be instantiated with this record, the return value is undefined.

```
%% Working here...
```

2.10.4 Pattern Matching

Pattern matching in parameter lists dramatically simplifies defining observer functions over type constructors. Parameter matching takes advantage of the mechanism used to create its input parameters. Consider the integer tree definition presented above. Two constructor functions, `empty` and `node` are defined to construct two different types of trees. Viewed differently, they also partition trees into the subclasses constructed by those individual functions. Specifically, the empty and nonempty trees. Viewed in this manner, it follows the the constructor functions can be used to generate types like any other types. For example:

```
nonemptyIntTree :: type is ran(node)
empty :: type is ran(empty)
```

Due to the nature of constructed types, the constructor for a particular instance of the type is always known. This fact can be utilized to perform pattern matching when instantiating function parameters. Consider the following definition of `is_empty` using selective union:

```
is_empty(t::tree(integer))::boolean is
   (<*(t::empty):boolean is true *> |
    <*(t::node(lt,v,rt))::boolean is false*>);
```

The first function accepts a single parameter of type `empty`. This shorthand is equivalent to saying that `t` is contained in the set of all trees generated by `empty`. Of course, this contains the single `empty` tree. In the second definition, the type `node(lt,v,rt)` refers to all trees that can be constructed with `node`. Furthermore, `lt`, `v` and `rt` become parameters in the function that are bound to the actual parameters of any invocation of `node`. Specifically, in the following function call:

```
<*(t::node(lt,v,rt))::boolean is false*>(node(empty,5,node(empty,6,empty)))
```

`lt = empty`, `v=5`, and `rt=node(empty,6,empty)` within the scope of the function. These values are determined by matching the constructor function `node` with the parameter specification for `t`. The parameters are implicitly defined and their associated types determined from the constructor specification. Specifically, `lt` and `rt` are of type `tree(integer)` while `v` is of type integer.

A more interesting case is defining accessor functions for the left and right subtrees of a nonempty tree. This is accomplished using the following definitions:

```
lTree(t::node(lt,v,rt))::tree(integer) is lt;

rTree(t::node(lt,v,rt))::tree(integer) is rt;
```

The utility of pattern matching is more obvious here. The two functions return actual parameters associated with the constructor function `node`. Furthermore, both functions are defined only over trees constructed with `node` and are not defined over trees constructed with `empty`. This is the desired result for high level specification.

In the definitions of `lTree`, `rTree` and `is_empty`, some or all of the constructor parameters are not used in the internal function. Thus, they need not be named in the definition. We use "_" to designate such a parameter as in the following:

```
is_empty(t::tree(integer))::boolean is
  (<*(t::empty):boolean is true *> |
   <*(t::node(_,_,_))::boolean is false*>);

lTree(t::node(lt,_,_))::tree(integer) is lt;

rTree(t::node(_,_,rt))::tree(integer) is rt;
```

In both cases, parameters that are not used are not named or available in the function definition.

Chapter 3

Facet and Package Basics

The basic unit of specification in Rosetta is termed a `facet`. Each facet defines a single aspect of a component or system from a particular perspective. To define facets completely, it is necessary to understand the basics of Rosetta declarations, functions and expressions. This chapter intends only to introduce the concept and simple examples of facet definition to motivate the descriptions in following chapters. If concepts are not fully presented here, assume they will be in chapters dealing with the specifics of facet definition.

A facet is a parameterized construct used to encapsulate Rosetta definitions. Facets form the basic semantic unit of any Rosetta specification and are used to define everything from basic unit specifications through components and systems. Facets consist of three major parts: (i) a parameter list; (ii) a collection of declarations; (iii) a domain; and (iv) a collection of labeled terms. This section introduces the facet syntax and an *ad hoc* facet semantics, and provides structure for the remainder of the document.

3.1 Facet Definition

Facets are defined using two mechanisms: (i) direct definition; and (ii) composition from other facets and functions. In this section we will deal only with direct definition and defer facet composition to Section 3.3. Direct definition is achieved using a traditional syntax similar to a procedure in a traditional programming language or a theory in an algebraic specification language. The general format for a facet definition is as follows:

```
facet <facet-label>(<parameters>)::<domain> is
   <declarations>
begin
  <terms>
end facet <facet-label>;
```

The facet definition is delineated by the `facet` keyword immediately followed by a $< facet - label >$ providing the facet with a unique name. The facet label is immediately followed by a comma separated parameter list denoted above by $< parameters >$, a domain that denotes the facet type by *¡domain¿* and the keyword `is` that opens the declarations section. The declarations section, denoted by $< declarations >$, is used to declare labeled items and define visibility of locally defined labels using an optional `export` clause. The keyword `begin` starts the definition section. Declarations follow in the form of labeled terms, denoted $< terms >$ that provide a definition for facet. The definition concludes with the keywords `end facet` and the facet label.

As an example, a specification for a `find` component follows:

```
facet register(i::input bitvector; o::output bitvector;
               s0::input bit; s1::input bit)::state_based is
  state::bitvector;
begin
  l1: if s0=0 then
         if s1=0 then state'=state
            else state'=lshr(state) endif
      else
         if s1=0 then state'= lshl(state)
            else state'=i endif
      end if;
  l2: o'=state';
end facet register;
```

This definition describes a facet `register` with data parameters `i`, and `o` of types `bitvector` and two control parameters of type `bit`. The variable `state` is defined to hold the internal state of the register and is of type `bitvector`. As can be deduced from examination of the specification, this register performs hold, logical shift right and left, and load operations given inputs of `00`, `01`, `10` and `11` on parameters `s0` and `s1` respectively.

All Rosetta variables and parameters are declared using the notation `v::T` where `v` is a variable name and `T` is a type. The "`::`" notation is used to indicate a declaration. The notation `x::T` creates an item labeled `x` whose values are associated with type `T`. The declaration can be viewed as declaring an item `x` whose possible values are selected from the set `T`. In the `register` specification, `state::bitvector` defines a variable labeled `state` whose values can be selected from the type `bitvector`.

Parameters are universally quantified variables visible over the scope of the facet. Parameter definitions are like traditional declarations with the addition of a kind indicator. Specifically, `i::input bitvector` defines a parameter `i` of type `bitvector` and declares the predicate `input(i)` to be true. The semantics of `input(i)` are defined by the semantic domain currently being used. Generally the `input` kind is used to denote a parameter used to provide input to the facet. The kinds `output` and `design` are also quite common in specifications with `output` indicating a parameter providing an output and `design` identifying a monotonic generic parameter. Parameters without a kind specifier are unqualified in a facet definition.

In the `register` example, the `state_based` domain defines the specification vocabulary used by the facet. The domain provides definitions and a basic model of computation for specification. The `state_based` domain provides a basic vocabulary for axiomatic specification and defines the "tick" notation (`state'`) to represent a label in the next state. It also defines `input` to explicitly make the value of input parameters invariant over state change. Domains are semantically facets and will be discusse in detail in Chapter 7.

The declaration section following the facet interface includes declarations local to the facet. Items defined in this manner are visible throughout the facet. Such declarations may be made visible outside the facet using an `export` statement. In this case, the exclusion of an `export` clause implies that no labels defined in the facet are visible outside the specification. The notation `export all` causes all defined labels to be visible outside the facet. Including a list of locally defined labels explicitly identifies what labels are and are not visible.

When referenced in the facet body, a term, variable or parameter is referenced by its label without decoration. When used outside the facet, labels are referenced using the facet name as a qualifier. In the modified register example below, `register.l1` refers to the first term in `register` while `register.state` refers to the variable `state`.

```
facet register(i::input bitvector; o::output bitvector;
               s0::input bit; s1::input bit):;state_based is
  state::bitvector;
  export all;
```

```
begin
  l1: if s0=0 then
         if s1=0 then state'=state
            else state'=lshr(state) endif
      else
         if s1=0 then state'= lshl(state)
            else state'=i endif
      end if;
  l2: o'=state';
end facet register;
```

The `begin-end` pair delimits the domain specific terms within the facet while the facet type defines the domain. The `begin` statement opens the set of terms. In the `register` the semantic domain is `state_based` providing the basic semantics for state and change in the traditional axiomatic style. Specifying a semantic domain indicates what domain theory the facet uses for its definition. Every facet must have an associated domain even if that domain is the `logic` domain common to all facets.

Terms in the term list define the behavior modeled by the facet. Each term is a labeled, well formed formula of type boolean or facet. Boolean terms define basic facet properties. Facet terms define facet properties by composing other facets. Both boolean and facet terms may be included in the same facet.

The general form associated with any term is:

```
l: term;
```

where `l` is the label associated with the term and `term` is the definition itself. The label is used to reference the term in other definitions as well as when the term is exported. As boolean terms are rarely referenced directly, their labels may be ommitted. All terms defined in scope of the `begin-end` pair are considered top level terms.

```
%% Add the use of a let clause for local definitions.
```

The `register` uses two terms to define behavior. The first, labeled `l1`, defines the register's next state in terms of its current state, input and control inputs. The `if` statement implements the various cases for hold, shift right, shift left, and load. It should be noted that the shift operations are implemented using the built in bitvector shift functions `lshr` and `lshl` that provide logical shifts over bitvector types. The second term, labeled `l2` defines the next output. This simple expression states that the next output is the same as the next state as defined by term `l1`.

It should be noted that both terms defined in `register` hold simultaneously. Thus, both the next state and output definitions must hold for the component to behave correctly. The structure of the specification is much like the structure of a VHDL specification. Each state variable and output parameter is handled individually. The distinction here is the variability of definition semantics. In this case, the Rosetta function semantics is used to calculate next values for each variable.

The domain extends the base definition semantics by adding new definitions specific to a specific domain. In the case of `state_based`, the basic addition is the concept of current and next state. Specifically in the register definition, `state` refers to the register contents in the current state while `state'` refers to the register contents in the following state. The `state_based` domain defines the semantics of "`x'`".

Parameter instantiation is achieved by traditional universal quantifier elimination. An object of the specified type is selected and the parameter replaced by that object. When formal parameters are instantiated with objects, those objects replace instances of parameters throughout the facet specification. When `A` is an actual parameter and `F` is a formal parameter, the notation `A=>F` allows direct assignment of actual parameters to formal parameters. This notation allows partial instantiation and is sometimes necessary when parameter ordering in constructed facets is ambiguous.

```
%% Parameter instantiation in a facet must be discussed further.
%% This is associated with bug 164
```

Consider the following modified `register` specification:

```
facet register(i::input bitvector; o::output bitvector;
               s0::input bit; s1::input bit)::state_based is
  export state;
  state::bitvector;
begin
  l1: if s0=0 then
        if s1=0 then state'=state
           else state'=lshr(state) endif
        else if s1=0 then state'= lshl(state)
           else state'=i endif
      end if;
  l2: o'=state';
end facet register;
```

This specification is identical to the previous definition except that only the `state` variable is visible outside the facet scope. The terms `l1` and `l2` are no longer visible as they are not listed in the export clause. The variable `state` is accessed using the name `register.state` because `register` is the label assigned to the facet.

3.1.1 Examples

Examples are included here to provide motivation for the facet syntax and to provide context for the following sections. It is intended that these examples provide an overview, not a detailed language description. It is suggested that these be referred to while reading subsequent chapters as a means for understanding the utility of Rosetta definition capabilities.

```
%% Examples are fried with the most recent syntax and language
%% semantics changes.
%%
%% sort - Too naive.  Complete the definition or delete.
%% array_utils - Update to reflect the fact that arrays are gone.
%% array_utils package - Probably update this and remove the facet
%% version.
%% sort - All the later sort stuff should be thought through.  We
%% don't have units or a constraint package at the moment.
%%
%% Include examples from the TDMA thingy.
```

Example 3 (Sort Definition) *A declarative specification for requirements and constraints associated with a* `sort` *function has the following form:*

```
use array_utils(integer);
facet sort_req(i::input sequence(integer);
               o::output sequence(integer))::state_based is
begin
  l2: permutation(o',i);
  l1: ordered(o');
end facet sort_req;
```

The facet ***sort_req*** *defines a view of a component that accepts an array of integers as input and outputs the array sorted. This simple specification demonstrates several aspects of Rosetta specification using the* ***state_based****, axiomatic style.*

Parameters for ***sort_req*** *are simply an input and output arrays of type* ***integer****. The facet uses the* ***state_based*** *domain allowing the use of* ***o'*** *to represent the output in the state following execution. The package* ***array_utils*** *(defined later) is included to provide definitions necessary for defining sort. Specifically,* ***permutation*** *and* ***ordered****. These functions could be defined in the declaration section of the package, however this definition is cleaner and allows reuse of the array utilities in other specifications. Note that the* ***array_utils*** *package is parameterized over a type. This parameterization is used to specialize the* ***array_utils*** *for any appropriate type.*

It is possible to write a ***sort_req*** *definition that is parameterized over the contents of the input and output array. This implementation sorts arrays of integers. Although this may be interesting from a pedagogical perspective, it is not particularly useful or reusable. The following definition parameterizes the facet definition over an arbitrary type,* ***T****:*

```
use array_utils(T);
facet sort_req(T::design type; i::input array(T);
               o::output array(T))::state_based is
begin
  l2: permutation(o',i);
  l1: ordered(o');
end facet sort_req;
```

In this new ***sort_req*** *facet, the type* ***T*** *associated with the contents of the input and output arrays is a parameter. This allows specialization of the* ***sort_req*** *facet for various array contents. The only restriction being that an ordering relationship must be defined on the array elements.*

The following instantiation of the parameterized ***sort_req*** *is equivalent to the original* ***sort_req*** *facet:*

```
    sort_req(integer,_,_);
```

This usage replaces all instances of ***T*** *in the facet with the type* ***integer****. The resulting facet is semantically identical to the original* ***sort_req*** *definition.*

Example 4 (array_utils Package) *Packages are a parameterized mechanism for grouping together definitions. They are defined using the semantics of facets and will be discussed fully in a later section. Here, the definition of the* ***array_utils*** *package used by the* ***sort_req*** *facet is defined:*

```
package array_utils(T::univ)::logic is

  // numin - return the number of occurrences of x in i
  numin(x::T; i::sequence(T)):: natural is
    if i=[] then 0
            else if x=i(0) then 1+numin(tail(i))
                           else numin(tail(i))
                 end if
    end if;

  // permutation - determine if a1 is a permutation of a2
```

```
  permutation(a1::sequence(T); a2::sequence(T)):: boolean is
    forall(x::T | numin(x, a1) = numin(x, a2));

  // ordered - determine if a1 is ordered.  =< must be defined on T
  ordered(a1::sequence(T)):: boolean is
    forall(i :: sel(x::natural| x =< #a1-1) | a(i) =< a(i+1));

  // tail - return the tail of an array.  based on sequence tail.
  tail(a1::sequence(T)):: sequence(T) is tl(a1);

end package array_utils;
```

The `array_utils` *package defines four general purpose functions for arrays: (i)* `numin`*; (ii)* `permutation`*; (iii)* `ordered`*; and (iv)* `tail`*. It is difficult to explain these definitions fully without deeper understanding of Rosetta function definition. However, some exploration will aid in understanding and writing more complex specifications.*

As an example, examine the definition of `permutation`*:*

```
  permutation(a1::sequence(T); a2::sequence(T)):: boolean is
    forall(x::T | numin(x, a1) = numin(x, a2));
```

This definition can be divided into two parts. First, the signature of `permutation` *is given as*

```
  permutation(a1::sequence(T); a2::sequence(T)):: boolean
```

The function name is `permutation`*,* `(a1::sequence(T); a2::sequence(T))` *are the domain parameters, and* `boolean` *is the return type.*

The second part of the definition, following the keyword `is`*, denotes the value of the return expression. The expression specifies the permutation. It is true when every element of* `T` *occurs in* `a1` *and* `a2` *the same number of times. It is false otherwise. The syntax of function declaration and the semantics of* `forall` *and other constructs are defined later.*

Other functions are similarly defined. `numin` *determines the number of occurrences of a value in an array using a simple recursive definition.* `ordered` *defines a predicate that is true when every element of its argument array is greater than or equal to the preceding element. Finally,* `tail` *for arrays is defined by extracting the elements into a sequence, finding the tail, and recreating an array. Remember, to fully understand these definitions requires further knowledge of Rosetta type and function semantics that will be presented later.*

Example 5 (Sort Constraints) *An alternative view of a component models performance constraints. The following definition models the power consumption constraints of a sorting component.*

```
facet sort_constr::constraints
  power::real;
begin
  p1: power =< 5mW;
end facet sort_constr;
```

The variable `power` *is a real number representing power consumed by the component. The facet body defines a single term that limits power consumption to be less than or equal to 5mW. Both the semantics of constraints and the unit constructors required to define 5mW are defined in the constraints facet.*

Example 6 (Timed Sort) *The facet* `sort_timed` *is an alternative definition of sort that places timing constraints on the definition. Here, instead of modeling what is true in an abstract next state, the sort is specified with respect to its behavior over time.*

```
use array_utils(T);
facet sort_timed(T::design type; i::input sequence(T);
                 o::output sequence(T))::continuous is
begin
  l2: permutation(o@(t+5ms),i);
  l1: ordered(o@(t+5ms));
end facet sort_timed;
```

This definition uses the `continuous` *domain rather than the* `state_based` *domain. The notation* `x@t` *refers to the value of* `x` *at time* `t`*. The term* `l2` *states that the output,* `o`*, 5ms in the future must be a permutation of the current input,* `i`*. The term* `l1` *states that the output must be ordered 5ms in the future.*

No notion of next state as used previously is defined. However, this specification provides more detail in the form of hard timing constraints. Using the `continuous` *domain, the user is allowed to define values of variables at specific times with respect to the current time* `t`*.*

Example 7 (Operational Sort) *The facet* `sort_op` *provides an operation definition for a sorting algorithm by "implementing" a quicksort algorithm that will sort the input. Specifically:*

```
facet sort_op(i::in T; o::out T)::continuous is

  qsort(i::sequence(T))::sequence(T) is
     let (pivot::T be t(0); t::sequence(T) be tail(i)) in
         if i=nil
            then i
            else qsort(lside(pivot,t)) & [pivot] & qsort(rside(pivot,t))
         end if;

  lside(pivot::T; i::sequence(T))::sequence(T) is
     if i=nil
        then nil
        else if i(0) =< pivot
                then cons(i(0),lside(pivot, tail(i)))
                else lside(pivot, tail(i))
             end if
     end if;

  rside(pivot::T; i::sequence(T))::sequence(T) is
     if i=nil
        then nil
        else if i(0) > pivot
                then cons(i(0),lside(pivot, tail(i)))
                else lside(pivot, tail(i))
             end if
     end if;

begin
  l1: o@(t+5ms) = qsort(i);
end facet sort_op;
```

This specification is interesting due to its similarity to a VHDL specification and its equivalence to ***sort_req****. The* ***sort_op*** *specification specifies that the output parameter 5ms in the future is equal to the result of applying quicksort to the input parameter* ***i****. The details of the application are unimportant. Suffice to say that excluding the concept of a wait statement, this is quite similar to how a VHDL specification might be defined.*

The function ***qsort*** *and the auxiliary functions* ***lside*** *and* ***rside*** *define a quicksort algorithm over sequences. The definition follows the classic recursive style. As with other function definitions in these examples, these functions require some further study to understand completely. At this point it is important only to understand that parameters to the function are specified as* ***param-var::param-type****, separated by the "****;****" token and enclosed within parantheses. The final expression defines the return value. In the case of* ***lside****, all values less than or equal to the pivot value are found recursively and returned.*

A potentially cleaner specification might have the form:

```
package work(T::univ)::logic is
  export sort_op;

  qsort(i::sequence(T))::sequence(T) is
     let (pivot::T be t(0); t::sequence(T) be tail(i)) in
         if i=nil
            then i
            else qsort(lside(pivot,t)) & [pivot] & qsort(rside(pivot,t))
         end if;

  lside(pivot::T; sequence(T))::sequence(T) is
     if i=nil
        then nil
        else if i(0) =< pivot
                then cons(i(0),lside(pivot, tail(i)))
                else lside(pivot, tail(i))
             end if
     end if;

  rside(pivot::T; i::sequence(T)):: sequence(T) is
     if i=nil
        then nil
        else if i(0) > pivot
                then cons(i(0),lside(pivot, tail(i)))
                else lside(pivot, tail(i))
             end if
     end if;

  facet sort_op(i::input T; o::output T) is
  begin continuous
    l1: o@(t+5ms) = qsort(i);
  end facet sort_op;

end package work;
```

Here the function specifications are removed from the facet specification. The facet and functions are included in the package ***work****. The similarity to VHDL here is intentional. Unlike VHDL, the package is parameterized allowing specialization for arbitrary types. Note the inclusion of the* ***export sort_op*** *clause. This causes the* ***sort_op*** *facet to be visible outside the package. Other declarations such as* ***qsort****,* ***lside*** *and* ***rside*** *are hidden in the package.*

Why the obsession with sort? Thus far, an axiomatic, continuous time and operational continuous time specification have been developed. Together, we can use all three specifications to define various characteristics of a single sorting component in a manner unique to Rosetta. Specifically, in the next section we will define how a designer can specify a sorting component by combining specifications from multiple domains. The result is a requirements specification, a temporally constrained requirements specification, an operational specification, and a power specification simultaneously describing a system. With the addition of facet composition operators, this provides a powerful mechanism for mixing and composing specifications.

Example 8 (Alarm Clock System) *Consider the following definition of an alarm clock taken from the Synopsys synthesis tutorial. This alarm clock provides a basic capability for setting time, setting alarm, sounding an alarm and keeping time. The specification states the following requirements:*

1. *When the* `setTime` *bit is set, the* `timeIn` *is stored as the* `clockTime` *and output as the display time.*
2. *When the* `setAlarm` *bit is set, the* `timeIn` *is stored as the* `alarmTime` *and output as the display time.*
3. *When the* `alarmToggle` *bit is set, the* `alarmOn` *bit is toggled.*
4. *When* `clockTime` *and* `alarmTime` *are equivalent and* `alarmOn` *is high, the alarm should be sounded. Otherwise it should not.*
5. *The clock increments its time value when time is not being set.*

The systems level description of the alarm clock is defined in the following facet:

```
use timeTypes;
facet alarmClockBeh(timeIn::input time; displayTime::output time;
          alarm::output bit; setAlarm::input bit;
          setTime::input bit; alarmToggle::input bit)::state_based is
  alarmTime :: time;
  clockTime :: time;
  alarmOn :: bit;
begin
  setclock: setTime=1 =>
              clockTime' = timeIn and displayTime' = timeIn;
  setalarm: if setAlarm=1
              then alarmTime' = timeIn and displayTime' = timeIn
              else alarmTime' = alarmTime
            end if;
  displayClock: setTime = 0 and setAlarm = 0 =>
                  displayTime' = clockTime';
  tick: setTime => clockTime' = increment_time(clockTime);
  armalarm: if alarmToggle = 1
              then alarmOn' = -alarmOn
              else alarmOn' = alarmOn
            end if;
  sound: alarm' = alarmOn and %(alarmTime=clockTime);
end facet alarmClockBeh;
```

Inputs correspond to data and control values for the clock. `timeIn` *contains the current time input and can be used to set either the alarm time or the clock time.* `displayTime` *is the time currently being displayed.* `alarm` *drives the audible alarm.* `setAlarm` *and* `setTime` *control whether the alarm time or clock time are currently being set.* `alarmToggle` *causes the alarm set state to toggle.*

Local variables correspond to the state of the clock. `alarmTime` *is the current time associated with sounding an alarm.* `clockTime` *is the current time.* `alarmOn` *is "1" when the alarm is set and "0" otherwise.*

Exploring the specification indicates that each requirement is defined as a labeled term. Each term can be traced back to a requirement from the English specification. Term `setclock` *handles the case where the clock time is being set. Term* `setalarm` *handles when the alarm time is being set. Term* `armalarm` *handles the toggling of the alarm set bit.* `tick` *causes the clockTime to be incremented. The clock time is incremented in the next state only when the clock time is not being set. Finally, the* `sound` *term defines the* `alarm` *output in terms of the* `alarmOn` *bit and whether the* `alarmTime` *and* `clockTime` *values are equal. The "%" notation transforms the boolean result of equals into a bit value. All terms must be simultaneously true. Thus, the specification has the same effect as using multiple processes in VHDL.*

The alarm clock facet uses the following collection of time manipulation functions and types:

```
package timeTypes::logic is
  hours :: subtype(natural) is sel(x::natural | x =< 12);
  minutes :: subtype(natural) is sel(x::natural | x =< 59);
  time :: type is data record(h::hours; m::minutes)::time?;

  increment_time(t:: time) :: time is
    record(increment_hours(t);  increment_minutes(t));

  increment_minutes(t:: time) :: minutes is
    if t(m) < 59
       then t(m) + 1
       else 0
    end if;

  increment_hours(t::time) :: hours is
    if t(m) = 59
       then if t(h) < 12
               then t(h) + 1
               else 1
            end if
       else t(h)
    end if;
end package timeTypes;
```

`hours` *and* `minutes` *are restricted subranges of natural number representing hours and minutes respectively. The notation* `type(natural)` *indicates that* `hours` *and minutes are bunches, not singleton values. The* `sel` *operation provides a comprehension operator and is used to filter natural numbers.* `time` *is a constructed type defined as a record containing an hours value and a minutes value.*

Three increment functions define incrementing time. `increment_time` *forms a record from the results of incrementing the current hours and minutes values.* `increment_hours` *and* `increment_minutes` *handle incrementing hour and minute values respectively. Note that the field names are used to reference hours and minutes values respectively.*

```
%% Remove this definition or fix it.
```

Example 9 (Stack definition) *For formal specification fans, a semi-constructive stack definition is included to describe an alternate means for function specification. Here, the traditional stack operations are declared, but are not defined directly. The distinction with other function definitions being that no constant definition appears in conjunction with the declaration. Assume here that there exist in the containing package declarations for EType and SType. Then the specification takes the form:*

```
facet stack::logic is
  push(E::Etype;S::Stype)::Stype;
  pop(S::Stype)::Stype;
  top(S::Stype)::Etype;
  is_empty(S::Stype)::boolean;
  empty::Stype;
begin
   ax1:forall(e::Etype|forall(s::Stype|pop(push(e,s))=s));
   ax2:forall(e::Etype|forall(s::Stype|top(push(e,s))=e));
   ax3:forall(e::Etype|forall(s::Stype|not(is_empty(push(e,s)))));
   ax4:is_empty(empty);
end facet stack;
```

This is a canonical constructive specification for a stack. In the declarations section, ***push****,* ***pop****, and* ***top*** *are defined to operate over stacks and elements. The axioms defined as* ***ax1*** *through* ***ax4*** *constrain the values of functions in the traditional declarative fashion.*

This specification style may prove uncomfortable for traditional VHDL users. An alternate definition uses sequences to represent the stack:

```
package stackAsSeq(E::type)::logic is
  S::subtype(sequence(universal)) is sequence(E);
  push(s::S; e::E) :: S is cons(e,s);
  pop(s::S) :: S is tl(s);
  top(s::S) :: E is hd(s);
  empty::S is nil;
  is_empty(s::S) :: boolean is s=empty;
end package stackAsSeq;
```

This stack definition uses the ***package*** *construct to present a series of direct definitions. No terms are needed to describe the behavior of the provided type. The stack type,* ***S****, is not an uninterpreted type but is defined as a sequence of type* ***E****. The basic stack operations are now defined on the stack type using concrete operations.*

An interesting exercise is to consider the meaning of:

```
stack(E,sequence(E)) and stackAsSeq(E)
```

As we shall see later, facet composition states that properties of both ***stack*** *and* ***stackAsSeq*** *must apply in the facet formed by* ***and****. Effectively, this new definition is consistent only if* ***stackAsSeq*** *obeys the axiomatic definition provided by* ***stack****. In essence,* ***stack*** *represents requirements while* ***stackAsSeq*** *represents an implementation of stack.*

Summary: A facet is the basic unit of Rosetta specification. It consists of a label, optional parameter list, optional declarations, a domain and terms that extend its domain. Variable declaration is achieved using the notation `v::T` interpreted as *the value of v is contained in T*. Constants are similarly defined using the notation `v::T is c` interpreted as *the value of v is contained in T and is equal to c*. Domains provide a vocabulary for defining specifications. Terms extend domains to provide definitions for the specific components. Terms are declarative constructs that are accompanied by a label. Any label defined in a Rosetta specification may be exported and referenced using the canonical `facet-name.label` notation. By default, all labels are exported. However, an explicit export statement may be used in the declaration section to selectively control label export.

3.2 Facet Aggregation

An important system level specification activity is aggregation of facets into general purpose architectures. Rosetta supports this directly using *facet inclusion* and *facet labeling*. Facet inclusion occurs when a facet name is referenced in a facet term. Facet labeling occurs when a facet is given a new label.

Consider the trivial example of defining a three input and gate from two input and gates:

```
facet andgate(x, y::input bit; z::output bit)::state_based is
begin state_based
  l1: z' = x and y;
end facet andgate;

facet andgate3(a,b,c::input bit; d::output bit) is
  i:: bit;
begin
  l1: andgate(a,b,i);
  l2: andgate(i,c,d);
end facet andgate3;
```

The resulting definition is quite similar to structural VHDL without explicit component instantiation. The first facet clearly defines the behavior of a simple *and* gate while the second seems to use facets as terms. The terms `l1` and `l2` both reference `andgate` and are interpreted as stating that the definitions provided by each are true. Thus, the first term instantiates `andgate` with items `a`, `b` and `i` where `i` is an internally defined variable of type `bit`. Thus, the facet asserts that `i` is equal to `a and b`. The second term does the same except it asserts that `d` is equal to `i and c`.

Communication between facets is achieved by sharing items. Here, the items are variable items defined either in the parameter list or in the body of the including facet. This models instantaneous exchange of information between facets via variables. Later, channels will be introduced to provide means for defining connections with properties such as storage and delay.

Although similar to VHDL structural definition, this Rosetta definition style is semantically quite different. To understand this requires some understanding of labels and item labeling. The notation `l: term` defines `term` and associates label `l` with it. Thus, the definition:

```
l1: andgate(a,b,i);
```

asserts `andgate(a,b,i)` as a term and associates label `l1` with it. Effectively, the definition renames `andgate` locally to `l1`. Thus, the terms `l1` and `l2` define facets equivalent to `andgate`, but with new names. The reasoning for this is demonstrated in any definition where components that locally define variables and constants have multiple instances. For example, consider the following incorrect specification:

```
facet register(i::in bitvector; o::out bitvector;
               load::in bit)::state_based is
  memory::bitvector;
begin
  load1: if %load then memory'=i else memory'=memory end if;
  output: o'=memory;
end facet register;

facet registerx2(i1,i2::input bitvector;
                 o1,o2::output bitvector;
```

```
                load::input bit) is
begin state_based
  register(i1,o1,load);
  register(i2,o2,load);
end facet registerx2;
```

Consider the `memory` variable associated with each register. In the above definition, `register.memory` reference to the `memory` variable in facet `register`. Unfortunately, there's no way to learn which register. Further, because the register variables share the same name in the facet, they must be equal.

The proper definition is:

```
facet register(i::input bitvector; o::output bitvector;
               load::input bit)::state_based is
  memory::bitvector;
begin
  load1: if %load then memory'=i else memory'=memory endif;
  output: o'=memory;
end facet register;

facet registerx2(i1,i2::input bitvector;
                 o1,o2::output bitvector;
                 load::input bit)::state_based is
begin
  r1:register(i1,o1,load);
  r2:register(i2,o2,load);
end facet registerx2;
```

In this definition, the facet `register` is "copied" and relabeled twice. In the first case, the new facet is named `r1` and in the second, `r2`. The `memory` variable associated with `r1` is referenced via `r1.memory` and similarly for `r2.memory`. Now there is no conflict and the elements of each component have unique references. This aspect of labeling is simple, but extraordinarily powerful.

Summary: Including facet definitions as terms supports structural definition through facet aggregation. Including and instantiating facets in definitions is achieved using relabeling. Instantiating facets replaces formal parameters with actual items. Unique naming forces these items to be shared among facets providing for communications. When a facet is renamed, all of its internal items are renamed making each instance of that included facet unique.

```
%% The following section is way out of date given the updates to the
%% facet algebra.  We need to rethink facet declaration to include
%% parameters (using a notation like functions) to make this happen
%% correctly.  I think we can use the same semantics.
```

3.3 Facet Composition

The essence of systems engineering is the assembling of heterogenous information in making design decisions. Rosetta supports this type of specification directly with operations collectively known as the *facet algebra*. The facet algebra provides mechanisms for defining new specifications by composing existing specifications using the standard operators `and`, `or`, and `not`.

In the context of facets, these are not logical operators. The operation `F1 and F2` does not have a boolean value. Instead, it defines a new facet with properties from both `F1 and F2`. Looking ahead, this operation provides us a mechanism for combining properties from several facets into a single facet.

Facets under composition must maintain the logical truths as specified by standard interpretations of logical connectives. For example, if `F3 = (F1 and F2)`, then `F3` is consistent if and only if `F1 and F2` is consistent (Note: `F1 and F3` are enclosed in parentheses because `=` has higher precedence than `and`). Facet composition is useful for specifying many systems level properties by combining properties from various facets. A new facet can be defined via composition by an expression of the following form:

```
<name>(<paramlist>) is <facet_expression>;
```

where $< name >$ is the new name, $< paramlist >$ is an optional parameter list, and $< facet\_expression >$ is an expression comprised of facet algebra operations.

The following examples describe several prototypical uses of facet composition. Please note that domains used in these examples are defined in an accompanying document.

`F1 and F2` Facet conjunction states that properties specified by terms `T1` and `T2` must by exhibited by the composition and must be mutually consistent. Further, the interface is $I_1 \cup I_2$ implying that all symbols visible in `F1` and `F2` are visible in the composition.

The most obvious use of facet conjunction is to form descriptions through composition. Of particular interest is specifying components using heterogeneous models where terms do not share common semantics. A complete description might be formed by defining requirements, implementation, and constraint facets independently. The composition forms the complete component description where all models apply simultaneously.

Example 10 (Requirements and Constraints) *Reconsider the previously defined facets* `sort_req` *and* `sort_const`*. Recall that* `sort_req` *defined requirements for a sorting component while* `sort_const` *defined a power constraint over the same component. A sorting component can now be defined to satisfy both facets:*

```
sort :: facet is sort_req and sort_const;
```

Informally, `sort`*: (i) outputs a sorted copy of its input; and (ii) consumes only 5mW of power. Formally, the new facet* `sort` *is the* product *of properties from* `sort_req` *and* `sort_const`*. In this example, the interaction between constraints domain and other requirements domains are unspecified. Therefore, analysis of interactions will reveal little additional information. However, it is certainly possible to define a relationship between the* `constraints` *and* `state_based` *domains if desirable.*

Example 11 (Postcondition Specifications) *Consider again the specifications for* `sort_req` *and* `sort_op`*. The first facet specifies the requirements for a sorting component using a black-box, axiomatic style. The second facet defines sorting using a specific, operational algorithm. Like the constraint model and requirements models previously,* `sort_req` *and* `sort_op` *can be combined into a single sorting definition:*

```
sort :: facet is sort_req and sort_op;
```

Here, the composition behaves much differently. The state-based and models do interact in interesting ways. The composition of `sort_req` *and* `sort_op` *provides a pre- and post-condition for the operational sorting definition. The net effect is like an assertion in VHDL. However, the requirements are specified distinctly and are not intermingled in the operational definition. Thus, for this composition to be consistent, the operational specification must hold along with it's real time constraints and the axiomatic specification must hold defining pre- and post-condition requirements on the composition.*

Similarly, a sort specification can be developed that combines requirements, operational and constraint models:

```
sort :: facet is sort_req and sort_op and sort_const;
```

`F1 or F2` Facet disjunction states that properties specified by either terms `T1` or `T2` must be exhibited by the composition. Note that this is logical or, not exclusive or. The most obvious use of facet disjunction is combining different component models into a component family. The following example illustrates such a situation.

Example 12 (Component Version) *Consider the following definitions using sort facets defined previously:*

```
multisort::facet is sort_req and (bubble_sort or quicksort);
```

The new facet `multisort` *describes a component that must sort, but may do so using either a bubble sort or quicksort algorithm. While* `and` *is a product operator,* `or` *is a sum operator over facets.*

Other facet operations are defined and include negation, implication and equivalence. These will be presented in detail in a later chapter. The objective here is simply to demonstrate various facet composition operations and where they might apply in a specification.

Summary: The facet algebra supports combining facet definitions into new facet definitions. The `and` and `or` operations corresponding to product and sum operations over facets combine facets under conjunction and disjunction respectively. The `and` operation defines new facets with all properties from both constituent facets. The `or` operation defines new facets with properties from either facet.

3.4 Packages

Packages provide a convenient way of aggregating similar Rosetta structures including facets, types, functions and other definitional elements. Semantically, a package is simply a facet with: (i) no terms section; and (ii) explict export of defined symbols. This, the package construct allows only the declaration of new items. The Rosetta package is intended function much like a VHDL package.

Packages are define using the `package` keyword and name, a parameter list, domain and collection of declarations. The `package` construct does not allow terms and the `begin` keyword is ommitted. The name labels the package and provides an access mechanism. The parameter list provides a means for defining models around a common parameter set. Only parameters of kind `design` are allowed. Leaving out the `kind` specifier causes a parameter definition to default to `design`. The domain defines a base domain for all contained definitions. Definitions may include any Rosetta definitional structure including constants, types, functions and relations, facets and other packages.

The form of a package is shown in the following example:

```
package mathops(w:natural)::logic is
 word::word(w);

 bv2nat(w::word)::natural;
 nat2bv(n::natural)::word;

 component adder(i1,i2::bitvector[w], o::bitvector[w+1]) is
 begin
  definition state-based
    bv2nat(o') = bv2nat(i1)+bv2nat(i2);
  end definition;
```

```
 end adder;

 component multiplier(i1,i2::bitvector[w], o::bitvector[2*w]) is
 begin
  definition logic
    bv2nat(o') = bv2nat(i1)*bv2nat(i2);
  end definition;
 end multiplier;

end mathops;
```

By default, all symbols from the package are visible by compilation units using the package. If an `export` clause is present, only listed labels are visible. Users are strongly encouraged to explicitly export symbols from packages. As with facets, exported package labels are referenced using the "package.label" notation.

Packages are included in other compilation units using the `use` keyword and a fully instantiated package name. To use the previous package definition contents within a second package, the following notation is used:

```
  use mathops(8);
```

The result is inclusion of the facet in the immediately following compilation unit. Note that all mathops parameters must be instantiated when it is included. The `adder` component in `mathops` is referenced using the notation `mathops.adder` unless the reference is unambiguous. In this case, simply using `adder` is appropriate. If a local definition of `adder` is declared in the including compilation unit or more than one definition of `adder` is present, then the dot notation must be used. If a facet includes multiple instances of `mathops`, parameters disambiguate definitions as in `mathops(8).adder`.

```
  %% Note that there are still examples remaining in the systems
  %% chapter that we might want to move

  %% Do we want to keep the concept of interface and body compilation
  %% units.  Entered as bug 163.
```

3.5 Label Visibility and Resolution

Rosetta is at its essence a statically scoped language where the declaration associated with a symbol being referenced in an expression can be found at compile time. When resolving a label instance, there are five basic sources for declarations that comprise the context: (i) the local declarative scope; (ii) the enclosing compilation unit; (iii) the facet domain; (iv) packages identifed in a `use` clause; and (v) the context of the enclosing compilation unit.

The local scope associated with an expression is defined as the parameter list associated with a function definition, let expression, quantifier or any expression construct that defines local parameters. If a label is used whose definition occurs in the local scope, that declaration always takes precidence over any enclosing scope. In the example:

```
  facet example::state-based is
    x::integer is 5;
    inc(x::integer)::integer is x+1;
  begin
  end facet example;
```

within the definition of `inc`, `x` used in the expression refers to the local parameter `x`, not the variable `x` defined in the outer scope. This is consistent with traditional programming languages.

Alternatively, in the example:

```
facet example::state-based is
  x::integer is 5;
  inc(x::integer)::integer is example.x+1;
begin
end facet example;
```

The dot notation is used to reference the `x` declared in the facet `example`'s declarative region. Such uses of the dot notation should be avoided, but it is semantically legal in this context.

The next scoping level is the containing compilation unit. Recall that a compilation unit is any structure that is a facet derivative. Specifically, facets, domains, packages and interactions are all facet derivatives and are thus compilation units. In the example:

```
facet example::state-based is
  x::integer;
  inc(x::integer)::integer is x+1;
begin
  t1: x' = inc(x);
end facet example;
```

the `x` appearing in term `t1` is the label declared in the facet's declarative region.

All facets extend a domain definition that provides a basis for defining the specification. Elements defined in the declarative region of a facet's domain are treated as if they are defined in the declarative region of the facet. This is consistent with the definition of facet extension used to define domain inclusion. In the example:

```
facet example::state-based is
  inc(x::integer)::integer is x+1;
begin
  S = integer;
  s' = inc(s);
end facet example;
```

The state type `S` and the state variable `s` are defined in the domain `state-based`. Thus they are referenced using their undecorated names without using the dot notation. Note `state-based.s` is not defined as the facet extends the domain rather than encapsulating the domain. In the example:

```
facet example::state-based is
  S::type is integer;
  inc(x::integer)::integer is x+1;
begin
  s' = inc(s);
end facet example;
```

the declaration of type `S` represents a redeclaration error because the declaration of `S` in `state-based` is treated as a local definition. Thus, the term `S = integer` is used in the previous facet definition to make the value of the state type concrete.

Packages used by a compilation unit represent the next source of scope and context information to consider when resolving a symbol. Three cases exist: (i) a label is declared locally and in a package; (ii) a label is declared in a single package; and (iii) a label is declared in multiple packages.

In the example:

```
package test::logic is
  x::integer;
end package test;

use test;
facet example::state-based is
  inc(x::integer)::integer is x+1;
begin
  t1: x' = inc(x);
end facet example;
```

the `x` instance in used in the term `t1` refers to the declaration in package `test`. It is used without the dot notation because there is only one possible source for the declaration. In the following example, multiple used packages define `x`:

```
package test0::logic is
  x::real;
end package test;

package test1::logic is
  x::integer;
end package test;

use test0;
use test1;
facet example::state-based is
  inc(x::integer)::integer is x+1;
begin
  t1: test1.x' = inc(test1.x);
end facet example;
```

Here the dot notation must be used to eliminate ambiguity in the determination of what declaration `x` refers to. If a local x is defined:

```
package test0::logic is
  x::real;
end package test;

package test1::logic is
  x::integer;
end package test;

use test0;
use test1;
facet example::state-based is
  x::type is integer;
  inc(x::integer)::integer is x+1;
```

```
  begin
    t1: x' = inc(test1.x);
  end facet example;
```

then the unqualified instance refers to the local definition. Note that definitions from packages can be referenced by explicitly using the package name.

Finally, it may be that a label is defined in the compilation unit containing facet `example`:

```
package scoping_example::logic is
  x::integer;

  package test0::logic is
    x::real;
  end package test;

  package test1::logic is
    x::integer;
  end package test;

  use test0;
  use test1;
  facet example::state-based is
    inc(x::integer)::integer is x+1;
  begin
    t1: scoping_example.x' = inc(test1.x);
  end facet example;
end package scoping_example;
```

when no local definition is present, the undecorated reference cannot be resolved to a single declaration of `x` as it is defined in the containing compilation unit and two packages. Thus, the package name must be explicitly included in the label reference. If no declaration is present except the declaration in `scoping_example`, then it may be used without the dot notation. If a local declaration is present, the local definition may always be referenced without the dot notation.

A rule of thumb for Rosetta scoping is that the local definition (in the facet or its domain) is always referenced without using the dot notation. If the local declaration is not present and only one declaration exists in the scope of the reference, then it may be used with our without the dot notation. If multiple declarations are present, then any declaration other than a local declaration must be referenced explicitly using the dot notation. Any active declaration may be referenced by using it's compilation unit name and the dot notation.

3.6 Compilation Units and Libraries

Rosetta treats each facet or facet derivative as a separate comilation unit. Thus, facets, packages, domains, components and interactions are defined as compilation units and may be processed as separate units. Even if multiple compilation units appear in a single file, they are process as individual units. Thus, a `use` or `library` clause applies only to the compilation unit immediately following.

The scope of a compilation unit is defined as the region between its declaration keyword (`facet`, `package`, `domain`, `interaction`, or `component`) and the `end` associated with the declaration keyword. In addition, the region immediatly preceding the declaration keyword back to the previous declaration is also included. Thus `use` and `library` clauses immediately preceding a compilation unit are treated as being in the scope of the

compilation unit. It bears repeating that `use` and `library` clauses apply only to the immediatly following compilation unit, not the entire file containing the compilation unit.

A library is a compilation unit with an associated logical location. Thus, a library is something that contains packages and other compilation units. Semantically, a library and a package are identical. It is only the association with a location that distinguishes a library. The implementation of libraries and dereferencing library names is implementation specific. However, the notation:

```
library ieee.ittc.ku.edu
```

refers to the library `ieee` located at `ittc.ku.edu`. The specifics of library resolution are left to the implementor. When defined in the scope of a compilation unit, any `library` definitions are used to find packages referenced in `use` clasues. Thus, the notation:

```
library ieee.ittc.ku.edu;
use floating_point;
```

specifies that library `ieee.ittc.ku.edu` should be added to the search path for the package `floating_point`.

```
%% The library notion is decidedly vague.  An alternative proposal
%% would be to prepend the library identifier to the package
%% identifier.  The only problem is determining where the library id
%% stops and the library id starts.  Linked to bug 165.
```

The outermost compilation unit of any specification must be a package. All facet, component, domain and interaction definitions must therefore be enclosed in a package. Packages may also be defined within packages, but may also form the root of a declaration hierarchy. It follows then that libraries must contain packages.

3.7 The Alarm Clock Example

In Section 3.1, the alarm clock example was introduced as an example systems level specification. In this section, the alarm clock example is examined more carefully and a structural definition introduced. The example is completely specified to provide an overall view of a Rosetta functional specification.

3.7.1 The `timeTypes` Package

`timeTypes` is a general purpose package introduced and explained in Section 3.1. It contains basic data types and functions used in the definition of the alarm clock system and structural definition. The only construct used in this definition that may require some explanation is the comprehension quantifier, `sel`. This function implements set comprehension for bunches. It does so by taking as its argument a function that maps a bunch onto the booleans and returning all domain elements for which the function is true. Thus, the statement:

```
sel(x::natural | x =< 12)
```

examines all elements of the natural numbers and returns those that are less than 12. Because its return type is bunch, its use in defining a type is perfectly legal. Further note that both `hours` and `minutes` are subtypes of `type(natural)`. This indicates that both have bunches as values, not singleton elements.

```
package timeTypes::logic is
  hours :: subtype(natural) is sel(x::natural | x =< 12);
  minutes :: subtype(natural) is sel(x::natural | x =< 59);
  time :: type is data record(h::hours; m::minutes)::time?;

  increment_time(t:: time) :: time is
    record(increment_hours(t);  increment_minutes(t));

  increment_minutes(t:: time) :: minutes is
    if t(m) < 59
       then t(m) + 1
       else 0
    end if;

  increment_hours(t::time) :: hours is
    if t(m) = 59
       then if t(h) < 12
               then t(h) + 1
               else 1
            end if
       else t(h)
    end if;
end package timeTypes;
```

3.7.2 Structural Definition

The structural definition begins by defining facets representing each of the alarm clock components. Specifically, this includes: (i) a multiplexor for defining what values are displayed; (ii) a store for internal state values; (iii) a counter for incrementing the current time; and (iv) a comparator for determining when the alarm should be sounded.

Multiplexor

The `mux` definition describes a component that determines which of its data inputs, `timeIn` or `clockTime`, should be displayed by the clock. This determination is made by examining the control signals `setAlarm` and `setTime`. Three terms are defined that select an output based on the control inputs.

```
// mux routes the proper value to the display output based on the
// settings of the setAlarm and setTime inputs.
use timeTypes;
facet mux(timeIn::input time; displayTime::output time;
          clockTime::input time; setAlarm::input bit;
          setTime::input bit)::state_based is
begin
  l1: %setAlarm => displayTime' = timeIn;
  l2: %setTime => displayTime' = timeIn;
  l3: %(-(setTime xor setAlarm)) => displayTime' = clockTime;
end facet mux;
```

Recall that the Rosetta operator `%` converts bit values into boolean values allowing bits to be used in implications directly.

Store

The `store` component is the store for the alarm clock's internal state. It operates by examining the control bit associated with each stored value. If the control bit is set, a new value is loaded from an appropriate input, or in the case of `alarmOn`, toggling the existing value. If the associated control bit is not set, then the stored value is retained.

```
// store either updates the clock state or makes it invariant based
// on the setAlarm and setTime inputs.  Outputs are invariant if
// their associated set bits are not high.
use timeTypes;
facet store(timeIn::input time; setAlarm::input bit; setTime::input bit;
            toggleAlarm::input bit;
            clockTime::output time; alarmTime::output time
            alarmOn::output bit)::state_based is
begin
  l1:: if %setAlarm
          then alarmTime' = timeIn
          else alarmTime' = alarmTime
       end if;
  l2:: if %setTime
          then clockTime' = timeIn
          else clockTime' = clockTime
       end if;
  l3:: if %toggleAlarm
          then alarmOn' = -alarmOn
          else alarmOn' = alarmOn
       end if;
end facet store;
```

Counter

The `counter` component is the simplest component involved in the definition. It states that each time the clock is invoked, its internal time is incremented.

```
// counter increments the current time
use timeTypes;
facet counter(clockTime :: inout time)::state_based is
begin
  l4:: clockTime' = increment_time(clockTime);
end facet counter
```

Comparator

The `comparator` implements the guts of the alarm clock's alarm function. It determines the appropriate value for the alarm output given the state of the alarm set bit and the values of the alarm time and the clock time. If the alarm is set and the alarm time and clock time are equal, then the alarm output is enabled. Again, the `%` operator is used to convert a boolean value into the bit value associated with the alarm output.

```
// comparator decides if the alarm should be sounded based on the
// setAlarm control input and if the alarmTime and clockTime are
```

```
// equal.
use timeTypes;
facet comparator(setAlarm:: in bit; alarmTime:: in time;
                 clockTime:: in time; alarm:: out bit)::state_based is
begin
  l1: alarm = %(setAlarm and (alarmTime=clockTime)) endif
end facet comparator;
```

3.7.3 Structural Definition

The actual structural definition instantiates each component and provides appropriate interconnections.

```
// The alarm clock structure is defined by assembling the components
// defined previously.
use timeTypes;
facet alarmClockStruct(timeIn::input time; displayTime::output time;
             alarm::output bit; setAlarm::input bit;
             setTime::input bit;  alarmToggle::input bit)::state_based is

  clockTime :: time;
  alarmTime :: time;
  alarmOn :: bit;
begin
  store_1 : store(timeIn,setAlarm,setTime,toggleAlarm,clockTime,
                  alarmTime,alarmOn);
  counter_1 : Counter(clockTime);
  comparator_1 : comparator(setAlarm,alarmTime,clockTime,alarm);
  mux_1 : mux(timeIn,displayTime,clockTime,setAlarm,setTime);
end facet alarmClockStruct;
```

3.7.4 The Specification

The final specification enclosed in a Rosetta `package` is shown in Figure 3.1.

```
package AlarmClock::logic is

  use timeTypes;
  facet mux(timeIn::input time; displayTime::output time; clockTime::input time;
            setAlarm::input bit; setTime::input bit)::state_based is
  begin
    l1: %setAlarm => displayTime' = timeIn;
    l2: %setTime => displayTime' = timeIn;
    l3: %(-(setTime xor setAlarm)) => displayTime' = clockTime;
  end facet mux;

  use timeTypes;
  facet store(timeIn::input time; setAlarm::input bit; setTime::input bit;
              toggleAlarm::input bit; clockTime::output time;
              alarmTime::output time alarmOn::output bit)::state_based is
  begin
    l1: alarmTime' = if %setAlarm then timeIn else alarmTime endif;
    l2: clockTime' = if %setTime then timeIn else clockTime endif;
    l3: alarmOn' = if %toggleAlarm then -alarmOn else alarmOn endif;
  end facet store;

  use timeTypes;
  facet counter(clockTime :: inout time)::state_based is
  begin
    l4:: clockTime' = increment-time clockTime;
  end facet counter

  use timeTypes;
  facet comparator(setAlarm:: in bit; alarmTime:: in time;
                   clockTime:: in time; alarm:: out bit)::state_based is
  begin
    l1: alarm = %(setAlarm and (alarmTime=clockTime)) endif
  end facet comparator;

  use timeTypes;
  facet alarmClockStruct(timeIn::input time; displayTime::output time;
                 alarm::output bit; setAlarm::input bit;
                 setTime::input bit; alarmToggle::input bit)::logic is
    clockTime :: time;
    alarmTime :: time;
    alarmOn :: bit;
  begin
    store_1 : store(timeIn,setAlarm,setTime,toggleAlarm,clockTime,
                    alarmTime,alarmOn);
    counter_1 : Counter(clockTime);
    comparator_1 : comparator(setAlarm,alarmTime,clockTime,alarm);
    mux_1 : mux(timeIn,displayTime,clockTime,setAlarm,setTime);
  end facet alarmClockStruct;

end package AlarmClock;
```

Figure 3.1: The complete alarm clock specification

Chapter 4

Labeling and Facet Inclusion

4.1 Labeling

Labeling is the process of assigning a name to a Rosetta item. Facet definitions, item declarations, and terms all define items and provide labels. Recall that all Rosetta items consist of a label, value and type. Where the value and type define current and possible values associated with the item, the label provides a name used to reference the item. Specifically, labels serve as names for terms, variables, constants, and facets. Any item may be referenced using it's label. This provides the basis of reflection in Rosetta allowing Rosetta specifications to reference elements of themselves.

4.1.1 Facet Labels

Facet labels name facets and provide a mechanism for controlling visibility within a facet. Facets are labeled when they are defined directly. Further, they are defined when labeled terms define new facets from existing definitions using the facet algebra described in Chapter 2 and later in Chapter 6. When that label appears within a definition, it references the defined facet.

In a traditional facet definition, the facet name following the `facet` keyword becomes the defined facet's label. Consider the following definition of `find`:

```
facet find(k::in keytype; i::in array[T]; o::out T)::state_based is
begin
  postcond1: key(o') = k;
  postcond2: elem(o',i);
end find;
```

This definition produces a facet item labeled `find` whose type is `facet` and whose value results from parsing all declarations and terms within the facet.

Items declared in and exported from a facet visible outside the facet. Such items are referenced using the standard notation *name.label* where *name* is the facet label and *label* is the item label. For example, `key(o') = k` is accessed using the name `find.postcond1`. Consider the following facet definitions:

```
facet find_power::constraint_requirements is
  power::real;
begin
  heatConst: heatDiss power <= 10mW;
end find_power;
```

```
facet find_emi::constraint_requirements is
  power::real;
begin
  emiConst: emi power;
end find_emi;
```

These facets describe electromagnetic interference (EMI) constraints and heat dissipation constraints in facets labeled `find_emi` and `find_power`. Both facets are defined over a physical variable representing power consumption. Consider the composition of these facets into a single electrical constraints facet. The new facet is defined by conjuncting the `find_power` and `find_emi` facets and providing a new label, `find_electrical`:

```
find_electrical :: facet is find_power + find_emi;
```

Note that this declaration is identical to all other Rosetta declarations. An item of type `facet` is declared and named `find_electrical`. Then, the value of `find_electrical` is constrained to be the product (conjunction) of `find_emi` and `find_power`. The declaration does not assert the facet in the current scope, but asserts that `find_electrical` references the new facet. This definition is the equivalent of saying:

```
  find_electrical :: static;
begin
  l1: find_electrical = find_emi + find_power;
```

Alternatively, a facet can be defined and referenced in a definition using the following form:

```
find_electrical:: find_power + find_emi;

%% Work on this.  It's a bit old and I think there's a much better
%% way to assert facets as terms.
```

As stated earlier, all terms are boolean valued expressions. However, in the earlier definition `+` is used to define a new facet. The distinction is this form defines a new facet and asserts it to be true in the current context. Again it is named `find_electrical` and all labels and variables defined in it are accessed using `find_electrical`, not their original names. Facet labels do not nest, but instead the new label always replaces the old. Because no export clause is specified, `power`, `heatConst` and `emiConst` are all visible using the `find_electrical.`*label* notation. Further discussion of facet inclusion and assertion is presented in Section 4.3. It suffices here to understand that a new facet is being defined and it's resulting label is the assigned term label.

4.1.2 Term Labels

Each term defined in a facet must have a label. The Rosetta syntax allows labels to be omitted, however the resulting term's label is simply undefined and may be constrained to particular value by language tools. The label identifies the term and is effectively equal to the term throughout the facet definition. All term definitions have the form:

l `:` *term* `;`

where l is the term label and *term* is the term body. Any reference to the label l in the scope of this definition refers to the term specified. Consider again the term from the earlier specification for EMI:

```
emiConst: emi power;
```

This simple definition defines a term `emiConst` that asserts `emi power`. Thus, the item referred to by `emi` instantiated with the item `power` is asserted as a true statement.

Consider the following term involving a `let` expression:

```
l1: let (x::natural = 1) in inc x;
```

The label `l1` refers to the term defined by the `let` expression. Simplifying this definition based on the definition of `let` results in the term `l1:  inc 1`. Given the classic definition of `inc`, this term is not legal as it asserts the value associated with `inc 1`. The only condition where this could be legal is if `inc` returns a boolean value or a facet.

4.1.3 Variable and Constant Labels

Labels for variable and constant items are labels for the objects they represent. Like term and facets, variables and constants are also made visible using their label. Like all other items, variables are referenced using the *name.label* notation where *name* is the facet label and *label* is the physical variable name. Consider the definition of `power` from the earlier constraint facet:

```
power::real;
```

This declaration defines a variable item referenced by the label `power`. Outside the facet definition, this variable is accessed using the notation `find.power`.

Constants work similarly. Consider the following constant definition:

```
pi :: real = 3.14159;
```

Within the scope of this definition, the label `pi` refers to the defined item whose value is the constant `3.14159`.

It is important to remember that functions, types and facets are all items that can be declared within a facet. Thus, they may all be referenced using their associated labels. Recall the definition of `increment_minutes` from the alarm clock specification:

```
increment_minutes(t::time)::minutes is
   if m(t) =< 59 then m(t) + 1 else 0;
```

This definition is interpreted exactly like the previous constant definition. The label `increment_minutes` refers to the item of type `time->minutes` whose value is specified by the constant function definition. Thus, `timeTypes.increment_minutes` is used in the body of including specifications to reference the functions. This practice of collecting declarations within facets will form the basis of the Rosetta `package` construct defined later.

4.1.4 Explicit Exporting

Visibility of labels is controlled using the `export` clause that appears in the declaration part of a facet. The convention for label exporting is that any label listed in the `export` clause is visible outside the enclosing facet using the standard *facet.label* notation. Labels not listed in the export clause are not visible and cannot be referenced. All labels within a facet can be exported using the special shorthand `export all` notation. If the export clause is omitted, then no labels from the facet are visible.

Summary: All Rosetta items are labeled and can be referenced in a specification by their associated label. Three major labeling operations are the definition of facets, the declaration of variables and constants, and the definition of terms within a facet. Any exported label may be referenced outside its enclosing facet using the canonical notation *facet.label* where *facet* is the containing facet's name and *label* is the label being access. Providing access is achieved using the `export` clause. If an `export` clause is present, all listed labels are visible and all unlisted labels are not. If `all` appears in the `export` clause, then all labels are exported. If no export clause is present, then no labels are visible outside the facet.

4.2 Label Distribution Laws

Given two labeled Rosetta items, distributive properties of labels over logical operations can be defined as follows:

Equivalence	*Name*
`l1:`*j* `and` *i* `= l1:`*j* `and l1:`*i*	`and` Distribution
`l1:(`*j* `or` *i*`) = l1:j or l1:i`	`or` Distribution
`l1:(not` *i*`) = not l1:`*i*	`not` Distribution
`l1:` *i*`; l1:` *j* `= l1:` *i* `and` *j*`;`	term Distribution
`l1::`*S*`; l1::`*T* `= l1::`*S* `and` *T*`;`	Declaration Distribution

Label distribution works consistently for any Rosetta definition. An identical label can be distributed into or factored out of any logical or collection operation regardless of the types of it's arguments. For example, labels distribute over `and` in exactly the same manner whether the arguments are expressions, facets, or terms. Let's examine distribution law in two general classes: (i) boolean operations; and (ii) term and declaration distribution.

4.2.1 Distribution Over Logical Operators

Label distribution over logical operations follows the same process regardless of the specific operation. Namely:

```
l1: A ∘ B == l1: A ∘ l1: B
```

for any logical operator `and`, `or` or `not`. The definition easily extends to cover cases for `=>`, `=` and other logical connectives. For example, the definition:

```
l1: P(x) or Q(y)
```

is equivalent to the definition:

```
l1: P(x) or l1: Q(y)
```

The semantics of each term depends on the specifics of the term value. In this case, if `P` and `Q` are both boolean valued operations, then the terms assert that the disjunction of the two properties holds. If the term types are facets, then the resulting definition defines and labels a new facet.

4.2.2 Distributing Declarations and Terms

Label distribution over semicolons occurs when two declarations or terms share the same label. Specifically, an example in the case of declarations:

```
x::integer;
x::character;
```

and in the case of terms:

```
b1: and_gate(x,y,z);
b1: constraint(p);
```

In both cases, the terms or declarations share the same label. In such circumstances, the semantics of distribution is the conjunction of the definitions. In the case of declarations:

```
v::S is c; v::T is d;
```

is equivalent to:

```
  v::S and T;
begin
  t: v=c and v=d;
```

The semantics of the declaration are such that `v` is the coproduct of `S` and `T`.[1] Specifically, any value associated with `v` has both the properties of `S` and the properties of `T`. This is not type intersection, but product in the category theoretic sense. The definition does not say that `v` is in the intersection of the original types, but says that it has a projection into both types.

The semantics of distribution over term declarations is similar. The definition:

```
b1: and_gate(x,y,z);
b1: constraint(p);
```

is equivalent to:

```
b1: and_gate(x,y,z) and constraint(p);
```

If the two conjuncts are boolean expressions, the definition of conjunction applies. If the two conjuncts are facets, then the new facet `b1` has the properties of both an `and_gate` and `constraint` simultaneously.

Summary: Label distribution is defined across all boolean operators as well as semicolons as used in declarations and term definitions. In all cases for boolean operations, identical labels distribute across operations. In all cases for semicolons, declarations and terms sharing labels can be combined into a single declaration or term resulting from the product (conjunction) of their definitions.

4.3 Relabeling and Inclusion

The ability to rename object in conjunction with label distribution laws allows definition of: (i) facet inclusion and instances; (ii) system structures; and (iii) type combination. Facet inclusion supports use of facets as units of specification modularity. If renamed when included, the new facet represents a renamed instance of the original. With inclusion, describing structural definitions becomes possible. Finally, using variable labels allows definition of type combination and interface union.

4.3.1 Facet Instances and Inclusion

Facet inclusion allows compositional definition in a manner similar to packages or modules in programming languages and theory inclusion in formal specification language. Whenever a facet label is referenced in a term, that facet is included in the facet being defined. Consider the following extended `find` specification:

[1] See Chapter 6 for details of conjunction usage.

```
facet find_primitives(T,K::subtype(univ)) is
  key(t::T)::K;
  elem(t::T,a::array(T))::boolean;
  export all;
begin logic
end find_primitives;

facet find(k::in keytype; i::in array(T); o::out T) is
begin state_based
  findpkg: find_primitives(T,keytype);
  postcond1: findpkg.key(o') = k;
  postcond2: findpkg.elem(o',i);
facet find;
```

In previous `find` specifications, definitions for `key` and `elem` remain unspecified. In this example, the facet `find_primitives` defines those operations. The `find` facet includes a copy of `find_primitives` in the term labeled `findpkg`. Semantically, this term includes a copy of `find_primitives` and re-labels the facet with `findpkg`.

In the resulting definition, elements of the newly renamed facet are accessed using `findpkg` as their associated facet name. Specifically, the `elem` and `key` functions defined in `find_primitives` are referenced using the `findpkg.elem` and `findpkg.key` notations respectively. `findpkg` is said to be an instance of the original facet because each newly named copy is distinct from the original. This includes physical variables as well as terms. Thus, two renamed copies of the same facet will not inadvertently interact. This is exceptionally important when defining structural definitions where many instances of the same component may be required.

Alternatively, a facet or package may be referenced in a `use` clause to make their definitions visible in the current scope. Consider the following definition:

```
facet find_primitives(T,K::subtype(univ)) is
begin requirements
  key(t::T)::K;
  elem(t::T,a::array(T))::boolean;
begin logic
  ...
end find_primitives;

use find_primitives(T,keytype);
facet find(k::in keytype; i::in array(T); o::out T) is
begin state_based
  postcond1: key(o') = k;
  postcond2: elem(o',i);
facet find;
```

Here the `use` clause makes all labels defined in `find_primitives` visible in the current scope. When using this approach, the "." notation is no longer necessary as the functions `key` and `elem` are now visible. In most situations, this is the desired mechanism for packaging and using definitions. The special `package` definition provides a facet construct specifically for this purpose. Please see Chapter 3 and Chapter 6 for more details on the semantics and use of packages.

4.3.2 Structural Definition

System structure is defined using facet inclusion and labeling in the same manner as defined previously. Facets representing components are included and interconnected by instantiating parameters with common objects.

Labeling provides name spaced control and supports defining multiple instances of the same component. Consider the following specification of a two bit adder using two one bit adders:

```
facet one_bit_adder(x,y,cin::in bit; z,cout::out bit) is
  delay::real;
  export delay;
begin state_based
  l1: z'= x xor y xor cin;
  l2: cout' = x and y;
end one_bit_adder;

facet two_bit_adder(x0,x1,y0,y2::in bit; z0,z1,c::out bit) is
  delay::real;
  cx::bit;
  export delay;
begin logic
  b0: one_bit_adder(x0,y0,0,z0,cx);
  b1: one_bit_adder(x1,y1,cx,z1,c);
  l1: delay = b0.delay+b1.delay;
end two_bit_adder;
```

Facet interconnection is achieved by sharing symbols between component instances. When a facet is included in the structural facet, formal parameters are instantiated with objects. When objects are shared in the parameter list of components in a structural facet, those components share the object. Thus, information associated with the object are shared between components. The *two_bit_adder* specification includes two copies of `one_bit_adder`. Parameters of the two adders are instantiated with parameters from `two_bit_adder` to associated signals with those at the interface. The internal variable `cx` is used to share the carry out value from the least significant bit adder with the carry in value from the most significant bit adder.

When the two `one_bit_adder` instances are included in the `two_bit_adder` definition, they are labeled with `b0` and `b1`. The result is that the first `one_bit_adder` is renamed `b0` and the second `b1`. The implication of the renaming is that the `delay` physical variable associated with the adder definition is duplicated. *I.e.* the values `b0.delay` and `b1.delay` are available for reference and represent distinct objects. Without renaming using labels, both `one_bit_adder` instances would refer to the same physical variable, `one_bit_adder.delay`. This is not appropriate as the adders should be distinct. The same result can be achieved using parameter for delay. In large specifications including parameters for physical variables representing constraint specifications becomes cumbersome. Further, delay is not a parameter but a characteristic of the component.

After including the two adder instances, the value of `delay` in the `two_bit_adder` specification is constrained to be equivalent to the sum of the `one_bit_adder` delays. In this way, it is possible to specify composition of non-behavioral characteristics across architectures.

Logical operators are defined to distribute across structure components. Assume the following facets defining power constraints on a one bit adder and an architecture defining constraints on a two bit adder composed of two one bit adders:

```
facet one_bit_adder_const is
   power::posreal;
begin constraints
   p0: power <= 5mW;
end one_bit_adder_const;
```

```
facet two_bit_adder_const is
  power::posreal;
begin constraints
  b0: one_bit_adder_const;
  b1: one_bit_adder_const;
  p0: power = b0.power + b1.power;
end two_bit_adder_const;
```

The facet conjunction `two_bit_adder = two_bit_adder and two_bit_adder_const` is equivalent to:

```
facet two_bit_adder(x0,x1,y0,y1::in bit; z0,z1,c::out bit) is
  delay::real;
  power::posreal;
  cx::bit;
  export delay,power;
begin logic
  b0: one_bit_adder(x0,y0,0,z0,cx);
  b0: one_bit_adder_const;
  b1: one_bit_adder(x1,y1,cx,z1,c);
  b1: one_bit_adder_const;
  d0: delay = b0.delay+b1.delay;
  p0: power = b0.power+b1.power;
end two_bit_adder;
```

This definition results from the definition of facet conjunction. The term set is simply the set of all defined terms in the two facets.

This definition results from the distributivity of labeling. The same result holds for disjunction, implication and logical equivalence. Application of label distribution results in:

```
facet two_bit_adder(x0,x1,y0,y1::in bit; z0,z1,c::out bit) is
  delay::real;
  power::posreal;
  cx::bit;
  export delay,power;
begin logic
  b0: one_bit_adder(x0,y0,0,z0,cx) and one_bit_adder_const;
  b1: one_bit_adder(x1,y1,cx,z1,c) and one_bit_adder_const;
  d0: delay = b0.delay+b1.delay;
  p0: power = b0.power+b1.power;
end two_bit_adder;
```

Here, conjunction distributes across the structural definition. Proper label selection allowed power constraints to be associated with each component. The result can be viewed as either the conjunction of a power and functional model or the composition of two component models both having constraint and functional models.

```
%% Working Here %%
```

Example 13 (Structural Example) *Consider the following facets:*

```
facet sort(x::in array(T); y::out array(T)) is
begin state_based
  l1: permutation(x,y');
  l2: ordered(y');
end sort;

facet binsearch(k::in keytype; x::in array(T); y:out T) is
begin state_based
  l1: ordered(y);
  l2: member(k, dom(x)) => member(y', dom(x)) AND key(y')=k;
end binsearch;
```

```
facet find_structure(k::in keytype; x::in array(T); y:out T) is
  buff::array(T);
begin logic
  b1: sort(x,buff);
  b2: binsearch(k,buff,t);
end find_structure;
```

The sort *and* binsearch *facets define requirements for sorting and binary search components. The* find-structure *facet defines a* find *architecture by connecting the two components. The state variable* buff *is shared by the binary search and sorting components and facilitates sharing information. Note that* new *does not generate a new copy of* buff *because new is called on both* sort *and* binsearch *before parameters are instantiated. Thus, the same object* buff *is references in the terms of both components and constrained by those terms.*

The following collection of examples are designed to demonstrate several configurations of a simple transceiver system. The following represent simple specifications of a transmitter and receiver used throughout the examples:

```
use signal_processing_requirements(T);
facet tx (data::in T; output::out T) is
begin state_based
  l1: output'=encode(data);
end tx;
```

```
use signal_processing_requirements(T);
facet rx (data::out T; input::in T) is
begin state_based
  l1: data'=decode(input);
end rx;
```

These specifications assume the following domain facet for signal processing:

```
facet signal_processing_requirements(T::subtype(univ)) is
  encode(t::T)::T;
  decode(t::T)::T;
  export encode,decode;
begin logic
  encode_decode: forall(t::T | decode(encode(t))=t);
end signal_processing_requirements;
```

Recall that in the presence of an `export` statement, only specified labels are visible outside the facet. Here, a facet is used rather than a facet to allow specification of the `encode_decode` axiom that states the inverse relationship between the encode and decode functions. The `use` clause makes the functions visible and available to the transmitter and receiver specifications. The axiom is not visible, but does remain present in the definition.

Example 14 (Transmit/Receive Pair) *The following defines the simplest possible communications channel transmitting and receiving encoded, baseband signals:*

```
facet tx_rx_pair (data_in::in T; data_out::out T) is
  channel::T;
begin logic
  txb: tx(data_in,channel);
  rxb: rx(data_out,channel);
end tx_rx_pair;
```

The resulting component represents a perfect transmitter receiver pair where input data is perfectly transmitted to an output data stream.

Example 15 (Transceiver) *The following defines a simple transceiver combining the transmitter and receiver functions into a single component:*

```
facet transceiver (data_in::in T; data_out::out T;
                   out_chan::out T; in_chan::in T) is
being structural
  txb: tx(data_in, out_chan);
  rxb: tx(data_out, in_chan);
end transceiver;
```

Note that in this specification, the transmitter and receiver do not interact. They simply operate in parallel on independent data streams.

Example 16 (Transceiver Pair) *Now consider a transceiver pair constructed from two transceivers:*

```
facet trx_pair (data_in1, data_in2::in T;
                data_out1, data_out2::out T) is
begin logic
  chan1,chan2::T;
  trx1: transceiver(data_in1,data_out1,chan1,chan2);
  trx2: transceiver(data_in2,data_out2,chan2,chan1);
end trx_pair;
```

Example 17 (Transceiver Pair - Common Channel) *An adaptation of a transceiver pair is one where transmission from both devices occurs on the same channel. Here, only one channel parameter is defined:*

```
facet trx_pair (data_in1, data_in2::in T;
                data_out1, data_out2::out T) is
  chan::T;
begin logic
  trx1: transceiver(data_in1,data_out1,chan,chan);
  trx2: transceiver(data_in2,data_out2,chan,chan);
end trx_pair;
```

Example 18 (Low Power Transmitter) *Define a new facet for transmitters and receivers that constrains power consumption:*

```
facet low_power is
  power::real;
begin constraints
  p0: power =< 10MW;
end low_power;

%% Cindy, look at this syntax...
```

One can now define a low power transmitter as:

```
tx_low_power(data::in T; output::out T)::facet is tx and low_power;
```

In this definition, a new facet called ***tx_low_power*** *is defined that is the composition of the transmitter functional facet and the low power constraints.*

```
%% The definition below should be contained in a facet definition as
%% it's really a function.
```

Example 19 (Transmitter Configuration) *Define a new facet for high power transmission:*

```
facet high_power is
  power::real;
begin constraints
  l1: power =< 100Mw;
end high_power;
```

Now define a configurable device that represents either the high or low power version:

```
tx_power_select(select::boolean)::facet is
   tx(data,output) and
   if select
      then low_power
      else high_power
   end if;
```

When the `select` *parameter is true, then the* `tx` *facet is composed with the* `low_power` *constraints facet. Otherwise, the* `tx` *facet is composed with the high power constraint facet.*

Chapter 5

Abstract Syntax, Typing and Static Consistency

The Rosetta typing relation has the form $p::\varphi$ and is read "p is of type φ" and asserts that the value associated with p must always be a value from the set resulting from the evaluation of φ. As φ is an arbitrary expression whose value is a set, types in Rosetta are first-class items and type checking cannot be performed conservatively for the full language.

Definition 1 (Possible Types) *The set $\mathcal{T}$ is the set of all possible Rosetta types and is defined as the set of all Rosetta sets:*

$$\mathcal{T} == \texttt{set}(\texttt{universal})$$

The concrete syntax for $\mathcal{T}$ is the Rosetta reserved word `type`.

Note that $\{\} in \mathcal{T}$, thus it is possible to have an empty type. Consistency theorems must account for this in determining static consistency of declarations.

Type Rule 1 (Of Type Definition) *φ is of type τ if it is an element of the set τ:*

$$\frac{\varphi \texttt{ in } \tau}{\varphi::\tau} \textsc{ElementIsType}$$

Type Rule 2 (Subtype Definition) *σ is a subtype of τ if σ is a subset of τ.*

$$\frac{\sigma =< \tau}{\sigma <: \tau} \textsc{SubsetIsSubtype}$$

Type Rule 3 (Subtype Definition) *If σ is the type of an expression and σ is a subtype of τ, then the expression is also of type τ.*

$$\frac{\varphi::\sigma \quad \sigma <: \tau}{\varphi::\tau} \textsc{Subtype}$$

5.1 Expressions

A Rosetta expression is constructed using operators and variables as defined in the current scope. Predefined operators and types are defined in Chapter 2 and form the basis of the Rosetta expression syntax. All Rosetta expressions are recursively defined in terms of unary and binary operations. Parenthesized operations have the highest priority followed by unary operations and binary operations in traditional fashion.

5.1.1 Expression Abstract Syntax

Expressions are formed using unary operations, binary operations, grouping operations, and with function calls. Given that p and q are meta-variables that range over legal Rosetta labels, the following general rules are used to define the abstract syntax of Rosetta expressions:

```
%% Still need an abstraction rule.
```

$$
\begin{array}{rcl}
\varphi & ::= & p \mid \text{pre}\varphi \mid \varphi_1 \circ \varphi_2 \mid \varphi\text{post} \mid f(\varphi_0, ..., \varphi_n) \mid (\varphi) \mid \\
& & \{\varphi_1, ..., \varphi_n\} \mid \{*\varphi_1, ..., \varphi_n*\} \mid [\varphi_1, ..., \varphi_n] \mid \\
& & \mathcal{Q}(p :: \varphi_1 \mid \varphi_2) \\
& & \texttt{if } \varphi_0 \texttt{ then } \varphi_1 \texttt{ else } \varphi_2 \texttt{ end if} \\
& & \texttt{let } p :: \varphi_1 \texttt{ be } \varphi_2 \texttt{ in } \varphi_3 \texttt{ end let} \\
& & \texttt{case } p \texttt{ of } \sigma_0 -> \varphi_0 \mid ... \mid \sigma_n -> \varphi_n \texttt{ end case} ;
\end{array}
$$

where preφ, φpost, and $\varphi_1 \circ \varphi_2$ are applications of prefix, postfix, and infix operations respectively; f is a function or facet label; and $\mathcal{Q}$ is a quantifier operation. The `if` , `let` and `case` forms are abstract syntax for associated functions. It is important to realize that these forms are expressions, not traditional programming language statements.

Precedence for Rosetta unary and binary operations follow the canonical style. The following table lists Rosetta operators in tabular form:

Operator	*Type*
`()`	Grouping
`{},{* *}[]`	Formers
`-, not, %, $, #, ~`	Unary operations
`^ , in`	Power and membership
`*, /, **`	Product operations
`+, -, ++, --, &`	Sum operations
`<, =<, >=, >, <<, >>`	Relational operations
`=, /=`	Equality operations
`min, and, nand`	Boolean product
`max, nmax, nmin, or, nor, xor, xnor, <=, =>`	Boolean Sum
`==`	Equivalence

Table 5.1: Precedence table for pre-defined Rosetta operations.

5.1.2 Expression Typing

Expression typing is achieved by examining the abstract syntax of expressions. Two cases are defined: (i) monolithic functions; and (ii) selective union of functions. In the monolithic case, the type of the function application is the range of the function if parameters instantiations are well-typed.

Type Rule 4 (Function Application) *Function application generates a value of the function return type when instantiated with parameters of the correct type. The type inference rule shows only a single parameter and generalizes to functions of multiple parameters.*

$$\frac{f :: <*(\mathsf{p}::\tau_1)::\tau_2*> \quad \varphi::\tau_1}{f(\varphi) :: \tau_2} \text{ Type-Function}$$

In the selective union case, there are two rules. Given $f = (f_0||f_1)$, the type of $f(\sigma)$ is the type of $f_0(\sigma)$ if $f_0(\sigma)$ is well-typed. If $f_0(\sigma)$ is not well-typed, then the type of $f(\sigma)$ is the type of $f_1(\sigma)$. If f_1 is defined using selective union the process repeats recursively until all functions involved in the selective union are not well-typed for f or a well-typed element is found.

Type Rule 5 (Selective Union Application) *The rules for typing functions defined using selective union select the type of the first function such that the actual parameter is an element of its type. The rules are defined so that the type of the function application* must *take the type of the first applicable function.*

$$\frac{f :: (f_0(p :: \tau_1) :: \tau_2||f_1) \quad \varphi :: \tau_1}{f(\varphi)::\tau_2} \text{ Type-SelectiveUnion1}$$

$$\frac{f :: (f_0(p :: \tau_1) :: \tau_2||f_1) \quad not(\varphi :: \tau_1) \quad f_1(\varphi) :: \tau_3}{f(\varphi)::\tau_3} \text{ Type-SelectiveUnion2}$$

Typing function application allows definition of typing theorems for other expression forms. The following forms can be typed by direct transformation into function application:

$$\frac{\phi\_\_(\varphi)::\tau}{\phi\varphi::\tau} \text{ Type-Prefix}$$

$$\frac{\_\_ \circ \_\_(\varphi_1, \varphi_2)::\tau}{\varphi_1 \circ \varphi_2::\tau} \text{ Type-InFix}$$

$$\frac{\_\_\phi(\varphi)::\tau}{\varphi\phi::\tau} \text{ Type-Postfix}$$

$$\frac{\varphi::\tau}{(\varphi)::\tau} \text{ Type-Parenthesis}$$

Technically the form (φ) is not a function application, but is included here due to its similarity to function application.

Set, sequence and multiset formers are also special forms of function application with variable numbers of arguments. The following theorems are derived for the three former special forms.

Type Rule 6 (Set, Sequence and Multiset Formers) *Formers generate constant data containers from their arguments. Thus, the type of a formed set, multiset or sequence is the shared type if its constituent elements.*

$$\frac{\varphi_0{::}\tau \quad \varphi_1{::}\tau \quad ... \quad \varphi_n{::}\tau}{\{\varphi_0, \varphi_1, ...\varphi_n\} :: \texttt{set}(\tau)} \text{ Type-SetFormer}$$

$$\frac{\varphi_0{::}\tau \quad \varphi_1{::}\tau \quad ... \quad \varphi_n{::}\tau}{\{*\varphi_0, \varphi_1, ...\varphi_n*\} :: \texttt{multiset}(\tau)} \text{ Type-MultisetFormer}$$

$$\frac{\varphi_0{::}\tau \quad \varphi_1{::}\tau \quad ... \quad \varphi_n{::}\tau}{[\varphi_0, \varphi_1, ...\varphi_\texttt{n}]{::}\texttt{sequence}(\tau)} \text{ Type-SequenceFormer}$$

Quantifiers represent another special form of function application. Recall that the semantics of a quantifier can be expressed as a higher order function. Specifically, for any quantifier function $\mathcal{Q}$, the following equivalence is defined:

$$\mathcal{Q}(\sigma{::}\tau \mid \varphi) \quad == \quad \mathcal{Q}(<*(\sigma{::}\tau){::}\texttt{boolean is } \varphi*>)$$

Using this equivalence, theorems for quantifier type inference can be defined. Forall q `in` $\{\texttt{forall}, \texttt{exists}\}$:

$$\frac{\phi(x{::}\varphi){::}\texttt{boolean}}{q(\phi){::}\texttt{boolean}} \text{ Type-QuantifierBasic}$$

$$\frac{\tau{::}\mathcal{T} \quad \varphi{::}\texttt{boolean}}{\phi(x{::}\tau \mid \varphi){::}\texttt{boolean}} \text{ Type-Quantifier}$$

Forall q `in` $\{\texttt{sel}, \texttt{min}, \texttt{max}, +, -\}$:

$$\frac{\phi(x{::}\tau){::}\texttt{boolean}}{q(\phi){::}\texttt{set}(\tau)} \text{ Type-FilterBasic}$$

$$\frac{\tau{::}\mathcal{T} \quad \varphi{::}\texttt{boolean}}{\phi(x{::}\tau \mid \varphi){::}\texttt{set}(\tau)} \text{ Type-Filter}$$

The `if` form is viewed as a function defined using selective union and is typed as such using the following theorem:

Type Rule 7 (If Expressions) *Typing* `if` *expressions is done in the classical fashion. If the* `then` *and* `else` *clauses share a common type, T, and the condition is* `boolean`*, then the type of the clause is the type, T.*

$$\frac{\varphi_1{::}\texttt{boolean} \quad \varphi_2{::}\tau \quad \varphi_3{::}\tau}{\texttt{if } \varphi_1 \texttt{ then } \varphi_2 \texttt{ else } \varphi_3 \texttt{ end if } ::\tau} \text{ Type-If}$$

Technically, the `let` expression is not a function application. However, a typing rule for `let` expressions is easily defined by treating the expression as a function:

Type Rule 8 (Let Expression) *The* `let` *expression provides a mechanism for defining local scope. Thus, the type of a* `let` *expression is the type of the expression it encapsulates.*

$$\frac{\varphi_2::\tau \quad \varphi_1 \texttt{ in } \tau_1}{\texttt{let } p::\tau_1 \texttt{ be } \varphi_1 \texttt{ in } \varphi_2 \texttt{ end let } ::\tau} \text{ TYPE-LET}$$

The `case` expression is a special syntax for selective union. Using the selective union typing rule, a theorem for the `case` form is defined:

Type Rule 9 (Case Expressions) *The* `case` *expression provides a specialized mechanism for defining complex conditions. Although not technically necessary, it can substantially simplify specifications. Like the* `if` *expression, the type of a* `case` *expression is the shared type of its conditional statements.*

$$\frac{\varphi_0::\tau \quad \varphi_1::\tau \ ... \ \varphi_n::\tau \quad p::\varphi_0 + \varphi_1 + ... + \varphi_n}{\texttt{case } p \texttt{ of } \varphi_{c0} -> \varphi_0 \mid \varphi_{c1} -> \varphi_1 \mid \ ... \ \mid \varphi_{cn} -> \varphi_n \texttt{ end case } ::\tau} \text{ TYPE-CASE}$$

5.2 Declarations

Declarations appear in parameter lists for functions, `let` expressions, and facets, and the declarative regions of facets, domains and packages. For a specification to be consistent, all declarations must be consistent. Defining an abstract syntax for declarations allows defining conditions for well-typed declarations.

5.2.1 Declaration Abstract Syntax

Declarations, as described in Chapter 2, take the general form:

$$\textit{declaration} ::= p::\varphi_1 \ \mid \ p::\varphi_1 \texttt{ is } \varphi_2$$

The context of a declaration, Γ, is the set of consistent declarations, $p::\varphi$, in the scope of the declaration. If any declaration in Γ is not typeable, then the declaration is not typeable.

Type Rule 10 (Variable Type) *The type of a variable is equal to the type of its most recent declaration. Given a sequence of declarations,* Γ*, and an item label,* σ*, the following holds:*

$$\frac{\sigma::\tau \in_\gamma \Gamma}{\Gamma \vdash \sigma::\tau} \text{ TYPEITEM}$$

where $\sigma::\tau \in_\gamma \Gamma$ *if* $\sigma::\tau$ *is the first declaration of* σ *moving right to left in the sequence.*

5.2.2 Declaration Consistency

Consistency Theorem 1 (Variable Consistency) *A declaration of the form:*

$$p::\tau$$

is considered sound if p has not been declared in the current scope and τ *is a nonempty set:*

$$\Gamma \vdash \tau::\mathcal{T} \texttt{ and } \varphi \neq \{\}$$

where Γ *is the context of the declaration.*

Consistency Theorem 2 (Constant Consistency) *A declaration of the form:*

$$p{::}\tau \texttt{ is } \varphi$$

is well-typed if $p{::}\tau$ *is well-typed and* $\varphi{::}\tau$*:*

$$\Gamma \vdash \tau{::}\mathcal{T} \texttt{ and } tau \neq \{\} \texttt{ and } \varphi{::}\tau$$

where Γ *is the context of the declaration.*

Function declaration is similar. Consider the expanded declaration form:

$$f{::}{<}{*}(\texttt{p} :: \tau_1){::}\tau_2{*}{>} \texttt{ is } {<}{*}(\texttt{q} :: \tau_3){::}\tau_4 \texttt{ is } \varphi{*}{>}$$

This declaration is well-typed as defined in Chapter 2 if:

$${<}{*}(\texttt{q} :: \tau_3){::}\tau_4 \texttt{ is } \varphi{*}{>}{::}{<}{*}(\texttt{p} :: \tau_1){::}\tau_2{*}{>}$$

where $f{::}g$ is defined for functions in the same manner as for other items and f `in` g has the definition provided in Chapter 2.

Consistency Theorem 3 (Function Consistency) *Given a function declaration of the form:*

$${<}{*}(\texttt{q} :: \tau_3){::}\tau_4 \texttt{ is } \varphi{*}{>}{::}{<}{*}(\texttt{p} :: \tau_1){::}\tau_2{*}{>}$$

the well-typed property is specified by the following theorem:

$$\Gamma, p{::}\tau_3 \vdash \varphi{::}\tau_4$$

where Γ *is the context of the declaration.*

Consistency Theorem 4 (Direct Declaration Consistency) *The expanded declaration is more typically defined using the direct declaration:*

$$f(p{::}\tau_1){::}\tau_2 \texttt{ is } \varphi$$

The well-typed property can be expressed for the direct declaration form as:

$$\Gamma, p{::}\tau_1 \vdash \varphi{::}\tau_2$$

where Γ *is the context of the declaration.*

5.3 Terms

A Rosetta term is a labeled expression that appears within the scope of a `begin-end` pair within a facet. All terms are asserted as true within the scope of the facet. Note that simply because a term is boolean valued does not imply the term cannot represent an operational specification. It simply says that the statements made within the term are declared to be true.

5.3.1 Term Abstract Syntax

The general format for a Rosetta term is a label, followed by an expression, terminated by a semicolon. Specifically, given that l is a meta-variable representing legal Rosetta labels, φ_b and φ_f are meta-variables representing `boolean` and `facet` typed expressions, the form of a term is:

$$\begin{array}{rcl} term & ::= & \varphi_b; \mid l : \varphi_b; \mid l : \varphi_f; \mid \\ & & \texttt{let}\ p{::}\varphi\ \texttt{in}\ term^*\ \texttt{end let}\ ; \end{array}$$

Thus, a term can be a labeled or unlabeled boolean expression, a labeled facet expression, or a collection of terms encapsulated in a `let` form. Note that the intent of the term `let` form is the same as the `let` expression, the semantics differs substantially.

For example, the following term states that `inc 3` is equal to `4`:

```
l1: inc(3) = 4;
```

Term label is `l1`, the term is `inc(3) = 4` and the semicolon terminates the term definition. The function of the semicolon is to terminate a labeled expression. Thus, the specification fragment:

```
l1: inc(3) = 4;
l2: forall(x::{1,2} | x<4);
```

defines two terms with labels `l1` and `l2` and term expressions `inc(3) = 4` and `forall(<*x::1,2 ->x<4 *>)` respectively. In contrast:

```
l1: inc(3) = 4
l2: forall(x::{1,2} | x<4);
```

is illegal as `l2:` is not an operation in the specification grammar and the first term is not terminated by a semicolon.

5.3.2 Term Consistency

Consistency Theorem 5 (Term Consistency) *Given a declaration context,* Γ*, a term,* t*, is well-typed if:*

$$\Gamma \vdash t{::}\texttt{boolean or}\ t{::}\texttt{null}$$

where `null` *is the base facet type.*

```
%% Operators do in fact distribute over semicolons.  This is,
%% unfortunately wrong and needs to be corrected.
```

Terms delineated by semicolons in the body of a specification are simultaneously true and form a set of terms associated with the facet. This the Term Consistency theorem status that the set of terms must be consistent with respect to the facet domain. A facet is consistent if and only if its domain, set of terms, and declarations are mutually consistent.

No term's semantic meaning can be inferred without reference to the including facet's domain. For example, the following definitions seem quite similar, but with proper interpretation mean quite different things. The following examples demonstrate this fact by showing how similarly defined terms have different semantics based on the definition domain. In each case, reference to the VHDL signal assignment semantics is mentioned to aide in understanding what is being specified. The various domains are explained in Chapter 7.

The following term asserts that `x` is equal to `f` of `x`:

```
begin
  l1: x = f(x);
  ...
```

The domain for this term is `null`, referring to Rosetta's basic mathematical system with no concept of state, time or computation model. Thus, `x = f(x)` is an assertion about `x` that must always hold. This domain is frequently termed the *monotonic* domain because change is not defined. If `f(x)` is not equal to `x`, then this term is inconsistent and the specification is inconsistent.

The following term asserts that `x` in the next state is equal to `f` of `x` in the current state:

```
begin
  l1: x' = f(x);
  ...
```

The `state-based` domain provides the basics of axiomatic specification. Specifically, the notion of current and next state. `x'` refers to the value of `x` in the state resulting from evaluating the facet's function. `x` refers to the value in the current state. This specification fragment has roughly the same semantics as an assignment statement as it specifies that `x` in the next state is equal to `f x`. Thus, if `x /= f(x)`, no inconsistency results. It is interesting to note that this statement is quite similar in nature to a basic signal assignment in VHDL. Specifically in VHDL:

```
  x <= f(x);
```

The following term asserts that `x` at current time plus `5e-6` is equal to `f` of `x` in the current state:

```
begin
  l1: x@(t+5e-6) = f(x);
  ...
```

This specification is quite similar to the previous specification in that the value of `x` in some future state is equal to `f(x)`. It differs in that the specific state is defined temporally. Specifically, in the state associated with `5e-6` units in the future, `x` will have the value associated with `f(x)` where the argument to `f` is the value of `x` in the current state. Again, this definition bears some resemblance to VHDL signal assignments. This time, a wait statement is specified in conjunction with the signal assignment:

```
  x <= f(x) after 5ms;
```

Other domains and semantics are available for discrete time, constraints and mechanical specifications. The intent here is to demonstrate only the relationship between a term and its associated domain.

Another example uses classical axiomatic specification to define a function. The function `inc` has been used repeatedly as an example of constant function definition. Here, the function is defined as an abstract function and constrained using a term in the specification body:

```
  inc(x::integer)::integer;
begin
  incdef: forall(x::integer | inc(x) = x + 1);
  ...
```

The definition states that for every integer, `x`, calling the function `inc` on `x` is equal to adding `1` to `x`. This is semantically equivalent to the previous definition, however it is more difficult for an interpreter to evaluate.

An alternate definition assigns a specific function to the function variable defined:

```
  inc(x::integer)::integer;
begin
  incdef: inc = <* (x::integer)::integer is x + 1 *>;
```

Semantically, this is identical to the standard definition. Like the previous definition, it is not as easy for the compiler to determine the value of `inc`.

The `let` form is also used to form terms. Consider the following definition:

```
l1: let (x::integer is a+1) in f(x,5);
```

When the `let` form is evaluated, the following term results:

```
l1: f((a+1),5);
```

The `let` form can be used to create local declarations that apply across several terms. Thus, a `let` form may define several terms and is never given a label explicitly. The following definition defines a single item, `c`, to be the result of evaluating a function. This item can then be referenced by all terms in the defining `let` form:

```
...
let c::real be 2*pi*r in
  t1: c>0.0;
  t2: v=c*h;
end let;
...
```

The terms `t1` and `t2` are semantically equivalent to replacing `c` with `2*pi*r` in each expression and removing the `let` form. Note that either or both labels may be ommitted.

The semantics of the `let` form over terms use the `let` form over expressions. Given the following definition:

$$\texttt{let}\ p{::}\tau\ \texttt{be}\ \varphi\ \texttt{in}\ t_0 : \varphi_0;\ t_1 : \varphi_1;\ \ldots\ t_n : \varphi_n;\ \ \texttt{end let}$$

its semantic equivalent is:

$$
\begin{aligned}
t_0 &: \texttt{let}\ p{::}\tau\ \texttt{be}\ \varphi\ \texttt{in}\ \varphi_0\ \texttt{end}\ \texttt{let};\\
t_1 &: \texttt{let}\ p{::}\tau\ \texttt{be}\ \varphi\ \texttt{in}\ \varphi_1\ \texttt{end}\ \texttt{let};\\
&\ldots\\
t_n &: \texttt{let}\ p{::}\tau\ \texttt{be}\ \varphi\ \texttt{in}\ \varphi_n\ \texttt{end}\ \texttt{let};
\end{aligned}
$$

Summary: A term is a `boolean` or `facet` typed expression defined within the body of a facet. Each term is separated by a semicolon and is simultaneously true within the facet scope. Terms must be evaluated with respect to the domain associated with their enclosing facet to be fully interpreted.

5.4 Facets

With definitions for the type relation defined for declarations and terms, it becomes possible to define the well-typedness of facets. A facet is a collection of parameters, declarations, terms, and a domain. A facet is well-typed if its components are well-typed in their scope.

5.4.1 Facet Abstract Syntax

The abstract syntax for a facet value can be defined as:

$$
\begin{aligned}
facet \quad ::= \quad & \texttt{facet}(\gamma_p){::}\gamma_d;\ |\\
& \texttt{facet}(\gamma_p){::}\gamma_d\ \texttt{is}\\
& \ \gamma_i;\\
& \ \gamma_e;\\
& \texttt{begin}\\
& \ \gamma_t;\\
& \ \texttt{end facet}
\end{aligned}
$$

where: (i) γ_p is the set of parameter declarations; (ii) γ_d is the set of declarations exported from the domain, d; (iii) γ_i is the set of local variable and constant declarations; (iv) γ_t is the set of term declarations; and (v) γ_e is the set of exported declarations from γ_i. For any context, $\gamma(p)$ defines the subset of declarations preceding p in the declaration sequence. If p does not appear in the sequence, then $\gamma(p) = \gamma$. The purpose of parameterizing γ in this way allows for declarations to appear in the context of declarations preceding it in its declarative region.

With definitions for declarations and terms, the consistency of a facet may be defined. Abstractly, a facet is a collection of parameter declarations (γ_p), a domain (γ_d), local item declarations ($gamma_i$), and term declarations ($gamma_t$). Declarations may also be provided by the context of the definition (γ_c). Specifically, `use` clauses that import symbols from packages.

5.4.2 Facet Consistency

Consistency Theorem 6 (Facet Consistency) *A facet is statically consistent if: (i) its domain is consistent; (ii) any included facets are consistent; and (iii) the following theorems can be verified:*

$$\forall(q{::}\varphi) \in \gamma_p \cdot \gamma_c, \gamma_p(q), \gamma_d \vdash \varphi \in \mathcal{T} \texttt{ and } \varphi \neq \{\}$$
$$\forall(q{::}\varphi) \in \gamma_i \cdot \gamma_c, \gamma_p, \gamma_d, \gamma_i(q) \vdash \varphi \in \mathcal{T} \texttt{ and } \varphi \neq \{\}$$
$$\forall(q{::}\varphi_1 \texttt{ is } \varphi_2) \in \gamma_i \cdot \gamma_c, \gamma_p, \gamma_d, \gamma_i(q) \vdash \varphi_1 \in \mathcal{T} \texttt{ and } \varphi_1 \neq \{\} \texttt{ and } \varphi_2{::}\varphi_1$$
$$\forall(l : \varphi;) \in \gamma_t \cdot \gamma_c, \gamma_p, \gamma_d, \gamma_i \vdash \varphi \in \texttt{boolean} + \texttt{null} \texttt{ and } \varphi \neq \{\}$$

Domains and packages follow directly from this definition as they are specializations of facet constructs.

Chapter 6

Facet Items, Facet Types and The Facet Algebra

```
%% Needs a good introductory paragraph

%% Still need to define syntax for the facet composition operators.
%% Also need some reference to the formal semantics.  Put a small
%% section in on the theory calculus from the interactions white
%% paper.
```

6.1 Facet Items

Because Rosetta is a reflective language, specification structures such as facets are items defined in the language. Like any other item, a facet item consists of a label, type and value. A facet's type is a set of facets that define the possible values of the facet item. A facet's value is simply an element of that set. The semantics of facet operations and types are defined in Chapter 8, but are included here as they are treated in the same manner as any Rosetta item.

In Chapter 3 a format for defining facets directly is provided. Specifically, the following defines a simple facet that increments an input value and outputs it:

```
facet inc(i::in integer; o::out integer)::state_based is
begin
  l1: o'=i+1;
end inc;
```

Treated as a Rosetta item, the label of this facet is `inc`, the type `state_based`, and the value the algebra associated with the definition.

6.2 Facet Types and Subtypes

A facet's type is defined by the use of a domain in its definition. For example, a facet `f` defined as follows:

```
facet f(x::in integer, z::out integer)::finite_state is
begin
  t1: ... ;
  t2: ... ;
end f;
```

is considered to be of type `finite_state`. Thus, the declaration:

```
f::finite_state;
```

could be used to declare the facet signature. Note that the details defined in terms and declarations from the previous facet are not included in this declaration.

The facet type defined by a facet domain is the collection of all consistent facets that are defined based on the domain. In effect, every facet that references a specific domain is an element of that domain.

Facet subtypes provide a *domain polymorphism* capability. In the same way that `integer` is a subtype of `real` because integers are defined by restricting reals, the `finite_state` domain is a sub-domain of the `state_based` domain. This is true because the `finite_state` domain is formed from the `state_based` domain by *extension*, adding new definitions to constrain the `state_based` domain. Thus, a homomorphism exists between the `state_based` and `finite_state` domains.

The signature of the `finite_state` domain is:

```
domain finite_state::state_based;
```

indicating that the domain `finite_state` is an extension of the domain `state_based`. It then follows that the facet type associated with `finite_state` is a subtype of the facet type associated with `state_based`. The semantics and uses of facet types and subtypes are defined in Chapters 7 and 8. Facet types and subtypes are among the most important language contributions of the Rosetta system.

—-

6.3 Facet Operations

The *facet algebra* defines a collection of operations over facet types. These operations allow composition of individual facets into new facets and the definition of relationships between facets.

Facet composition operators can be used to define new facets as compositions of other facets. To achieve this, a facet is declared and assigned to the composition of other facets. An example from Chapter 3 describes the composition of requirements and constraints for a sorting component. Specifically:

```
sort :: logic is sort_req + sort_const;
```

This declaration follows the definitional style used for all Rosetta declarations. The label `sort` names the facet while the built-in type `logic` defines the facet type. In this case `+`, pronounced sum, forms a new facet from `sort_req` and `sort_const`. Specifically, sum forms the co-product of `sort_req` and `sort_const`.

Like types, parameterized facets may be defined using the function notation. The `facet` type is a type like any other and can be returned by functions. Thus, the signature of a parameterized `sort` facet definition is:

```
sort(qs::boolean)::facet is
  sort_const + (if qs then quick_sort_req else sort_req);
```

In this definition, the parameter `qs` selects whether requirements for a quicksort or more general sorting requirements are included in the sum.

The following operators are defined over facets:

Operation	*Format*	*Definition*
Sum	`F + G`	*co-product of F and G*
Product	`F * G`	*product of F and G*
Implies	`F => G`	*homomorphism from G to F*
Equivalence	`F = G`	`F => G and G => F`
Functor	`F(f::domain)::domain`	Mapping from one facet type to another

The properties of sum and product are defined by the category theoretic notations of co-product and product. When the co-product of two items is formed, the new item must have the properties of both the original items. Specifically, the facet `F+G` must have all properties of both `F` and `G`. When the product of two items is formed, the new item must have the properties of one or the other of the original items. Specifically, the facet `F*G` must have either properties of `F` or `G`.

Facet implication is a relation between facets that occurs when a homomorphism exists between them. When `F=>G`, pronounced "F implies G" holds, all properties of `F` are also properties of `G` and a homomorphism exists from `G` to `F`. Note that the homomorphism works in the opposite direction as implication.

Facet equivalence is defined as the existence of an isomorphism between two facets. If `F=>G` and `G=>F`, then an isomorphism exists between `F` and `G` and `F=G`.

A functor is a special function that maps elements of one facet type onto elements of another facet type. Functors play an important role in Rosetta as mechanisms for moving information between domains. Application of functors and their semantics are defined fully in Chapter 7 and Chapter 8 respectively.

The following sections describe several prototypical uses of facet composition. Please note that domains use in these examples are defined in Chapter 7. In the following definitions, assume that all F_n are facets where T_n, D_n and I_n are the term set, domain and interface associated with F_n respectively.

6.3.1 Facet Conjunction

Facet conjunction, $F_1 \wedge F_2$, states that properties specified by terms T_1 and T_2 must by exhibited by the composition and must be mutually consistent. Further, the interface is $I_1 + +I_2$ implying that all symbols in the parameter lists of F_1 and F_2 are also visible in the parameter list of the composition.

The most obvious use of facet conjunction is to form descriptions through composition. Of particular interest is specifying components using heterogeneous models where terms do not share common semantics. A complete description might be formed by defining requirements, implementation, and constraint facets independently. The composition forms the complete component description where all models apply simultaneously.

Example 20 (Requirements and Constraints) *Consider the following facets describing a sorting component:*

```
facet sort_req(i::input T;
     o::output T)::state_based is
begin
  l2: permutation(o',i);
  l1: ordered(o');
end facet sort_req;
```

```
facet sort_const::constraints is
  power::real;
begin
  p1: power =< 5mW;
end facet sort_const;
```

A sorting component can now be defined to satisfy both facets:

```
sort::facet is sort_req and sort_const;
```

Alternatively, the following definition can be used to define ***sort****:*

```
  sort::facet;
begin
  l1: sort = sort_req and sort_const;
  ...
end facet facet;
```

Another alternative is using relabeling to define a single sort component in a structural Rosetta description:

```
begin
  sort: sort_req and sort_const;
  ...
end facet facet;
```

In each case, the resulting ***sort*** *definition is the conjunction of the* ***sort_req*** *and* ***sort_const*** *definitions.*

Summary:

6.3.2 Facet Disjunction

Facet disjunction, $F_1 \vee F_2$, states that properties specified by either terms T_1 in domain D_1 or T_2 in domain D_2 must be exhibited by the resulting facet. Like conunction, the interface of the resulting facet is $I_1 + + I_2$, the union of the facet interfaces.

The most obvious use of facet disjunction is the definition of cases. Two situations are of particular interest: (i) using predicatative semantics to define component behavior; and (ii) defining families of components.

Example 21 (Case Specification) *Given a container C defined as a collection of key (K), element (E) pairs, naive requirements for a simple search algorithm are defined as:*

```
facet search(c::input C; k::input K; o::output E)::state_based is
begin
  member((k,o'),c);
end facet search;
```

Clearly, this specification will be inconsistent if there is no element in c corresponding to k. Thus, it is traditional to break the requirements into two cases: (i) the element is present and is returned; and (ii) the element is not present. Such a situation is modeled by the following two specifications:

```
facet searchOK(c::input C; k::input K;
          o::output E)::state_based is
begin
  l1: exists(x::E | member((k,x), c));
  l2: member((k,o'),c);
end facet searchOK;
```

```
facet searchErr(c::input C; k::input K;
              o::output E)::state_based is
begin
  l1: -exists(x::E | member((k,x), c));
end facet searchErr;
```

Facet ***search*** *is now defined:*

```
search::facet is searchOK or searchErr;
```

Example 22 (Component Version) *Another excellent example of disjunction use is representing a family of components. Consider the following definitions using sort facets defined previously:*

```
multisort::facet is sort_req and (bubble_sort or quicksort);
```

The new facet `multisort` *describes a component that must sort, but may do so using either a bubble sort or quicksort algorithm.* [1]

A more interesting definition configures a component to represent both low and high power configurations of a device:

```
facet low_power::constraints is
  power::real;
begin
  power =< 1mW;
end facet low_power;

facet tx_req(d::input data;
         s::output signal)::continuous is
begin
  <transmitter definition here>
end facet tx_req;
```

```
facet power::constraints is
  power::real;
begin
  power =< 5mW;
end facet power;

low_power_tx::facet is
   tx_req and low_power;

high_power_tx::facet is
   tx_req and power;
```

In this example one specification for a transmitter function is provided along with two definitions of low and high power versions. Facet conjunction is used to combine power constraints with functional transmitter properties.

Consider the following specification:

```
tx(select::boolean)::facet = if select then
                                low_power_tx
                                else high_power_tx
                             endif;
```

Here a generic parameter is introduced into the definition to select one version over another. When `select` is instantiated, then `tx` resolves to the appropriate model. A more interesting case occurs when `select` is skolemized to an arbitrary boolean constant `a`:

```
tx(a) ==  if a then low_power_tx else high_power_tx endif;
```

Whenever facet `tx` is used in this manner, both specifications must be considered. Effectively, `tx` defines two transmitter models. When instantiated in a structural facet, both models must be considered in the analysis activity. It must be noted that the parameter `select` is a boolean valued parameter and not a facet. It is tempting to attempt a definition of `if-then-else` that uses facets as all its parameters. However, such a definition has been shown to have little utility.

```
%% These must be dealt with separately because they do not result in
%% facets.  Although implication should if it's defined in terms of
%% disjunction.
```

[1] Assume the facet quicksort has been defined in the canonical fashion.

Summary:

6.3.3 Facet Implication

Facet implication, $F_1 \Rightarrow F_2$, states that properties specified by term T_1 must imply properties specified by term T_2. Note that $F_1 \Rightarrow F_2 \equiv \neg F_1 \vee F_2$. The most obvious use of refinement is showing that one facet "implements" the properties of another. Specifically, if $F_1 \Rightarrow F_2$, then the theory of F_2 is a subset of the theory of F_1.

Example 23 (Implementation) *Given the requirements defined for sort in* ***sort_req****, any legal implementation of a sorting algorithm must implement these properties. We say that* ***sort_req*** *can be* refined *into* ***bubble_sort*** *and state this as:*

```
sort_ref::facet is bubble_sort => sort_req;
```

Additional constraints may be added by conjuncting facets in the consequent of the implication. The following is an example of adding a low power constraint to the specification:

```
constrained_sort_ref::facet is bubble_sort => low_power and sort_req;
```

This is an interesting result because it insists that ***bubble_sort*** *be a low power solution to the sorting problem.*

As an aside, the definition of conjunction requires that $F_1 and F_2 => F_1$.

Summary:

6.3.4 Facet Equivalence

Facet equivalence, $F_1 \Leftrightarrow F_2$, states that properties specified by terms T_1 and T_2 in domains D_1 and D_2 must be equivalent. The formaly definition of equivalence can be expressed in terms of implication. Formally:

$$F_1 \Leftrightarrow F_2 = F_1 \Rightarrow F_2 \wedge F_2 \Rightarrow F_1$$

Summary:

6.4 Parameter List Union

Throughout the defintion of the facet algebra, reference is made to the union of parameter lists. Specifically, when facets are combined the parameter list of the new facet is defined as $\mathtt{I}_1 + + I_2$. Viewed as bunches, this definition is literally true where all parameters from both facets become parameters in the new facet.

Given the facet declarations:

```
facet F1(x::R, y::S, t::T) is
  ...
end facet F1;
```

```
facet F2(w::Q,x::R) is
  ...
end facet F2;
```

The parameter list of `F1 and F2` is `(x::R, y::S, t::T, w::Q)`. Note that the declaration of parameter `x` is shared in both facet declarations. Bunch union implies that a single `x` appears in the result of parameter list union.

6.4.1 Type Composition

The more interesting case occurs when a parameter is shared betweeen facets, but the declarations specify different types. Consider the following two facet declarations:

```
facet F1(x::R, y::S, t::T) is
  ...
end facet F1;
```

```
facet F2(w::Q,x::P) is
    ...
end facet F2l;
```

In this case, the parameter list of `F1 and F2` is `(x::R, x::P, y::S, t::T, w::Q)`. Note that two declarations of `x` exist in the parameter list definition. Recall that parameter declarations are simply terms appearing in the parameter list. Specifically, a variable or parameter declaration is shorthand for:

```
x:e::R
```

Viewing the parameter definition in this way allows application of label distribution laws. This application yeids the parameter list `(x::R and P, y::S, t::T, w::Q)`. Note that in this parameter list `x` is of type `R and P` implying that `x` can be viewed both as type `R` and type `P`.

When conjuncting and disjunction facets, care must be taken to assure that parameters having the same name represent the same physical quantity. The type declaration `R and P` results in a type that, in principle, behaves like the result of facet conjunction. Specifically, an item of this type is simultaneously viewed as being of both types. It is also important to understand that type composition is not type union. Specifically `R and P` is not equal to `R ++ P`. In the latter case, elements of `R ++ P` can take values from either `R` or `P`.

An excellent example of type composition occurs when looking at a circuit component such as a simple gate from multiple perspectives. Consider a simple `and` gate viewed in both the analog and digital domains:

```
facet and_discrete(x,y::input bit;
   z::output bit)::state_based is
begin
  l1: z' = x*y;
end facet and_discrete;
```

```
facet and_cont(x,y::input real;
    z::output real)::continuous is
begin
  <and gate definition here>
end facet and_cont;
```

The definition of a completely modeled `and_gate` gate becomes:

```
and_gate :: facet is and_discrete and and_cont;
```

The parameter list resulting from this definition is:

```
(x,y::input bit and real, z::output bit and real)
```

Thus, each parameter can be viewed as either a real or discrete value.

```
%% Need discussion of parameter interaction here.  Defer semantics
%% to the semantics guide, but some discussion must occur.
```

6.4.2 Parameter Ordering

The pragmatics of using parameter list union insist that some ordering be placed on the results. Typically, specifiers use the order of parameters in parameter list to associated actual parameters with formal parameters. Rosetta provides two mechanisms for handling this situation. The first is for the user to define an ordering and the second is to use explicit parameter assignment.

To explicitly define parameter ordering in the facet resulting from a conjunction the user specifies parameters in the facet declaration. For our `and_gate` gate example previously, the following definition specifies an ordering for resulting parameters:

```
and_gate(z,y,x::null)::facet is and_discrete and and_cont;
```

In this definition, the parameter ordering in the definition of `and_gate` defines the parameter ordering. The `null` type is used to specify the parameter types as for any type `T`, `T and null == T`. Thus, the parameter definitions add ordering information, but add no additional type information to the definition.

Users may also allow Rosetta to order the types for them.

```
%% Ordering definition here
```

Examples:

Summary:

Chapter 7

Domains and Interactions

Domains are special facets that define domain theories for defining facets. Three types of domains are identified: (i) unit of semantics; (ii) model of computation; and (iii) engineering design. Unit of semantics domains identify the vocabulary or domain of discourse for classes of specifiations. Model of computation domains provide basic unerlying computational models for defining specifidations. Engineering design domains take models of computation domains and define working engineering modeling domains. Domains serve as types for regular facets by generating a category that is all extensions of the domain. The set of objects in this category is defined as the type associated with the domain.

Interactions are functors between domains that support moving information from one domain into another. The functor maps every element of the domain category to a member of a destination domain type. Thus, the information contained in the domain facet is transformed into a new range facet of the new domain type.

7.1 Domains

A *domain* is a special purpose facet that defines a domain theory for facets. All facets extend domains that are considered to be their types. Each domain represents a model of computation and a vocabulary for domain specification. When writing a specification, a designer chooses a domain appropriate for the model being constructed. The domain is extended by adding declarations and terms that use the base domain's predefined model of computation. Alternatively, a designer can define their own domain by extending an existing domain or start completely from scratch. The advantage of using an existing domain is reuse of the domain and its interaction with other domains. The abstract syntax for a domain definition is defined as:

$$
\begin{array}{lll}
\textit{domain} & ::= & \texttt{domain}(\gamma_p)::\gamma_d;\ | \\
 & & \texttt{domain}(\gamma_p)::\gamma_d\ \texttt{is} \\
 & & \quad \gamma_c; \\
 & & \quad \gamma_v; \\
 & & \texttt{begin} \\
 & & \quad \gamma_t; \\
 & & \texttt{end domain};
\end{array}
$$

where γ_p is a seqence of parameter declarations of kind `design`, γ_d is the domain extended by the new definition, γ_c and γ_v are constant and variable items defined in the domain, and γ_t are terms that define the new domain by extending γ_d. Given a domain called `state_based_semantics` used in the following facet:

```
facet register(i::input bitvector; o::output bitvector;
               s0::input bit; s1::input bit) :: state_based_semantics is
  state::bitvector;
begin
  l1: if s0=0
        then if s1=0 then state'=state
                else state'=lshr(state) end if
        else if s1=0 then state'= lshl(state)
                     else state'=i end if
      end if;
  l2: o'=state';
end facet register;
```

the parameter `f` in `state_based_semantics` refers to the including facet `register`. Thus, the domain definition can generically reference elements of the including facet in its definition. For example, it is possible to reference `meta_labels(f)` or `meta_items(f)` to reference the labels and items defined in `f` respectively.

As with traditional facet definition, a domain definition extends the theory provided by its referenced domain. It is therefore possible to define a lattice of domains that inherit and specialize each other. Figure 7.1 shows one such specification lattice including pre-defined domain definitions. Solid arrows represent extension between domains. Clear arrows represent morphisms where all information is conserved from domain at tail of arrow to domain at head. Dotted arrows represent morphisms where some information is lost as a result of the morphim.

The root of the lattice is the `null` domain which is directly extended by the `static` domain. The `static` domain contains declarations of all Rosetta types, constructs and operators. Effectively, it serves as prelude. The rest of the domains are classified into 3 groups: *unit of semantics*, *model of computation* and *engineering model*. A *unit of semantics* represents a unifying semantic domain, i.e. it provides a basic set of semantic objects that can be used to represent different computation models. The *model of computation* group consists of domains that provide semantics for defining objects, goals, relations, i.e. the ontology, of a design paradigm. An *engineering model* is a domain that defines semantics specific to an engineering field, e.g. it may contain definition of units.

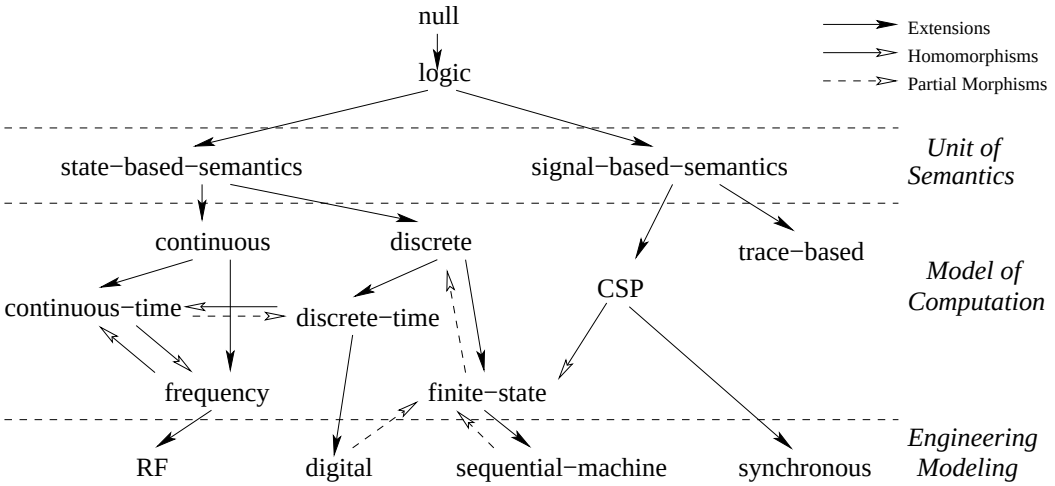

Figure 7.1: Lattice of pre-defined specification domains.

The following sections provide basic definitions and usage examples for each pre-defined domain.

7.1.1 Null

The `null` domain refers to the empty domain. It is included to provide a basis for defining domains that inherit nothing from other domains. The `static` domain that provides basic mathematics will use `null` as a domain to indicate that it is self contained. There is no constructive definition of `null` because it has no domain definition.

```
domain null is
end domain;
```

7.1.2 Prelude Package

All basic Rosetta language constructs and definitions are provided in the package `prelude`.

```
%% Update to include all new Rosetta functions as defined in the
%% types chapter.

package prelude::static is
  universal :: type;
  univ :: subtype(universal);
  item, constructed, function, element,
    set, array, facet :: subtype(univ);
  domain, package, component, interaction :: subtype(facet);
  character, enumeration, number :: subtype(element);
  string :: array(character);
  label::type;
  boolean :: subtype(number) is enumeration[true | false];
  complex :: subtype(number);
  real :: subtype(complex);
  rational :: subtype(real);
  integer :: subtype(rational);
  natural :: subtype(integer);
  bit :: subtype(natural) is enumeration[0 | 1];
  bitvector :: subtype(array(bit));
  j :: complex;
  im(x :: complex) :: real;
  rl(x :: complex) :: real;
  fl(x :: real) :: integer;
  abs(x :: real) :: real;
  sqrt(x :: real) :: complex;
  sqr(x :: real) :: real;
  log(x :: real) :: real;
  __=__ :: <* (lhs, rhs :: univ) :: boolean *>;
  __/=__ :: <* (lhs, rhs :: univ) :: boolean *>;
  __==__ :: <* (lhs, rhs :: univ) :: boolean *>;
  #__ :: <* (rhs :: set(univ)) :: natural *>;
  __++__ :: <* (lhs, rhs :: set(univ)) :: set(univ) *>;
  __**__ :: <* (lhs, rhs :: set(univ)) :: set(univ) *>;
  __--__ :: <* (lhs, rhs :: set(univ)) :: set(univ) *>;
  __in__ :: <* (lhs :: univ; rhs :: set(univ)) :: boolean *>;
  $__ :: <* (rhs :: array(univ)) :: natural *>;
```

```
  __;;__ :: <* (lhs, rhs :: array(univ)) :: array(univ) *>;
  // polymorphic
  subst(a :: array(univ); i :: natural; v :: univ) :: array(univ);
  %__ :: <* (rhs :: bit) :: natural *>;
  __<__ :: <* (lhs, rhs :: number) :: boolean *>;
  __>__ :: <* (lhs, rhs :: number) :: boolean *>;
  __>=__ :: <* (lhs, rhs :: number) :: boolean *>;
  __=<__ :: <* (lhs, rhs :: number) :: boolean *>;
  __>=__ :: <* (lhs, rhs :: number) :: boolean *>;
  __max__ :: <* (lhs, rhs :: number) :: number *>;
  __min__ :: <* (lhs, rhs :: number) :: number *>;
  __+__ :: <* (lhs, rhs :: complex) :: complex *>;
  __-__ :: <* (lhs, rhs :: complex) :: complex *>;
  __/__ :: <* (lhs, rhs :: complex) :: complex *>;
  __*__ :: <* (lhs, rhs :: complex) :: complex *>;
  __^__ :: <* (lhs, rhs :: complex) :: complex *>;
  not__ :: <* (rhs :: boolean) :: boolean *>;
  __implies__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  __=>__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  __nmax__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  __nmin__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  __and__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  __nand__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  __or__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  __nor__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  __xor__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  __xnor__ :: <* (lhs, rhs :: boolean) :: boolean *>;
  posreal :: subtype(real) is sel(x::real | x >= 0);
  tl(seq :: sequence(univ)) :: sequence(univ);
  nil :: sequence(univ);
  pi :: real is 3.1415927;
  // and so on ...
end package prelude;
```

7.1.3 Static

The `static` domain serves as the basis for all pre-defined Rosetta domains. All primitive Rosetta constructs are provided in `static` by including the `prelude` package and are available in any facet or domain that extends `static`. Any term defined in the static domain expresses a fact that is monotonic because definition are unaware of any underlying state change.

```
use rosetta.prelude
domain static::null is
begin
end domain static;
```

Examples

Summary

7.1.4 State Based

The `state_based_semantics` domain defines the base semantics for systems that change state. This domain extends the mathematical capabilities provided by the `static` domain to include the concept of state and change. Recall that in the `static` domain, the values associated with items could not change. Doing so created inconsistencies with the original definitions. The `state_based_semantics` domain provides the basis for modeling the concepts of state and change by defining: (i) the state of a facet; (ii) the current state; and (iii) a next state function that derives the next state from the current state.

Consider the following trivial definition of a counter that counts from 0 to 7 and repeats:

```
facet counter(v::output natural)::state_based_semantics is
  n::natural;
begin
  next: if n < 7 then n'=n+1 else n'=0 end if;
  output: v' = n;
end facet counter;
```

This definition uses a natural number, `n`, to maintain the current counter value and uses two terms to define the next state and output respectively. The first term, labeled `next`, defines the next state given the current state:

```
next: if n < 7 then n'=n+1 else n'=0 end if;
```

In this term, `n` refers to the value of `n` right now in the current state. The notation `n'` refers to the value of `n` in the next state after the component or system represented by the facet has completed its computation. Understanding this convention, the term can now be interpreted as a conditional statement stating 'if `n` is less than seven in the current state, then `n` in the next state is `n+1`, else `n` in the next state is `0`.' This is precisely how a counter calculates its next value.

Similarly, the second term defines the next output:

```
output: v' = n;
```

Using the same interpretation mechanism, the next value of `v` will be the current value of `n`. This is somewhat interesting as the output lags the current state by one value. If such behavior is not desired, then this term can be modified to state `v'=n'`.

It is exceptionally important to recognize that the following term similar to a C-like programming statement is not correct:

```
next: if n < 7 then n=n+1 else n=0 endif;
```

Remember that terms state things that are true. These are not executed and there is no notion of assignment. Although legal in C where `=` is an assignment operator, in Rosetta this statement asserts that if $n < 7$, then $n = n+1$ is also true. Looking at `=` as equality rather than assignment makes the second statement inconsistent as there is no natural number that is equal to itself plus one. The key to using `state_based_semantics` domains is recognizing that no label tick indicates the current state and label tick indicates the next state.

The `state_based_semantics` tick notation is defined based on the `state_based_semantics` domain definitions of current state and the next state function. In reality, the notation `x'` is shorthand notation for `x@next(s)` where: (i) `@` refers to the value of a label in a state; (ii) `next` defines the state following a given state, and `s` is the current state. Specifically:

```
x == x@s
```

and

```
x' == x@next(s)
```

The previously defined counter specification is equivalent to the following expansion:

```
facet counter(v::out natural)::state_based_semantics is
  n::natural;
begin
  next: if n@s < 7 then n@next(s)=(n@s)+1 else n@next(s)=0 end if;
  output: v@next(s) = n@s;
end facet counter;
```

where the tick notation is replaced by it's definition and references to labels in the current state are expanded to explicitly reference the state. Readers curious about the actual definition of the `state_based_semantics` domain should refer to Section 7.1.4 defining the semantics of `state_based_semantics`. Readers needing only to understand use of the `state_based_semantics` domain may safely skip Section 7.1.4.

Examples

```
%% Steal the examples from the tutorial and add a few more.
```

Semantics

This and subsequent semantics sections may be skipped by readers who do not wish to see the internals of a domain definition.

The `state_based_semantics` domain provides a basic definition of state and change of states. Two basic mechanisms are provided: (i) a definition of state; and (ii) a definition of what next state means. The definition of state provides a state type that can be referenced in definitions. In the `state_based_semantics` domain, relatively few restrictions are placed on the state definition. The next state function provides the concept of change by sequencing states. Like the state type, the `state_based_semantics` domain places few restrictions on the next state function.

The state type, `States`, is defined as an uninterpreted type representing possible states. Information can be added, but base definitions cannot be changed. A specific instance, `s::States`, is defined as the *current state*. Effectively, `s` provides a name that can be referenced in definitions rather than quantifying over all states. The next state function, `next`, is defined as a function that maps one state to another. No other constraints are defined for the `next` function. In other domains related to `state_based_semantics`, `next` will be restricted and specialized to model varying definitions of time and state change. In the basic domain, only the existence of a current and next state are defined.

```
domain state_based_semantics :: static is
  States :: subtype(univ);
  s :: States;
  next :: <* (x :: States) :: States *>;

  __@__ :: <* (lhs :: label; rhs :: States) :: item *> is
     <* (lhs :: label; rhs :: States) :: item is meta_getItem(lhs, rhs) *>;
  __' :: <* (lhs :: label) :: item *> is
```

```
      <* (lhs :: label) :: item is lhs@next(s) *>;

  event(x :: label; st :: States) :: boolean is (x@st /= x@next(st));

  input(x :: label) :: boolean;
  output(x :: label) :: boolean;
  inout(x :: label) :: boolean;
  design(x :: label) :: boolean;

  export all;
begin

end domain state_based_semantics;
```

The Rosetta Semantics Guide defines the function `meta_parse` as a function that assigns semantics to Rosetta structures. When applied to an arbitrary Rosetta structure, `meta_parse` is equal to the semantic definition of that structure. In the `state_based_semantics` domain, the definition of `meta_parse` is specialized to provide a mapping from labels to their associated values *in a particular state*. The function is specialized only for atomic items forming the leaves of the parse tree. In other words, items associated with lexical tokens.

`Meta_parse` for a given label is defined to return the value associated with that label in the current state. The meta-function `meta_getItem(l,c)` refers to the item associated with `l` in the state (or context) `c`. In `meta_parse`, `meta_getItem` is instantiated to reference specifically the state `s` defined in the `state_based_semantics` domain to be the current state. `Meta_value` is used to access the value of the item. Thus, any label appearing in a `state_based_semantics` facet refers to the value of the item named by the label in the current state.

The infix function `@` is used to reference a label's value in an arbitrary state. The notation `x@s5` refers to the value of `x` in a state referred to by `s5`. The definition of `@` is nearly identical to `meta_parse` using `meta_getItem` and `meta_value` to obtain the value of a label in an arbitrary state. As anticipated, the theorem that `x==x@s` is easily proven by instantiating the state variable in `@` with the current state `s`. Of special note is that the first argument to `@` is a label. Because the label rather than the value is desired in this situation, the `meta_parse` function is not applied to the first argument of `@`.

In the `state_based_semantics` domain, the dominant specification methodology is axiomatic specification. Thus, the primary use of `@` is to refer to label values in the next state. Specifically, statements such as `R(x,x@next(s))` are used to constrain the value of `x` in the next state based on the value of `x` in the current state. In axiomatic specification, the standard notation `x'` is used as a shorthand for `x@next(s)`. Thus, Rosetta provides a shorthand notation for any symbol, `x`, in the next state as `x'`. The previous example specification can thus be rewritten in a more compact notation as `R(x,x')`.

Summary

The `state_based_semantics` domain provides a mechanism for specifying how a component or system changes state. It provides a basic type, `States`, for states and a state variable, `s::States`, that represents the current state. In addition, the STATE_BASED_SEMANTICS domain provides a definition for `next`, a function that generates a new state from a given state. Thus, if `s` is the current state, then `next(s)` is the next state.

To refer to the value of a variable in a state, the "`@`" operation dereferences a label in a bunch of items. Specifically, given an item bunch `c` and a label `l`, the notation `l@c` refers to the value of the item associated with `l` in the context `c`. As states are defined as bunches of items, the "`@`" operation is easily used to obtain values of items in a state.

Because relationships between the current and next state are frequently the objective of state based specification, the `state_based_semantics` domain provides a shorthand for referencing items in the next state.

Specifically, the notation `x'` is equivalent to `x@next(s)` providing a convenient shorthand for referencing next state values.

As the `state_based_semantics` domain represents a unifying semantic domain, it should mainly be used as basic domain for definition of new domains. However, at early design stages when working at high levels of abstraction, the `state_based_semantics` domain provides a mechanism for describing state transformations without unnecessary details.

The `state_based_semantics` domain should not be used when details such as timing are involved in the specification. Furthermore, the `state_based_semantics` domain provides no automatic mechanism for composing component states when developing structural models.

7.1.5 Discrete

The `discrete` domain constrains the `States` type defined in `state_based_semantics` to a set of discrete values. As `discrete` extends `state_based_semantics`, all definitions from `state_based_semantics` remain valid in the new definition.

Examples

Semantics

The `discrete` domain constrain the states to be discrete.

```
domain discrete(f::facet) is
begin state_based_semantics
  d1: exists(fnc::<*(st::States)::natural*> |
             forall(s1,s2::States|
             (s1 /= s2) implies (fnc(s1) /= fnc(s2))));
end domain discrete;
```

7.1.6 Finite State

The `finite_state` domain provides a mechanism for defining systems whose state space is known to be finite. As `finite_state` extends `state_based_semantics`, all definitions from `state_based_semantics` remain valid in the new definition. Specifically, `next`, "`@`", and tick retain their original definitions. The only addition is the constraint that `States` must be a finite set. Consider the simple definition of a counter:

```
facet counter(c::output natural)::finite_state is
  States :: subtype(natural);
begin
  state-space: States=0,1,2,3;
  next-state: next(s)=if s=<2 then s+1 else 0 endif;
  output: c'=next(s);
end facet counter;
```

In this counter, the state space is explicitly defined as the set containing 0 through 3. Because `#S=4`, the state space is clearly finite causing no inconsistency with the axiom added by the `finite_state` domain. Here, instead of defining the next state function in terms of properties of the next state, it is explicitly defined as modulo 4 addition on the current state. This is quite different from the previous `state_based_semantics` specifications where the actual value of the next state was not defined.

The final term in the counter asserts that the output in the next state is the value of the next state. Remember that here state is defined as a collection of numeric values from 0 through 4. Expanding the final term reveals the following:

```
c@next(s)=next(s)
```

The output in the next state is equal to the value of the next state.

Examples

Semantics

The domain `finite_state` is an extension of the `state_based_semantics`. Specifically, in the `finite_state` domain the number of defined states is finite. The domain definition extends the `finite_state` facet by simply adding the assertion that the size of the state type, `States`, is finite:

```
domain finite_state(f::facet)::state_based_semantics is
begin
  l1: #States < TRUE;
end domain finite-state;
```

Because `finite_state` is an extension of `state_based_semantics`, all other definitions remain true.

Summary

The `finite_state` domain is simply an extension of the `state-based` domain where the set of possible states is known to be finite. Using the `finite_state` domain is exactly the same as using the `state-based` domain with the additional restriction assuring that the state bunch is finite. Note that all `finite_state` specifications can be expressed as `state_based_semantics` definitions with the added restriction on the state space size.

The `finite_state` domain is useful when defining systems known to have finite states. Whenever a sequential machine is the appropriate specification model, the `finite_state` domain is the appropriate specification model. Typically, the elements of the state type are specified by extension or comprehension over another bunch to assure the state type is finite. Most RTL specifications can be expressed using the `finite_state` domain if desired.

The `finite_state` domain should not be used when the state type is not known to be finite. If the additional finite state property does not add to the specification, then the `state_based_semantics` domain should be used. Of particular note is that `finite_state` specifications should not typically be used when timing information is specified as a part of function. In such circumstances, the set of possible states is almost always know to be infinite.

7.1.7 Infinite State

Like the `finite_state` domain, the `infinite_state` domain extends the `state_based_semantics` domain by restricting the state definition. Instead of making the state type finite, the `infinite_state` domain explicitly makes the state type infinite by adding the term `next(s) > s`. With this definition, two concepts are defined: (i) an ordering on states; and (ii) the next state is always greater than the current state. Strictly, the latter definition is somewhat too restrictive as it implies no loops in the state transition diagram. However, the restricted model does lend itself to systems level specification.

Examples

Semantics

The `infinite_state` domain extends the `state_based_semantics` domain by adding the assertion that the next state is always greater than the current state. This is expressed in term `l1` in the following domain definition:

```
domain infinite_state(f::facet)::state_based_semantics is
  __>__(s1::S,s2::S)::boolean;
begin state_based_semantics
  l1: forall(<*(s1::S)::boolean is next(s1) > s1 *>);
  l2: forall(<*(s1::S)::boolean is -(s1 < s1) *>);
  l3: forall(<*(s1::S,s2::S)::boolean is s1 < s2 => -(s2 < s1) *>);
  l3: forall(<*(s1::S,s2::S,s3::S)::boolean is
        s1 < s2 and s2 < s3 => (s1 < s1) *>);
end domain infinite_state;
```

The addition of `l1` to the definition implies a potentially infinite number of states as a side effect of state ordering. Specifically, a total order, "`>`", must be defined over any bunch of items used to define state. To assure the total order property, terms constrain the "`<`" to be a total order. Note that although this declaration is local to the `infinite_state` domain, the specifier is responsible for assuring the definition of the operation. Specifically, the specifier must define what "`<`" means for the state bunch `S`. When specifying time models in the `discrete-time` and `continuous-time` domains later, the elements of `S` will be fixed in such a way that the ordering property is assured.

Like the `finite_state` domain, because the `infinite_state` domain extends `state_based_semantics`, all definitions remain true. Expanding the `state_based_semantics` domain gives the following definition of `infinite_state`:

```
domain infinite_state(f::facet)::static is
  S::bunch(items);
  s::S;
  next(s::S)::S;
  __>__(s1::S,s2::S)::boolean;
  M__parse(l::M__labels(f))::universal is M__value(M__item(l,s));
  __@__(l::label; s1::S)::universal is M__value(M__item(l,s1)));
  __'(x::label)::item is x@next(s);
begin
  a1: forall(<*(s::S)::boolean is
        forall(<*(l::M__labels(f))::boolean is
          M__item(l,f) = M__item(l,s)*>)
  l1: forall(<*(s1::S)::boolean is next(s1) > s *>);
  l2: forall(<*(s1::S)::boolean is -(s1 < s1) *>);
  l3: forall(<*(s1::S,s2::S)::boolean is s1 < s2 => -(s2 < s1) *>);
  l4: forall(<*(s1::S,s2::S,s3::S)::boolean is
        s1 < s2 and s2 < s3 => (s1 < s1) *>);
end infinite_state;
```

Summary

The `infinite_state` domain is another extension of the `state_based_semantics` domain where the set of possible states is known to be infinite and ordered. Using the `infinite_state` domain is exactly the same

as using the state_based_semantics domain with the additional restriction of assuring the existence of a state ordering and that next generates new states in order. Note that all infinite_state specifications can be expressed as state_based_semantics definitions with appropriate added restrictions on state ordering.

The infinite_state domain is useful when defining systems where states are ordered and potentially infinite numbers exist. For example, representing a discrete event simulation system is appropriate for the infinite_state domain.

The infinite_state domain should not be used when the state type is know to be finite or if no state sequencing is known. Nor should the infinite_state domain be used when timing models are known. As such, most specifiers will choose to use the discrete-time or continuous-time domain over the infinite_state domain in most modeling situations.

7.1.8 Discrete Time

The discrete-time domain is a special case of the infinite_state domain where: (i) each state has an associated time value; and (ii) time values increased by a fixed amount. Specifically, in the discrete-time domain, time is a natural number denoted by t and discrete time quanta is a non-zero natural number denoted by delta. The next state function is defined as next(t)=t+delta and following from previous domain definitions, x@t is the value of x and time t and x' is equivalent to x@next(t).

Specifications are written in the discrete time domain in the same fashion as the infinite and finite state domains. The additional semantic information is the association of each state with a specific time value. Thus, the term:

```
t1: x' = f(x)
```

constrains the value of x at time t+delta to be the value of f(t) in the current state. This specification style is common and reflects the general syntax and semantics of a VHDL signal assignment.

Examples

Semantics

The discrete-time domain extends the infinite_state domain by: (i) refining state to be of type natural; and (ii) refining the next. These definitions are provided in the definition section while the association between state and time is made in term l1 in the following definition:

```
domain discrete-time(f::facet)::infinite_state is
  T::natural;
  t::T;
  delta::natural--0;
  next(t::T)::T is t+delta;
begin
  l1: T=S and t=s;
end domain discrete-time;
```

Expanding fully the definition of infinite_state results in the following specification:

```
domain discrete-time(f::facet)::static is
  S::bunch(items);
  s::S;
```

```
  T::natural;
  t::T;
  delta::natural--0;
  next(s::S)::S;
  next(t::T)::T is t+delta;
  __>__(s1::S,s2::S)::boolean;
  M__parse(l::M__labels(f))::universal is M__value(M__item(l,s));
  __@__(l::label; s1::S)::universal is M__value(M__item(l,s1)));
  __'(x::label)::item is x@next(s);
begin static
  a1: forall(<*(s::S)::boolean is
        forall(<*(l::M__labels(f))::boolean is
          M__item(l,f) = M__item(l,s)*>)
  l1: forall(<*(s1::S)::boolean is next(s1) > s *>);
  l2: forall(<*(s1::S)::boolean is -(s1 < s1) *>);
  l3: forall(<*(s1::S,s2::S)::boolean is s1 < s2 => -(s2 < s1) *>);
  l4: forall(<*(s1::S,s2::S,s3::S)::boolean is
        s1 < s2 and s2 < s3 => (s1 < s1) *>);
  l5: T=S and t=s;
end domain discrete-time;
```

Simplifying and removing redundant terms results in: (Note that the simplification is achieved by rewriting `S` and `s` with `T` and `t` throughout the specification.)

```
domain discrete-time(f::facet)::static is
  T::natural;
  t::T;
  delta::natural--0;
  next(t::T)::T is t+delta;
  __>__(s1::T,s2::T)::boolean;
  M__parse(l::M__labels(f))::universal is M__value(M__item(l,t));
  __@__(l::label; s1::T)::universal is M__value(M__item(l,s1)));
  __'(x::label)::item is x@next(t);
begin static
  a1: forall(<*(s::T)::boolean is
        forall(<*(l::M__labels(f))::boolean is
          M__item(l,f) = M__item(l,s)*>)
  l1: forall(<*(s1::T)::boolean is next(s1) > t *>);
  l2: forall(<*(s1::T)::boolean is -(s1 < s1) *>);
  l3: forall(<*(s1::T,s2::T)::boolean is s1 < s2 => -(s2 < s1) *>);
  l4: forall(<*(s1::T,s2::T,s3::T)::boolean is
        s1 < s2 and s2 < s3 => (s1 < s1) *>);
end domain discrete-time;
```

Of particular note is the axiom asserting that `next(s) > s` from the infinite_state domain. It is a simple matter to show that specializing `next(s)` with `t+delta` satisfies this requirement. Specifically, `delta` is constrained to be a non-zero natural number. Adding `delta` to any natural number `t` results in a value that is greater than `t` by the canonical definition of natural number addition.

Summary

The `discrete-time` domain is an extension of the `infinite_state` domain where the set of possible states is the set of natural numbers and the next state function is constrained to be the addition of a discrete value

to the current state. Using the `discrete-time` domain is exactly the same as using the `infinite_state` domain with the addition of discrete time values to the definition of state. Note that all `discrete-time` specifications can be expressed as `infinite_state` and `state_based_semantics` definitions with appropriate added restrictions on state ordering.

The `discrete-time` domain is the workhorse of the state_based_semantics specification domain. It is exceptionally useful when defining digital systems of all types. Using the expression notation:

```
x' = f(x,y,z)
```

provides a mechanism for constraining the value of variables in the next state. Furthermore, the notation:

```
x@t+(n*delta) = f(x,y,z)
```

provides a mechanism for looking several discrete time units in the future. Such mechanisms are useful when defining delays in digital circuits.

The `discrete-time` domain should not be used when no fixed timing constraints are known. In such situations, the `infinite_state` or `state_based_semantics` domains may be more appropriate and will help avoid over-specification.

7.1.9 Continuous Time

Continous time specifications provide a mechanism for defining temporal specifications using a maximally general notion of time. Unlike discrete time specifications, continuous time specifications allow reference to any specific time. Time becomes real-valued.

The `continuous-time` facet provides a type `T` representing time that is real valued. This differs from the `discrete-time` domain where time values were restricted to the natural numbers, a countably infinite set. The function `next` is defined as is `x'` for any variable `x`. Using these concepts, the time derivative, or instantaneous change associated with `x` is defined as `deriv(x@t)` or simply `deriv(x)` by viewing `x` as a function of time. An n_{th} order time derivative can be referenced by recursive application of `deriv`. The second derivative is defined `deriv(deriv(x))`, the third derivative `deriv(deriv(deriv(x)))`, and so forth.

If `x` is defined as a function over time, `x=f(t)`, the derivative is defined in the canonical fashion as:

$$\frac{dx}{dt} = \frac{df(t)}{dt} = \texttt{deriv(f)}$$

It is interesting to note that the definition `x=f(t)` expands to `x@t=f(t)@t`. Although this notation may be a bit awkward, it is consistent with the definition of the state of a Rosetta specification at a particular time `t`. It should be noted that `deriv(f)` defined in this context is identical to the derivative `deriv(f,t)` using the general derivative structure provide in the `static` domain. The same applies for any of the derivative and integral functions provided in the time domain.

The indefinite integral with respect to `t` is defined as `antideriv(x)` and behaves similarly when `x` is a function over time. Note that the antiderivative with respect to time assumes an integration constant of zero. Making the integration constaint different is a simple matter of adding or subtracting a real value from the indenfinite integral. As with the standard indefinite integral, the following is defined:

```
antideriv(deriv(x)) == x
```

The indefinite integral of the derivative of any function over time is the original function.

The definite integral is provided as `integ(x,l,u)` and is defined as:

```
integ(x,l,u) == antideriv(x)(u) - antideriv(x)(l)
```

This is the canonical definition of the definite integral over a specified time period.

Examples

Semantics

```
%% There are still numerous problems with the following definitions.

domain continuous-time(f::facet)::static is
  T::real;
  __@__(x::label; t::T)::univ is M__value(x,M__items(f,t));
  next(t::T)::T;
  __'(x::label)::item is x@next(s);
  deriv(x::real)::real is deriv(x,t);
  antideriv(x::real)::real is antideriv(x,t,0);
  integ(x,u,l::real)::real is integ(x,t,u,l);
begin
  t1: forall(<*(t::T)::boolean is
        forall(<*(d::M__type(x,M__items(f)))::boolean is
          forall(<*(e::T)::boolean is
            abs(x@(t+e)-x') < d *>)*>)*>)
  t2: forall(<*(t::T)::boolean is
        deriv(x@t) = lim((x@next(t) - x@t) / (t - next(t)),next(t),0))
end domain continuous-time;

%% Original definition of next from discussions earlier in the
%%  year.
%% deriv(x@t) = x@next(t) - x@t) / (t - next(t))
```

Summary

7.2 Interactions

An *interaction* is a special purpose facet that defines situations where two domains interact. Unlike domain and traditional facet definitions, signficant syntactic sugar is added to the interaction definition syntax to simplify the special characteristics of the interaction definition.

Facets, Domains and Terms

An atomic facet, F_k, is a pair, (D_k, T_k), where D_k is the *domain* of F_k and T_k is the *term set* of F_k. The domain of a model is its semantic basis. The term set of a model is a set of terms that extend its domain to describe a more specific system. Thus, D_k provides meaning and inference capabilities to terms expressed in T_k. We say that a term, t, is a consequence of a model if $M_k \vdash_{D_k} t$, where $\vdash_{D_k}$ is inference as defined by domain D_k. Specifically, a term follows from a model if it can be inferred using the model's inference mechanism. The theory, Θ_k, of a model, M_k, is defined as the closure with respect to domain specific inference:

$$\Theta_k = \{t \mid M_k \vdash_{D_k} t\}$$

The complete calculus for models is given in the semantics guide and is taken largely from existing model theoretic research. It is sufficient for this effort to understand that each model consists of a semantic domain model and a set of terms extending that domain.

Interaction Semantics

A *composite facet* is a set of facets that are simultaneously true. Generally, a composite facet is defined using facet conjunction. Given two facets F_j and F_k, we define $F = F_j and F_k$ as:

$$F = \{(D_j, T_j \cup M\_\_I(F_j, F_k)), (D_k, T_k \cup M\_\_I(F_k, F_j))\}$$

where $M\_\_I$ is an *interaction function* defining the domain interaction. $M\_\_I(F_j, F_k)$ defines a set of terms, called the *interaction term set* or simply *interaction set*, in the semantic domain of F_j. The interaction set defines the impact of F_k on F_j using terms defined in the semantic domain D_j. Under composition, the terms of F_j are unioned with the interaction term set to augment the original model with interaction results. Again, the key to the approach is that the interaction term set is expressed in the affected domain. In a design flow, composing models in this way corresponds to putting a design in its operational environment.

The Rosetta syntax for the default domain interaction function is:

```
M__I(F1::domain1; F2::domain2)::set(term) is empty;
```

Interaction definitions overload `M__I` for various domains. The default interaction defined by the function above is the empty term set. Specifically, if no interaction has been defined between domains, then the interaction is assumed to be null.

The projection of a composite model into a domain, π_D, is the atomic facet assocated with domain D resulting from the interaction. While composition combines models, projection pulls them back apart maintaining the effect of the interaction. Specifically:

$$\pi_D(F) = F_k \Leftrightarrow F_k \in F \wedge D = D_k$$

The Rosetta syntax for the projection function is:

```
M__pi(D::domain,F::facet)::facet;
```

The projection function retrieves domain aspects of a composite facet specific to a domain. To find the projection, the composition is formed using the interaction function and the projection with respect to the domain in question is extracted. If there is no model in the composition associated with D, then the projection is undefined. In a design flow, taking projections in this way corresponds to assessing the results of putting a design in its operational environment from one particular perspective.

A special case of facet composition exists when $D_j = D_k$. In this case:

$$F = (D_j, T_j \cup T_k)$$

Specifically, the resulting model is an atomic model with the domain shared by the original models and term set equal to the union of the original term sets. Further, only π_{D_j} is defined. It is not possible to retrieve the original theories from the composition.

They syntax for an interaction is defined as:

```
interaction operator(domain1,domain2::domain) is
begin interaction
  l1: M__I(facet::domain1; facet::domain2) is term-expression;
  l2: M__I(facet::domain2; facet::domain1) is term-expression;
end operator;
```

In this definition, *operator* is the facet algebra operation associated with the interaction while the *domain1* and *domain2* values specify parameters representing the two interacting domains. Typically, interactions are defined only over conjunction operations. Other operations may introduce interactions in the future and the notation is flexible to allow such definitions. Note that the definition semantics differs from traditional facet definition in that the parameter types are the domains associated with facets, not types in the traditional sense.

The domain of an interaction definition is the special `interaction` domain where the interaction operations and the projection operation are defined. By building from the `interaction` domain, designers start with a basic set of interaction definition capabilities. The *terms* associated with an interaction are a set of traditional terms that define the interaction functions. To completely specify the interaction, the definition of `M__I` must be included for both perumtations of the domain parameters. If this is not done, then the interaction definition will not be complete. The interaction definition applies any time two facets of the appropriate types are specified in a facet algebraic operation involving *operator*.

One of the simplest and most powerful interactions defined is the interaction between monotonic static (mathematics) and a state_based_semantics semantics. Two interactions define the relationship defined in the *static* domain and temporal claims defined in the `state_based_semantics` domain.. Assume that static simply provides a mathematical domain that is monotonic, *i.e.* unchanging. In contrast the state based domain provides the concepts of current and next state. Intuitively, if both models describe the same system, each assertion made in the monotonic (static) model must be true in every state of the temporal (state based) domain.

```
interaction and(state_based_semantics,static::domain) is
begin interaction
  l1: M__I(f::static,g::state_based_semantics)::set(term) is
      dom(t::M__terms(f) | dom(s::g.S | t@s));
  l2: M__I(g::state_based_semantics,f::static)::set(term) is
      sel(t::M__terms(g) | forall(s::g.S | t@s));
end and;
```

The first interaction equation defines the impact of an interaction between static and state based specifications on the static domain:

```
M__I(f::static,g::state_based_semantics)::set(term) is
      dom(t::M__terms(f) | dom(s::g.S | t@s))
```

This function states that every predicate defined in the facet of type static `static` is an invariant in the `state_based_semantics` facet. Specifically, the interaction is defined as the set of all terms from the `static` facet asserted in every state of the `state_based_semantics` facet. The notation `t@s` is used to represent the term `t` asserted in state `s`. The set `S` is the set of all possible states as defined by the `state_based_semantics` domain. Assuming that:

```
forall(x::integer | P(x) is a term in the static domain,
```

then:

```
forall(s::g.S | forall( x:integer | P(x)@s))
```

holds in the state based domain and the set:

```
dom(s::S | forall(x::integer | P(x))@s)
```

defines the interaction. Specifically, because `forall(x::integer | P(x))` is monotonic, it must hold in every state.

This interaction is extremely useful in defining system constraints such as power consumption. Many constraints are defined in a monotonic fashion. By composing a constraints model for a component and a functional model using a state based semantics, the interaction asserts that the constraint is true in all states.

The second interaction equation is the dual of the first:

```
M__I(g::state_based_semantics,f::static)::set(term) is
     sel(t::M__terms(g) | forall(s::g.S | t@s));
```

The interaction set is defined as all those terms in the `state_based_semantics` facet that are true in every state. The property expressed is that if a term is true in every state, that term is invariant over all states. It is, in effect, invariant and can be stated without reference to state. The generation of the interaction set is analogous to the previous example.

Similar in nature to its dual, this interaction discovers invariant properties in a state_based_semantics specification. Although discovering a constraint is an interesting concept, its usefulness arises typically when modeling properties such as safety and liveness conditions.

As noted, this is the simplest of the interactions defining relationships between domains using different temporal semantics. Other currently defined domains provide pair-wise relationships between monontonic, state based, finite state, infinite state, discrete time and continuous time domains. Not all of these domains are isomorphic, thus many interactions define only partial transformations of information.

Just a few interesting interactions useful for defining constraints and requirements include:

- *Monotonic constraints interpreted as moving averages* — Rather than treating monotonic specifications as absolute limits, check moving averages over time. Useful for specifying constraints whose instantaneous values are not as important as values over time.
- *Axiomatic specifications interpreted as assertions in operational specifications* — Preconditions and postconditions specified in the state based domain become assertions checked at the initiation and termination of an operationally specified process. Useful for mapping "black box" requirements onto detailed specifications.
- *Temporal specifications interpreted as temporal constraints in operational specifications* — Like axiomatic specifications, but checked at specific temporal instances. Useful for mapping real time constraints onto detailed specifications.

In the design flow, interactions provide information to designers whenever models are composed using the projection operators. When the model composition occurs is a matter of style, however the projection operators deliver back to the domain specific designers the implications of the interaction. Specifically, the designers working with the state based, functional model learn what impacts constraints have on their design without requiring access to the constraint model. Conversely, the constraints engineer understands the impact of the functional design on constraints issues.

Systems and Components

Editor's Note: This section is (clearly) under construction. The working version is not ready for presentation. We anticipate an incremental update on or before 7/7/99. Comments and suggestions are welcome but may need to be deferred until after the section is released.

An integral part of systems design is the representation and aggregation of components. The facet semantics presented this far provides necessary support for component semantics. However, component-based design is so pervasive the inclusion of a component structure is necessary.

A *component* consists of three information elements representing: (i) *assumptions* made in its design; (i) *definitions* providing component descriptions; and (iii) *verifications* describing what must be true about the component. Assumptions provide necessary context for component use. They include assumptions made on the operational environment, input and output data, or any other constraint placed on a component's use. Definitions describe characteristics of the component and include both functional requirements and performance constraints. Example facets presented earlier in this document exemplify the construction and uses for facets in descriptions. Verifications describe conditions that must be true for the component to be considered correct. Each verification may be accompanied by a *justification* that provides support for the verification.

A component is defined using the following syntax:

```
component d2a(bits::in bitvector[width], sig:: out real, width::nat)
    power::real;

  use conversion-functions;

  definitions
  begin logic
    b0: bv2r(bits,sig);
  end definitions;

  assumptions
  begin logic
    a1:
  end assumptions;

  verifications
  begin logic
    v1: sig <= bv2nat(max(bits)) * quantum;
  end verifications;
end d2a;

package conversion-functions;
  <conversion function definitions here>
end conversion-functions;
```

The *component* definition provides a template for defining three facets associated with definitions, verifications and assumptions respective. The keywords `assumptions`, `definitions`, and `verifications` delineate facet definitions. The format of each respective section is identical to a facet definition that assumes the same parameter list as the component parameter list. Adding parameters additional parameters is not allowed.

Assumptions

Terms defined in the assumptions facet represent conditions that must be true for the component to be used. Assumptions are used to document design assumptions, usage conditions, and other information elements assumed true by the designer. When a component is used in a system definition, the designer is obligated to show all assumptions are true in that context. Within the context of the facet, assumptions are treated as definitions.

Definitions

Terms defined in the definition facet represent the component's basic functional characteristics. As the name implies, definitions are treated as axioms and are true both in the facet context and when included in other

designs.

Verifications and Justifications

The verification language allows attaching *verification sequences* to verification terms. Each verification sequence is a sequence of related justifications that support the verification term. A justification is either a single term supporting the truth of the verification or a pair of terms related by a justification operator.

This needs to be tightened up and completed - wpa

Each term in the verification section may be accompanied by a verification sequence. The form of terms in the verification section is:

```
l: term <== justification;
```

where `justification` a sequence of terms, justification operations and reasons.

```
T_1
[==|==>|<==] <reason>;
T_2
[==|==>|<==] <reason>;
...
T_n
```

where $T_1 \ldots T_k$ represent terms and ==, ==>, <== represent equivalence, entailment and reverse entailment respectively. Each verification step is documented by a free form $< reason >$ that justifies the verification step. Such reasons may take various forms including formal verification, simulation, testing, inspection, and assumptions. No restriction is placed on the reason to allow heterogeneous justifications for verification steps.

Component Semantics

Component semantics are defined as a facet where: (i) the assumptions and verifications sections become locally defined facets; and (ii) the definitions facet defines terms for the facet. Specifically, the component `d2a` is equivalent to the facet:

```
facet d2a(bits::in bitvector[width], sig:: out real, width::nat)
    power::real;

  use conversion-functions;

  facet assumptions
  begin logic
    a1:
  end assumptions;

  facet verifications
  begin logic
    v1: sig <= bv2nat(max(bits)) * quantum;
  end verifications;

begin logic
  b0: bv2r(bits,sig);
end d2a;
```

When included in a definition, the d2a facet equivalent to the component is included in the standard manner. The assumptions and verifications facets are accesed using the standard *facet.label* notation. The automatic reference to the functional definition from the name is the intuitive interpretation.

The notation:

```
b0: d2a(I,O,8);
```

includes a facet called d2a with parameters instantiated with I, O and 8 respectively.

The assumptions and verifications component elements are accesses using the canonical access mechanism. Specifically, d2a.assumptions refers to the facet:

```
facet d2a.assumptions(bits::in bitvector[width], sig:: out real, width::nat)
  power::real;
begin logic
  a1:
end d2a.assumptions;
```

while d2a.verifications refers to the facet:

```
facet d2a.verifications(bits::in bitvector[width], sig:: out real,
  width::nat)
  power::real;
begin logic
  v1: sig <= bv2nat(max(bits)) * quantum;
end d2a.verifications;
```

Note that it is possible to access term v1 in component d2a using the notation d2a.verifications.v1.

Example 24 (One bit adder) *Consider the following one bit adder component definition:*

```
component one-bit-adder(x,y,cin::in bit; z,cout::out bit)
  begin

  definition state-based
    z'= x xor y xor cin;
    cout' = x and y;
  end definition;

end one-bit-adder;

component two-bit-adder(x0,x1,y0,y2::in bit; z0,z1,c::out bit)
  begin

  definition logic
    cx::bit
    b0: one-bit-adder(x0,y0,0,z0,cx);
    b1: one-bit-adder(x1,y1,cx,z1,c);
    delay = b0.delay+b1.delay;
  end definition;

  verification logic
    <definition of two bit adder correctness here>
  end verification;

end two-bit-adder;
```

Interfaces and Bodies

A common program writing scheme is the separate presentation of a module *interface* and module *body*. The interface represents visible module aspects while the body presents their implementations. Using the export clause, Rosetta allows the explicit definition of any facet's interface. However, no explicit means is provided thus far for separately defining a facet's interface and body.

Separate specification units are achieved using the `interface` and `body` specification directives. Both are defined as:

```
facet interface find(k::in keytype; i::in array[T]; o::out T) is
  power::real;
begin state-based
end find;
```

The interface specification defines visible labels and the facet interface. The `find` interface defines a parameter list and the physical variable `power`.

```
facet body find(k::in keytype; i::in array[T]; out::out T) is
begin state-based
    l1: key(o') = k;
    l2: elem(o',i);
end find;
```

The body specification defines labels that are not visible. Here, two terms are defined that will not be accessible outside the `find` facet. Together, the interface and body specifications define the same `find` facet defined in the introduction. Specifically, they are together equivalent to the conjunction of the interface and body facets with all labels defined in the interface exported. For the `find` specification, this results in:

```
facet interface find(k::in keytype; i::in array[T]; o::out T) is
  power::real;
  export power;
begin state-based
    l1: key(o') = k;
    l2: elem(o',i);
end find;
```

Because both packages and components are defined using facet semantics, defining interface and body specifications for these constructs follows from the above definitions.

The interface and body constructs are purely syntactic as their semantics is directly defined using existing facet construct. Thus, all facet operations and restrictions apply to interfaces and bodies. The primary difference being that reference to a facet automatically references the conjunction of it's associated interfaces and bodies. Given the previous definition of a `find` interface and body, referring to `find` implicitly refers to their conjunction.

More generally, given interface $(F_{I_1}, F_{I_2}, ..., F_{I_n})$ and body $(F_{B_1}, F_{B_2}, ..., F_{B_m})$ specifications for any construct, referencing the associated facet label refers to the conjunction of all facet interfaces and bodies associated with that label. Formally:

$$F = \bigwedge_{i=1..m} F_{I_i} \wedge \bigwedge_{j=1..n} F_{B_j}$$

This is the intuitive definition as reference to the name cannot differentiate between the several defining body and interface constructs.

Although interfaces and bodies are unrestricted facets, stylistically several rules apply. Specifically: (i) do not introduce new parameters in body specifications; (ii) do not associate different definitions with the same label; and (iii) do not use export clauses in interface and body specifications. It is unwise to introduce new interface parameters in a body specification. Conjunction defines the semantics of such an addition, but it clearly violates the spirit of the interface specification by defining a new visible label.

Labels associated with parameters, physical variables and terms should not be associated with different definitions in body and interface specifications. It is certainly possible to define the same label differently in the body and interface. However, facet conjunction rules and label distribution rules imply that the *entire* definition will be visible. This has the same effect as adding a new parameter in a facet body - a new, visible label is being added outside the interface.

Explicit use of export clauses in interfaces and bodies is to be avoided. An export clause in an interface does nothing. An export clause in a body usurps the information hiding achieved by the definition constructs.

Example 25 (Adder Interface and Body) *Example goes here...*

Example 26 (Component Interface and Body) *Example goes here...*

Example 27 (Package Interface and Body) *Example goes here...*

Example Specifications

Note: The following definitions have not been updated since the elimination of the system construct - wpa

Example 28 (Telecommunication System) *The following is a specification of a simple telecommunication system. This specification parallels the definitions provided as examples earlier in the document.*

```
system commPackage
  definitions
    -- Define a library of basic communications functions
    -- parameterized over data and transmission types.
    facet codeLib(D,T::TYPE)
      export encode, decode, decode_encode;
    begin requirements
      -- Define encoding and decoding functions
      encode: D->T;
      decode: T->D;
      -- Define a duality theorem for encode and decode
      decode_encode: theorem (forall d::T) decode(encode(d))==d;
    end codeLib;
end commPackage;
```

The `commPackage` system is a package that defines facets for reuse in other systems. The `commPackage` system provides a facet `codeLib` that defines properties of encoding and decoding functions. The specifier provides definitions for encoding and decoding functions as parameters to the `codeLib` facet. A single theorem is defined in `codeLib` that states the decode function is the inverse of the encode function. In this example, there are no assumptions or verifications.

```
system telecom(D,T::TYPE);
  -- Define systemwide types for data and transmission values
  includes
    commPackage(D,T);
  definitions
    -- Define a very simple ideal transmitter
    facet tx(D,T::TYPE, data::in D, output::out T)
    begin requirements
      codeLib(D,T);
      txdef: output = codeLib.encode(data);
    end tx;

    -- Define a very simple ideal receiver
    facet rx(D,T::TYPE, data::in T, output::out D)
    begin requirements
      codeLib(D,T);
      rxdef: output = codeLib.decode(data);
    end rx;

    -- Define a very simple ideal channel
    facet channel(T::TYPE, datain::in T, dataout::out T)
    begin requirements
      l: dataout=datain;
    end channel;

    -- Put it all together
    facet commSys(D,T::TYPE, datain::in D, dataout::out D)
    begin requirements;
      chan1,chan2::T;
      tx: tx(datain,chan1);
      ch: channel(chan1,chan2);
      rx: rx(dataout,chan2);
    end commSys;

  assumptions
    -- There are no assumptions for this system
    true;

  verifications
    -- Verify that given an input value, the output value will always
    -- be equal to it.
    bisim: (forall (d1,d2::D)
                commSys(D,T,d1,d2)) => d1=d2 <== <PVS proof>;
end telecom;
```

The telecomm system uses information from system `commPackage` *to define a simple telecommunications system. First,* `commPackage` *is included to make general telecommunications systems information available. In the definitions section, facets are defined for a transmitter, receiver and ideal channel. In the first two facets,* `codeLib` *is included to provide encoding and decoding functions. The final facet,* `commSys` *connects a transmitter and receiver together over a single channel. The verification section includes a single term stating that the input to the commSys is equal to its output. The justification for this verification is simply stated to be a proof using the PVS system.*

The `telecomm` system is an exceptionally naive specification of a telecommunications system include only to demonstrate system specification capabilities. We will now specify a similar, but more complex system representing a more complex system with more interesting systems requirements.

Example 29 (Temporal Telecomm System Specification) *Consider the definition of a telecommunications system using temporal specification techniques. Specifically, the discrete time requirements specification and monotonic logic specification capabilities are used together to define a heterogenous system specification.*

First, define two systems representing ideal transmitter and receiver configurations. Effectively, these specifications define baseline requirements.

```
system transmitter(D,T::TYPE)
defines
  facet tx(d::D; t::T)
  begin discrete-time
    t'=encode(d);
  end tx;
end transmitter;

system receiver(D,T::TYPE)
defines
  facet rx(t::T; d::D)
  begin discrete-time
    d'=decode(t);
  end rx;
end transmitter;
```

Define constraint facets for representing components.

```
system constraints;
definitions
  -- Define a component power constraint template
  facet power(c::real);
    p::real;
  begin requirements
    p <= c;
    compose:[x::set[real]]::real = sum(x);
  end power;

  -- Define a component clock speed constraint template
  facet clockspeed(c::natural);
    freq::natural;
  begin requirements
    freq <= c;
    compose:[x:set[natural]:natural = min(x);
  end clockspeed;
end constraints;
```

The constraints facets define templates for specifying constraints for components. They will be used by specifying values for parameters that indicate design constraints. The compose operators indicate how constraints are composed across several components. This will be used to determine if an architecture specification meets higher level constraint requirements.

Using the two sets of systems, it is possible to define a constrained transmitter/receiver pair using an ideal channel:

```
system commSys(D,T::type)
includes
  -- Include functional transmitter and receiver systems as well as
  -- constraint models.
  transmitter(D,T), receiver(D,T), constraints;
```

The ***includes*** *section lists the systems defined previously for representing transmitters, receivers and constraint blocks. The transmitter and receiver systems are instantiated with data types* `D` *and* `T` *from the* ***commSys*** *system interface. When* ***commSys*** *is used in a specification, the* `D` *and* `T` *types are instantiated throughout the system definition.*

```
defines
  -- Define functional correctness as output in the next state is
  -- input in the current state
  facet functional-system(i::in D; o::out D)
  begin requirements
    o'=i;
  end functional-system;

  -- Define systems level constraints
  facet constrained-system
    p::real;
    freq::natural;
  begin requirements
    power::power(10);
    clockspeed::clockspeed(5);
    p=power.p;
    freq=clockspeed.freq;
  end constrained-system;

  -- Define system level requirements as functional correctness
  -- combined with constraints
  system-req = functional-system and constrained-system;
```

The initial facets in the ***defines*** *section specify system level function and constraints. The functional specification is trivial specifying that a communication system simply takes its input and produces it as output. The systems level constraints specify top level constraints of clockspeed and power consumption. Here the facets from the* ***constraints*** *system are used to specify relationships. Finally, a new facet is defined to represent the overall systems requirements.* ***system-req*** *combines constraints and functional requirements into a single system definition using facet conjunction.*

```
  -- Define an architecture for the transmission system as transmitter
  -- connected directly to a receiver
  facet functional-arch(d1,d2::D)
  begin requirements
    t::T;
    tx: transmitter.tx(d1,t);
    rx: receiver.rx(t,d2);
  end functional-arch;
```

```
  -- Define an architecture for constraint requirements, one block for
  -- each component
  facet constraint-arch;
    p::real;
    freq::natural;
  begin requirements
    tx: constraints.power(5.0) and constraints.clockspeed(50);
    rx: constraints.power(3.0) and constraints.clockspeed(50);
    l1: p = power.compose({tx.p,rx.p});
    l2: freq = clockspeed.compose({tx.freq,rx.freq});
  end constraint-arch;

  -- Define the complete architecture specification by combining the
  -- functional architecture and the constraint architecture
  architecture = constraint-arch and functional-arch;
```

With system level specifications in place, it is possible to define an architecture representing an initial design decomposition. In this architecture, three components are connected in series. Facets from the `transmitter` *and* `receiver` *systems are used in conjunction with the locally defined* `channel` *facet to define a simple system. The transmitter outputs values that appear at the channel as inputs. The channel the outputs the same value which appears at the input of the receiver. Finally, the receiver decodes its input and produces data.*

The constraint architecture is similar except here power and clockspeed are defined. Note that labels are shared between the functional and constraint architecture facets. When conjuncted together, distribution laws cause the constraint specifications to be associated with their appropriate transmitter or receiver specifications. This combination is accomplished in the `architecture` *facet definition.*

```
verifications
  -- Simple property inclusion style verification.  The properties of
  -- the system requirements must be exhibited by the architecture.
  behavioral-equivalence: architecture implies system-req;

end commSys;
```

Finally, a verification condition is specified that defines a relationship between systems level specifications and architecture specifications. Here a simple property inclusion relationship is defined. Specifically, the architecture facet should imply all properties of the system requirements facet.

Removing commentary from the system specification results in:

```
system commSys(D,T::type)
includes
  -- Include functional transmitter and receiver systems as well as
  -- constraint models.
  transmitter(D,T), receiver(D,T), constraints;

defines
  -- Define functional correctness as output in the next state is
  -- input in the current state
  facet functional-system(i::in D; o::out D)
  begin requirements
    o'=i;
```

```
  end functional-system;

  -- Define systems level constraints
  facet constrained-system
    p::real;
    freq::natural;
  begin requirements
    power::power(10);
    clockspeed::clockspeed(5);
    p=power.p;
    freq=clockspeed.freq;
  end constrained-system;

  -- Define system level requirements as functional correctness
  -- combined with constraints
  system-req = functional-system and constrained-system;

  -- Define an architecture for the transmission system as transmitter
  -- connected directly to a receiver
  facet functional-arch(d1,d2::D)
  begin requirements
    t::T;
    tx: transmitter.tx(d1,t);
    rx: receiver.rx(t,d2);
  end functional-arch;

  -- Define an architecture for constraint requirements, one block for
  -- each component
  facet constraint-arch;
    p::real;
    freq::natural;
  begin requirements
    tx: constraints.power(5.0) and constraints.clockspeed(50);
    rx: constraints.power(3.0) and constraints.clockspeed(50);
    l1: p = power.compose({tx.p,rx.p});
    l2: freq = clockspeed.compose({tx.freq,rx.freq});
  end constraint-arch;

  -- Define the complete architecture specification by combining the
  -- functional architecture and the constraint architecture
  architecture = constraint-arch and functional-arch;

verifications
  -- Simple property inclusion style verification.  The properties of
  -- the system requirements must be exhibited by the architecture.
  behavioral-equivalence: architecture implies system-req;

end commSys;
```

Chapter 8

Semantic Issues

```
%% This chapter is undergoing so many changes that you might as well
%% put change bars around the whole thing....
```

8.1 Co-algebraic Semantcs of Facets

8.1.1 Co-algebra Abstract Syntax

The semantic basis for a facet is a co-algebra defined with respect to the system state. The definition of the co-algebra provides definitions of possible observations of a system state and constraints on those observations. The abstract syntax for the co-algebra has the following form:

$$\begin{array}{rcl} \texttt{coalgebra} & ::= & \langle \Gamma \rangle : \mathcal{S} \texttt{ where } \Pi \mid \mathcal{S} \\ \Gamma & ::= & \gamma \mid \gamma, \Gamma \\ \gamma & ::= & \sigma :: \tau \\ \Pi & ::= & \varphi \mid \varphi; \Pi \end{array}$$

where $\mathcal{S}$ is the system state over which observations are defined and each γ in Γ defines an observer on $\mathcal{S}$. Specifically, each γ of the form $\sigma :: \tau$ in Γ defines a function $\sigma(s :: \mathcal{S}) :: \tau$. Π defines a collection of boolean expressions over elements of Γ and $\mathcal{S}$ that constrain the observers of state. If a state type, $\mathcal{S}$, is specified without oberservers then the resulting co-algebra has no observable behaviors. Although legal, the definition is vacuous an is included only for completeness. Note that a co-algebra where $\Pi =$ `true` places no constraints on the observations and results in a signature definition. Note also that a co-algebra where $\Pi \vdash$ `false` is inconsistent.

The co-algebra has no concrete semantics in Rosetta and supports only defining facet values. It is important not to confuse the value of a facet with its associated abstract syntax tree representation. Specifically, using a constructed type to specify a facet specifies the abstract syntax of the facet, not its value.

8.1.2 Facet Semantics

The abstract syntax for a facet item as defined in Chapter 5 is:

$$
\begin{array}{lll}
facet & ::= & \texttt{facet}(\gamma_p)::\gamma_d;\mid \\
 & & \texttt{facet}(\gamma_p)::\gamma_d\ \texttt{is}\ \gamma_i;\gamma_e;\pi_t;
\end{array}
$$

Working from the abstract syntax, the co-algebra associated with the abstract syntax can be defined in terms of a signature a term set. The signature is defined as:

$$\langle \gamma_d, \gamma_i\gamma_e \rangle : \mathcal{S}\ \texttt{where}\ \pi_t;\ \pi_d;$$

where γ_p, γ_i, and γ_d are declarations from the facet parameter list, declarative region and included domain, respectively.

Elements of the co-domain signature must satisfy the termset formed from terms defined in the facet, π_t, and terms defined in the domain, π_d under the denotation function, δ. Give the a state, $s :: \mathcal{S}$ and the label for a declaration, l, the denotation function has the following form:

$$\delta(l@s) = l(s)$$

To interpret π_t and π_d with respect to a state,s, the denotation function is applied to each label. Specifically, the expression $x' = x + 1$ is interpretted in state s:

$$
\begin{array}{rcl}
x' = x + 1 & == & x@((next@s)(s)) = x@s + 1 \\
 & == & x(next(s)(s)) = x(s) + 1
\end{array}
$$

Recall that a declaration of the form $\sigma::\tau\ \texttt{is}\ \varphi$ defines a constant value, σ. When interpreting terms, the denotation function will not be explicitly applied to these definitions as they will be constant in all states. This clarifies the above notation:

$$
\begin{array}{rcl}
x' = x + 1 & == & x@next(s) = x@s + 1 \\
 & == & x(next(s)) = x(s) + 1
\end{array}
$$

The co-algebra signature defines a mapping from the system state, $\mathcal{S}$ to the type of each parameter, constant, and variable declaration. Thus, a declared item's value in a given state can be referenced by applying its associated function to that state. This includes function values defined in the state. For example, assume the following facet definition:

```
facet(i::input integer)::state_based is
  z::integer;
begin
  t1: p(i,z);
end facet;
```

results in the following co-algebraic definition:

$$\langle i{::}\texttt{integer}, z{::}\texttt{integer}, \mathcal{S}{::}\texttt{type}, next{::}{<}*(\texttt{s} :: \mathcal{S}){::}\mathcal{S}*{>}\rangle : \mathcal{S}\ \texttt{where}\ p(i,z);$$

Expanding the co-algebra notation results in the following declarations and type specification:

$$\begin{array}{rcl} \mathcal{S} & :: & {<}*(\texttt{s} :: \mathcal{S}){::}\mathcal{S}*{>} \\ i & :: & {<}*(\texttt{s} :: \mathcal{S}){::}\texttt{integer}*{>} \\ z & :: & {<}*(\texttt{s} :: \mathcal{S}){::}\texttt{integer}*{>} \\ next & :: & {<}*(\texttt{s} :: \mathcal{S}){::}{<}*(\texttt{s}{::}\mathcal{S}){::}\mathcal{S}*{>}*{>} \\ \\ t1 & : & \texttt{forall}(i{::}\texttt{integer} \mid p(i,z)); \end{array}$$

8.2 Preliminary Definitions

Definition 2 (Rosetta Term Language) *$\mathcal{R}$ is the language consisting of all legal Rosetta strings.*

Definition 3 (Rosetta Variable Free Termlanguage) *$\mathcal{R}_\rfloor$ is the variable free termlanguage associated with Rosetta. In the language, the variable free termlanguage is synonymous with the type* `universal` *containing all Rosetta things.*

8.3 Items

The basic unit of Rosetta semantics is an *item* used to represent all Rosetta constructs. An item behaves as a 3-tuple, $\langle l, t, v\rangle$ consisting of a: (i) label; (ii) type; and (iii) value.

Definition 4 (Item) *An* item *is an abstract data structure defined as follows:*

- `l` – *a* label *naming the item referenced by the function* `meta.label(i::item)::label`
- `t` – *a* set *defining the* type *of* `i` *referenced by the function* `meta.type(i::item)::set(universal)`
- `v` – *a* value *referenced by the function* `meta.value(i::item)::meta.type(i)>`

A Rosetta item is consistent if its value is an element of its type. This definition must hold for both static definitions and during evaluation.

Axiom 1 (Value Consistency) `forall(i::item | meta.value(i) in meta.type(i))`

Any item's value is an element of it's associate type set.

All parsed Rosetta definitions are internally represented as items. The function `meta.parse` is a predefined operation that takes any element of $\mathcal{R}$ and returns the item associated with it. `meta-string` is the inverse function producing a string representation in the concrete syntax associated with `parse`:

Definition 5 (Parsing) `meta.parse` *transforms a string into its associated Rosetta item. Effectively, meta.parse associates semantics with string representations of Rosetta items. Many* `meta.parse` *representations may exist for various concrete Rosetta syntaxes.* `meta.parse(s::string)::item` *obeys the following axioms:*

Axiom 2 (Parse Consistency) *`forall(i::item | meta.parse(meta.string(i))=i)`*

Thus, parsing the string representation of an item results in the original item.

Axiom 3 (String Consistency) *`forall(s::string | s in R=>(meta.string(meta.parse(s))=s)`*

If s is a syntactically correct Rosetta language structure, then the string representation of the parsed string is the parsed string. Note that many `meta.parse` and `meta.string` pairs may exist that satisfy this property.

Labels are used to reference items in Rosetta specifications. To aid in the dereferencing process, any label used in a specification for any action other than labeling new constructs refers to the value of the item associated with the label.

Definition 6 (Dereferencing) *Let σ_k be the label of a Rosetta item declared in facet, F, where:*

$$\delta(F) = \langle \sigma_0 :: \tau_0, \ldots, \sigma_m :: \tau_m \rangle :: \mathcal{S}$$

The value of σ_k with respect to an arbitrary $s :: \mathcal{S}$ is define as:

$$\sigma_k @ s == \sigma_k(s)$$

where $\sigma_k @ s$ is the abstract syntax for an item reference and $\sigma_k(s)$ is the application of σ_k from the co-algebra, $\delta(F)$ to a state, s.

The meta function `meta.value(l::label,s::S)` embodies the deferencing operation such that:

```
x@s == meta.value(x,s)
```

where `x` is a label and `s` is a system state.

The deferecing function maps an item label to its associated value with respect to a facet co-algebra over some state. given this definition, the following definitions hold where l is an item label:

$$\begin{aligned}
\delta(l) &== \delta(l @ s) \\
&== l(s) \\
\delta(l') &== \delta(l @ next(s)) \\
&== (l(next(s))(s)) \\
\delta(l @ (t+2)) &== (l(next(s))(t+2)) \\
\delta(f(x)) &== (f(s))(x(s))
\end{aligned}$$

Because Rosetta is statically scoped, the context of a declaration can be determined at analysis time regardless of state. Thus, given a item label in context it is possible to determine the item type using the meta-function `meta.type`.

Definition 7 (Typing) *The type of an item defined in a context, Γ, is defined based on the most recent declaration in scope. Specifically, if Γ is the context of a label reference and $\Gamma \vdash$ `x::T`, then the meta function `meta.type` is defined as `meta.type(x)=T`.*

```
%% Removed Variable and Constant Items Section in deference to
%% the types chapter.
```

8.4 Value Item

A *value item* is a constant item representing a specific, atomic Rosetta value. Specifically, a value item is an item whose value is constant and known. Each value item's label is the same as the string associated with its value. Thus, the label for the item associated with the value 5 is the string "5". When the label "5" appears in a Rosetta specification, it resolves to the value 5. `literal` values are necessarily of type `element`.

Example 30 (Value Item Example) *The value `5` is represented by the item `i` such that:*

```
meta.label(i)= ``5'';
meta.type(i)=element;
meta.value(i) = 5;
```

Axiom 4 (Value Parse and String) *If an item's label is equivalent to the string associated with it's value, then printing the item prints only the label.*

```
forall(i::item | meta.value(i)=meta.label(i) => meta.string(i)=meta.value(i))


%% I don't like this definition.  It should be replaced with
%% something more substantial.
```

Definition 8 (Literal Items) *The `meta.literal` function is true if and only if its argument is or references a value item. In general, a literal item is an item that satisfies the Literal Parse and String axiom and `meta.literal` can be defined as:*

```
meta.literal(i::item)::boolean is
   meta.value(i)=meta.label(i) => meta.string(i)=meta.value(i);


%% I'm not sure this literal function is necessary.  Even if it is,
%% this definition is pretty lame.
```

8.5 Type Items

A *type item* is a variable or constant item representing a Rosetta type. The type item label is the name of the type while the type item value is the set representing the possible values associated with the type. The type item's type is the supertype of the type. For example, the definition of `natural` is represented as:

```
natural::subtype(integer) is sel(i::integer | i>=0);
```

In this declaration, the label is `natural`, the subtype is the powerset of `integer` and the value is the set of all elements of `integer` greater than or equal to `0`.

In Rosetta, an uninterpreted type is defined as a variable type while interpreted types are typically defined as constants. It should be noted that the definition of these types parallels that of variable and constant definition. Rosetta types are simply variables and constants whose values are sets.

8.5.1 Uninterpreted Types

An uninterpreted type definition is achieved in Rosetta by the following declaration:

```
T::subtype(universal);
```

where `T` is the name of the type and `subtype(universal)` represents an arbitrary set. Rosetta provides a special keyword, `type`, that is equivalent to `subtype(universal)`. The above definition is equivalent to:

```
T::type;
```

An uninterpreted subtype definition is achieved in Rosetta by the declaration:

```
T::subtype(R);
```

where `R` is a known type. In this definition, `T` is contained in `set(R)`, but it's actual value is left unspecified. Although `T` is known to be a subset of `R`, its actual value is not known. The distinction between the definitional styles is that when the supertype is known, some type compatibility decisions can be made. When the supertype is not known, the type is not guaranteed to be compatible with any other type. Recall that when defining types, `subtype == set`. Thus `T::subtype(R)` is equivalent to `T::set(R)`.

Example 31 (Uninterpretted Type Item) *The type `R::type` is represented by an item `t` such that:*

```
meta.label(t) = ``R'';
meta.type(t) = set(universal);
meta.value(t) = undefined;
```

Example 32 (Uninterpretted Subtype Item) *The type `R::subtype(integer)` is represented by the item `t` such that:*

```
meta.label(t) = ``R'';
meta.type(t) = set(integer);
meta.value(t) = undefined;
```

8.5.2 Interpreted Types

An interpreted type definition is achieved in Rosetta by defining a value for a type variable. Specifically, by defining a value for the type it is made constant can can be interpreted by tools. Such a type is defined using following declaration:

```
T::subtype(R) is S;
```

where `T` is the type name, `R` is the supertype, and `S` is the set defining the type value. As with other constant definitions, this type definition is equivalent to the declaration:

```
T::subtype(R);
```

and the associated term:

```
T = S;
```

It should be noted that the set `S` may be any expression of type set such that `S in subtype(R)` or equivalently `S in set(R)`. Specifically, types may be expressed by comprehension over the subtype or any other type so long as the result is contained in `R`.

Example 33 (Interpretted Type Item) *The predefined type* `bit` *can be defined as:*

```
bit::subtype(natural) is 0,1;
```

is represented by an item `t` *such that:*

```
meta.label(t) = ''bit'';
meta.type(t) = set(natural);
meta.value(t) = 0,1;
```

It is important to note that types behave as variables and constants in all ways. Keeping this in mind makes this section somewhat redundant as rules for variable and constant definition are simply repeated for types.

```
%% Removed type compatibility section in deference to the labeling
%% chapter work.
```

8.6 Facet Items

A facet item is an item whose value is of type facet. Like all Rosetta specification structures, facets are simply values constructed in the language. In this case, a facet is a theory representing one model of a system. Note that as packages, domains, interactions and components are defined from the basic facet semantics, these items represent subclasses of facet items.

The basis of the facet semantics is drawn from algebraic semantics. Each facet can be viewed as a collection of axioms defined over a signature. Thus, the semantic definition of a facet can be viewed as the quotient algebra associated with the facet with respect to equational reasoning.

8.6.1 Algebras in Rosetta

We begin by clearly defining the notion of a *domain algebra*. To do this we need to define sorts, constant and operation symbols and operation symbols with arguments from the set of sorts.

For an arbitrary algebra A, let $\boldsymbol{A_S}$ represent the set of sorts of A, with members $S_1 ... S_k$, where S_i is the i'th sort of A. Let $\boldsymbol{A_N}$ represent the set of constant symbols in $\boldsymbol{A_S}$. Alternatively, $n \in \boldsymbol{A_N}$ can be said to represent a function $n :\rightarrow S_i$ for some $i \leq k$.
Let $\boldsymbol{A_O}$ represent the set of functions of A with arguments from S_i for some $S_i \in \boldsymbol{A_S}$. The result of evaluating any function $f(s_p, ..., s_q) \in \boldsymbol{A_0}$ must be within S_j for some $j \leq k$.

Given two algebras A and B, the signature of A is said to extend that of B precisely when all the following conditions are fulfilled:

1. $\forall S_i \in \boldsymbol{B_S}, S_i \in \boldsymbol{A_S}$
2. $\forall n \in \boldsymbol{B_N}, n \in \boldsymbol{A_N}$
3. $\forall f(s_p, ..., s_q) \in \boldsymbol{B_O}, f(s_p, ..., s_q) \in \boldsymbol{A_O}$

The presentation of A is said to extend that of B precisely when any axiom in the presentation of B exists in the presentation of B.
If both the presentation and signature of A extend those of B we say that the algebra A is an extension of the algebra B.

In the theory of Rosetta, the algebras we encounter will be both domain and facet algebras. When we refer to an algebra, we intend the algebra which is denoted by both a signature and a presentation.

Domains

For a domain D_1 we define sorts, constant symbols and functions as below.

The sorts, making set $\boldsymbol{D_{1S}}$ are defined to be:

1. All declarations of types and variables that are defined in the Rosetta domain definition of D_1 or a parent of it in the domain hierarchy.

The constant symbols, making set $\boldsymbol{D_{1N}}$ are defined to be all declarations of constants that are defined in the Rosetta domain definition of D_1 or a parent of it in the domain hierarchy.

The function symbols, making set $\boldsymbol{D_{1O}}$ are defined to be:

1. All declarations of domain-specific functions in the Rosetta domain definition of D_1 or a parent of it in the domain hierarchy

Facets

For a facet algebra F_1 of a facet specification f_1 that extends a domain D_1 we define sorts, constant symbols and functions are defined as below. The sorts, making set $\boldsymbol{F_{1S}}$ are defined to be:

1. Parameters of f_1.
2. All declarations within f_1.
3. All declarations from D_1 or a parent of it in the domain hierarchy

The constants, making set $\boldsymbol{F_{1S}}$ are defined to be those facet constants defined in the Rosetta facet specification for any facet that conforms to this facet algebra, as well as any constants from D_1 or a parent of it in the domain hierarchy.

The operations, making set $\boldsymbol{F_{1O}}$ are defined to be:

1. Any function that is declared in f_1.
2. All domain-specific functions in the Rosetta domain definition of D_1 or a parent of it in the domain hierarchy.

8.6.2 Facet Abstract Syntax

Facet types are defined by providing a *signature* defining items representing variables and constants, a *domain* extended by the facet, and a collection of *terms* that define axioms over items defined in the signature and domain. Thus, we define a facet's abstract semantics in terms of these elements.

Definition 9 (Facet Type) *A* facet*, F, is an item with the following properties:*

```
%% Need an image function for the labels operation.  Range might
%% work.  It's still rather fouled up.
```

- *meta.signature(F) is the* signature *of the facet and consists of a set of declared and visible items over which terms are defined.*
- *meta.terms(F)=T is the set of* terms *defined by the facet and consists of a set of items of type term.*
- *meta.domain(F) is the* domain *extended by the facet and is a single facet of type domain.*

The type `facet` is defined as the set of all possible facets. Thus, a facet item, `f`, is an item whose value is of type facet. Specifically:

```
f::facet;
```

declares a variable item, `f`, of type `facet`. A facet constant is defined in the canonical Rosetta fashion:

```
f::facet is <exp>;
```

where <*exp*> is an expression of type facet, typically defined using the facet algebra.

Subtypes of the `facet` type are defined based on the domain used by a facet. Two facets are a member of the same subtype if they extend the same domain. If *domain* is the name of a specific Rosetta domain, then the facet subtype *domain* is defined as follows:

```
domain :: subtype(facet) is sel(f::facet | meta.domain(f)=domain);
```

Example 34 (Facet Subtype) *The following definition specifies that* `sb` *is a facet variable of subtype* `state_based`*:*

```
sb :: subtype(state_based);
```

```
%% Don't forget to define domain subtypes/inclusion as well...
```

Most facets are defined using the concrete facet syntax:

```
facet facet-label(param-list) is
  export-list;
  declarations;
begin domain-name
  term-list;
end facet facet-label;
```

where *facet-label* names the facet, *param-list* is an optional set of facet parameters, *export-list* is the list of exported labels, *declarations* is the set of local declarations, *domain-name* is the name of the domain extended by the facet, and *term-list* is the list of facet terms defined over its signature.

Example 35 (Facet Item) *Let the following be a hypothetical facet item:*

```
facet f(p1::T) is
  p2::R;
begin logic
  t1:P(p1,p2);
end f;
```

Given that `i=meta.parse(f)`*, the following definitions hold:*

- `meta.label(i)=f`
- `meta.type(i) = logic`
- `meta.value(i) = v`

where the following definitions hold for the value, `v`*, of* `i`*:*

- `meta.signature(v)={p1::T,p2::R}+meta.signature(domain)+{t1}`
- `meta.domain(v)=logic`
- `meta.terms(v)={t1}+meta.terms(domain)`

Facet Signature

```
%% Intro to the signature here.  The elements of the signature
%% follow - parameter, defined and included items.
```

Definition 10 (Paramter List) *The* `meta.parameters(v)` *function is defined as the set of labels included in the parameter list of facet value* `v`*.*

Definition 11 (Export List) *The* `meta.visible(v)` *function is defined as the set of labels from facet* `v` *that are visible in an including facet. When one facet is included in another, these labels are included in the signature of the including facet using the notation* facet-name.label*. If the included facet is a domain or package, the exported labels of the included facet are added to the signature without modification.*

Facet Domain

A facet's domain is itself a facet that defines a specification vocabulary for writing specifications. Sematically, the facet extends its domain by adding declarations and terms. The signature and term set start with declarations from the domain and add item and term declarations to them. Unlike inclusion of facets in terms, the inclusion of a domain adds directly to the signature of the including facet. Specifically, the dot notation is not used to access domain labels allowing them to be referenced directly.

Facet Terms

```
%% Need to define both a facet instance term item and a boolean term
%% item.  Also make the label optional on boolean term items.  Add
%% the let over terms as well.
```

Terms are special Rosetta objects that represent declarations within the facet body. Specifically, any declaration using the syntax

```
l:t;
```

where `l` is a label and `t` is either a boolean expression or a facet expression declares a *term item*.

For any term item, t, the term item label names the term and its value is of type boolean or facet. If a term's label is undefined, the term cannot be referenced by name. Unlabled terms are allowed only in the case of boolean valued terms.

Example 36 (Boolean Term Item) *The term `l:F(x)=5;` defined in the logic domain is represented by an item t such that:*

```
meta.label(i) = l;
meta.type(i) = boolean;
meta.value(i) = F(x)=5;
```

This term is a boolean term because its value is defined to be a boolean expression.

Facet valued terms must be labled to achieve a form of elaboration called *relabling*. When a facet is included as a term, the facet is elaborated a relabeled. The effect is that the instantiated facet is a fresh facet. Renaming the facet assures that no name conflicts will occur. Consider the following terms:

```
and1: and-gate(x1,x2,z2);
and2: and-gate(x3,x4,z2);
const: and-gate.power =< 5mw;
```

Two `and-gate` devices are instantiated and a term constraining the power consumption is defined over an internal variable. Without relabeling, it is impossible to determine which power constraint is being set. If both are assumed, then it becomes impossible to set the power consumption differently for both. Thus, the `and-gate` device is copied twice and renamed by the label. In this case, the constraint is obvious:

```
and1: and-gate(x1,x2,z2);
and2: and-gate(x3,x4,z2);
const1: and1.power =< 5mw;
const2: and2.power =< 6mw;
```

Relabeling removes and ambiguity or aliasing that might occur due to the inclusion of multiple copies of the same facet.

```
%% Facet Algebra discussion here...  Probably move later in the
%% document after facets are discussed.  May need to move the entire
%% term discussion as well...
```

Example 37 (Facet Term Item) *The term* `and1:and-gate(x1,x2,z);` *defined in the logic domain is represented by an item t such that:*

```
meta.label(i) = and1;
meta.type(i) = facet;
meta.value(i) = and-gate(x1,x2,z);
```

This term is a facet term because its value is defined as a facet type. Note that the labeling of the term effectively renames the facet.

The Facet let Form

Rosetta defines a `let` form in the definition of expressions that allows a local variable to be defined and used in the scope of the `let` form. This concept is extended to terms allowing the definition of local variables across multiple terms. The syntax for the facet `let` form is:

```
let declaration be expression in
  term-list
end let;
```

This form is equivalent to adding *declaration* `is` *expression* to the delcarative section, but making declarations visible only in the terms from *term-list*. Note that the declaration is shared among the terms and is not equivalent to putting each term in its own let form. Variables in let forms can never be exported.

Example 38 (Facet Let Form) *In the following facet, a* `let` *form is used to define a common declaration,* `pi`*, used to calculate circumference and radius:*

```
facet letform(r::in real) is
  c,a::real;
begin null
  let pi::integer be 3.14159 in
    t1: c = 2*pi*r;
    t2: a = pi*r^2;
  end let;
end facet letform;
```

This facet is equivalent to:

```
facet letform(r::in real) is
  c,a::real;
  pi::real is 3.14159;
begin null
  t1: c = 2*pi*r;
  t2: a = pi*r^2;
end facet letform;
```

Note that if terms appear in the facet declaration, but outside the let form, the variable defined by the let form is not visible.

8.6.3 Facet Semantics

A facet's semantics is defined based on its *signature*, *domain* and *terms* that extend that domain. This corresponds to the concepts of a formal system and theory presentation in traditional formal systems. In traditional definitions, the presentation is defined with the formal system implicitly present. As Rosetta supports interaction between domains, the formal system must be explicitly present in the facet specification.

Definition 12 (Facet Semantics) *Using the abstract syntax functions* `signature`, `domain`, *and* `terms`, *the semantics of a facet* `f` *is defined as the quotient algebra of the presentation* `signature(f)` *and* `terms(f)`.

```
%% Check to see if the following really is true...
```

The quotient algebra is obtained by finding the closure of the presentation with respect to classical equational reasoning. Specifically, the algebra resulting from all possible substitutions of equal expressions in the terms of the facet. The quotient algebra as defined in this manner is from this point forward refered to as the theory of `f` and is denoted `meta.theory(f)`. Note that the function `meta.theory(f)` has no complete realization in any Rosetta implementation.

The definition of semantic correctness is simply consistency of terms with respect to the specified semantic domain. If no term or declaration introduces an inconsistency, then the facet definition is semantically correct. The function `meta.consistent` denotes consistency of a facet.

Definition 13 (Semantic Correctness) *A facet is semantically correct,* `meta.consistent(F)`, *if its associated theory does not contain* `FALSE` *as an element. Specifically:*

$$\neg(FALSEinmeta.theory(f))$$

The semantic correctness of any given facet is dependent on both its term set and its domain. Thus, it is impossible to determine semantic correctness without knowing the specification domain. This is expected as in Rosetta, the domain is specified explicitly with each facet.

Semantic correctness as consistency is not decidable in the general case. Thus, pragmatics of semantic checking insist on a human assisted process. Where appropriate, Rosetta will be restricted to assure automatic semantic correctness determination. Such situations necessarily include operational facets where executability needs to be insured.

Definition 14 (Binding) *A binding is an association between an item and a value. A single binding is expressed by the term* `i=v` *where* `i` *is the item label and* `v` *is an expression. A binding is consistent in the context of a facet,* `f`, *if:*

```
v in meta.signature(f)
v in meta.value(i)
meta.consistent(f')
```

where `f'` is a facet such that:

```
meta.signature(f') = meta.signature(f)
meta.domain(f') = meta.domain(f)
meta.terms(f') = meta.terms(f) + i=v
```

Definition 15 (Environment) *An environment is a set of bindings associated with a facet. An environment,* `E`, *is consistent if:*

```
forall(i=v in E | i in meta.signature(f))
forall(i=v in E | v in meta.value(i))
meta.consistent(f') where f' is a facet such that:

  meta.signature(f') = meta.signature(f)
  meta.domain(f') = meta.domain(f)
  meta.terms(f') = meta.terms(f) + E
```

Definition 16 (State and Possible States) *A state, `S`, of a facet, `f`, is a consistent environment such that every item is bound to a value:*

```
forall(i in meta.signature(f) | exists(v in meta.type(i) | i=v in S))
```

A facet's state set, `meta.states(f)`, is the set of states associated with the facet. If `meta.states(f)=empty`, then the facet is not satisfiable and is inconsistent. If `#meta.states(f)=1`, the facet is satisfiable and deterministic. If `#meta.states(f)>1`, the facet is satisfiable, but not deterministic.

```
%% Working here...
```

8.6.4 Parameterization and Instantiation

A formal parameter can be replaced by an actual parameter if the actual parameter is *type compatible* with the formal parameter. A formal parameter is type compatible with an actual parameter if all legal instances of the actual parameter are type safe with respect to the formal parameter. Although substitution is a purely syntactic operation, the objects associated with labels must be referenced to determine the safeness of the substitution.

Definition 17 (Type Compatbility) *An actual parameter, `a`, is type compatible with respect to a formal parameter, `p`, if and only if:*

```
meta.compatible(a,p::label) :: boolean is meta.ran(a) :: meta.type(p);
```

Facet parameters as all Rosetta parameters are treated as universally quantified variables. The definition:

```
facet A(x::T,y::R) is
  ...
end A;
```

can be viewed conceptually as:

```
forall(x::T |
  forall(y::R |
    facet A is
      ...
    end A;
  )
);
```

Although not a legal Rosetta definition, the facet reflects the behavior of a parameter. Instantiating parameters is a process of applying the standard universal elimination operation in classical logic. Specifically, replacing a formal parameter with an item of compatible type and eliminating the universal quantifier associated with the variable. This process is referred to as *instantiation* of a facet.[1]

Definition 18 (Facet Instance) *A facet instance is defined as a collection of terms that are consistent with the facet definition and potentially extend the facet definition. Specifically, given a facet F_n, F_m defines an instance of F_n if and only if:*

$$\texttt{meta.consistent}(\texttt{F}_\texttt{m}) \wedge \forall \texttt{t} :: \texttt{meta.terms}(\texttt{F}) \cdot \texttt{meta.terms}(\texttt{G}) \vdash_{\texttt{D}_\texttt{n}} \texttt{t} \tag{8.1}$$

F_m is an instance of F_n if it is consistent and every term in F_n can be derived from F_m.

Instantiating a parameterized item is replacement of a formal parameter with the label of an actual parameter that is type compatible. Instantiating parameters is the only syntactic mechanism for generating facet instances.

Definition 19 (Instantiation) *Given a facet with formal parameter `i` and an actual parameter `j`, such that `meta.compatible(j,i)` holds, the following defines the result of instantiating `i` with `j`:*

- *`meta.items(meta.instantiate(f,i,j))=meta.items(f)-{meta.item(i)}+{meta.itemj}`*
- *`meta.domain(meta.instantiate(f,i,j))=meta.domain(f)`*
- *`meta.terms(meta.instantiate(f,i,j))=meta.terms(f)[i/j]`*
- *`meta.types(meta.instantiate(f,i,j))=meta.types(f)[i/j]`*
- *`meta.pvars(meta.instantiate(f,i,j))=meta.pvars(f)[i/j]`*
- *`meta.visible(meta.instantiate(f,i,j))=meta.visible(f)[i/j]`*
- *`meta.params(meta.instantiate(f,i,j))=meta.params(f)-{i}`*

8.7 Facet Contexts

```
%% Still needs some work to deal with the static nature of types and
%% terms over time.  Specifically, l1: P(x) defines a constant of
%% type term whose value is P(x).  It cannot change over time.  The
%% value of x may change over time.  This needs to be cleaned up and
%% and some text added.
```

To this point, facets are static items with no temporal properties. When describing systems, it is necessary to specify how sequences of input changes effect the system's state and output. The Rosetta *context* provides the ability to reference facet instantiations at various points in time. This mechanism is provided syntactically using the "`@`" operation to explicitly reference a context.

Various domains use the notation `x@t` to reference the value of item `x` in some context `t`. This usually refers to a state or time value. Within domain specifications, the behavior of context objects is described without reference to their structure or interface behavior. Here we provide mechanisms that allow specifying the semantics of expressions as well as referencing items in various contexts.

`x@t` refers to the value of `x` in context `t`. Using the concept of context introduced in Section 8.2, the semantics of `x@t` is defined as `meta.derefValue(x,t)` or the value of `x` in context `t`.

Definition 20 (x@s) *The semantics of `x@s` is defined simply as `x` in context `t`:*

[1] Note that facet instantiation is defined identically to function application.

```
x@t == meta.derefValue(x,t);
```

Thus, when the notation `x@t` appears in an expression, it is in interpreted as the value of `x` at `t`. When a variable appears without explicit reference to context, the default context is defined as the current context. Thus, `x` refers to the value of `x` at the current time or in the current state.

8.8 Composition Operations

8.9 Label Distribution

Label distribution states and labeling operations distribute over declarations. Thus, `L:T1` $\circ$ `L:T2 = L:T1` $\circ$ `T2` for any Rosetta logical operator.

Definition 21 (Label Distribution) *For any logical operator* $\circ$*, label* `L` *and item definitions* `T1` *and* `T2`*, the following distribution law holds:*

$$\texttt{L} : \texttt{T1} \circ \texttt{L} : \texttt{T2} = \texttt{L} : (\texttt{T1} \circ \texttt{T2})$$

Example 39 (Label Distrubution Over Terms) *Assume the following term definitions in a facet:*

`L:P(x); L:Q(x);`

By label distribution, this is equivalent to:

`L:P(x) and Q(x)`

8.10 Type Composition

A direct application of label distribution is type composition. In Rosetta, an item is frequently viewed from multiple, interacting specifications. One such example occurs when facets are composed and parameter lists are unioned. In such cases, a label referring to an item may be interpreted differently in the domains of both facets. In such situations, the type of the parameter in the newly formed facet is the conjunction of the original two types. This follows directly from the definition of type composition:

Definition 22 (Type Composition) *Assume the following two variable definitions:*

`v::T1; v::T2;`

By label distribution, this is equivalent to:

`v::(T1 and T2)`

Note that in this context `and` is not a logical connective, but a composition operator. `v::(T1 and T2)` means that the item `v` is both of type `T1` and `T2` simultaneously. This does not imply that the resulting type is the intersection of `T1` and `T2`, but is similar to a set of ordered pairs of elements from `T1` and `T2`. Elements of the composed type can be viewed as either type. The semantics of this will be better understood when considering facet composition operators later in this section.

8.11 Facet Composition

```
%% White paper composition definitions here...
```

8.12 Domain Interaction

8.12.1 Categories

A category is said to be defined by facet-type. Formally, this means that a category is defined by a domain algebra. That is, we speak of the category $D_1 - CAT$, in which the objects are domain algebras and facet algebras, with the morphisms being extensions. That is, there is a morphism from object A to object B in $D_1 - CAT$ precisely when the algebra A extends the algebra B. By this definition both facet and domain algebras can be the source of morphisms. This morphism is in fact unique. That is, given a source algebra F corresponding to a facet f, and a target algebra G corresponding to a facet g, there is only one extension we can use to form G from F. This extension corresponds to

1. Adding to the signature of F those elements of $\boldsymbol{G_S}, \boldsymbol{G_N}$ and $\boldsymbol{G_O}$ which are not already present.
2. Adding to the presentation of F those axioms which exist in the presentation of G but not the presentation of F.

Up to the order in which elements are added (something about which we do not care), this extension is unique. This is in part due to the fact that an algebra corresponds to a particular Rosetta facet specification so the *only* way to form the facet algebra G of facet g is as above - adding different sorts, terms, axioms or functions will result in a facet algebra corresponding to a facet which, although it may be functionally equivalent to g, is not in fact g. In this sense we see that, for example, re-labelling a function or using a derived set of axioms within the presentation results in a facet algebra that by our definition is a distinct object from the one prior to re-labelling/deriving axioms.

The category $D_1 - CAT$ is defined as being the category with set of objects Obj and morphisms $Morph$ where:

1. $\forall A \in Obj, A \neq D_1 \exists f \in Morph$ such that $targ(f) = A$
2. there is no $f \in Morph$ such that $targ(f) = D_1$.

It is important to realise that, although each category is a category of algebras, the algebras may have different signatures. We see that a category $D_1 - CAT$ consists of all facets that extend domain D_1 (remembering that in Rosetta domains are also facets). Thus, if domain D_2 extends domain D_1, then $D_2 - CAT$ is a subcategory of $D_1 - CAT$. One thing to note is that here sorts of the first and second type in the D_1 algebra are the instantiations of the algebras which are objects in $D_1 - CAT$.

A proof that this does indeed define a category follows:
$\forall f, g, h \in Morph$ and $A, B, C, D \in Obj$:

1. **Unique-type**
RTP: $f : A \to B$ and $f : A' \to B' \Rightarrow A = A'$ and $B = B'$.
Proof
All morphisms in the category as defined above are extensions. An extension is defined uniquely by its target and source, so it is impossible for the same extension (morphism) to have more than one target or source.

2. **Composition**
RTP: $f : A \to B$ and $g : B \to C \Rightarrow f \circ g : A \to C$

Proof
By definition of our morphisms, we know the signature and presentation of C extend the signature and presentation of B, which itself extends the signature and presentation of A. From the definition of extension, it is then clear that the signature and presentation of C extend the signature and presentation of A. Thus, our construction of the category implies that there is indeed a morphism between A and C. We denote this morphism $f \circ g$ and construct it by effecting both the extensions of B, and the subsequent extensions of C upon our object A.

3. **Identity-existence**
RTP: $\forall A \in Obj \exists id_A \in Morph$ such that $id_A : A \rightarrow A$
Proof
By definition of extension, every algebra is said to extend itself. Thus, by how we have constructed the category, we know there exists such an identity morphism.

4. **Associativity**
RTP: $f : A \rightarrow B, g : B \rightarrow C, h : C \rightarrow D$, then $(f \circ g) \circ h = f \circ (g \circ h)$
Proof
This holds from the definition of $f \circ g$, above as the simple addition of sorts, operations and constants (addition is an associative operation).

5. **Identity-operation**
RTP:$\forall f : A \rightarrow B, id_A \circ f = f = f \circ id_B$
Proof
We define the identity morphism as the morphism which returns the identical morphism as that which it operated on (up to isormorphism). This clearly provides the above property.

Thus, the definitions above fulfill requirements for a category.

8.12.2 Transferring Information

We often need to transfer information from one domain to another, in the process of what is termed in Rosetta a *projection* or *interaction*. In the most general terms, we can represent this by means of functors operating on the categories specified by the domains in question. That is, we can define a functor

$$\theta : D_1 - CAT \rightarrow D_2 - CAT$$

to be used when we want to transform a facet f_1 that extends domain D_1 to a facet that extends domain D_2. This is most often used when we want to view a facet from one domain in a second domain. We achieve this - in theory - by applying the functor θ and viewing the facet $\theta(f_1)$.

8.12.3 Facet Composition

To provide the theoretical basis of facet composition of two facets with algebra F_1 and F_2 we need to express this operation and resultant facet algebra, F_3, in terms of category theory. Let $D_3 - CAT$ represent the smallest category such that F_3 is an object within $D_3 - CAT$. We define D_3 as the algebra with signature composed of the following:

1. The set of sorts of D_3:
   $\boldsymbol{D_{3S} = D_{1S} \bigcup D_{2S}}$.
2. The set of constant symbols:
   $\boldsymbol{D_{3N} = D_{1N} \bigcup D_{2N}}$.
3. The set of functions of D_3 with arguments from S_i for some $S_i \in \boldsymbol{D_{3S}}$:
   $\boldsymbol{D_{3N} = D_{1N} \bigcup D_{2N}}$.

The presentation of algebra D_3 is also the union of the presentations of algebras D_1 and D_2.

This domain is generally not explicitly stated in the domain hierarchy and no details are provided on how to construct it. However, it is simple to show that $D_3 - CAT$ is a sub-category of both $D_1 - CAT$ and $D_2 - CAT$. To facilitate this proof we remember that D_1 and D_2 are themselves objects in a category, denoted by L. WLOG, we will prove this for $D_1 - CAT$ only. By definition of extension, the D_3 algebra is an object in $D_1 - CAT$. By composition of extensions, any objects which are extensions of D_3 and therefore in $D_3 - CAT$ are also extensions of D_1 and therefore objects in $D_1 - CAT$. Since morphisms are defined purely by source and target object, we define the morphisms in $D_3 - CAT$ to be precisely those morphisms which exist between the objects of $D_3 - CAT$ when considered as objects of $D_1 - CAT$.

Facet and domain composition can be theoretically represented as a co-product. That is, we claim that D_3 and the morphisms $f_1 : D_1 \rightarrow D_3$ and $f_1 : D_2 \rightarrow D_3$, above, form the co-product of D_1 and D_2.
Firstly, the morphisms from D_1 and D_2 clearly exist in L, as D_3 is an extension of both D_1 and D_2. So D_3 is clearly *a* co-product of D_1 and D_2. To prove that it is indeed *the* co-product we need to prove initiality.
RTP: $\forall D_4$ in the class of co-products in L of D_1 and D_2, $\exists! g : D_3 \rightarrow D_4$
Proof
Since D_4 is in the class of co-products in L of D_1 and D_2 we know there exist morphisms from these algebras to D_4 in L. Thus, D_4 is an extension of D_1 (D_2). Now, we defined D_3 to be the algebra with the smallest signature such that it is an extension of D_1 and D_2 both. Thus, a morphism g in the category L does exist such that $g : D_3 \rightarrow D_4$. Now, our definition of the category said that precisely one morphism existed from each algebra to each of its extensions. Thus, g is the unique such morphism and we have proved initiality.

Having established a domain for the facet represented by the composition of facets with facet algebra F_1 and F_2, we are now in a position to define the actual facet f_3 (which is $f_1 + f_2$), with facet algebra F_3. As expected, we define F_3 to be the coproduct of F_1 and F_2, with the proof as above. We can therefore see that F_3 is an element in $D_3 - CAT$ by associativity of extension.

```
%% White paper interaction semantics here...
```

8.13 Types and Values

8.14 Open Issues

- Assuming that `A::B` is true when `A` is an atomic element and in `B`. Specifically, that `A` is not a bunch. If this axiom is removed, then bunch values for atomic items becomes possible. For example, if `i::integer` is defined, then `i=1,2` is fine if `1,2::integer` is true.
- String functions are defined over items, not just over values. The string/parse thing needs some thought. It's not clear to me that it's as simple as we're assuming it to be at this point.