An Introduction to Rosetta

Syntax, Semantics, and Specification

Dr. Perry Alexander
The University of Kansas
alex@ittc.ukans.edu

Mr. David Barton
AverStar, Inc
dlb@averstar.Com

Introduction

- This presentation overviews the basics of Rosetta specification
- You will learn
  - Basic Rosetta type definition
  - Rosetta functions and function definition
  - Facets, Packages and Domains
  - Available domain types
  - Specification composition
- You should be familiar with at least one HDL or high level programming language before attempting this tutorial

Agenda

- Rosetta Introduction
- Declarations, Types, and Functions
- Facets and Packages
- Domains and Domain Interactions
- Examples
- Advanced Topics

What is Systems Engineering?

- **Managing and integrating information from multiple domains when making design decisions**
- **Managing constraints and performance requirements**
- **Managing numerous large, complex systems models**
- **Working at high levels of abstraction with incomplete information**
- **…Over thousands of miles and many years**

"…the complexity of systems… have increased so much that production of modern systems demands the application of a wide range of engineering and manufacturing disciplines. The many engineering and manufacturing specialties that must cooperate on a project no longer understand the other specialties. They often use different names, notations and views of information even when describing the same concept. Yet, the products of the many disciplines must work together to meet the needs of users and buyers of systems. They must perform as desired when all components are integrated and operated."

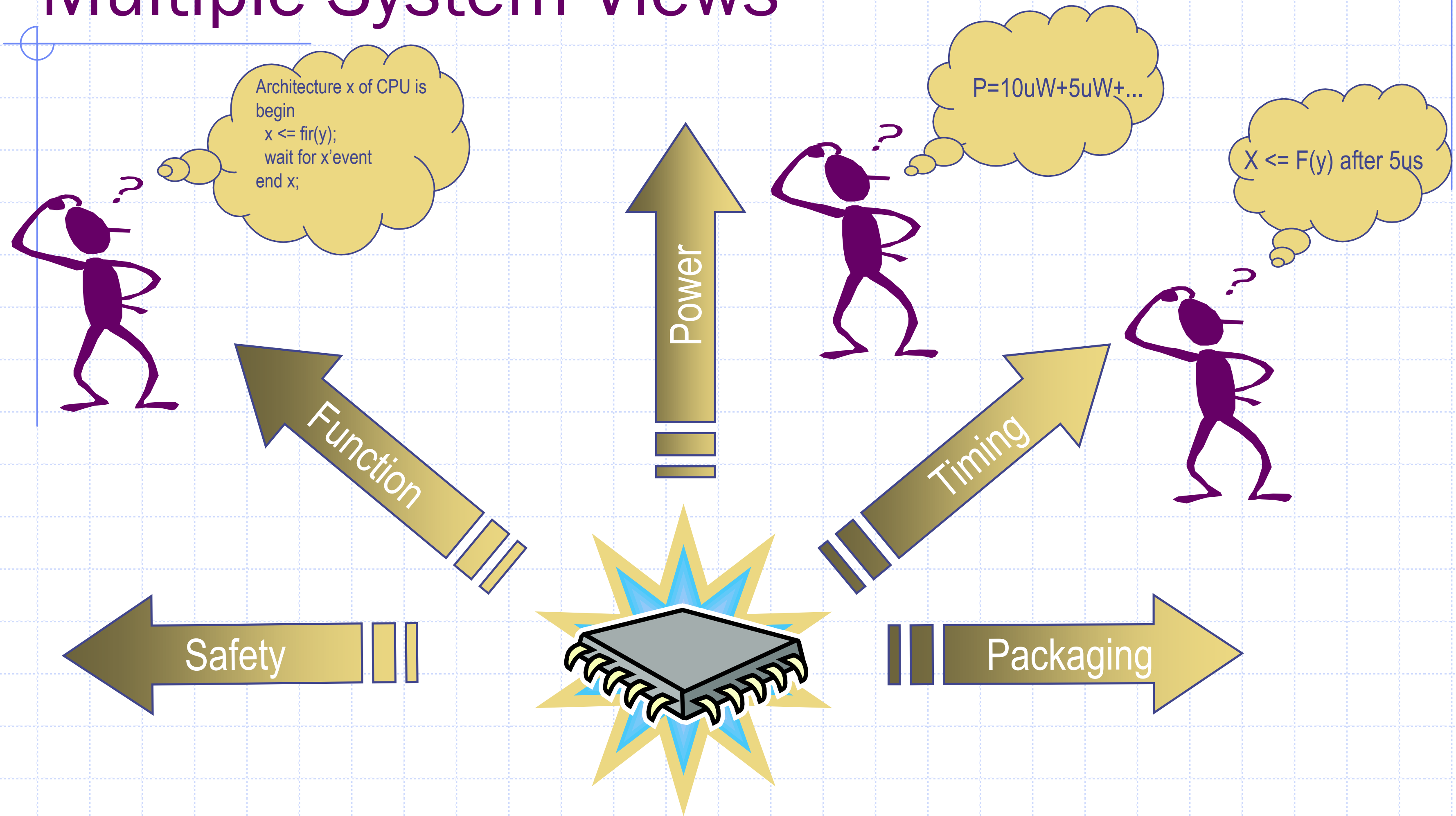
D. Oliver, T. Kelliher, J. Keegan, Engineering Complex Systems, McGraw-Hill, 1997.

The Systems Level Design Problem

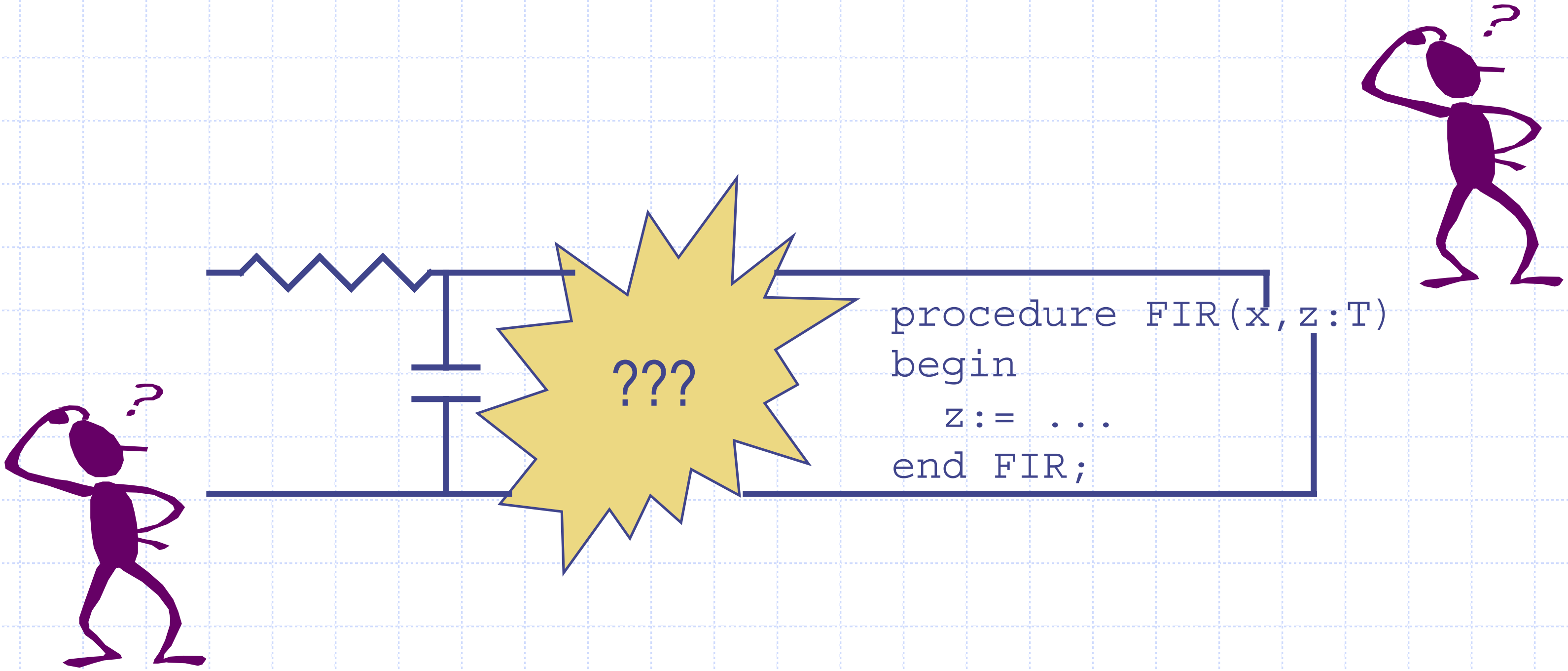
The cost of systems level information is too high…

- Design goals and system components interact in complex and currently unpredictable ways
- Interrelated system information may exist in different engineering domains (intellectually distant)
- Information may be spread across the system specification, in separate parts of the description
- Representation and analysis of high level systems models is difficult and not well supported
- Representation and analysis of interactions between system elements is not supported at all

Multiple System Views



Multiple Semantic Models



Rosetta Objective

Provide a means for defining and integrating systems models throughout the design lifecycle…

- Define *facets* of components and systems
- Provide *domains* for facet description
- Provide mechanisms for composing *components* and *facets*
- Specify *interactions* between domains reflected in facet composition

Rosetta Design Goals

Provide a means for defining and integrating systems models throughout the design lifecycle…

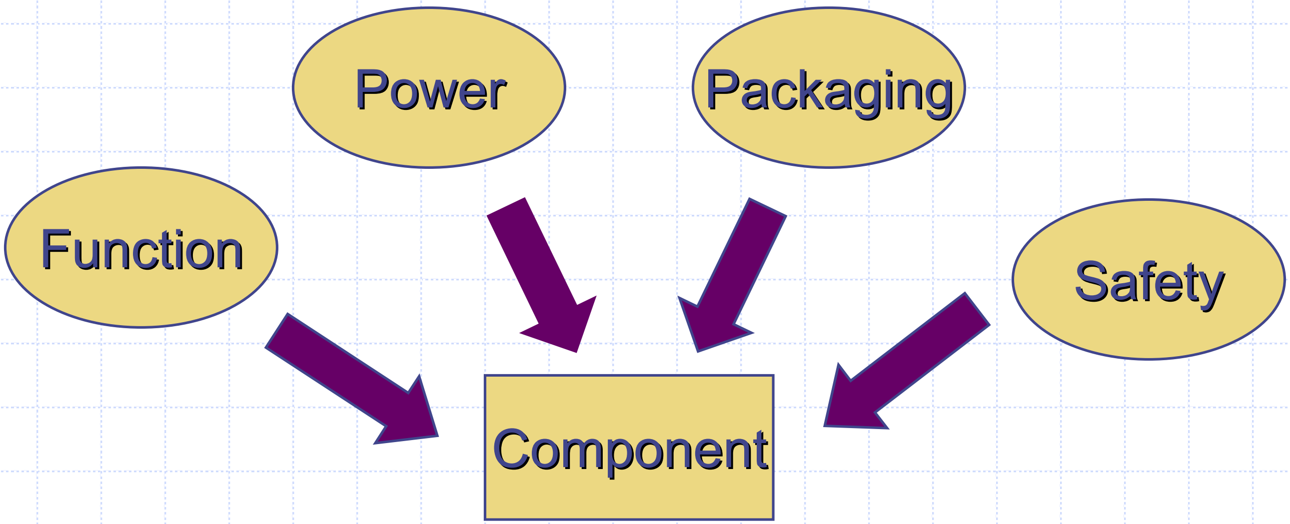
- Support for multi-facet modeling
  - Multiple views of the same component
  - Representation of functional and constraint information
- Support for multiple semantic domains
  - Integrate components from multiple domains
  - Integrate component views from multiple domains
- Support for complexity management
  - Verification condition representation
  - Support for verification

Multi-Faceted Modeling

- Support for systems level analysis and decision making
- Rosetta *domains* provide modeling abstractions for developing facets and components
- Examples include:
  - Performance constraint modeling
  - Discrete time modeling
  - Continuous time modeling
  - Finite state modeling
  - Infinite state modeling

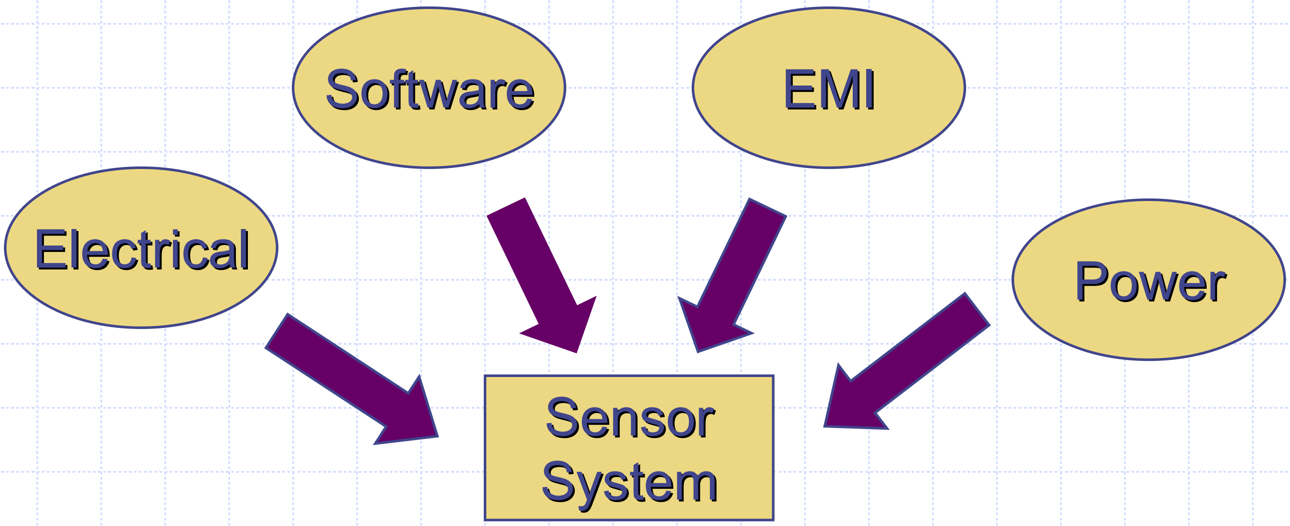
Multi-Faceted Modeling

- Support for modeling in heterogeneous domains
- Rosetta *facets* model different views of a system or component



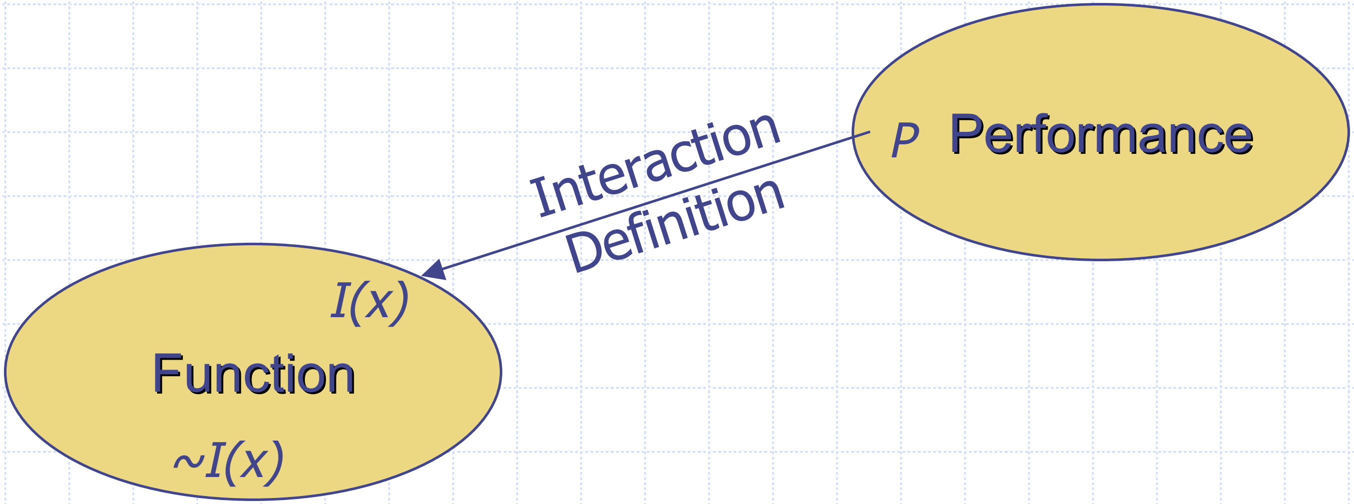
A Simple Example…

- Construction of system involves multiple specialists
  - Each specialist works from their set of plans
  - Each specialist uses their own domain-specific information and language
- The systems engineer must manage overall system construction using information from all specialist domains



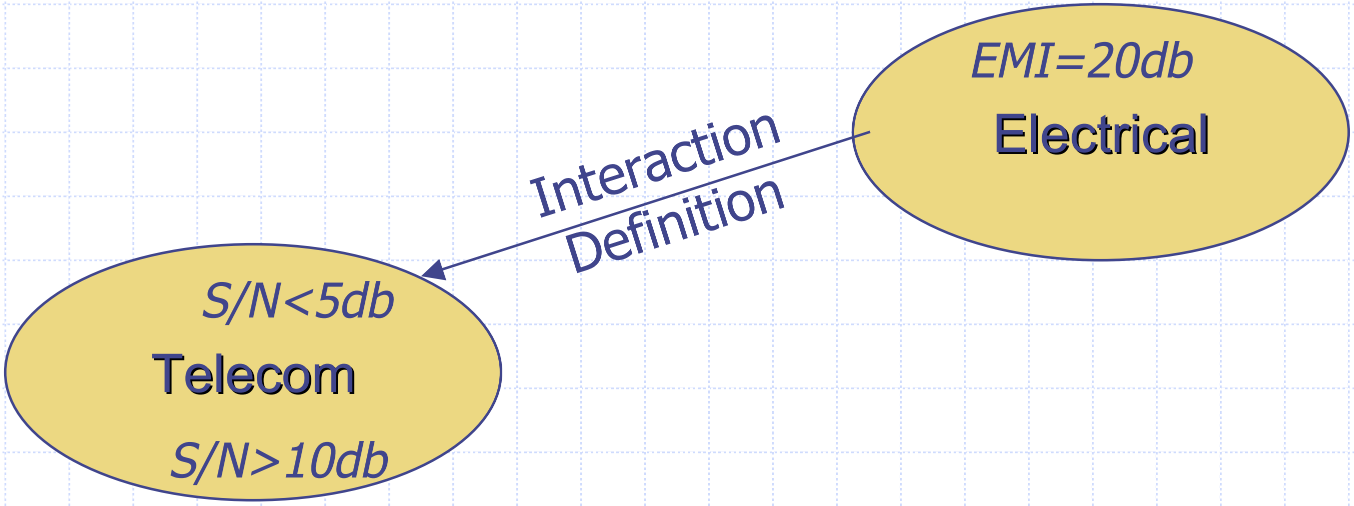
Multi-Faceted Modeling

- Support for modeling facet interaction
- Rosetta *interactions* model when information from one domain impacts another



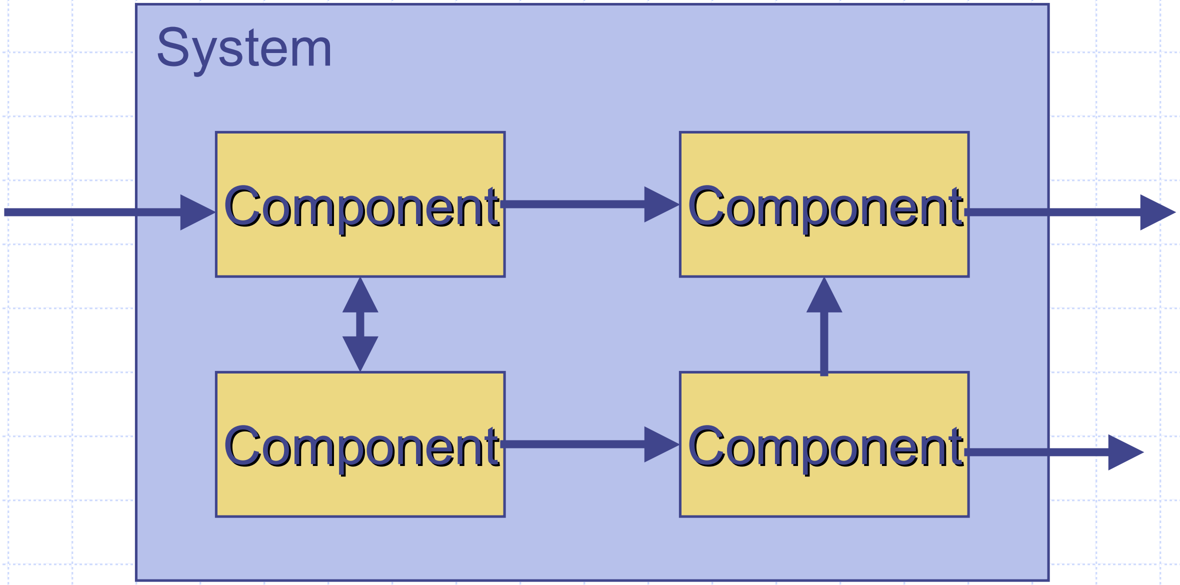
A Simple Example...

- EMI specified at 20db
- Generates S/N ratio less than 5db
- Requirements for S/N greater than 10db



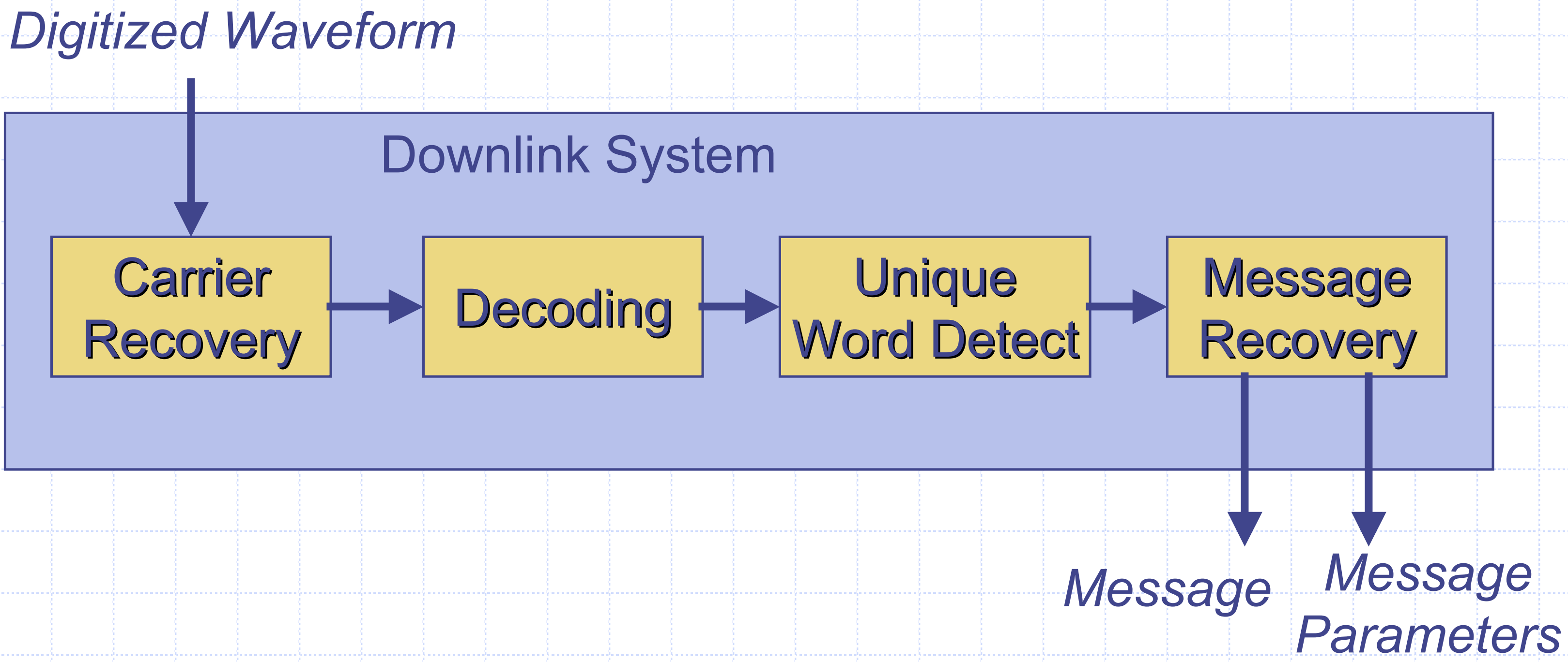
Multi-Faceted Modeling

- Support for heterogeneous component assembly
- Rosetta *components* model system structure



A Simple Example

- Simple Satellite Download
- Each component is a Rosetta facet or component
- Each component may use its own domain for requirements specification

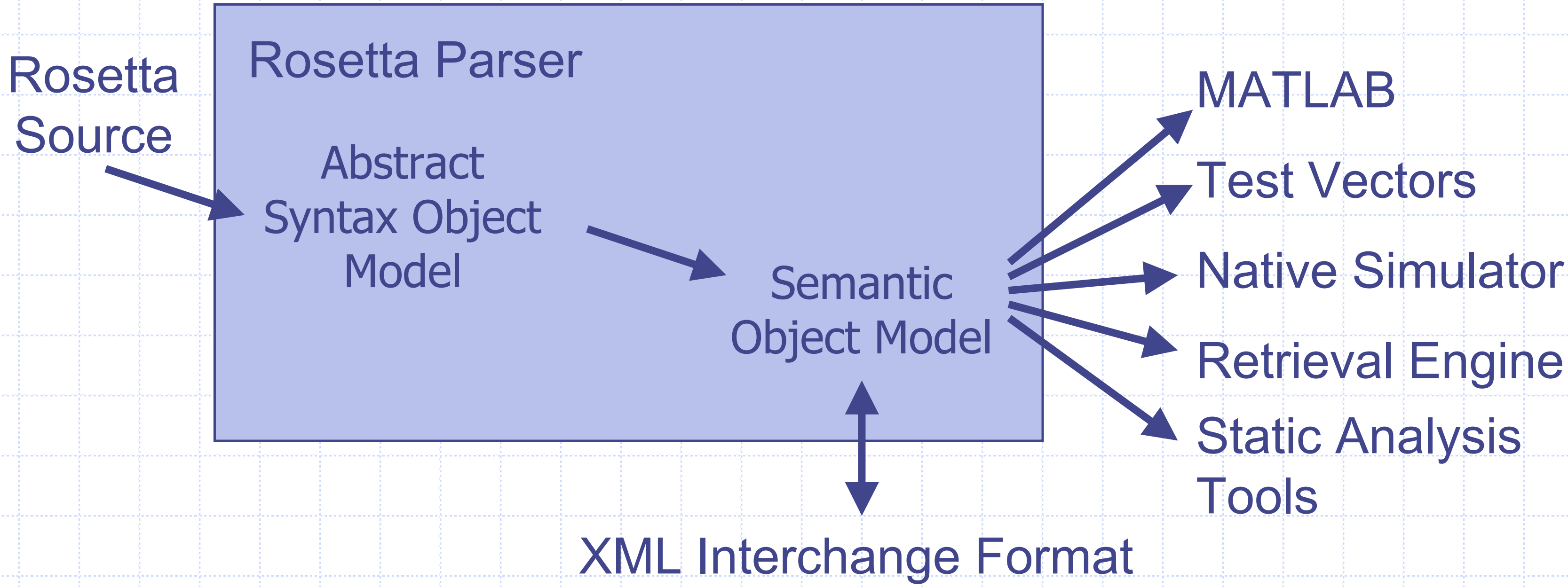


What Rosetta Provides

- A Language for model representation
  - Simple syntax for parameterized model representation
  - Language support for information hiding and component definition
  - Representation of verification conditions and justifications
- A Semantics for system modeling
  - Representation of system models
  - Representation of application domains
  - Representation of interactions between domains
  - Highly extensible and customizable

Rosetta Tool Architecture

- Front-end parser generating a semantic object model
- Back-end tools supporting various design capabilities
- MoML compatible XML interchange format



Rosetta Modeling Flow

- Choose *domains* for component modeling
- Define component *facets* using domains
- Assemble facets into individual *components*
- Assemble components into *systems* using structural assembly techniques
- Analyze components and systems using
  - Domain specific tools
  - Domain interaction tools

Vocabulary

- Item – The basic unit of Rosetta semantics
- Type or Bunch – A collection of items
- Operation or Function – A mapping from an element of a domain bunch to a range bunch
- Variable – An item whose value is not explicitly specified
- Constant – An item whose value is explicitly specified
- Label – A name for an item
- Facet – An item specifying a system model

Items

- Every Rosetta definition unit is defined as an *item*
- Each item consists of three critical elements:
  - A *label* naming the item
  - A *value* associated with the item
  - A *type* enumerating possible values
- For any item, I, M__value(I) :: M__type(I)
- If an item's value is fixed at parse time, it is a *constant* item
- If an item's value is unknown at parse time, it is a *variable item*

Bunches and Types

- The Rosetta type system is defined semantically using bunches
  - A bunch is simply a collection of objects
  - Any item A is a bunch as is any collection A,B,C
- The notation A::B is interpreted as "bunch A is contained in bunch B"
  - Contained in is both "element of" and "subset"
  - Type correctness is defined using the "contained in" concept
- The notation A++B is the bunch union of A and B
- Examples:
  - 1::1++2
  - 1++2::integers
  - integers::numbers

Declarations

- Declarations create and associate types with items
- All Rosetta items must be declared before usage
- Declarations occur:
  - In parameter lists
  - In the facet declaration section
  - In packages
  - In **let** constructs

Declarations

- Items are created using *declarations* having the form:

label :: *type* [is *value*];

- *Label* is the item's name
- *Type* is a bunch defining the item's type
- *Value* is an expression whose value is constraint to be an element of *type*

Constant vs Variable Declaration

- Using the is construct, an item's value is fixed
  - The following example defines an item and sets its value

```
Pi::real is 3.14159;
```

- Omitting the is construct, an item's value is variable
  - The following example defines an item and leaves its value unspecified

```
counter::natural;
```

Example Declarations

```
i::integer;                         // variable i of type integer
bit_vector::sequence(bit);          // variable bit_vector
T::type(univ)                       // uninterpreted scalar type

natural::type(integer) is           // natural number definition
   sel(x::integer | x =< 0);

inc(x::integer)::integer is         // Constant function inc
   x+1;

pos?(x::integer)::boolean;          // Function signature
```

Types and Bunches

- Rosetta types are defined semantically as bunches
- The notation x::T used to declare items is the same as bunch inclusion
- Any bunch may serve as a type
  - Bunch operations are used to form new types from old
  - Functions returning bunches define parameterized types

Predefined Scalar Types

- Rosetta provides a rich set of scalar types to choose from:
  - number, real, rational, integer, natural,
  - boolean, bit
  - character
  - null
- The type **element** is a supertype of all scalars
- The types **boolean** and **bit** are subtypes of integer
  - TRUE is the greatest and FALSE is the least **integer**
  - 0 and 1 are shared among **bit** and **integer**

Number Types

- Numerical types are all subtypes of number
- Standard operators on numbers are available:
  - +,−,*,/ - Mathematical operations
  - min, max – Minimum and maximum
  - <,=<,>=,> - Relational operators
  - abs, sqrt – Absolute value and square root
  - sin,cos,tan – Trig functions
  - exp,log – Exponentiation and log functions
- Subtype relationships between numbers are defined as anticipated

The Boolean Type

- Booleans are the subtype of integers that includes TRUE and FALSE
  - TRUE is a synonym for the maximum integer
  - FALSE is a synonym for the minimum integer
- Booleans *are not* bits
- Operations include:
  - max, min
  - and, or, not, xor
  - implies
- Note that min and max are and and or respectively
  - X min Y = X and Y
  - X max Y = X or Y

The Boolean Type

- The semantics of boolean operations follow easily from min and max
  - TRUE and FALSE = TRUE min FALSE = FALSE
  - TRUE or FALSE = TRUE max FALSE = TRUE
- TRUE and FALSE are not infinite, but use infinite mathematics:
  - TRUE+1 = TRUE
  - TRUE = -FALSE
  - FALSE = -TRUE

The Bit Type

- Bits are the subtype of natural numbers that include 0 and 1
- Operations include similar operations as boolean:
  - max, min
  - and, or, not, xor
  - Implies
- The operation % transforms between bits and booleans
  - %TRUE = 1
  - %1 = TRUE
  - For any bit or boolean, b, %(%b))=b
- The semantics of bit operations is defined by transforming arguments to booleans
  - 1 and 0 = %1 and %0 = TRUE and FALSE = FALSE

Compound Types

- Compound types are formed from other types and include bunches, sets, sequences, and arrays
- Ordered compound types define ordering among elements
  - Sequences and arrays are ordered
  - Bunches and sets are not ordered
- Packaged types have distinct inclusion and union operators
  - Sets and arrays can contain other sets and arrays
  - Bunches and sequences cannot contain sequences

Predefined Compound Types

- bunch(T) – The bunch of bunches formed from T
- set(T) – The bunch of sets formed from T
- sequence(T) – The bunch of sequences formed from T
  - bitvector – Special sequence of bits
  - string – Special sequence of characters
- array(T) The bunch of arrays formed from T

The bunch Type

- Defined using **bunch(T)** where T is a type
- Operations on bunch types include:
  - A++B – Bunch union
  - A**B – Bunch intersection
  - A--B – Bunch difference
  - A::B – Bunch containment or inclusion
  - $S – Size
  - **null** – The empty bunch
- Examples:
  - 1++(2++3) = 1++2++3
  - 1**(1++2++3) = 1
  - 1++2::1++2++3 = TRUE
  - 1++2::1++3 = FALSE

The set Type

- Defined using set(T) where T is a type
- Operations on set types include:
  - {A} – The set containing elements of bunch A
  - ~A – The bunch containing elements of set A
  - A+B, A*B, A–B – Set union, intersection, difference
  - A in B – Set membership
  - A<B,A=<, A>=B, A>B – Proper Subset and Subset relations
  - #A – Size
  - empty – The empty set
- Sets are formed from bunches
  - The semantics of set operations is defined based on their associated bunches
  - S++T = {~S ++ ~T}

The set Type

- Example set operations
  - {1++2} + {3} = {1++2++3}
  - ~{1++2++3} = 1++2++3
  - {1++2} =< {1++2++3}
  - (A < A) = FALSE
  - (A =< A) = TRUE
  - {null} = empty
  - {1++2} = {2++1}

The sequence Type

- Defined using sequence(T) where T is a type
- Operations on sequence types include:
  - 1;;2 – Catenation
  - **head, tail** – Accessors
  - S(5) – Random access
  - A<B, A=<B, A>=B, A>B – Containment
  - $S – Size
- Sequences cannot contain sequences as elements

The sequence Type

- Examples:
  - head(1;;2;;3) = 1, tail(1;;2;;3) = 2;;3
  - 1;;2;;3 < 1;;2;;3;;4 = TRUE
  - 1;;3 < 1;;2;;3 = FALSE
  - If s=4;;5;;3;;2;;1 then s(2)=5
- Strings and bit vectors are special sequences
  - bitvector :: type(universal) is sequence(bit);
  - string :: type(universal) is sequence(character);

The array Type

- Declared using array(T) where T is a type
- Operations on array types include:
  - [1;;2;;3] – Forming an array from a sequence
  - ~A – Extracting a sequence from an array
  - A(1) – Random access
  - #A – Size of array A
- Arrays are to sequences as sets are to bunches
  - Arrays are formed from sequences
  - The semantics of array operations are defined based on sequences

The array Type

- Examples (assume A=[1;;2;;3]):
  - A(1) = 2
  - #A = 3
  - ~A = 1;;2;;3
  - A;;A = [~A;;~A] = [1;;2;;3;;1;;2;;3]

Aggregate Types

- Aggregate types are formed by grouping elements of potentially different types in the same structure
- Aggregate types include
  - Tuples – Structures indexed using an arbitrary type
  - Records – Structures indexed using naturals

Predefined Aggregate Types

- Tuple [T1 | T2 | T3 | …] – The bunch of tuples formed from unlabled instances of specified types
  - Tuple elements are accessed using position as in t(0)
  - Specific tuples are formed using the notation [v1 | v2 | v3 | …]
- Example: Complex Numbers
  - c1::tuple[real | real]
  - c1 = tuple[1.0 | 2.0]
  - c1(0) = 1.0
- Tuple declarations and formers have the same form

Predefined Aggregate Types

- Record [F1::T1 | F2::T2 | F3::T3 | …] – The bunch of records formed from labeled instances of types
  - Record elements are accessed using field name as in R(F2)
  - Specific tuples are formed using the notation [v1 | v2 | v3 | …]
- Example: Complex Numbers
  - c1::record[r::real | c::real]
  - c1 = record[r is 1.0 | c is 2.0]
  - c1(r) = 1.0
- Record declarations and formers have the same form

Functions and Function Types

- Functions provide a means of defining and encapsulating expressions
- Functions are pure in that no side effects are defined
  - No global variables
  - No “call by reference” parameters
- A Rosetta function is an item whose
  - Type is a function type
  - Value is an expression

Unnamed Functions and Types

- A unnamed functions support defining local functions and function types
- The notation:

  <* (d::D) :: R *>

  defines a function type that includes all mappings from D to R.
- The notation:

  <* (d::D) ::R is *exp(d)* *>

  defines a single function mapping d to *exp(d)*

Formally Defining Functions

- A function of a particular type is defined like any other structure:

  f::<*(d::D)::R *> is <* (d::D)::R is *exp(d)* *>

- For example:

  inc::<*(j::integer)::integer*> is <*(j::integer)::integer is j+1*>

- This is somewhat strange and painful, so...

Function Definition Syntax

- A convenient concrete syntax is defined as:

  f::<*(d::D)::R *> is <* (d::D)::R is *exp(d)* *>

  ==

  f(d::D)::R is expression;

- Increment can now be defined much more compactly as:

  inc(j::integer)::integer is j+1;

Basic Function Types

- Functions are declared using the notation:
  F(d::D) :: R;
- D is a type defining the function *domain* and R is an expression defining the function's *return type* and ran(F) is the range of F
  - dom(F) = D
  - ret(F) = R
  - ran(F) = {All possible return values}
- Example: Increment
  - inc(i::integer)::integer;
  - ret(inc) = dom(inc) = integer
  - ran(inc) = sel(i::integer | 0 < i)

Functions of Multiple Arguments

- Functions with multiple arguments are define by recursive application of the function definition:

  F(d1::D1; d2::D2; d3::D3 …)::R

- For engineering purposes, this defines a function that maps multiple values onto a single value
- Example: Add
  - add(n1 :: natural; n2 :: natural) :: natural ;
  - dom(add) = natural;
  - ret(add) = natural;
  - ran(add) = <* (n2::natural) :: natural *>;

Function Type Semantics

- Function types provide a means of defining signatures
- The semantics of a function type definition state:
  - The function is defined only over elements of its domain
  - The function must return an element of its range
- The increment example is a function that takes an integer as input and returns the integer bunch
- The add example is a function that
  - Takes an integer as input and returns a new function
  - Applies the new function to the second integer argument
  - 99.9% of the time, you can simply think of this as a two argument function

Examples

- sqrt(i::integer)::integer;
- ord(c::character)::natural;
- element(e1::E; s::set(E)) :: boolean;
- top(e1::E; s1::Stack)::E;
- cat(s1::sequence(E); s2::sequence(E))::sequence(E);

Defining Functions

- Specific functions are defined using roughly the same mechanism as other items:

  F(d::D) :: R is value;

  where the type is a function type and value is a function type that interprets as an expression

- Example: increment
  - inc(n::natural)::natural is n+1;
  - n names the input parameter
  - n+1 defines the return value

Interpreting function definitions

- The function definition names parameters and defines a return value

Function Name

Function Type

```
add(j::natural; k::natural)::natural is j+k;
```

Parameter Names and Types

Return Expression

Example Functions

```
// Simple 2-1 multiplexor
mux(s::bit; i0::bitvector; i1::bitvector)::bitvector is
        if s=0 then i0 else i1 endif;

// Hours increment function
incrementHours(h::hours; m::minutes)::hours is
    if m = 59 then
       if h = 23 then 0 else h+1 endif
       else h endif;
```

Example Functions

```
//Parameterized linear search function
search(E::type(univ); s::sequence(E); p::<*(e::E)::boolean*> is
    if s/=null then
      if p(s(0)) then s(0) else search(E,tail(s),p) endif
    endif;

search(integer,_,_) ==
   <*(s::sequence(integer),p::<*(e::integer)::boolean*> is
         if s/=null then
           if p(s(0)) then s(0) else search(integer,tail(s),p) endif;
         endif; *>
```

- Note the use of function and type parameters in the search definition allowing multiple criteria and search results

Applying Functions

- Applying a function is a two step process
  - Replace formal parameters with actual parameters in the value expression
  - Evaluate the value expression
- Example: inc(5)
  - inc(5) = <*5+1*> = 6
  - add(5,2) = <*(m::natural) ::natural is 5+m*>(2) = <*5+2*> = 7
- Simply replace and simplify
- All parameters need not be instantiated!!

Partial Evaluation

- Partial evaluation is achieved by instantiating only some of the variables
  - Use “_” as a placeholder for uninstantiated parameters
- Consider the function definition:

```
searchInt(s::sequence(integer);
      p::<*(e::integer)::boolean*>)::boolean;

searchInt = search(integer,_,_);
```

defines a new function that is a specialization of the general purpose `search` function

Functions on Functions

- Many classical specification functions are defined as "functions on functions"
  - min, max - Minimum or maximum in a bunch
  - forall and exits – Universal and existential quantification
  - dom, ran - Domain and range over a function
  - sel - Select or comprehension over a bunch

The Domain and Range Functions

- Domain is a simple extraction of the function domain
  - dom(<* (d::D)::R *>) = D
- Range is the bunch formed by application of the function to each defined domain value
  - ran(<* (d::D)::R *>)= The bunch of the function applied to all domain values
  - Frequently used to implement the image of a function over a bunch or set
- Examples:
  - dom(inc) = natural
  - ran(inc) = natural -- 0;

The Minimum and Maximum Functions

- The min and max functions take the minimum and maximum values of a function's range, respectively
- Examples:
  - min(inc)=1
  - max(inc)=TRUE

The Quantifier Functions

- The **forall** and **exists** functions are shorthands for **min** and **max** respectively
  - Both take arguments of boolean valued functions
  - Both apply the function to each domain element
  - The **forall** function takes the minimum value while **exists** takes the maximum value
- Examples
  - forall(<*(x::integer)::boolean is x>0 *>) = FALSE
  - exists(<*(x::integer)::boolean is x>0 *>) = TRUE

The Quantifier Functions

- Because **forall** and **exists** are so common, we define a special syntax for their application:

```
forall(x::integer | x>0) ==
    forall(<*(x::integer)::boolean is x>0 *>) = FALSE

exists(x::integer | x>0 ==
    exists(<*(x::integer)::boolean is x>0 *>) = TRUE
```

where the the “|” separates a variable declaration from a boolean expression defined over that variable.

The Selection Function

- The `sel` function performs comprehension over the domain of a function
  - Use the select function whenever comprehension or filtering is desired
- Examples:
  - `sel(<* (x::integer)::boolean is x>=0 *>)=natural`
  - `sel(<* (x::1++2++3++4) :: boolean is 2*x=4 *>) = 2`
  - `natural::bunch(integer) is sel(<* (x::integer)::boolean is x >= 0*>)`

The Selection Function

- The `sel` function also uses a special syntax to aid comprehension

```
sel(x::integer | x>=0) ==
sel(<* (x::integer)::boolean is x>=0 *>)

natural::bunch(integer) is sel(<* (x::integer)::boolean is x
   >= 0*>) ==
natural::bunch(integer) is sel(x::integer | x >= 0);
```

Functions as Type Definitions

- Functions can be used to define parameterized types

```
boundedBitvector(n::natural)::type(bitvector) is
    sel(b::bitvector | $b = n);
```

- The function can now be used to define new types because its return value is a bunch:

```
bitvector8::type(bitvector) is boundedBitvector(8);
```

- bitvector8 is the type containing all bitvectors of length 8

Facets, Packages and Components

- Facets define basic system and component models
  - Parameters provide communications and design specialization
  - Declaration areas provide for local item definitions
  - A domain identifies the vocabulary
  - Terms provide for the semantics of the facet model
- Packages group definitions
  - Packages are special facets without terms
  - Packages group definitions into parameterized modules
- Components provide a standard definition style for system components
  - Components record design assumptions
  - Components record correctness assertions

Understanding Facet Definitions

- Facets and packages provide mechanisms for defining models and grouping definitions

Facet Name *Parameter List*

Variables

```
facet trigger(x::in real; y::out bit) is
  s::bit;
begin continuous
  t1: if s=1 then if x>=0.4 then s'=1 else s'=0 endif;
               else if x=<0.7 then s'=0 else s'=1 endif;
  t2: y@t+10ns=s;
end trigger;
```

Domain

Terms

Facet Name and Parameters

- The facet name is a label used to reference the facet definition
- Facet parameters are formal parameters that represent an interface to the component
  - Parameters provide a means for model specialization
  - Parameters provide a means for model communication
- Parameters are instantiated by providing actual values when a facet is used

```
trigger(input,output);
```

Trigger Label and Parameters

- The label **trigger** names the facet
- The parameters **x** and **y** define inputs and outputs for the facet

```
facet trigger(x::in real; y::out bit) is
```

- The direction indicators **in** and **out** define the behavior of parameters by asserting **in(x)** and **out(x)** as terms

Facet Declarations

- Facet declarations are items defined within the scope of the facet
  - When exported, declarations are referenced using the canonical "." notation as in `ad2.s`
  - When not exported, declarations cannot be referenced outside the facet
  - Declarations are visible in all facet terms
- Items are declared in the manner defined previously
  - Item values may be declared constant
  - Item types include all available Rosetta types including facets, functions and types

Trigger Facet Declarations

- The local variable s declares a bit visible throughout the facet
- No export clause is present, so all labels are visible

```
facet trigger(x::in real; y::out bit) is
  s::bit;
begin continuous
```

- In this specification, s defines the instantaneous state of the component

Facet Domain

- The facet domain provides a base vocabulary and semantics for the specification
- Current domains include
  - Logic and mathematics
  - Axiomatic state based
  - Finite state
  - Infinite state
  - Discrete and continuous time
  - Constraints
  - Mechanical

Trigger Domain

- The trigger facet uses the continuous domain for a specification basis
- The continuous domain provides a definition of time as a continuous, real value

```
facet trigger(x::in real; y::out bit) is
  s::bit;
begin continuous
```

Facet Terms

- A term is a labeled expression that defines a property for the facet
  - Simple definition of factual truths
  - Inclusion and renaming if existing facets
- Terms may be, but are not always executable structures
  - Terms simply state truths
- Term visibility is managed like variable visibility
  - If exported, the term is referenced using the "." notation
  - If not exported, the term is not visible outside the facet

Trigger Terms

- Terms define the state value and the output value

```
t1: if s=1 then if x>=0.4 then s'=1 else s'=0 endif;
                else if x=<0.7 then s'=0 else s'=1 endif;
t2: y@t+10ns=s;
```

- Term t1 defines the state in terms of the current state and the current inputs
- Term t2 defines that the output should be equal to the state value 10ns in the future
- The continuous domain provides the semantics for time and the semantics of the reference function @

Trigger Terms

- Neither trigger term is executable, but state equalities
  - T1 places constraints on the value of state with respect to the current inputs
  - T2 places constraints on the value of output 5 nanoseconds in the future
- Other domains provide other mechanisms for specification semantics

Packages

- A package is a special purpose facet used to collect, parameterize and reuse definitions

Package Name *Package Parameters*

```
package wordTypes(w::natural) is
begin logic
  word::type(bitvector) is bitvector(w);
  word2nat(w::word)::natural is
     ...
  facet wordAdder(x,y::word)::word is
     ...
end wordTypes;
```

Package Domain

Package Definitions

Package Semantics

- Packages are effectively facets without terms
  - All elements of the package are treated as declarations
  - All package definitions are implicitly exported
- The package domain works identically to a facet domain
- Instantiating the package replaces formal parameters with actual parameters

The wordType Package

- The **wordType** package defines
  - A type **word**
  - A function for converting words to naturals
  - A facet defining a word adder
- All definitions are parameterized over the word width specified by **w**
  - Using **wordType(8)** defines a package supporting 8 bit words

Domains and Interactions

- Domains provide domain specific definition capabilities
  - Design abstractions
  - Design vocabulary
- Interactions define how specifications in one domain affect specifications in another

The Logic Domain

- The logic domain provides a basic set of mathematical expressions, types and operations
  - Basic types and operations with little extension
  - Best thought of as the domain used to provide basic mathematical structures
  - Currently, all domains inherit from the logic domain

The State-Based Domain

- The `state-based` domain supports defining behavior using axiomatic semantics
- Basic additions in the `state-based` domain include:
  - `S` – The state type
  - `next(s1::S)::S;` – Relates the current state to the next state
  - `x@s` – Value of `x` in state `s`
  - `x'` – Standard shorthand for `x@next(s)`

Defining State Based Specifications

- Define important elements that define state
- Define properties in the current state that specify assumptions for correct operation
  - Frequently called a precondition
- Define properties in the next state that specify how the model changes it’s environment
  - Frequently called a postcondition
- Define properties that must hold for every state
  - Frequently called invariants

Pulse Processing Example

- State is the last pulse received and its arrival time or none
- The initial pulse arrival time must be greater than zero
  - L1: pulseTime >= 0;
- If the initial pulse is of type A1 and the arriving pulse is of type A2, reset and wait for another pulse
  - L2: pulse=A1 and inPulse=A2 implies pulse'=none
- If the initial pulse is of type A1 and the arriving pulse if of type A1, then output command
  - L3: pulse=A1 and inPulse=A1 implies pulse'=none and o'=interpret(pulseTime,inPulseTime);
- No state should ever have a negative time value
  - L4: forall(s1::S | pulseTime@s >=0)

The Pulse Processor Specification

```
facet pp-function(inPulse:: in PulseType;
                  inPulseTime:: in time;
                  o:: out command) is
  use timeTypes; use pulseTypes;
  pulseTime :: time;
  pulse :: PulseType;
begin state-based
  L1: pulseTime >= 0;
  L2: pulse=A1 and inPulse=A2 => pulse'=none;
  L3:pulse=A1 and inPulse=A1 => pulse'=none and
      o'=interpret(pulseTime,inPulseTime);
  L4: forall(s1::S | pulseTime@s >=0);
end pp-function;
```

When to Use the State-based Domain

- Use state-based specification when:
  - When a generic input/output relation is known without detailed specifics
  - When specifying software components
- Do not use state-based specification when:
  - Timing constraints and relationships are important
  - Composing specifications is anticipated

The Finite State Domain

- The finite-state domain supports defining systems whose state space is known to be finite
- The finite-state domain is a simple extension of the state-based domain where:
  - S is defined to be or is provably finite

Trigger Example

- There are two states representing the current output value
  - S::type(integer) = 0++1;
- The next state is determined by the input and the current state
  - L1: next(0) = if i>=0.7 then 1 else 0 endif;
  - L2: next(1) = if i=<0.3 then 0 else 1 endif;
- The output is the state
  - L3: o'=s;

The Trigger Specification

```
facet trigger(i:: in real; o:: out bit) is
  S::type = 0,1;
begin state-based
   L1: next(0) = if i>=0.7 then 1 else 0 endif;
   L2: next(1) = if i=<0.3 then 0 else 1 endif;
   L3: o'=s;
end trigger;
```

When to Use the Finite State Domain

- Use the `finite-state` domain when:
  - Specifying simple sequential machines
  - When it is helpful to enumerate the state space
- Do not use the `finite-state` domain when
  - The state space cannot be proved finite
  - The specification over specifies the properties of states and the next state function

The Infinite State Domain

- The `infinite-state` domain supports defining systems whose state spaces are infinite
- The `infinite-state` domain is an extension to the `state-based` domain and adds the following axiom:
  - next(s) > s
- The `infinite-state` domain asserts a total ordering on the state space
  - A state can never be revisited

The Pulse Processor Revisited

- Is the arrival time and the type of the last received pulse
- The initial pulse arrival time must be greater than zero
  - L1: pulseTime >= 0;
- Adding the infinite state restriction assures that time advances
- If the initial pulse is of type A1 and the arriving pulse is of type A2, reset and wait for another pulse
  - L2: pulse=A1 and inPulse=A2 implies pulse'=none
- If the initial pulse is of type A1 and the arriving pulse if of type A1, then output command
  - L3: pulse=A1 and inPulse=A1 implies pulse'=none and o'=interpret(pulseTime,inPulseTime);
- No state should ever have a negative time value
  - L4: forall(s1::S | pulseTime@s >=0)

The Discrete Time Domain

- The **discrete-time** domain supports defining systems in discrete time
- The **discrete-time** domain is a special case of the **infinite-state** domain with the following definition
  - next(t)=t+delta;
- The constant **delta>=0** defines a single time step
- The state type **T** is the set of all multiples of **delta**
- All other definitions remain the same
  - next(t) satisfies next(t)>t

Time Constrained Pulse Processor

- State is the last pulse received and its arrival time or none
- The initial pulse arrival time must be greater than zero
  - Guaranteed by definition of time
- If the initial pulse is of type A1 and the arriving pulse is of type A2, reset and wait for another pulse
  - L2: pulse=A1 and inPulse=A2 implies pulse@t+delta=none
- If the initial pulse is of type A1 and the arriving pulse if of type A1, then output command in under 2 time quanta
  - L3: pulse=A1 and inPulse=A1 implies pulse@t+delta=none and o@t+2*delta=interpret(pulseTime,t);
- No state should ever have a negative time value
  - Guaranteed by the definition of time

Discrete Time Pulse Processor

```
facet pp-function(inPulse::in PulseType;
                            o::out command) is
  use pulseTypes;
  pulseTime :: T;
  pulse :: PulseType;
begin discrete-time
  L2: pulse=A1 and inPulse=A2 => pulse@t+delta=none;
  L3:pulse=A1 and inPulse=A1 => pulse@t+delta=none and
      o@t+2*delta=interpret(pulseTime,t);
end pp-function;
```

Understanding the Discrete Time Pulse Processor

- Each state is associated with a discrete time value
  - Event times are constrained
  - Time properties account for preconditions and invariants
- The `next` function is defined as previously
  - Can reference arbitrary time spaces

When to Use the Discrete Time Domain

- Use the discrete-time domain when:
  - Specifying discrete time digital systems
  - Specifying concrete instances of systems level specifications
- Do not use the discrete-time domain when:
  - Timing is not an issue
  - More general state-based specifications work equally well

The Continuous Time Domain

- The **continuous-time** domain supports defining systems in continuous time
- The **continuous-time** domain has no notion of next state
  - The time value is continuous – no next function
  - The “@” operation is still defined
    - Alternatively define functions over t in the canonical fashion
  - Derivative, indefinite and definite integrals are available

Continuous Time Pulse Processor

- Not particular interesting or different from the discrete time version
  - Can reference arbitrary time values
  - Cannot use the `next` function
  - No reference to discrete time – must know what delta is

Continuous Time Pulse Processor

```
facet pp-function(inPulse::in PulseType;
                          o::out command) is
  use pulseTypes;
  pulseTime :: T;
  pulse :: PulseType;
begin continuous-time
  L2: pulse=A1 and inPulse=A2 => pulse@t+5ms=none;
  L3: pulse=A1 and inPulse=A1 => pulse@t+5ms=none and
      o@t+10ms=interpret(pulseTime,t);
end pp-function;
```

Understanding the Continuous Time Pulse Processor

- Discrete time references are replaced by absolute time references with respect to the current time
  - Using 5ms and 10ms intervals rather than the fixed time quanta

Using the Continuous Time Domain

- Use the continuous-time domain when
  - Arbitrary time values must be specified
  - Describing analog, continuous time subsystems
- Do not use the continuous-time domain when:
  - Describing discrete time systems
  - State based specifications would be more appropriate

Specialized Domain Extensions

- The domain `mechanical` is a special extension of the logic and continuous time domains for specifying mechanical systems
- The domain `constraints` is a special extension of the logic domain for specifying performance constraints
- Other extensions of domains are anticipated to represent:
  - New specification styles
  - New specification domains such as optical and MEMS subsystems

Specification Composition

- Compose facets to define multiple models of the same component
  - Using the facet algebra
  - Components
- Compose facets to define systems structurally
  - Including facets as terms
  - Instantiating facets
  - Channels and models of computation

Facet Semantics

- The semantics of a facet is defined by its domain and terms
  - The domain defines the formal system associated with the facet
  - The terms extend the formal system to define the facet
- An interaction defines when information from one domain effects another
- A Rosetta specification defines and composes a collection of interacting models

Formal Systems

- A *formal system* consists of the following definitions:
  - A *formal language*
    - A set of *grammar rules*
    - A set of *atomic symbols*
  - An *inference mechanism*
    - A set of *axioms*
    - A set of *inference rules*
  - A *semantic basis*
- In Rosetta, these elements are defined in the domain specification
  - Language and inference mechanism are relatively fixed
  - Semantics varies widely from domain to domain

Semantic Notations

- The semantics of a facet F is defined as an ordered pair (D_F, T_F) where:
  - D_F defines the domain (formal system) of the specification
  - T_F defines the term set defining the specification
- A facet definition is consistent if T_F extends D_F conservatively
  - FALSE cannot be derived from T_F using D_F

Facet Conjunction

- Facet conjunction defines new facets with properties of both original facets

$$F \longrightarrow F\ \text{and}\ G \longleftarrow G$$

- Facet F and G reflects the properties of both F and G simultaneously
- Formally, conjunction is defined as the *co-product* of the original facets

Facet Conjunction Example

```
facet pp-function(inPulse::in PulseType;
                  o::out command) is
  use pulseTypes;
  pulseTime :: T;
  pulse :: PulseType;
begin discrete-time
  L2: pulse=A1 and inPulse=A2 => pulse'=none;
  L3:pulse=A1 and inPulse=A1 => pulse'=none and
     o@t+2*delta=interpret(pulseTime,t);
end pp-function;

facet pp-constraint is
  power::real;
begin constraints
  c1: power=<10mW;
  c2: event(inPulse) <-> event(o) =< 10mS;
end pp-constraint;
```

Facet Conjunction Example

- Define a new pulse processor that exhibits both functional and constraint properties:

```
facet::pp is pp-function and pp-constraints;
```

- The new pp facet must exhibit correct function and satisfy constraints
  - Functional properties and heat dissipation are independent
  - Functional properties and timing constraints are *not* independent

When to Use Facet Conjunction

- When a component or system should exhibit two sets of properties
- When a component or system should exhibit two orthogonal functions

Understanding Facet Conjunction

- Given two facets F and G the conjunction F and G might be defined formally as:

  $$F \text{ and } G = \{(D_F, T_F), (D_G, T_G)\}$$

- The conjunction is literally a facet consisting of both models
- If F and G are orthogonal, this definition works fine
  - F and G are rarely orthogonal
  - Orthogonality makes things rather uninteresting
- Thus we define an *interaction*

Understanding Facet Conjunction

- Given two facets F and G the conjunction F and G is defined formally as:
  $F \text{ and } G = \{(D_F, T_F + M\_\_I(G,F)), (D_G, T_G + M\_\_I(F,G))\}$
- The interaction function, M__I, maps terms from one domain into terms from another
- An *interaction* defines the function M__I for a domain pair
- A *projection* extracts the model associated with a domain from a facet:

$$M\_\_proj((F \text{ and } G), D_G) = (D_G,\ T_G + M\_\_I(F,G))$$

Domain Interaction Semantics

Interaction defines affects of information from facet from D_j on D_k defining F_k'

Composition is coproduct

$$\begin{tikzcd}
F_0 \arrow[dd] \arrow[dr] & & F_1 \arrow[dl] \arrow[dd] \\
& F_0 \text{ and } F_1 \arrow[dl] \arrow[dr] & \\
F_0' & & F_1'
\end{tikzcd}$$

Interaction defined using Rosetta's reflective capabilities

Projections form product

Understanding Facet Conjunction

- Composing facets from the same domain uses the same semantics as Z specification composition
  - A and B – All terms from both facts are true
  - A or B – Conjunctions of terms from facets are disjuncted
- If the conjunction of two facets does not generate new terms, then those facets are orthogonal with respect to conjunction
  - This is important as it can reduce analysis complexity stubstantially

Interaction Definitions

- An interaction is a special package that defines M__I for two domains:

Interaction Operation *Domains*

```
interaction and(state-based,logic::domain) is
begin interaction
  M__I(f::logic,g::state-based)::set(term) is
   {ran(t::M__terms(f) | ran(s::g.S | t@s))}
  M__I(g::state-based,f::logic)::set(term) is
   {sel(t::M__terms(g) | forall(s::g.S | t@s))}
end and;
```

Interaction Domain

Interaction Function

Understanding Facet Conjunction

- After taking a deep breath…

- The interaction explains how domains affect each other
- The projection extracts a facet from a particular domain from another facet
- To understand how domains interact
  - Form the composition using interactions
  - Project the result into the domain of interest
  - The results of the interaction are presented in the domain of interest

Facet Disjunction

- Facet disjunction defines a new facet with properties of either original facet

$$F \longleftarrow F \text{ or } G \longrightarrow G$$

- Facet F or G reflects the properties of either F or G
- Formally, disjunction is defined as the product of the original facets

Facet Disjunction Example

```
facet pp-function(inPulse::in PulseType;
                                 o::out command) is
  use pulseTypes;
  pulseTime :: T;
  pulse :: PulseType;
begin discrete-time
  L2: pulse=A1 and inPulse=A2 => pulse'=none;
  L3:pulse=A1 and inPulse=A1 => pulse'=none and
       o@t+2*delta=interpret(pulseTime,t);
end pp-function;

facet pp-constraint is
  power::real;
begin constraints
  c1: power=<10mW;
  c2: event(inPulse) <-> event(o) =< 10mS;
end pp-constraint;
```

Facet Disjunction Example

```
facet pp-lp-constraint is
  power::real;
begin constraints
  c1: power=<5mW;
  c2: event(inPulse) <-> event(o) =< 15mS;
end pp-constraint;
```

- A component that satisfies functional requirements and either constraint set is defined:
  pp::facet is pp-function and
      (pp-lp-constraint or pp-constraint)

- pp is a component that represents either the normal or low power device

When to Use Facet Disjunction

- When a component may exhibit multiple sets of properties
- When representing a family of components

Facet Declarations

- Facets may be defined in the same manner as other items:

  f::facet [is *facet-expression*];

- The type facet is the bunch of all possible facets
- *facet-expression* is an expression of type facet

- Can also define a variable facet without a predefined value:

  f::facet;

Component Aggregation

- System decomposition and architecture are represented using aggregation to represent structure
  - Include and rename instances of components
  - Interconnect components to facilitate communication
- Propagate system properties onto components
  - Label distribution
- Aggregation approach
  - Include facets representing components
  - Rename to preserve internal naming properties
  - Communicate through sharing actual parameters
  - Use label distribution to distribute properties among components

Facet Inclusion

- Include and rename facets to represent components
  - rx:rx-function(i,p) includes rx-function and renames it rx
  - Access labels in rx using rx.*label* not rx-function.*label*
  - Achieves instance creation with little semantic effort
- Use internal variables to achieve perfect, instant communication

```
facet iff-function(i::in signal; o::out signal) is
  p::pulseType; c::command;
begin logic
  rx: rx-function(i,p);
  pp: pp-function(p,c);
  tx: tx-function(c,o);
end iff;
```

Facet Inclusion

- The same technique works for facets of any variety
- Consider a structural definition of component constraints

```
facet iff-constraint is
  power::real;
begin logic
  rx: rx-constraint;
  pp: pp-constraint;
  tx: tx-constraint;
  p: power = rx.power+pp.power+tx.power;
end iff;
```

Label Distribution

- Labels distribute over most logical and facet operators:
  - L: term1 and L:term2 == L: term1 and term2
  - L: term1 or L:term2 == L: term1 or term2
  - L: term1 => L:term2 == L: term1 => term2
  - L: term1 = L:term2 == L: term1 = term2
- Consequences when conjuncting structural definitions are interesting

Conjuncting Structural Definitions

```
facet iff-function(i::in signal; o::out signal) is
  p::pulseType; c::command;
begin logic
  rx: rx-function(i,p);
  pp: pp-function(p,c);
  tx: tx-function(c,o);
end iff;
```

```
facet iff-constraint is
begin logic
  rx: rx-constraint;
  pp: pp-constraint;
  tx: tx-constraint;
  p: power = rx.power+...;
end iff;
```

iff::facet is iff-function and iff-constraint

Combining Terms

```
facet iff-function(i::in signal; o::out signal) is
  p::pulseType; c::command;
begin logic
  rx: rx-function(i,p);
  pp: pp-function(p,c);
  tx: tx-function(c,o);
  rx: rx-constraint;
  pp: pp-constraint;
  tx: tx-constraint;
  p: power = rx.power+...;
end iff;
```

Applying Label Distribution

```
facet iff-function(i::in signal; o::out signal) is
  p::pulseType; c::command;
begin logic
  rx: rx-function(i,p) and rx: rx-constraint;
  pp: pp-function(p,c) and pp: pp-constraint;
  tx: tx-function(c,o) and tx: tx-constraint;
  p: power = rx.power+...;
end iff;
```

```
facet iff-function(i::in signal; o::out signal) is
  p::pulseType; c::command;
begin logic
  rx: rx-function(i,p) and rx-constraint;
  pp: pp-function(p,c) and pp-constraint;
  tx: tx-function(c,o) and tx-constraint;
  p: power = rx.power+...;
end iff;
```

Label Distribution Results

- In the final specification, component requirements coalesce based on common naming
  - Using common architectures causes components to behave in this way
  - Systems level requirements are “automatically” associate with components

Component Families

- Parameters can be used to select from among component options when combined with if constructs
- Leaving the select parameter open forces both options to be considered.

```
facet iff-constraint(lp::in boolean) is
begin logic
  rx: rx-constraint;
  pp: if lp then pp-lp-constraint
               else pp-constraint
        endif;
  tx: tx-constraint;
  p: power = rx.power+...;
end iff;
```

Avoiding Information Hiding

- The `use` clause allows packages and facets to be included in the declaration section
- All exported labels in used packages and facets are added to the name space of the including facet
- The `use` clause must be used carefully:
  - All encapsulation is lost
  - Includes at the item level rather than the facet level
  - Used primarily to include definitions from standard packages and libraries

An Example Rosetta Specification

- To be provided interactively at the tutorial

Advanced Topics

- Reflection and meta-level operations
- Interactions
- Architecture definition
- Communication and Models of Computation

- Information to be provided at the tutorial based on student interest

Meta-Level Operations

- Rosetta Meta-level operations allow specifications to reference specifications
- All Rosetta items have the following meta-level information:
  - M__type(I) – Type of item I
  - M__label(I) – Label of item I
  - M__value(I) – Value of item I
  - M__string(I) – Printed form of item I
- Specific items are defined using specialized operators

Defining Interactions and Domains

- A principle use for meta-functions is defining interactions and domains
  - Most users never see interaction or domain definitions
- A simple interaction defines the relationship between terms in logic and terms in state-based descriptions:

```
Interaction and(f1::logic; f2::state-based) is
begin logic
 l1: forall(t::M__terms(f1) |
        forall(s::State is t@s :: M__terms(f2)))
end and;
```

Architecture Definition

- Facets parameterized over facets supply architecture definitions:

```
facet batch-seq(x::in T; z::out T, f1,f2::facet) is
  a::M__type(f1.x)**M__type(f2.x);
begin logic
  c1::f1(x,a);
  c2::f2(a,z);
end batch-seq;
```

Instantiating Architectures

- Define instances of the architecture by instantiating the facet parameters

```
search(x::in T; z::out T) :: facet is
  batch-seq(_,_,sort,binary_search)
```

- Instantiate the component parameters but not the input and output parameters
- Naming conventions must be maintained

Summary

- Tutorial provides a basic introduction to the nuts and bolts of Rosetta specification
- Defines types and functions available
- Defines facets and packages as specification units
- Defines domains available to specifiers
- Defines specification composition
- Examples and Exercises provided interactively
  - Please contact authors for hard copies

Where to Get More Information

- The Rosetta web page contains working documents and examples:

  http://www.sldl.org
- Working documents include
  - Usage Guide
  - Semantic Guide
  - Domain and Interaction definition
- Tool downloads include
  - Java-Based Rosetta Parser