Composition Semantics of the Rosetta Specification Language

BY

Megan E. Peck

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Date Defended: August 30, 2012

The Thesis Committee for Megan E. Peck
certifies that this is the approved version of the following thesis:

Composition Semantics of the Rosetta Specification Language

Chairperson Dr. Perry Alexander

Date Approved:

Abstract

The Rosetta specification language aims to enable system designers to abstractly design complex heterogeneous systems. To this end, Rosetta allows for compositional design to facilitate modularity, separation of concerns, and specification reuse. The behavior of Rosetta components and facets can be viewed as systems, which are well suited for coalgebraic denotation. The previous semantics of Rosetta lacked detail in the denotational work, and had no firm semantic basis for the composition operators. This thesis refreshes previous work on the coalgebraic denotation of Rosetta. It then goes on to define the denotation of the composition operators. A real-world Rosetta example using all types of composition serves as a demonstration of the power of composition as well as the clean, modular abstractness it affords the designer.

Acknowledgements

This thesis may be a long time coming, but that is not for lack of support. I first must thank my husband, Wesley. You've encouraged me from before I even thought I could do grad school, up to the very end, and have had more faith in me than I have had in myself. Thank you for your love, encouragement and help, and thank you for taking care of our family and giving me the opportunity to pursue my education.

Thank you to my son Eli, and my son on the way. You've given me the motivation I needed to finish and to be the best mom I can be to you. I hope you are free to pursue everything you want to in life, and that I can be supportive of you in the ways that I have been supported. Thank you to the rest of my family who helped establish every building block to get me here today.

Thank you to my advisor, Dr. Perry Alexander. You helped convince me that graduate school was for me, and that a higher degree was an attainable goal. You've also been very patient with my long route to a degree and understood that for me, being a wife and mom was central in my life, and never pressured me or made me feel guilty about that. I would have left long ago without such a wonderful mentor who cares about what is important to his students.

Thank you to my committee for getting me to my final goal, and all of the professors who have taught me so much along the way. Thank you to the students in the CSD lab throughout the years, who have given me great friendship and lots of laughs as well as commiseration.

Thank you to my God, who knows every desire of my heart, and every way in which I am pulled. You created me with purpose, and above any success on earth, I hope that I can be known as one like David, who is after your heart above all else. Thank you for every blessing in my life, and thank you that you find satisfaction in me apart from any achievement.

The work in this thesis was partially funded by The Kansas Partnership for Graduate

Fellows in K-12 Education (NSF GK-12 Award Number DGE-0742523) and by Cadstone, LLC (Simulation of Rosetta Representations and Evaluations, Project Number IND41660) .

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-world systems tend to be too large and complex to reason about in a single specification language or domain vocabulary. For one, systems are typically heterogeneous in nature, which necessitates different vocabularies for each aspect of the heterogenous design. System designs are also naturally done on a component-level basis, in a block-diagram style. The complexities of an entire system cannot realistically be expressed at the top level of a design. Rather, to be a scalable, feasible design representation, a specification language must allow for many kinds of composition to enable the designer to separate concerns and build up a system in pieces. Composition in system-level specification languages not only aids the specifier in managing the size and heterogeneity of real-world systems, but also supports predictive analysis. Automated reasoning techniques can be applied to the smaller pieces within the appropriate domain vocabulary and knowledge, rather than trying to apply automated reasoning to a monolithic specification [Frisby et al., 2011].

The Rosetta specification language [Alexander, 2006, Alexander et al., 2000] provides a semantic structure for system-level design that centers on heterogeneous model composition. It aims to aid the specifier in meaningfully composing components of the system into the whole, or alternatively decomposing the whole system into specific parts. Built-in composition operators give the designer the ability to design complex, heterogeneous systems in a component-level fashion, building the entire system out of domain-specific pieces. The basic unit of specification in Rosetta is a *component* that collects a set of declarations and states assumptions, definitions, and implications about those declarations within a single domain. In Rosetta, components are first-class structures and can be manipulated as data. Three primitive operations are defined for component composition: structural composition constructs a component that includes the operand facets as components; conjunctive composition defines a component that satisfies all given operand components; and disjunctive composition defines a component that satisfies one or more of the given operand facets.

Formalizing the semantics of a specification language gives us assurances as to the validity of the specifications we write with the language. This thesis refreshes previous work on the coalgebraic denotation of components and facets. While the ideas of the Rosetta composition operators are not new, they have yet to be formally denoted. This work fills that semantic void by defining the denotation of the composition operators. A real-world Rosetta example using all types of composition serves as a demonstration of the power of composition as well as the clean, modular abstractness it affords the designer.

Chapter 2

Rosetta Background

Many complex systems have fundamental requirements that cannot be expressed in one common domain. Thus, such systems cannot be easily specified using one common semantics. For example, embedded systems may contain digital and analog components that must communicate; a house is at once a load-bearing structure, an electrically wired system, an HVAC system, and must obey certain accessibility, zoning, and safety regulations; and an aircraft must fly, consume power, and deliver entertainment to its passengers. In each case, different views of the same entity must be satisfied to successfully construct an iPod, a house that passes all inspections, or an aircraft in which people will fly. Yet, each view requires some specialized or expert knowledge to express the requirements. No single language can encompass all of these views simultaneously for any system of even marginal complexity.

Rosetta accommodates these multiple specification views by providing a framework for characterizing these views, called *domains* [Streb et al., 2006, Streb and Alexander, 2006]. Each domain defines a domain-specific modeling vocabulary and semantics for representing information related to a specification aspect. The basic building blocks of Rosetta are *components* and *facets*. A component defines assumptions, definitions, and implications over a set of declarations. A facet is a component with no assumptions or implications. Every facet models a specific system aspect by extending a domain with problem specific definitions. For example, a digital adder might be written in the `discrete_time` domain while a filter might be written in the `frequency` domain. In these cases, the domain is like the *type* of the facet.

Rosetta includes a pre-defined domain hierarchy shown in Figure 2.1. Additionally, a designer may extend the domain hierarchy to include a new domain with more specific knowledge or to write models using a new semantic basis. This knowledge can be as simple as a set of declarations and related axioms or as complicated as the domain requires.

The Rosetta domain hierarchy forms a complete lattice ordered by theory homomorphism. In Figure 2.1, arrows define extension resulting in more concrete domains. Inverse arrows define abstractions resulting in more abstract domains. Together, these relationships form a Galois connection [Streb et al., 2006, Streb and Alexander, 2006] between adjacent domains. This relationship is critical when viewing facet transformation as moving among domains in the lattice.

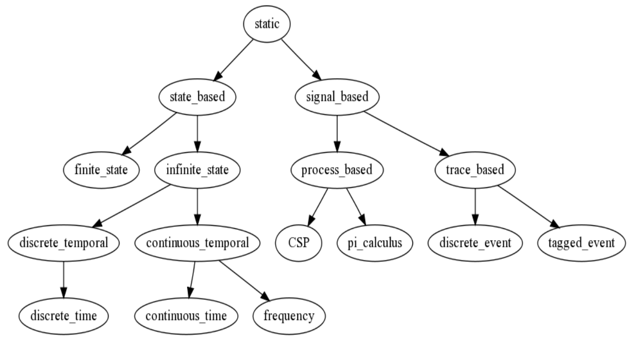

Figure 2.1.: The lattice of domains in Rosetta.

As specifications can be written in multiple domains, a mechanism is needed for meaningfully combining information represented in facets from those various domains that construct the entire specification. In Rosetta, *interactions* provide ways to reason about specifications from different domains. Just as users can extend domains with new domains, they can define interactions between domains. The interaction construct describes how information flows between two domains – as the name suggests, how two domains interact with each other. In this way, the design of the Rosetta language does not attempt to support all conceivable domains by itself, but instead offers a framework in which the appropriate domains and interactions may be constructed.

2.1 Building Blocks

The basic specification structuring unit in Rosetta is the *component* or *facet*. A component definition extends a particular domain with definitions, assumptions, assertions, and implications. Components may have inputs and outputs that allow them to be parameterized and to communicate with other specification constructs. The terms within a component may either be Boolean expressions written in Rosetta's expression language or may instantiate other components to define structural, hierarchical specifications. Since it is common to not need the assumptions or implications of a component, facets are more commonly seen in Rosetta than the more general components, as facets are simply components with no assumptions or implications.

```
facet halfAdder (x,y::input bit;
                 s,c::output bit)
               :: state_based is
begin
   s'=x xor y;
   c'=x and y;
end facet halfAdder;
```

Figure 2.2.: Half adder specification.

The example `halfAdder` facet shown in Figure 2.2 has two input bits, `x` and `y`, and sum and carryout outputs, `s` and `c`, respectively. The domain of the facet is `state_based`, as the sum and carry are computed for the next state, given the current inputs. The next states, denoted by a "ticked" symbol, of the outputs are constrained by Boolean expressions that equate them with values calculated from current state variables. This intentionally follows closely the convention used by hardware designers writing VHDL [Ins, 1994] or Verilog [IEE, 1995].

In the following sections we use these basic constructs – components, facets, and domains – to explore three types of composition built in to Rosetta. The three types of composition allowed in Rosetta are *structural*, *conjunctive*, and *disjunctive* composition. Each type of composition is motivated, described, and a Rosetta example of each type demonstrates its use.

2.2 Structural Composition

Structural, or *hierarchical composition* allows the designer to specify a system as a collection of components. Structural decomposition and composition is a common method of system design where components are composed into systems. This type of composition is readily understood as component instantiation or inclusion. It is extensively used by hardware design languages such as VHDL, Verilog, SystemVerilog [Acc, 2002] and SystemC [Grötker et al., 2002] or software architecture specification languages [Allen and Garlan, 1997, Nuseibeh et al., 2003].

Structural composition allows the designer to reuse models of components, and to build up systems out of these components. The concept of structural composition provides the semantic tool for specifying systems in a manner that reflects how they are already designed, allowing direct representation of the structure already inherent in the system being specified. Thus, the reusable units in a specification will mirror the recurring units throughout the actual structure of the system. This is common for composable elements in a hierarchical specification that is particularly popular in hardware design.

Continuing with the halfAdder example from the previous section, we now create the fullAdder facet by instantiating two halfAdders and appropriately interconnecting their input and output parameters. The resulting specification is shown in Figure 2.3.

```
facet fullAdder (x,y,ci::input bit;
                 s,co::output bit)
               :: state_based is
  s1,c1,c2::bit;
begin
  ha1: halfAdder(x,y,s1,c1);
  ha2: halfAdder(s1,ci,s,c2);
  co' = c1 or c2;
end facet fullAdder;
```

Figure 2.3.: Structural full adder.

The fullAdder uses three internal variables, s1, c1 and c2 to share information between the instantiated halfAdders. The fullAdder uses structural composition implemented as facet inclusion to create the traditional implementation of a full adder using two half adders. The output, s, of fullAdder is constrained through the constraints given by halfAdder, while co is constrained through a new Boolean expression.

By *instantiating* the halfAdder facet, we get a distinct unit with the given parameters applied and with the equality constraints over that facet enforced. The fullAdder facet instantiates two halfAdders. Their instantiations are separate from one another, and the only relation between them is their sharing of the parameters s1 and c1. There is no depth limit to hierarchical composition; we could just as easily use fullAdder instantiations to create ripple carry adders for inclusion in an ALU, and could then instantiate an ALU in a CPU design, instantiate the CPU in an embedded systems design, continuing as far as necessary.

This composition is available at any level of the specification from simple combinational circuits through entire processors, embedded systems, and systems-of-systems. As a more complex example, Figures 2.4 and 2.5 show how a simple structural CPU model is constructed from components in the canonical fashion.

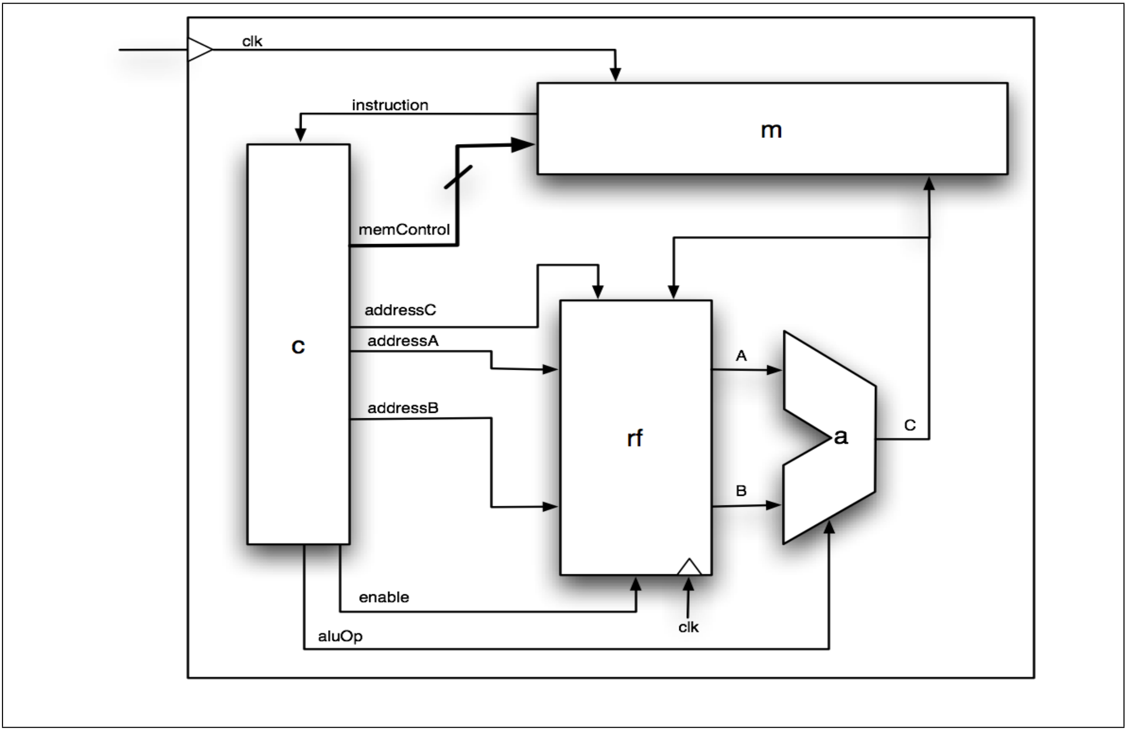

Figure 2.4.: Structural CPU block diagram.

Limiting ourselves to just hierarchical composition has drawbacks, however. We are required to define all behavior and constraints in place. Any non-functional behaviors would necessarily be involved in the same specifications that defined the behavior. Conjunctive composition alleviates this problem by adding a 'horizontal' composition that allows two specifications to represent the *same* unit, both constraining what that system is and how that system behaves.

```
facet controlUnit (..)::HW ..
facet alu     (..)::HW ..
facet regFile    (..)::HW ..
facet memory (..)::HW ..

facet cpu(clk) :: HW is
 enable::bit;
 instruction,A,B,C::word;
 addressA,addressB,addressC,aluOP::nibble;
 memControl::bitVector;
begin
  c: controlUnit (instruction,memControl,
                  addressA,addressB,addressC,
                  enable,aluOP);
  a: alu     (aluOp,A,B,C);
  rf: regFile    (clk,enable,addressA,addressB,
                  addressC,A,B,C);
  m: memory (clk,memControl,C);
end facet cpu;
```
Figure 2.5.: Structural CPU example.

2.3 Conjunctive Composition

Specification conjunction allows the designer to specify multiple views of a single component and compose them into a single model. As such, specification composition provides language level support for separation of concerns. Conjunctive composition is done through the product operator, $*$. We can define a facet $f3$ as the product of facets $f1$ and $f2$ by saying

```
facet f3 = f1 * f2;
```

Using specification conjunction a system designer may specify functional requirements that define what a system does separately from physical constraints such as resource limitations, available implementation fabrics, and usage assumptions. Conjoining the resulting specifications allows concurrent design, modeling all aspects simultaneously.

Similarly, using specification conjunction, a designer may specify system behavior separately from implementation architecture specifics. This is a positive feature supporting co-design applications where a system designer should define the functional requirements of a component without tying the specification to particular hardware or software architecture details [Peck, 2011]. Conjoining the resulting specifications allows mapping of system requirements to individual system components.

As an example of how Rosetta supports specification at the language level, consider the functional behavior definition for a QAM modulator with encryption [Kimmell et al., 2008] in the `qamAESArch` facet in Figure 2.6. In the `structure1` and `structure2` facets, we define two alternative non-functional views of the same system that differ by requiring the sub-components to be implemented in hardware or software in different configurations.

As specified, the two models are independent – nothing has composed the functional and non-functional models. We use the facet product operator `*` to define the conjunctive composition of the `qamAESArch` facet with each non-functional requirements facet to describe two separate implementations in Figure 2.7.

This composition requires any resulting implementation to satisfy both facet specifications simultaneously – both in terms of constraining the domains and all definitions in the conjoined facets. Thus, `implementation1` is constrained by domains `static` and `fabric`, and by the definitions of `code`, `buff1`, `enc`, `buf2` and `modulate` of `behavior`, and the `code`, `buff1`, `enc` and `modulate` as defined in `structure1`. This composition searches for terms within the two facets with the same names, that are then similarly conjoined. The overall composed entity therefore must have within it one term for each shared name that satisfies both sets of requirements.

```
facet qamAESArch
     (i::input word(2); o::output real;
      f::input frequencyType;
      length::design keyLengthType;
      k::input word(length))::static is
     ho::bit;
     aesi::word(16);
     mi::word(2);
begin
     code: huffEncoder(i,ho);
     buff1: buffer(ho,aesi);
     enc: aesEncryptor(aesi,aeso,length,k);
     buff2: buffer(aeso,mi);
     modulate: qamModulator(mi,o,f);
end facet qamAESArch;

facet structure1() :: fabric is
begin
     code: hardware(fpga);
     buff1: hardware(fpga);
     enc: hardware(crypto);
     buff2: hardware(fpga);
     modulate: hardware(fpga);
end facet structure1;

facet structure2() :: fabric is
begin
     code: hardware(fpga);
     buff1: hardware(fpga);
     enc: software(proc1);
     buff2: software(proc2);
     modulate: software(proc2);
end facet structure2;
```

Figure 2.6.: Rosetta conjunction co-design example for a QAM modulator with encryption.

```
facet implementation1::static is qamAESArch * structure1;
facet implementation2::static is qamAESArch * structure2;
```

Figure 2.7.: Rosetta conjunction specifying two implementations of a QAM modulator with encryption.

With hierarchical and conjunctive composition, we are approaching a designer's goals of reuse and separation of concerns. However, we are still limited in that a conjunctive facet must meet all definitions and constraints of each facet in the product. The designer may desire the ability to compose separate aspects of a function, while requiring only one of several sets of definitions and constraints to hold. In the next section, we show how disjunction fulfills this desire to maintain separation of concerns under this situation.

2.4 Disjunctive Composition

Specification disjunction allows the designer to specify alternate views of a system where only one view must be satisfied at a given time. Disjunctive composition is a means of separately defining alternatives of behavior or constraints within a system in such a fashion that they can be connected afterwards. Disjunctive composition is done through the sum operator, $+$. We can define a facet $f3$ as the sum of facets $f1$ and $f2$ by saying

```
facet f3 = f1 + f2;
```

The disjunction, or sum, of two facets is itself a facet where alternate definitions are provided. At least one definition needs to be valid in a valid sum, though it does not necessitate mutual exclusivity. All "alternate" definitions might hold in a valid sum. In the running co-design example, we might use disjunction to define possible structures.

```
facet anyStructure::fabric is structure1 + structure2;
```

Figure 2.8.: Rosetta disjunction example allowing multiple possible structures.

In Figure 2.8, `anyStructure` is a composed facet that must satisfy either `structure1`'s behavior and constraints or `structure2`'s behavior and constraints. The disjunction allows for both to be satisfied at once – it is not an exclusive `or`. Often, in practice the terms of the facets being composed will themselves disallow both being satisfied at once.

We can now define a system specification that shows that both structures might be valid in our co-design specification. Two available (and semantically equivalent) implementations are show in Figure 2.9.

```
facet implementation3::static is
  qamAESArch * anyStructure;
facet implementation4::static is
  qamAESArch * (structure1 + structure2);
```

Figure 2.9.: Co-design implementations using disjunction.

In these implementations, we require `behavior` to be satisfied, and we require *either* `structure1` or `structure2` to be satisfied. This sort of implementation would allow us to consider a larger system containing this implementation without having to select one structure and exclude the other. We can define the behavior once, and have reuse with respect to the different possible implementation structures and details of the system. At the same time, this larger system is not leaving the structure entirely abstract as it explicitly lists the structures that are allowed, without selecting exactly one.

Disjunction allows the designer to specify multiple pieces of a component and compose them to make the whole. In this way, the designer can isolate functionalities. For instance, if an instruction or command in a system can be one of many options, the designer can specify each separately and compose them to create the entire instruction. This approach is also used in the specification language Z, via disjoints [Jim Woodcock, 1995].

Consider the design of a simple processor with a variety of different instructions shown in Figure 2.10. When writing the system, the designer may want to write each of the processor's behavior for each instruction separately, and compose them to create the processor's complete behavior.

```
domain processor :: state_based is
   registers   :: array(16, word);
   pc          :: word;
   instruction :: word is memFetch (pc);
   op          :: nibble is decodeOp (instruction);
begin
end domain processor;

facet plus :: processor is
   src1 :: nibble is decodeSrc1 (instruction);
   src2 :: nibble is decodeSrc2 (instruction);
   dest :: nibble is decodeDest (instruction);
begin
   op = plusOp;
   registers'= replace(registers,dest,
                         registers[src1] +
                           registers[src2]);
   pc' = pc + x"0002";
end facet plus;

facet jmp :: processor is
begin
   op  = jmpOp;
   registers' = registers;
   pc' = newPC(instruction);
end facet jmp;

facet processorBeh :: processor is plus + ... + jmp;
```

Figure 2.10.: Processor disjunction example.

The `processor` domain is defined to extend the `state_based` domain with declarations for a 16-register register file, `registers`, and a program counter, `pc`. Using a function `memFetch :: word -> word`, it constrains the `instruction` to be the value fetched from memory at address `pc`. Similarly, using a function to decode the instruction, `decodeOp :: word -> nibble`, the domain constrains the `op`.

The behavior for each `op` can then be written in its own facet with this new `processor` domain. For instance, the `plus` facet enforces the constraint that `op` be the operation for addition. Since the domain defines `registers` and `pc`, the `plus` facet must provide the next state constraints for these. The register file is updated using the `replace` function, which replaces the given index parameter with the new given value, and leaves the rest alone. In this way, we correctly define the framing rules by updating the destination register while leaving the rest unchanged. The facet also updates the `pc`'s next state simply by adding 2 to the current state.

Similarly, the `jmp` facet constrains that `op` must be the `jmpOp`, explicitly states that `registers` does not change, and updates `pc` with the newly calculated program counter value given by the function `newPC :: word -> word`. The other processor instructions would be written in the same fashion.

Given the individually written facets for each instruction, we can disjunct them to create a new facet, `processorBeh`, that defines the behavior of the processor for all possible operations. Since each individual facet has the domain `processor`, the disjuncted facet will as well. It should be noted that, although facet disjunction does not outright force mutual exclusivity, in this case only one facet in the conjunction can be consistent, since the decoded `op` will only match one possible processor operation.

The disjunction in this processor example illustrates two major benefits. The first benefit is the notion that the designer can separate the concerns of the different operations and focus on the behavior of one instruction at a time. The second benefit is the ease of extensibility of the processor design. New instructions can be written and added in a clear way by adding one more facet to the disjunction. This allows for the modularity a programmer is accustomed to utilizing, at a per-behavior level, and ensures that every facet in the disjunction is constraining the needed pieces of the facet — in this case, the `registers` and `pc`. This strategy can be of great value for more complicated specifications.

Chapter 3

Coalgebraic Denotation

This chapter is heavily based on previous work [Kong et al., 2003] and builds upon that work. The previous work sets up two parts to denoting a facet – defining the coalgebraic system structure, and denoting the syntactic pieces of a facet. The definition of coalgebraic system structure lacked a framework for the general case of denoting any facet. It describes the structure and denotes some specific facets. This work expands upon that, giving the framework for the general case. The previous work thoroughly describes the general case of denoting the syntactic pieces of a facet. However, we have updated these with some newer Rosetta requirements. The previous work also lacked discussion of components, rather defining the denotation for only facets. This work adds the details of component denotation.

3.1 Facet Syntactic Denotation, Refreshed

Facets are denoted in two parts. One part is defining its behavior as a coalgebraic structure. The abstract state of a facet is its observable behavior, which itself can be viewed as a system, and is therefore a coalgebra. The other part looks at the facet pieces, and denotes those syntactic pieces. This step goes inside the facet definitions and gives the semantic details of the facet.

First consider the denotation of the syntactic pieces of a facet. Recall the general syntactic parts of a typical facet f are

```
facet f (#parameters#)::#domain# is
   #variables#
begin
   #terms#
end facet f;
```

Rosetta facets consist of observers (from parameters and variables and any variables that come from the domain definition), a domain, and sets of definitions, or terms. Every facet has only one set of observers. These observers are essentially the interface of the facet. There can be multiple sets of terms, though it is most typical to only have one set.

Though in previous work, facets were considered 4-tuples, <l,O,D,T>, we now define them as 3-tuples, <O,D,T> where

- l is the label of the facet, or the facet name.
- O contains the observers (parameters and variables, including all domain variables) of f. So, $O = (O_1, O_2, ..., O_n)$ where each O_i is an observer of f.

- D is the domain of the components.
- $T = [T_1, T_2, ... T_n]$, where each T_j is a set of denotations of f's terms.

We now choose to exclude the label to treat it more like a lambda, in the lambda calculus. Essentially, we are creating an interface, and the label of that interface is irrelevant in describing the system behavior. Each facet instance would be unique, and have its own label. The label of the facet definition is extraneous, and thus now omitted. Also, this work adds the notion of allowing multiple bodies within a facet or component. The motivation for this change is explained in Section 4.2.1.2. The denotation has been refreshed to reflect these changes.

Any specification of a facet is consistent if at least one of the sets of terms is consistent. Note that the majority of facets will only have one set of terms. The denotations for the terms themselves has already been done [Kong et al., 2003]. The following section describes these denotations to finish off the details of facet denotations.

There are three valuation functions, E, O, and V, for expressions, operators, and values, respectively, where

- $E[\![\varepsilon]\!] : Environment \rightarrow Values$
- $O[\![\Omega]\!] : Universal \rightarrow Universal \rightarrow Universal$ (where $Universal$ includes all values, including lambdas)
- $V[\![v]\!] : Constants \rightarrow Values$

As an example of the use of these valuations, consider the denotation of some prelude terms. As part of the language prelude, here is a sampling of typical Rosetta terms.

- $E[\![\xi]\!](envt) \equiv (envt\_value(\xi))$

- $E[\![v]\!](envt) \equiv V[\![v]\!]$
- $E[\![\varepsilon\Omega\varepsilon']\!](envt) \equiv O[\![\Omega]\!]\langle E[\![\varepsilon]\!](envt), E[\![\varepsilon']\!](envt)\rangle$
- $O[\![=]\!] \equiv \lambda\langle v1, v2\rangle.if\ v1 = v2\ then\ True\ else\ False$

where

- ξ corresponds to identifiers
- *envt* is the environment
- v corresponds to constant values
- ε is an expression
- Ω is a binary operator
- $\lambda\langle parameters\rangle.body$ is a lambda expression
- $(envt\_value(\xi))$ means the value of ξ in the environment

While this does not explicitly denote every possible syntactical term in Rosetta, these prelude terms give a general basis for denoting terms. The remaining terms would be denoted in the same fashion, with these three valuation functions.

3.2 Component Syntactic Denotation

In preious Rosetta work, a component was denoted as three facets – one for assumptions, one for definitions, and one one for implications. Since then, the component has become the basic building block, while a facet is now a special case of a component with no assumptions or implications. Therefore, the denotation of components has not previously been addressed. The observers and domain details of components and facets are identical, so the denotations only differ in that the bodies of the component denotation must also include the assumptions and implications.

Recall the general syntactic parts of a typical component c are

```
component c (#parameters#)::#domain# is
  #variables#
begin
   assumptions
     #assumptions#
   end assumptions;
   definitions
     #terms#
   end definitions;
   implications
     #implications#
   end implications;
end c;
```

Again, we will consider a component a 3-tuple. $< O, Dom, Bodies >$ and where

- O contains the observers (parameters and variables, including all domain variables) of c. So, $O = (O_1, O_2, ..., O_n)$ where each O_i is an observer of c.
- Dom is the domain of the components.
- $Bodies = [(A_1, D_1, I_1), (A_2, D_2, I_2), ..., (A_p, D_p, I_p)]$, a list of triples containing each of the following:

- Each A_i is the set of denotations of the assumptions in the ith triple in c
- Each D_i is the set of denotations of the definitions in the ith triple in c
- Each I_i is the set of denotations of the implications in the ith triple in c

So rather than just sets of terms, there are now sets of triples containing all assumptions, definitions, and implications. A consistent component is one in which every term within one set of (assumptions, definitions, implications) in the component is consistent. Note that the majority of components will only have one set of assumptions, definitions, and implications. All assumptions, definitions, and implications are denoted with the same valuation functions previously described for facets.

3.3 Abstract Syntax as Coalgebra

The coalgebraic structure of facets is previously described [Kong et al., 2003], but lacked a denotation for the general case. These techniques were used to denote the coalgebraic structure of specific facets, but there was no general denotation to apply to any facet. Thus, this work generalizes what was previously done.

The behavior, or abstract state, of a component or facet is defined by its observers. This means there is no distinction between the coalgebra denoted by a component and a facet. Consider the abstract state, S, of the facet f. S is defined by all possible observations of f, meaning S is the same as $O^{\mathbb{N}}$. We will see that S is the coalgebra denoted by f. The behavior of a facet is what we observe of that facet over transitions. So, the system can be thought of as all possible observations. Given a transition function, ξ, we take a step, which results in the observations from that transition as well as the rest of the system behavior. So we can define the structure of f, $O^{\mathbb{N}} \xrightarrow{\xi} O \times O^{\mathbb{N}}$, where we describe the behavior of the facet as a sequence of observations of the facet.

We can then give the commuting diagram.

$$\begin{tikzcd}
S \arrow[r, "Beh"] \arrow[d, "\xi"'] & O^{\mathbb{N}} \arrow[r, "map(f)"] \arrow[d, "obs\_fnc"'] & O_D^{\mathbb{N}} \arrow[d, "\langle obs\_domain, rest \rangle"'] \\
O \times S \arrow[r, "id \times Beh"] & O \times O^{\mathbb{N}} \arrow[r, "f \times map(f)"] & O_D \times O_D^{\mathbb{N}}
\end{tikzcd}$$

where the objects in the diagram are

- S : The abstract state of the system corresponding to f's behavior
- $O^{\mathbb{N}}$: All possible behaviors/observers of each state of the system
- $O_D^{\mathbb{N}}$: The observations of the system state abstracted to f's domain
- $O \times S$: The observations on state transitions
- $O \times O^{\mathbb{N}}$: The behavior of the system, including the behavior of the observation on a transition as well as the behaviors of the possible next states
- $O_D \times O_D^{\mathbb{N}}$: The observations of system states, including the observation on a transition, as well as the observations of possible next states, abstracted to f's domain

the arrows are

- Beh : Gives the behavior of the system states in terms of the observation of each state
- map(f) : The function f gives the abstraction from an observable behavior in f of a system state to the observation of the domain variables. Mapping f over the observable behaviors then gives the observations of the domain variables for each observable state behavior of f.

- ξ : The transition function
- obs_fnc : A function to give the observation from taking one step; this includes the observation of the current state as well as the possible next states (the rest of the behavior)
- $\langle obs\_domain, rest \rangle$: Abstracted to the domain level, *obs_domain* describes the function to take an observation, and *rest* is the function that gives all possible future observations of the system (for the state_based domain, this is $< curStateD, possNextD >$)
- $id \times Beh$: id is the identity function, *Beh*, as described above
- $f \times map(f)$: f and map(f) as described above

The commuting diagram shows that the facets are described via their observations over transitions. Furthermore, we can abstract these observations to observations within their domains, or their domain coalgebra. Every facet with a given domain observes the variables defined by that domain. When we abstract a facet's coalgebra to its domain coalgebra, we get the behavior from observing only the domain variables. Essentially, we start with the abstract state of the facet. We abstract to get the behavior of the facet, or facet coalgebra. We can abstract once more to get the behavior from only observing domain variables. The commuting diagram shows that domain coalgebras are final, meaning they are complete in the sense that any facet in that domain can be uniquely mapped/abstracted to that domain. Essentially, this reiterates that every facet with a given domain extends that domain, and therefore observes the variables of that domain.

3.4 Example Facet Denotation

Consider the *counter* facet as an example denoted to a coalgebra as described above.

```
facet counter( en    ::  input bit;
               clk   ::  input bit;
               rst   ::  input bit;
               out   ::  output word(3)
               ) :: state_based is

  internal :: word(3);

begin

t1:   if rst=1 then internal' = "000"
      elseif (en=1 and clk=1 and clk'event) then
            internal' = case internal is
                             b"000" -> b"001" |
                             b"001" -> b"010" |
                             b"010" -> b"011" |
                             b"011" -> b"100" |
                             b"100" -> b"101" |
                             b"101" -> b"110" |
                             b"110" -> b"111" |
                             b"111" -> b"000"
                        end case;
      else internal'=internal;
      end if;
t2:   out' = internal';
end facet counter;
```

Figure 3.1.: Example Facet Denotation

Let *S* be the set of states, or all the observations, of the *counter* facet. The system defined by *counter* is a stream from *S* to *S*, exhibiting observations. The observations are the 3 input bits, the output 3-bit bit vector (word(3)), and the internal 3-bit bit vector of the facet, as well as the the set of states used in the specification of counter.

The coalgebraic structure shows the observation from taking one transition with the rest of the possible transitions, and follows in Figure 3.2.

$$\boxed{\begin{array}{l}(States \times bit \times bit \times bit \times word(3)\ \times word(3))^{\mathbb{N}} \xrightarrow{\xi} \\ \qquad (States \times bit \times bit \times bit \times word(3) \times word(3)) \times \\ \qquad\quad (States \times bit \times bit \times bit \times word(3) \times word(3))^{\mathbb{N}}\end{array}}$$

Figure 3.2.: Coalgebraic Structure of *counter* Facet

We can then give the commuting diagram. For ease of reading, let $Obs = (States \times bit \times bit \times bit \times word(3) \times word(3))$.

$$\begin{tikzcd}
X \arrow[r, "Beh"] \arrow[d, "\xi"'] & Obs^{\mathbb{N}} \arrow[r, "map(f)"] \arrow[d, "obs\_fnc"'] & States^{\mathbb{N}} \arrow[d, "\langle curStateD, possNextD \rangle"'] \\
\langle Obs \rangle \times X \arrow[r, "id \times Beh"] & Obs \times Obs^{\mathbb{N}} \arrow[r, "f \times map(f)"] & States \times States^{\mathbb{N}}
\end{tikzcd}$$

Where the objects in the diagram are

- X : The set of states of the system
- $Obs^{\mathbb{N}} = (States \times bit \times bit \times bit \times word(3) \times word(3))^{\mathbb{N}}$: The behaviors of each state of the system
- $States^{\mathbb{N}}$: The observations of system states abstracted to the state_based domain
- $\langle Obs \rangle \times X = \langle States \times bit \times bit \times bit \times word(3) \times word(3) \rangle \times X$: The observations on transitions
- $Obs \times Obs^{\mathbb{N}} = (States \times bit \times bit \times bit \times word(3) \times word(3)) \times (States \times bit \times bit \times bit \times word(3) \times word(3))^{\mathbb{N}}$: The behavior of the system, including the

behavior of the observation on a transition as well as the behaviors of the possible next states

- $States \times States^{\mathbb{N}}$: The observations of system states, including the observation on a transition as well as the observations of possible next states, abstracted to the state_based domain

And the arrows are

- Beh : Gives the behavior of the system states in terms of the observation of each state.
- map(f) : The function f gives the abstraction from an observable behavior in add_beh of a system state to the observation of the domain variables. Mapping f over the observable behaviors then gives the observations of the domain variables for each observable state behavior of add_beh.
- ξ : The transition function
- obs_fnc : A function to give the observation from taking one step; this includes the observation of the current state as well as the possible next states (the rest of the behavior)
- $\langle curStateD, possNextD \rangle$: Like obs_fnc, but for the state_based domain
- $id \times Beh$: id is the identity function, Beh as described above
- $f \times map(f)$: f and map(f) as described above

The denotation of each term in counter is as follows:

t1	$T[\![if\ rst = 1$ $\quad then\ internal' = ”000”$ $\quad elseif\ (en = 1\ and\ clk = 1\ and\ clk'event)$ $\quad then\ internal' = case\ internal\ is$ $\qquad\qquad b”000” -> b”001”\mid$ $\qquad\qquad b”001” -> b”010”\mid$ $\qquad\qquad b”010” -> b”011”\mid$ $\qquad\qquad b”011” -> b”100”\mid$ $\qquad\qquad b”100” -> b”101”\mid$ $\qquad\qquad b”101” -> b”110”\mid$ $\qquad\qquad b”110” -> b”111”\mid$ $\qquad\qquad b”111” -> b”000”$ $\qquad\quad end\ case;$ $\quad else\ internal' = internal;$ $\quad end\ if;]\!]\ envt$ $\equiv if\ ((envt\_value(rst)) = 1)$ $\quad then\ (envt\_value(internal))((envt\_value(next))$ $\qquad\qquad (envt\_value(\alpha))) = b”000”$ $\quad elseif\ ((envt\_value(en) = 1\ and\ envt\_value(clk) = 1)$ $\qquad\qquad and\ ((envt\_value(clk))(envt\_value(event))))$ $\quad then\ (envt\_value(internal))((envt\_value(next))$ $\qquad\qquad (envt\_value(\alpha))) =$ $\qquad case(envt\_value(internal))is$ $\qquad\qquad b”000” -> b”001”\mid$ $\qquad\qquad b”001” -> b”010”\mid$ $\qquad\qquad b”010” -> b”011”\mid$ $\qquad\qquad b”011” -> b”100”\mid$ $\qquad\qquad b”100” -> b”101”\mid$ $\qquad\qquad b”101” -> b”110”\mid$ $\qquad\qquad b”110” -> b”111”\mid$ $\qquad\qquad b”111” -> b”000”$ $\qquad end\ case;$ $\quad else\ (envt\_value(internal))((envt\_value(next))(envt\_value(\alpha)))$ $\qquad\qquad = envt\_value(internal)$
simplified t1	$if\ (rst = 1)$ $then\ internal(next(\alpha)) = b”000”$ $elseif\ ((en = 1\ and\ clk = 1)\ and\ clk(event))$ $then\ internal(next(\alpha)) =$ $\quad case\ internal\ is$ $\qquad b”000” -> b”001”\mid$ $\qquad b”001” -> b”010”\mid$

	$b"010" -> b"011"\|$ $b"011" -> b"100"\|$ $b"100" -> b"101"\|$ $b"101" -> b"110"\|$ $b"110" -> b"111"\|$ $b"111" -> b"000"$ $end\ case;$ $else\ internal(next(\alpha)) = internal$
t2	$T[\![out' = internal']\!]envt$ $\equiv if\Big((envt\_value(output))((envt\_value(next))(envt\_value(\alpha))) =$ $(envt\_value(internal))((envt\_value(next))(envt\_value(\alpha)))\Big)$ $then\ True\ else\ False$
simplified t2	$output(next(\alpha)) = internal(next(\alpha))$

So the denotation of *counter* is

$$< (en : input\ bit, clk : input\ bit, rst : input\ bit, out : output\ word(3),$$
$$internal : word(3)),$$
$$State\_based,$$
$$[[simplified t1, simplified t2]]$$
$$>$$

with coalgebraic structure given above in Figure 3.2.

The first part of the denotation gives all observers of *counter*, which includes all parameters and variables of the *counter*. Next is the domain, which is *State_based*. Then is the list of denoted term lists. As with most facets, this facet only has one term body. Therefore, there is only one list of terms. That list contains the denotations for $t1$, and $t2$, shown above. The denotation of *counter* follows the two parts of a facet denotation. First, the observers form the coalgebraic structure of the denotation. This consists of all possible states of *counter*'s behavior, and the observations over the possible transitions. Second, the syntactic pieces of *counter* make up the rest of the denotation. These pieces are the observers, domain, and denotations of all terms within *counter*.

3.5 Component Consistency

Components and facets may have more than one body, but only one body needs to be consistent. Therefore, a consistent component is one in which every term within one set of (assumptions, definitions, implications), corresponding to one body, in the component is consistent. Similarly a consistent facet is one in which every term within at least one set of terms, corresponding to one body, in the facet is consistent. A Boolean term, or assertion, is consistent if it is true. In other words, if no term is false, then false has not been asserted, so the component is consistent. In components, the assumptions and definitions are typically used in the implications, i.e. $(assumptions \wedge definitions \implies implications)$. All instantiated facets within the instantiating facet must themselves be consistent given their formal parameters replaced with the actual parameters. Essentially, consistency is a structurally recursive or inductive concept, in that something is consistent if all of its parts are consistent (the base case being true boolean assertions). So a facet with only assertion terms is consistent if all of its assertions hold true. Once that facet is instantiated in another facet, the instantiating facet is consistent if its assertions hold and if the instantiated facet holds under the instantiation.

Using our *counter* example, an implementation of *counter* is consistent if the assertions $simplified\ t1$ and $simplified\ t2$ hold true. Any facet that instantiates *counter* is only consistent if that instantiation of *counter*, which replaces *counter*'s formal parameters with actual parameters, is consistent. An inconsistent component is invalid, in that we can say nothing about it. There is no basis for reasoning about an inconsistent component.

Chapter 4

Composition Semantics

We've already defined the denotation for components and facets. This section will show how we denote composed components and facets. First, we'll look at sum and product. For each, we give the denotation of the resulting sum or product, respectively. We explain how the result is constructed, as well as the validity of the domain of the result. We also describe how the result is still a valid component/facet, and exhibits the desired behavior appropriate to sum and product. Each section gives an example to illustrate the construction and denotation of the sum or product. We then move on to instantiation and inclusion, noting the subtle distinction between the two. We give an example of each, followed by each denotation. We end the chapter with a discussion of homomorphisms. While not a true composition operator, it is an important factor in relating and reasoning about multiple components and facets.

4.1 Preliminaries

Product and sum are binary operators. As such, we will define two components, $c1$ and $c2$ as operands in future discussion. Say $c1$ and $c2$ are components denoted as

$< O_1, Dom_1, Bodies_1 >$ and $< O_2, Dom_2, Bodies_2 >$, where

- O_1 and O_2 are the observers (parameters and variables, including all domain variables) of c_1 and c_2, respectively. So, $O_1 = (O_{11}, O_{12}, ..., O_{1n})$ and $O_2 = (O_{21}, O_{22}, ..., O_{2m})$, where each O_{1i} is an observer of $c1$ and each O_{2j} is an observer of $c2$.

- Dom_1 and Dom_2 are the domains of the components.

- $Bodies_1 = [(A_{11}, D_{11}, I_{11}), (A_{12}, D_{12}, I_{12}), ..., (A_{1p}, D_{1p}, I_{1p})]$, a list of triples containing each of the following:

  - Each A_{1i} is the set of denotations of the assumptions in the ith triple in $c1$.

  - Each D_{1i} is the set of denotations of the definitions in the ith triple in $c1$.

  - Each I_{1i} is the set of denotations of the implications in the ith triple in $c1$.

- $Bodies_2 = [(A_{21}, D_{21}, I_{21}), (A_{22}, D_{22}, I_{22}), ..., (A_{2q}, D_{2q}, I_{2q})]$, like in $Bodies_1$

Similarly, when needed, we will define two facets, $f1$ and $f2$, as the operands in future facet composition discussion. Say $f1$ and $f2$ are facets denoted as $< O_1, D_1, Terms_1 >$ and $< O_2, D_2, Terms_2 >$ where

- O_1 and O_2 are the observers (parameters and variables, including all domain variables) of $f1$ and $f2$, respectively. So, $O_1 = (O_{11}, O_{12}, ..., O_{1n})$ and $O_2 = (O_{21}, O_{22}, ..., O_{2m})$, where each O_{1i} is an observer of $f1$ and each O_{2j} is an observer of $f2$.

- D_1 and D_2 are the domains of the facets.
- $Terms_1 = [T_{11}, T_{12}, ...T_{1m}]$, where each T_{jk} is a set of terms (like sets of definitions in a component) in $f1$, likewise for $Terms_2 = [T_{21}, T_{22}, ...T_{2n}]$ in $f2$.

4.2 Sum

The sum operator allows for disjunctive composition of components and facets. It gives us the ability to define multiple views of a system, where only one must hold. As described in Section 2.4, the specifier can separate alternative functionalities and sum them to create the entire functionality. This section explains how the sum is constructed. We look at the denotation, explain the validity of the pieces of the denotation and the resulting component or facet, and give a full example.

4.2.1 Component Sum

Using the definitions of $c1$ and $c2$ from above, when we take the sum of two components, we get the following:

Say $c3 = c1 + c2$, then $c3$ is denoted as $< O_1 + +O_2$, $Dom_1 \sqcap Dom_2$, $Bodies_1 + +Bodies_2 >$ where

- $O_1 + +O_2 = (O_{11}, O_{12}, ..., O_{1n}, O_{21}, O_{22}, ...O_{2m})$. Note that duplicates (i.e. some $O_{1i} = O_{2j}$) are excluded. Also note that it is common that $O_1 \equiv O_2 \equiv O_1 + +O_2$. All parameters and variables of both operands are included in the sum.
- $Dom_1 \sqcap Dom_2$ is the least common domain of Dom_1 and Dom_2. It is also common for $Dom_1 \equiv Dom_2 \equiv Dom_1 \sqcap Dom_2$. The details of the least common domain are described in Section 4.2.1.1

- $Bodies_1 ++ Bodies_2$ is a simple append yielding $[(A_{11}, D_{11}, I_{11}), (A_{12}, D_{12}, I_{12}), ..., (A_{1p}, D_{1p}, I_{1p}), (A_{21}, D_{21}, I_{21}), (A_{22}, D_{22}, I_{22}), ..., (A_{2q}, D_{2q}, I_{2q})]$. The details of this append are in Section 4.2.1.2

4.2.1.1 Exploring the Least Common Domain

Rosetta domains and the transformations between domains form the Rosetta *domain lattice* [Lohoefener, 2011]. As domains are extended (down the lattice), constraints are added. Since the domains form a lattice, any two domains on the lattice have a *least common domain*, or *meet*, above them in the hierarchy. So if $c1$ has domain $Dom1$ and $c2$ has domain $Dom2$ as described above, when we disjoin them to produce $c3 = c1 + c2$, we can safely say that $c3$ has the domain $Dom3 = Dom1 \sqcap Dom2$.

Furthermore, because any domain lower in the lattice is more constrained, all definitions in $c1$ meet all the constraints of $Dom3$ and likewise, the definitions of $c2$ meet all constraints of $Dom3$. So no work is necessary to transform the terms in either component of the sum into the new domain – they are already in that domain.

It should be noted that the specifier can safely explicitly abstract (moving up the lattice) or concretize (moving down the lattice) a component into a desired domain prior to taking the sum to control the domain of the resulting component. For instance, the designer may want the specificity provided by a more constrained domain. They may safely transform one of the components into the domain of the other component prior to taking their sum to gain that specificity.

4.2.1.2 Exploring Bodies1++Bodies2

We can think of sum as a disjunction of components. We have two components that we sum together, and we know the result is either the first part of the sum or the second part of the sum. We need a way of separating the disjoint parts within a summed facet. Essentially, we need a way of separating entire sets of assumptions, definitions, and implications, and enforcing that only one of those sets needs to be consistent. This need prompted the addition of multiple bodies within a component, which was not previously supported in Rosetta, and is the reason that only one body needs to be consistent for the component to be consistent.

Say you know your system will behave in one of two ways. With the addition of multiple bodies, you have two choices for specifying this system. You could explicitly write a facet with two bodies – one for each behavior.

```
facet system(#parameters#)::#domain#
begin
  #body describing behavior 1#

begin
  #body describing behavior 2#

end system;
```

Alternatively, you could define two components, *behavior*1 and *behavior*2. This way you are able to separate the assumptions, definitions, and assumptions of these two behaviors into their own components. When *behavior*1 and *behavior*2 are summed to describe the entire system, their assumptions, definitions, and assumptions need to be reflected in the entire system, though, in a way that allows for the situation that only one behavior at a time need be enforced. To that end, we have chosen to append each set of (assumptions,definitions,implications) onto the list of possible behaviors. One or more of these triples needs to hold in a consistent component. Appending these (assumptions,definitions, implications) triples gives us exactly the notion of disjunction we need. Note that it is not always possible to know which body (or bodies) is “active” within a specification. There is no notion of tagging that would identify which set of (assumptions,definitions,implications) is consistent at that time. This is intentional, as some specifications may not be constrained enough to determine which option is currently “active.” We opted for a strategy that allows for expressivity and under-constrained specifications.

4.2.2 Facet Sum

Facets are simply components without sets of assumptions or implications. So the facet sum is a simplified version of the component sum.

We’ll use the previously defined facets, $f1$ and $f2$ as the operands of the sum. Say $f3 = f1 + f2$. Then it is denoted as $< O_1 ++ O_2, D_1 \sqcap D_2, Terms_1 ++ Terms_2) >$ where

- $O_1 ++ O_2 = (O_{11}, O_{12}, ..., O_{1n}, O_{21}, O_{22}, ...O_{2m})$ as in component sum. Note that duplicates (i.e. some $O_{1i} = O_{2j}$) are excluded. Also note that it is common that $O_1 \equiv O_2 \equiv O_1 ++ O_2$. All observers from each operand are included in the result.

- $D_1 \sqcap D_2$ is the least common domain of D_1 and D_2 as in component sum. It is also common for $D_1 \equiv D_2 \equiv D_1 \sqcap D_2$ Details of the least common domain are described in Section 4.2.1.1.

- $Terms_1 ++ Terms_2$ would be $[T_{11}, T_{12}, ..., T_{1m}, T_{21}, T_{22}, ..., T_{2n}]$, where each T_{ij} is a set of term. This is similar to the component sum in Section 4.2.1.2, except since there are no assumptions or implications, there are only sets of terms (definitions) to append.

4.2.2.1 Example

Recall our *processor* example from Figure 2.10 in Section 2.4. Instructions are fetched and decoded, and based on the decoding, different operations are executed. We modularly define the execution for each possible instruction. The processor is defined as the sum of each instruction execution.

Note, this is an example where the different parts of the sum are in fact mutually exclusive. The first line of each facet asserts that the *op* is equal to that particular instruction. That assertion will hold in only one of the facets. For instance, if the *op* is decoded as the $plusOp$, then the assertion $op = plusOp$ will hold in the facet $plus$, but in any other facets those assertions will fail. In the jmp facet, the assertion $op = jmpOp$ will hold, etc. The facet *processorBeh* is consistent if any of the facets in the sum are satisfied.

The denotation of this facet is as follows:

```
< (registers::array(16,word),pc::word,instruction::word is memFetch(pc),
            op::nibble is decodeOp(instruction)),
  processor,
  [[ T ⟦ op= plusOp; ⟧ ,
              T ⟦ registers'=replace(registers,dest,registers[src1]+registers[src2]);⟧,
              T ⟦ pc' = pc + x"0002";⟧],
  ... , [ ... ] , ...
  [ T ⟦ op= jumpOp;⟧,
             T ⟦ registers'=registers;⟧,
             T ⟦ pc' = newPC(instruction);⟧]]
>
```

All observers are combined. The domain for all of the operands was *processor*, so the domain of the result is still *processor*. The list of all body terms in the result contains separate lists of terms from each operand – the lists of terms are all appended in the result.

4.2.2.2 Sharing

In sum, sharing clauses give us a similar power as domain definitions. In domain definitions, we add constraints based on our knowledge of that domain. For example, in *state_based*, we add the constraints of having a current state and next state. When we do a facet sum, we have the potential of losing domain information since we may have to go up the domain lattice to find the least common domain of the summed facets. However, we may have knowledge of certain constraints that should still be part of each of those facets. We can add that information to the sharing clause of the conjunction to enforce those constraints. Note that with facet sum, nothing is automatically shared to avoid name capture issues. Anything that should be shared must be explicitly added to the sharing clause.

4.2.2.3 The Resulting Component/Facet

It should be noted that the result of summing two components (or facets) is a valid component. A valid component/facet would contain valid observers, a valid domain, and a list of valid bodies/terms. We have appended all observers from each operand to form the observers of the sum. Since those operand observers were all the parameters and variables of the operands, the appending of the observers gives us valid parameters and variables for the sum. We've already discussed that the domain of the new facet exists and is valid. Since we have appended sets of bodies/terms from the operands together for the sum, we get a list of valid sets of bodies/terms. These are the three parts of a component's (or facet's) denotation.

Also, since the structure of the component coalgebra is defined over the observers, we still have a valid coalgebra as the observers,$(O_1 + +0_2)$, are valid. Therefore, the behavior of the summed component still denotes a coalgebra. The structure of this coalgebra is

$(O_1 + +0_2)^N \xrightarrow{\xi} (O_1 + +O_2) \times (O_1 + +O_2)^N$

4.3 Product

The product operator allows for conjunctive composition of components and facets. It gives us the ability to define multiple views of a system, where all views must hold. As described in Section 2.3, the specifier can separate concurrent requirements, often in different domain vocabularies, and take the product to address all requirements. This section explains how the product is constructed. We look at the denotation, explain the validity of the pieces of the denotation and the resulting component or facet and give an example.

4.3.1 Component Product

Using the definitions of $c1$ and $c2$ from above, when we take the product of two components, we get the following:

Say $c3 = c1 * c2$, then $c3$ is denoted as $< O_1 + +O_2, Dom_1 \sqcap Dom_2, Bodies_1 * *Bodies_2 >$ where

- $O_1 + +O_2 = (O_{11}, O_{12}, ..., O_{1n}, O_{21}, O_{22}, ...O_{2m})$. Note that duplicates (i.e. some $O_{1i} = O_{2j}$) are excluded.
- $Dom_1 \sqcap Dom_2$ is the least common domain of Dom_1 and Dom_2, as discussed

in Section 4.3.1.1.

- $Bodies_1 ** Bodies_2$ is essentially the cross product of $Bodies_1$ and $Bodies_2$, however there are some intricacies in combining shared items. In the case of no shared items,

  $Bodies_1 ** Bodies_2 =$

  $[(A_{11} ++ A_{21}, D_{11} ++ D_{21}, I_{11} ++ I_{21}), ..., (A_{1p} ++ A_{21}, D_{1p} ++ D_{21}, I_{1p} ++ I_{21}),$

  $(A_{11} ++ A_{22}, D_{11} ++ D_{22}, I_{11} ++ I_{22}), ..., (A_{1p} ++ A_{22}, D_{1p} ++ D_{22}, I_{1p} ++ I_{22}),$

  $...,$

  $(A_{11} ++ A_{2q}, D_{11} ++ D_{2q}, I_{11} ++ I_{2q}), ..., (A_{1p} ++ A_{2q}, D_{1p} ++ D_{2q}, I_{1p} ++ I_{2q})]$

  The rest of the details of $**$ are addressed in Sections 4.3.1.2 and 4.3.2.

4.3.1.1 Exploring the Least Common Domain

For the same reasons as explained for component sum in Section 4.2.1.1, any two components have a least common domain and it is safe to use this domain as the domain for the product, with no additional work necessary as all terms in the product will be in the least common domain.

4.3.1.2 Exploring Terms1**Terms2

Regardless of shared items in the definitions sections, there is no sharing in the assumptions or implications. Therefore, we always use the simple append operator for these sets, as done above. Within the sets of definitions, we *can* have sharing. Since there is no sharing in the assumptions or implications, we will describe the details of sharing within the confines of the simpler case of a facet product in Section 4.3.2, as facets have no assumptions or implications. The same notion is applied to the appending of definitions in the case of component product.

4.3.2 Facet Product

Facets are simply components without sets of assumptions or implications. So the facet product is a simplified version of the component product.

We'll use the previously defined facets, $f1$ and $f2$, as the operands of the product. Say $f3 = f1 * f2$. Then $f3$ is denoted as $< O_1 + +O_2, D_1 \sqcap D_2, Terms_1 + +Terms_2) >$, where

- $O_1 + +O_2 = (O_{11}, O_{12}, ..., O_{1n}, O_{21}, O_{22}, ...O_{2m})$. Note that duplicates are excluded (e.g. O_{2j} is excluded if some $O_{1i} = O_{2j}$).

- $D_1 \sqcap D_2$ is the least common domain of D_1 and D_2. We discuss the details and validity of the least common domain in Section 4.3.1.1.

- $Terms_1 * *Terms_2 =$

  $[T_{11} + +T_{21}, T_{12} + +T_{21}, ..., T_{1m} + +T_{21},$

  $T_{11} + +T_{22}, T_{12} + +T_{22}, ..., T_{1m} + +T_{22},$

  $...,$

  $T_{11} + +T_{2n}, T_{12} + +T_{2n}, ..., T_{1m} + +T_{2n}]$

  if there is no sharing. However, for any shared items, their definitions must be conjoined.

If there are no shared definitions in $Terms_1$ and $Terms_2$, then $Terms_1 * *Terms_2$ is the cross product of all of the sets of terms (each T_{jk}) in $Terms_1$ and $Terms_2$, where all of the terms in each part of the cross product are appended, as done above. We are constraining $f3$ with all of the constraints of $f1$ and all of the constraints of $f2$. Often, these definitions are constraining the same item. Any terms that have the same labels are considered shared items. Consider one cross product, say $T_{ab} ++ T_{cd}$ within $T_1 ** T_2$. If T_{ab} and T_{cd} each contain a term with label *sharedItem*, then those items are conjoined into one item within $T_{ab} ++ T_{cd}$. Let's look at some examples of the kinds of shared items encountered.

Rosetta terms are either boolean assertions or are instantiated facets. When we conjoin facets, we must essentially conjoin their terms. The conjunction of two boolean assertions $a1$ and $a2$ would then instinctively be $a1$ *and* $a2$. For conformity, we can use the $*$ operator in Rosetta, which subsumes *and*. Consider two simple facets $g1$ and $g2$ that both have an item *sum*, where $g1$ defines

sum: z'=input+z;

and g2 defines

sum: power'=power+loss;

Note that these definitions of *sum* are boolean assertions. Then $g3 = g1 * g2$ would contain the item with the boolean assertion:

sum: (z'=input+z) * (power'=power+loss);

The conjunction of two facet instantiations is done using facet product. Shared facet instantiations can be seen in the following example of the denotation of a facet product. Note that it is invalid to have a shared item where one is an assertion and the other is a facet instantiation.

4.3.2.1 Example of facet product

Recall the QAM example from Figure 2.6 in Section 2.3. We defined the behavior in *qamAESArch*, and the implementation details in *structure*1 and *structure*2. We then composed the behavior and structure to get fully constrained implementation details in *implementation*1, and *implementation*2.

The denotations for *qamAESArch* is

```
< (i::input word(2),o::output real, f::input frequencyType; length::design keyLengthType,
                    k::input word(length), ho::bit,aesi::word(16), mi::word(2)),
   static,
   [(⟦code: = huffEncoder(i,ho);⟧,
                    ⟦buff1: buffer(ho,aesi);⟧,
                    ⟦enc: aesEncryptor(aesi,aeso,length,k);⟧,
                    ⟦buff2: buffer(aeso,mi);⟧,
                    ⟦modulate: qamModulator(mi,o,f);⟧)]
>
```

and the denotation of *structure* is

```
< ()
   fabric,
   [(⟦code:= hardware(fpga);⟧,
     ⟦buff1:hardware(fpga);⟧,
     ⟦enc: hardware(crypto);⟧,
     ⟦buff2: ,hardware(fpga);⟧,
     ⟦modulate: hardware(fpga);⟧)]
>
```

Note each of these facets have only one set of terms. We are taking the cross product of two sets with cardinality one, which yields a set with cardinality one. The shared items in our QAM example are *code*, *buff*1, *enc*, *buff*2, and *modulate*. These terms are facet instantiations. Conjoining these terms involves taking their facet products. So, for the *code* item, we have the new *code* item $code:\ huffEnconder(i,ho) * hardware(fpga)$;. The denotation of $implementation1$ is

```
< (i::input word(2), o::output real, f::input frequencyType; length::design keyLengthType,
                        k::input word(length), ho::bit,aesi::word(16), mi::word(2)),
  static,
  [(T⟦code: huffEncoder(i,ho) * hardware(fpga);⟧,
             T⟦buff1: buffer(ho,aesi) * hardware(fpga);⟧,
             T⟦enc: aesEncryptor(aesi,aeso,length,k) * hardware(crypto),⟧;
             T⟦buff2: buffer(aeso,mi) * hardware(fpga);⟧,
             T⟦modulate: qamModulator(mi,o,f) * hardware(fpga);⟧)]
>
```

Assertion terms have type *Boolean*, while instantiations have the type of the instantiated facet. Rosetta does not support heterogeneity with respect to types. Thus, it is considered invalid to take the product of a term that is an assertion and a term that is a facet instantiation.

4.3.2.2 Sharing clause for explicit sharing

The previous example has many shared items, but they are implicitly shared. Sharing clauses explicitly force facets in a facet product to each define every item in the sharing clause. In other words, a valid specification of $f3 = f1 * f2\ sharing\ x1, x2, ...xn$ explicitly forces $f1$ and $f2$ to each define items $x1$, $x2$,..., and xn. As discussed in Section 4.2.2.2, sharing enables the addition of domain-specific constraints.

4.3.2.3 The Resulting Component or Facet

It should be noted that the product of two components (or facets) is a valid component. We have appended all observers, giving the resulting component a valid set of observers. We've already discussed that the domain of the new facet exists and is valid. We have appended the assumptions, implications, and unshared definitions of each cross-product of bodies, which gives valid new assumptions, implications and definitions. Any shared definitions are either boolean assertions that are multiplied to give a valid boolean assertion, or are facet instantiations, where we take the product of these instantiations. Using structural induction, we can assume that we start with the product of any instantiated facets being valid. With that assumption, we can show that for all products of two components, the result is a valid components.

Also, since the structure of the component coalgebra is defined over the observers, we still have a valid coalgebra as the observers,$(O_1 + +0_2)$, are valid. Therefore, the behavior of the product still denotes a coalgebra. The structure of this coalgebra is

$$(O_1 + +0_2)^N \xrightarrow{\xi} (O_1 + +O_2) \times (O_1 + +O_2)^N$$

4.4 Instantiation

We can specify systems structurally or hierarchically using facet instantiation. Facet instantiation happens within a body of another facet, by replacing the formal parameters of the instantiated facet with actual parameters. A facet declaration, say fd, is similar to a class. An instantiation of a facet is a value whose type is the facet it instantiates. If f is an instance of fd, f is a value with type fd.

4.4.1 Instantiation Example

Let's look at an example of facet instantiation. Recall the $halfAdder$ facet from Figure 2.2 in Section 2.1. Also recall from Figure 2.3 in Section 2.2 that we can now structurally define a $fullAdder$ facet that instantiates two $halfAdder$s. We did this by connecting the correct inputs to the correct outputs, i.e. by assigning the correct actual parameters to the formal parameters of each $halfAdder$.

4.4.2 Denotation

The denotations of both the instantiated facets and the the instantiating facet are the same as for any facet. We'll illustrate this using our example.

The denotation of $halfAdder$ is

$< (x : input\ bit, y : input\ bit, s : output\ bit, c : output\ bit),$

$State\_based,$

$[(T[\![s' = x\ xor\ y]\!], (T[\![c' = x\ and\ y]\!])]$

$>$

The denotation of each term in $fullAdder$ goes in its denotation. So, the above denotation of $halfAdder$ will appear twice in the denotation of $fullAdder$, with the formal parameters replaced with the actual parameters. So, the denotation of $fullAdder$ is

$< (x : input\ bit, y : input\ bit, ci : input\ bit, s : output\ bit, co : output\ bit,$

$s1 : bit, c1 : bit, c2 : bit),$

$State\_based,$

$[(ha1 :< (x : input\ bit, y : input\ bit, s1 : output\ bit, c1 : output\ bit),$

$State\_based,$

$[(T[\![s' = x\ xor\ y]\!], T[\![c' = x\ and\ y]\!])]$

$>,$

$ha2 :< (s1 : input\ bit, ci : input\ bit, s : output\ bit, c2 : output\ bit),$

$State\_based,$

$[(T[\![s' = x\ xor\ y]\!], T[\![c' = x\ and\ y]\!])]$

$>,$

$T[\![co = c1\ or\ c2;]\!]$

$)]$

Note that since the instantiation occurs in the body of the instantiating facet, it has no direct bearing on the observers of the instantiating facet. It therefore has no bearing on the coalgebraic structure of the instantiating facet. Rather the denotation of the instantiated facet just becomes part of the denoted terms within the body of the instantiating facet.

4.5 Inclusion

We saw how facet instantiation is done in Section 4.4. The $fullAdder$ example in Figure 2.3 instantiates two $halfAdder$s. Notice that each instantiation is given an item label, namely $ha1$ and $ha2$. These items labels are essentially the facet inclusion. So $ha1$ is a facet inclusion, and $halfAdder(x, y, s1, c1)$ is the facet instantiation. What this does is allow the observable behavior of what is included to be observable by the facet inclusion. For instance, any observable behaviors of the $halfAdder(x, y, s1, c1)$ instantiation are observable by $ha1$. The inclusions $ha1$ and $ha2$ in effect rename the instance allowing for multiple instances of the same facet.

4.5.1 Inclusion Example

Here's a more detailed example of facet inclusion, that will help illustrate the distinction between instantiation and inclusion, as well as what is in scope within the including facet [Ros, 2008].

```
facet pf(x::input integer; y::output integer)::static is
  export power;
  power :: real;
begin
  power = 0.2;
  y = x + 3;
end facet pf;

facet example(x1, x2 :: input integer;
              y1, y2 :: input integer;
              z :: output integer)::static is
  export power;
  power :: real;
begin
  power = f1.power + f2.power + 0.2;
  z = y1 + y2;

  f1: pf(x1, y1);
  f2: pf(x2, y2);
end facet example;
```

Figure 4.1.: Inclusion Example

We've essentially put a box around the pf instantiation and called it $f1$ (and $f2$). Anything observable from pf is now observable in the $f1$ and $f2$ inclusions. Therefore, $f1.power$ is in scope. And while the facet definition pf is in scope in the *example* facet, $pf.power$ is not in scope, because pf is a facet definition, or class, and not a value.

4.5.2 Denotation

As we saw in Section 4.4.2, the labels are left alone in the denotation, whereas the actual facet instantiation is what is denoted. So in our example, we left $ha1$ and $ha2$ alone, and then denoted the two instantiations of $halfAdder$. So, within the terms of the denotation of $fullAdder$ we had

$$
\begin{aligned}
&ha1 :< (x : input\ bit, y : input\ bit, s1 : output\ bit, c1 : output\ bit),\\
&\qquad State\_based,\\
&\qquad [(T[\![s' = x\ xor\ y]\!], T[\![c' = x\ and\ y]\!])]\\
&\quad >
\end{aligned}
$$

In essence, there is no real denotation of the inclusion, but rather, the instantiation of what is included. The effect of the included instantiation was discussed in 4.4.2.

4.6 Homomorphism

A *homomorphism* is a relationship expressed between facets rather than a composition operator, but is worthy of discussion due to the fact that it gives us a means for reasoning about and relating multiple facets. A homomorphism $A => B$ exists between two facets A and B when all properties of B can be derived from A. Facet homomorphism is frequently referred to as implication because the behaviors of B are implied by A. An *isomorphism* exists between two facets A and B when both $A => B$ and $B => A$. Facet isomorphism is frequently referred to as facet equivalence because A and B have the same properties and are indistinguishable.

Both homomorphism and isomorphism are used to define correctness conditions and express inheritance relationships among facets. In an algebraic sense, if A represents a system and B represents a minimal set of properties that system must exhibit, then $A => B$ formally defines a correctness condition on A that would be checked using theorem proving techniques. Similarly, if *B* represents a system and *A* represents a maximal set of properties the system is allowed to exhibit, then $A => B$ formally defines a correctness condition on *B* that would be checked using model checking techniques. In essence, we are able to express both algebraic and coalgebraic correctness conditions.

Homomorphism also defines the partial ordering used to specify the domain lattice. Specifically, for a set to represent a lattice, a partial order on the set, a minimum element, and a maximum element must be defined. For the Rosetta domain lattice, homomorphism is the partial order while the `static` and `bottom` domains represent the minimum and maximum elements respectively.

Chapter 5

A Case Study

Here we will define and denote a complete system using facet composition as a case study on system design using the composition operators and their semantics. Our system, based on KURM [Alexander, 2009], will contain a dual port RAM and a CPU. We will specify both the functional/behavioral design as well as implementation details, and use conjunction to combine these different design aspects to fully specify our entire system. This example will use instantiation and inclusion, product and sum. The functional design of our system is shown in Figure 5.1, with all descriptions of signals in Table 5.1.

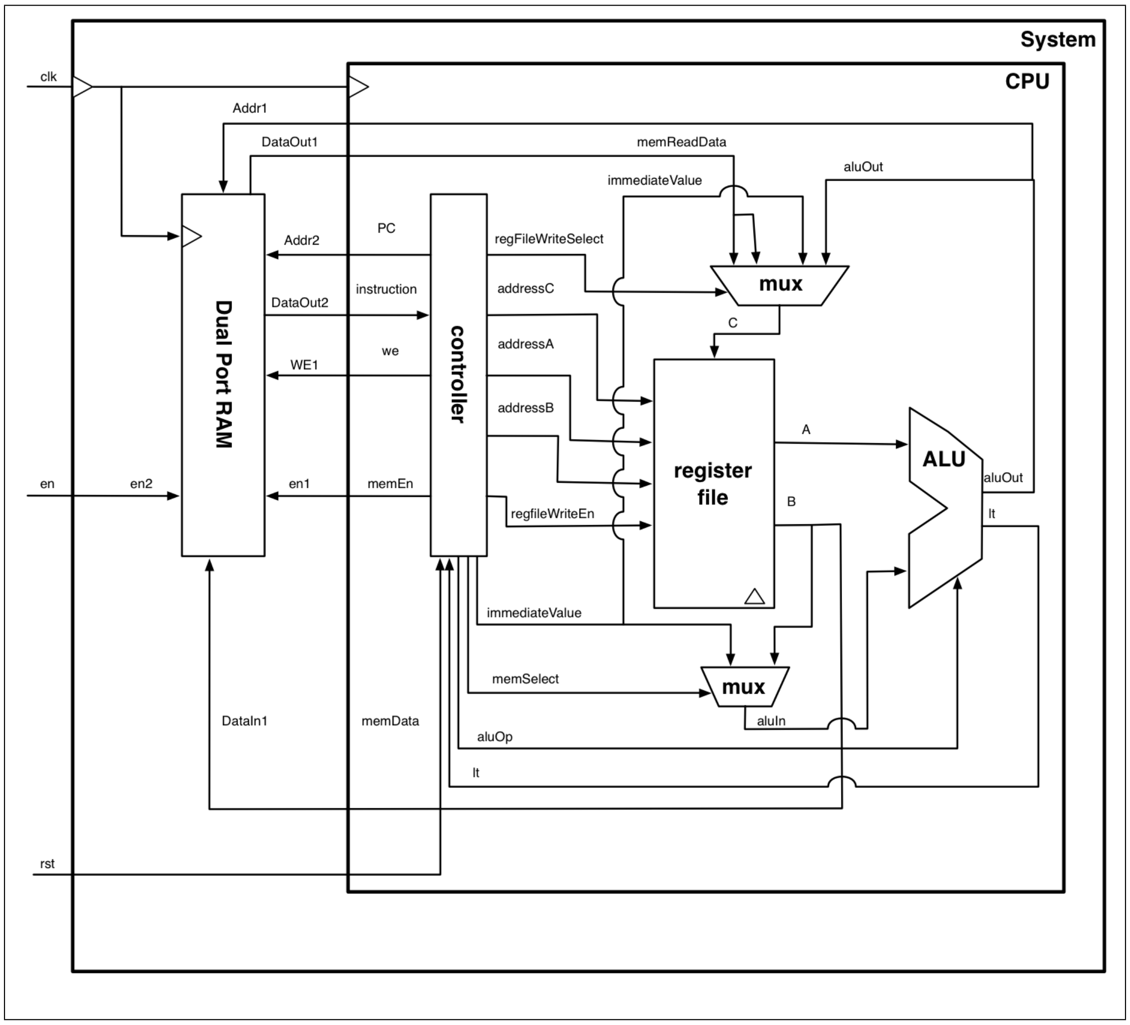

Figure 5.1.: Functional system design

Table 5.1. Description of signals in System Design

Signal Name	Description
clk	System clock
en	System enable
rst	System reset
Addr1	Address to read and write to data memory
DataOut1	Data from memory, used for LW from memory
Addr2	Address to read instructions from memory (Program Counter)
DataOut2	Instructions read from memory
WE1	Memory write enable, used for SW to memory
en1	Enable data memory, active for LW or SW
DataIn1	Data to write to memory, used for SW to memory
en2	Memory enable for Instructions–tied to system enable
memReadData	Data from memory for SW, tied to DataOut1 of memory
PC	Program Counter, tied to Addr2 of memory
instruction	Instruction fetched from memory, tied to DataOut2 of memory
we	Enable write to data memory, tied to WE1 of memory
memEn	Enable data memory, tied to en1 of memory
memData	Data to write to memory, tied to DataIn1 of memory
regFileWriteSelect	Selects between data from memory, immediate data, or result from ALU to write back to register file, depending on instruction type
addressC	Register to write to in register file
addressA	First register to read from in register file
addressB	Second register to read from in register file
regFileWriteEn	Enable write back to register file
immediateValue	Value to be used for immediate and memory instructions
memSelect	Selects between immediate value or register value for ALU, input, depending on instruction type
aluOp	Operation for ALU to perform
lt	Status reflecting when ALU's first input is less than second input
C	Value to write to register file
A	First register read from register file, and first input to ALU
B	Second register read from register file
aluIn	Second input to ALU
aluOut	Result from ALU calculation

5.1 Controller Functional Design

The controller lends itself well to being modularly designed using facet sum. We can write a facet for each instruction type plus one for when there is a reset, and disjoin them to create the entire controller.

Table 5.2. Instruction Set for CPU

Instruction	Meaning	Op	Source1	Source2	Source3
Add R_s, R_t, R_d	$R_d{}' = R_s + R_t$	0000	0-15	0-15	0-15
Sub R_s, R_t, R_d	$R_d{}' = R_s - R_t$	0001	0-15	0-15	0-15
And R_s, R_t, R_d	$R_d{}' = R_s \wedge R_t$	0010	0-15	0-15	0-15
Or R_s, R_t, R_d	$R_d{}' = R_s \vee R_t$	0011	0-15	0-15	0-15
LW R_s, R_t, off	$R_t{}' = M(R_s + off)$	0100	0-15	0-15	off
LI $R_s, immed$	$R_s{}' = extend(immed)$	0101	0-15	$immed_{7-4}$	$immed_{3-0}$
SW R_s, R_t, off	$M(R_s + off)' = R_t$	0110	0-15	0-15	off
BLT R_s, R_t, off	$if(R_s < R_t)$ $PC' = PC + off$	0111	0-15	0-15	off
Jmp $addr$	$PC' = extend(addr)$	1000	$addr_{11-8}$	$addr_{7-4}$	$addr_{3-0}$

We'll define the facet definitions for the *Reset*, *AddOp*, *LW*, *SW*, *LI*, *BLT*, and *Jmp* facets. The *SubOp*, *AndOp*, and *OrOp* facets only differ from *AddOp* in their *op*s and the *aluOp*. In this example, these disjoined facets are mutually exclusive. Only one facet will be consistent when there is a reset, and one for when there is no reset and the instruction is, for example, add. Each facet starts with assertions that the inputs to the facets have the values associated with that facet. For the *reset* facet, there is an assertion that $rst = 1$. This is only a consistent facet under the condition of a reset. Similarly, each facet for a particular instruction operator has an assertion that $rst = 0$ and $instruction = \#Op\ for\ that\ instruction\#$. Following the assertions are the assignments that are appropriate for each instruction. So, in the *reset* facet, the facet first asserts that there is a reset, followed by the assertions that the next state of each output signal is assigned to all 0's. In this fashion, we write each facet separately with the assertions and assignments appropriate to the desired functionality. When we sum them, only one must be consistent. For each instruction, there is a facet that will be consistent and give the appropriate assignments for that instruction.

```
facet Reset(rst::input bit;
            we,memEn,memSelect,
                     regFileWriteEn::output bit;
            aluOp,refFileWriteSelect::output word(2);
            addressA,addressB,addressC::output word(4);
            PC,immediateValue::output word(16)
          )::State_based is
begin

    rst=1; //Assertion that there is a reset

    we'=0;
    memEn'=0;
    memSelect'=0;
    regFileWriteEn'=0;
    regFileWriteSelect'=b"00";
    aluOp'= b"00";
    addressA'=x"0";
    addressB'=x"0"
    addressC'=x"0";
    PC'=x"0000";
    memAddr'=x"0000";
    immediateValue'=x"0000";

end facet Reset;
```

Figure 5.2.: Facet for defining behavior for reset

```
facet AddOp(rst::input bit;
          instruction::input word(16);
          we,memEn,memSelect,regFileWriteEn::output bit;
          aluOp,refFileWriteSelect::output word(2);
          addressA,addressB,addressC::output word(4);
          PC,immediateValue::output word(16)
         )::State_based is
begin

    rst=0; //Assertion of no reset
    instruction(15 downto 12)=b"0000";//Assert add op

    we'=0;
    memEn'=0;
    memSelect'=0;
    regFileWriteEn'=1;
    regFileWriteSelect'=b"00";
    aluOp'= b"00";
    addressA'=instruction(11 downto 8);
    addressB'=instruction(7 downto 4);
    addressC'=instruction(3 downto 0);
    PC'=PC+x"0002";
    immediateValue'=x"0000";

end facet AddOp;
```

Figure 5.3.: Facet for defining behavior for Add instruction

```
facet LW(rst::input bit;
         instruction::input word(16);
         we,memEn,memSelect,regFileWriteEn::output bit;
         aluOp,refFileWriteSelect::output word(2);
         addressA,addressB,addressC::output word(4);
         PC,immediateValue::output word(16)
        )::State_based is
begin

    rst=0; //assertion of no reset
    instruction(15 downto 12)=b"0100";//assert LW op

    we'=0;
    memEn'=1;
    memSelect'=1;
    regFileWriteEn'=1;
    regFileWriteSelect'=b"11";
    aluOp'= b"00";
    addressA'=instruction(11 downto 8);
    addressB'=x"0";
    addressC'=instruction(7 downto 4);
    PC'=PC+x"0002";
    immediateValue'=signExtend(instruction(3 downto 0));


end facet LW;
```
Figure 5.4.: Facet for defining behavior for LW instruction

```
facet LI(rst::input bit;
          instruction::input word(16);
          we,memEn,memSelect,regFileWriteEn::output bit;
          aluOp,refFileWriteSelect::output word(2);
          addressA,addressB,addressC::output word(4);
          PC,immediateValue::output word(16)
         )::State_based is
begin

    rst=0; //assertion of no reset
    instruction(15 downto 12)=b"0101"; //assert LI op

    we'=0;
    memEn'=0;
    memSelect'=0;
    regFileWriteEn'=1;
    regFileWriteSelect'=b"01";
    aluOp'= b"00";
    addressA'=x"0";
    addressB'=x"0";
    addressC'=instruction(11 downto 8);
    PC'=PC+x"0002";
    immediateValue'=signExtend(instruction(7 downto 0));

 end facet LI;
```

Figure 5.5.: Facet for defining behavior for LI instruction

```
facet SW(rst::input bit;
          instruction::input word(16);
          we,memEn,memSelect,regFileWriteEn::output bit;
          aluOp,refFileWriteSelect::output word(2);
          addressA,addressB,addressC::output word(4);
          PC,immediateValue::output word(16)
         )::State_based is
begin

    rst=0; //assertion of no reset
    instruction(15 downto 12)=b"0110"; //assert SW op

    we'=1;
    memEn'=1;
    memSelect'=1;
    regFileWriteEn'=0;
    regFileWriteSelect'=b"11";
    aluOp'= b"00";
    addressA'=instruction(7 downto 4);
    addressB'=x"0";
    addressC'=x"0";
    PC'=PC+x"0002";
    immediateValue'=signExtend(instruction(3 downto 0));


end facet SW;
```

Figure 5.6.: Facet for defining behavior for SW instruction

```
facet BLT(rst, lt :: input bit;
          instruction :: input word(16);
          we, memEn, memSelect, regFileWriteEn :: output bit;
          aluOp, refFileWriteSelect :: output word(2);
          addressA, addressB, addressC :: output word(4);
          PC, immediateValue :: output word(16)
         ) :: State_based is
begin

    rst=0; //assertion of no reset
    instruction(15 downto 12)=b"0111"; //assert BLT op

    we'=0;
    memEn'=0;
    memSelect'=0;
    regFileWriteEn'=0;
    regFileWriteSelect'=b"00";
    aluOp'= b"00";
    addressA'=x"0";
    addressB'=x"0";
    addressC'=x"0";
    PC'= if(lt=1)
         then
           PC+instruction(3 downto 0);
         else
           PC+x"0002";
    immediateValue'=x"0000";


end facet BLT;
```

Figure 5.7.: Facet for defining behavior for BLT instruction

```
facet Jmp(rst::input bit;
          instruction::input word(16);
          we,memEn,memSelect,regFileWriteEn::output bit;
          aluOp,refFileWriteSelect::output word(2);
          addressA,addressB,addressC::output word(4);
          PC,immediateValue::output word(16)
         )::State_based is
begin

    rst=0; //assertion of no reset
    instruction(15 downto 12)=b"0000"; //assert of Jmp op

    we'=0;
    memEn'=0;
    memSelect'=0;
    regFileWriteEn'=0;
    regFileWriteSelect'=b''00'';
    aluOp'= b"00";
    addressA'=x"0";
    addressB'=x"0";
    addressC'=x"0";
    PC'=signExtend(instruction(11 downto 0);
    immediateValue'=x"0000";


end facet Jmp;
```

Figure 5.8.: Facet for defining behavior for Jmp instruction

Then summing those, we can create the *controller* facet.

```
facet controller::State_based = Reset + AddOp + SubOp +
                                 AndOp + OrOp + LW + LI +
                                 SW + BLT + Jmp;
```

Figure 5.9.: Facet for defining behavior for entire Controller. Parameters omitted for ease of reading

5.1.1 Denotation

The *controller* facet's coalgebraic structure is

$$(bit, word(16), bit, bit, bit, bit, word(2), word(2), word(4), word(4), word(4),$$
$$word(16), word(16))^N$$
$$\xrightarrow{\xi} (bit, word(16), bit, bit, bit, bit, word(2), word(2), word(4), word(4), word(4),$$
$$word(16), word(16))$$
$$\times (bit, word(16), bit, bit, bit, bit, word(2), word(2), word(4), word(4), word(4),$$
$$word(16), word(16))^N$$

and its denotation is

$< (rst :: input\ bit, lt :: input\ bit,$

$instruction :: input\ word(16),$

$we :: output\ bit, memEn :: output\ bit, memSelect :: output\ bit,$

$regFileWriteEn :: output\ bit,$

$aluOp :: output\ word(2), refFileWriteSelect :: output\ word(2),$

$addressA :: output\ word(4), addressB :: output\ word(4),$

$addressC :: output\ word(4),$

$PC :: output\ word(16), immediateValue :: output\ word(16)),$

$State\_based,$

$[[\#denotations\ of\ all\ terms\ in\ Reset\#],$

$[\#denotations\ of\ all\ terms\ in\ AddOp\#],$

$[\#denotations\ of\ all\ terms\ in\ SubOp\#],$

$[\#denotations\ of\ all\ terms\ in\ AndOp\#],$

$[\#denotations\ of\ all\ terms\ in\ OrOp\#],$

$[\#denotations\ of\ all\ terms\ in\ LW\#],$

$[\#denotations\ of\ all\ terms\ in\ LI\#],$

$[\#denotations\ of\ all\ terms\ in\ SW\#]$,

$[\#denotations\ of\ all\ terms\ in\ BLT\#]$,

$[\#denotations\ of\ all\ terms\ in\ Jmp\#]]$

$>$

As an example, $[\#denotations\ of\ all\ terms\ in\ Reset\#]$ would be

$[rst = 1;$,

$we(next(\alpha)) = 0;$,

$memEn(next(\alpha)) = 0;$,

$memSelect(next(\alpha)) = 0;$,

$regFileWriteEn(next(\alpha)) = 0;$,

$regFileWriteSelect(next(\alpha)) = b$"00";,

$aluOp(next(\alpha)) = b$"0)";,

$addressA(next(\alpha)) = x$"0";,

$addressB(next(\alpha)) = x$"0";,

$addressC(next(\alpha)) = x$"0";,

$PC(next(\alpha)) = x$"0000";,

$memAddr(next(\alpha)) = x$"0000";,

$immediateValue(next(\alpha)) = x$"0000";,

]

Other denotations follow similarly and will not be repeated here.

All observers from every part of this sum, meaning all parameters and variables of each facet, are now part of the observers for the *controller*. The domain of each piece is *State_based*, so the domain of the *controller* is still *State_based*. Because we are taking a sum, the terms from each piece of the sum are kept separate in their own list of terms. The separate lists of terms are appended in the *controller*'s list of term lists. In this way, only one instruction (or reset) is addressed at a time.

5.2 CPU Functional Design

The functional design of the CPU is done structurally. The *controller* facet is defined in section 5.1. The *mux*es, *registerFile*, and *ALU* are all straight-forward, well-known components, so we will assume those facets are already designed. All signal descriptions are defined in Table 5.1.

```
facet cpu(clk,rst::input bit;
            memReadData,instruction::input word(16);
            we, memEn::output bit;
            pc, aluOut,memData:: output word(16)) :: HW is

 memLoadSelect, regFileWriteEn, immediateSelect::bit;
 aluOp,aluStatus::word(2);
 addressA,addressB,addressC::word(4);
 immediateValue,A,B,C,aluIn::word(16);

begin
  c: controlUnit (rst,lt,
                  instruction,
                  we,memEn,memSelect,regFileWriteEn,
                  aluOp,refFileWriteSelect,
                  addressA,addressB,addressC,
                  PC,immediateValue);

  a: alu     (aluOp,A,aluIn,aluStatus,aluOut);

  rf: regFile    (clk,regFileWriteEn,addressA,
                  addressB,addressC,A,B,C);

  mux1: mux (memReadData,AluOut,memLoadSelect,C);

  mux2: mux (B,immediateValue,immediateSelect,aluIn);
end facet cpu;
```

Figure 5.10.: CPU Functional Specification

5.2.1 Denotation

The *cpu* facet's coalgebraic structure is

$$(bit,bit,word(16),word(16),bit,bit,word(16),word(16),word(16)$$

$$bit,bit,bit,word(2),word(2),word(4),word(4),word(16),word(16),$$

$$word(16),word(16),word(16))^N$$

$$\xrightarrow{\xi} (bit,bit,word(16),word(16),bit,bit,word(16),word(16),word(16)$$

$$bit, bit, bit, word(2), word(2), word(4), word(4), word(16), word(16),$$
$$word(16), word(16), word(16))$$
$$\times\ (bit, bit, word(16), word(16), bit, bit, word(16), word(16), word(16)$$
$$bit, bit, bit, word(2), word(2), word(4), word(4), word(16), word(16),$$
$$word(16), word(16), word(16))^N$$

and its denotation is

$$< (clk :: input\ bit, rst :: input\ bit,$$
$$memReadData :: input\ word(16), instruction :: input\ word(16),$$
$$we :: output\ bit, memEn :: output\ bit,$$
$$pc :: output\ word(16), aluOut :: output\ word(16), memData :: output\ word(16),$$
$$memLoadSelect :: bit, regFileWriteEn :: bit, immediateSelect :: bit,$$
$$aluOp :: word(2), aluStatus :: word(2),$$
$$addressA :: word(4), addressB :: word(4), addressC :: word(4),$$
$$immediateValue :: word(16), A :: word(16), B :: word(16),$$
$$C :: word(16), aluIn :: word(16))$$
$$HW,$$
$$[[\#denotation\ of\ controlUnit\ instantiation\#,$$
$$\#denotation\ of\ alu\ instantiation\#, \#denotation\ of\ regFile\ instantiation\#,$$
$$\#denotation\ of\ mux\ instantiation\#, \#denotation\ of\ mux\ instantiation\#]$$
$$]$$
$$>$$

This denotation involves the simple case of instantiations. The observers are the parameters and variables from the *cpu* itself. Its domain is HW. There is one list of terms, which contains the five facet instantiations in *cpu*.

5.3 System Functional Design

The functional design of the system is a matter of instantiating and interconnecting the previously defined CPU and a dual port RAM with write capabilities on one of the ports.

```
facet system(clk,en,rst:input bit) :: HW is

 we1,en1 :: bit;
 dataOut1 ,dataOut2 ,dataIn ,Addr1 ,Addr2 :: word(16);


begin
 processor: cpu(clk ,rst ,
                    DataOut1 ,DataOut2 ,
                    WE1,en1 ,
                    Addr2 ,Addr1 ,
                    DataIn1 );

 mem: memory(clk ,
                 we1,en1 ,Addr1 ,DataOut1 ,DataIn1 ,
                 en ,Addr2 ,DataOut2
                 );

end facet system;
```

Figure 5.11.: System Functional Specification

5.3.1 Denotation

The *system* facet's coalgebraic structure is

$$
\begin{aligned}
&(bit, bit, bit, bit, bit,\\
&\quad word(16), word(16), word(16), word(16), word(16))^N\\
&\xrightarrow{\xi} (bit, bit, word(16), word(16), bit, bit, word(16), word(16), word(16))\\
&\quad \times (bit, bit, word(16), word(16), bit, bit, word(16), word(16), word(16))^N
\end{aligned}
$$

and its denotation is

$$
\begin{aligned}
&<(clk, en, rst :: input\ bit;\\
&\qquad we1, en1 :: bit;\\
&\qquad dataOut1, dataOut2, dataIn, Addr1, Addr2 :: word(16);)\\
&\quad HW,\\
&\quad [[\#denotation\ of\ processor\ instantiation\#,\\
&\qquad \#denotation\ of\ memory\ instantiation\#]\\
&\quad]\\
&>
\end{aligned}
$$

This denotation involves the simple case of instantiations. The observers are the parameters and variables from the *system* itself. Its domain is HW. And there is one list of terms, which contains the two facet instantiations in *system*.

5.4 Power Consumption Constraints

The designer might be interested in specifying implementation details such as power constraints of the system. We can write facets to model these constraints. We can make these as course-grained or fine-grained as desired. For instance, we can have a facet to model the power constraint to correspond with each block in Figure 5.1. Below we give these facet definitions for the power consumption of the ALU, the CPU, the RAM, and the entire system. The muxes, register file, controller, and memory would be done in a similar manner to the ALU. Alternatively, we could have defined a very course-grained model of power consumption at the top level, or any level of detail in between.

```
facet aluPower(rst:input bit;
                wordSize,operandSize,
                      statusSize::input natural;
                calculation,switch,leakage::design real;
                power::output real) ::  is

begin

  power'=   if rst=1
              then leakage
              else wordSize*(switch +
                            operandSize*calculation) +
                   switch*statusSize;
           end if;

end facet aluPower;
```

Figure 5.12.: ALU Power Consumption Specification

```
facet cpuPower(rst:input bit;
                calculation,switch,leakage::design real;
                power::output real) :: state_based is

aluP,mux1P,mux2P,regFileP,controllerP::real;
controllerOutputBits::natural;

begin

  cob: controllerOutputBits = 1*4 + 2*2 + 4*3 + 16*2;

  a: aluPower(rst,16,2,1,calculation,switch,aluP);

  mux1: muxPower(rst,2,16,switch,mux1P);

  mux2: muxPower(rst,4,16,switch,mux2P);

  rf: regFilePower(rst,16,16,switch,leakage,regFileP);

  c: controllerPower(rst,10,16,controllerOutputBits,
                     switch,leakage);

  p: power'=  if rst=1
                then leakage
                else aluP + mux1P + mux2P +
                           regFileP + controllerP;
              end if;

end facet cpuPower;
```
Figure 5.13.: CPU Power Consumption Specification

```
facet systemPower(rst:input bit;
                  calculation,switch,leakage,
                              threshold::design real;
                  power::output real,
                  overThreshold::output boolean)
                    :: state_based is

cpuP,memP,nextPower::real;

begin

  processor: cpuPower(rst,calculation,switch,
                          leakage,cpuP);

 mem: memPower(rst,switch,leakage,memP);

 mp: nextPower = if rst=1
                 then leakage
                 else cpuP + memP;

  ot: overThreshold' = nextPower > threshold;

  p: power'=  nextPower;

end facet systemPower;
```
Figure 5.14.: System Power Consumption Specification

The implementation details of the power-consumption facets model the same general strategy of the behavioral model. The power consumption of the system is defined in a block-diagram fashion using instantiation.

5.5 Entire System: Functional Behavior and Power Consumption

We now have a model for the behavior of our system and for the power consumption of our system. We can conjoin them to create the desired specification of the system. A consistent implementation would be one in which the behavior were consistent and the power consumption were consistent. In this case, given the appropriate design parameters, the power consumption for each clock cycle must remain under the given threshold.

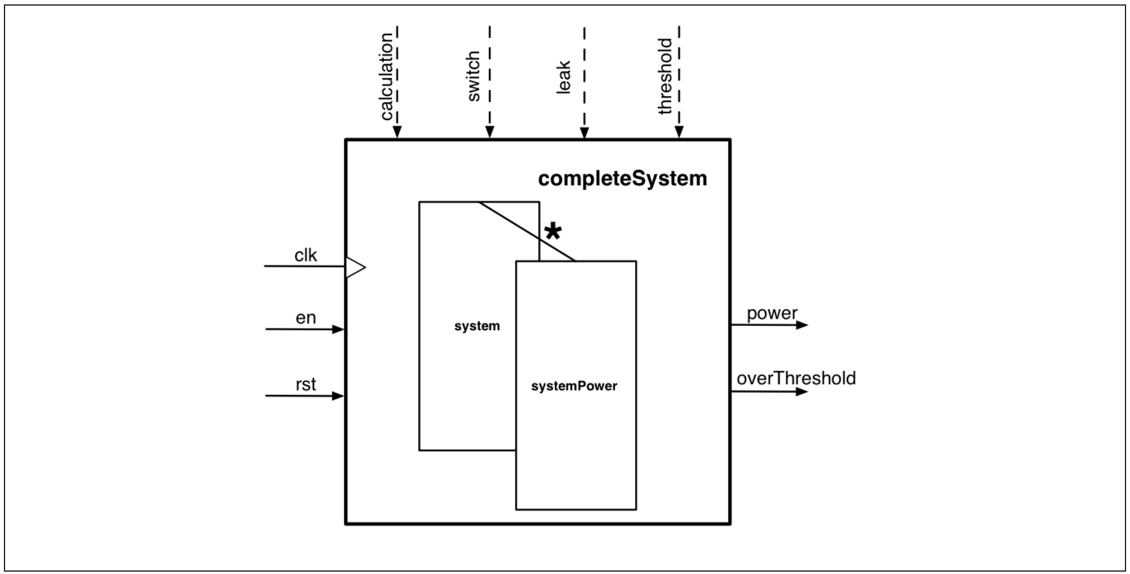

Figure 5.15.: Complete System Design

```
facet completeSystem = system(clk,en,rst) *
                          systemPower(rst,calculation,
                                      switch,leakage,
                                      threshold,power,
                                      overThreshold);
```

Figure 5.16.: Complete System Specification

5.5.1 Denotation

The *completeSystem* facet's coalgebraic structure is

$$(bit, bit, word(16), word(16), bit, bit, word(16), word(16), word(16),$$
$$bit, bit, bit, word(2), word(2), word(4), word(4), word(4),$$
$$word(16), word(16), word(16), word(16), word(16),$$
$$real, real, real, real, real, real, real, real, real)^N$$

$$
\begin{aligned}
\xrightarrow{\xi}\ & (bit, bit, word(16), word(16), bit, bit, word(16), word(16), word(16),\\
& \quad bit, bit, bit, word(2), word(2), word(4), word(4), word(4),\\
& \quad word(16), word(16), word(16), word(16), word(16),\\
& \quad\quad real, real, real, real, real, real, real, real, real)\\
& \times (bit, bit, word(16), word(16), bit, bit, word(16), word(16), word(16),\\
& \quad bit, bit, bit, word(2), word(2), word(4), word(4), word(4),\\
& \quad word(16), word(16), word(16), word(16), word(16),\\
& \quad\quad real, real, real, real, real, real, real, real, real)^N
\end{aligned}
$$

and its denotation is

```
<(clk,en,rst::input bit; we1,en1 :: bit;
      dataOut1,dataOut2,dataIn,Addr1,Addr2::word(16);
      calculation,switch,leakage,threshold :: design real;
      power :: output real; overThreshold :: output boolean;
      cpuP,memP,nextPower :: real),
  State_based,
  [[#denotation of facet product of cpu and cpuPower#,
    #denotation of facet product of memory and memPower#,
    #denotation of np term#,
    #denotation of ot term#
    #denotation of p term#]]
>
```

Figure 5.17.: Denotation of completeSystem

where the denotation of the facet product of *cpu* and *cpuPower* (and similarly for the product of *memory* and *memPower*) is

```
<(clk::input bit,rst::input bit,
     memReadData::input word(16),instruction::input word(16),
     we::output bit,memEn::output bit,
     pc::output word(16),aluOut::output word(16),
     memData::output word(16),
     memLoadSelect::bit,regFileWriteEn::bit,immediateSelect::bit,
     aluOp::word(2),aluStatus::word(2),
     addressA::word(4),addressB::word(4),addressC::word(4),
     immediateValue::word(16),A::word(16),B::word(16),C::word(16),
     aluIn::word(16),
     calculation::design real,switch::design real,
     leakage::design real;
     power::output real,
     aluP::real,mux1P::real,mux2P::real,regFileP::real,
     controllerP::real)
 State_based,
 [[#denotation of facet product of controlUnit and controllerPower#,
   #denotation of facet product of controlUnit and controller#,
   #denotation of facet product of alu and aluPower#,
   #denotation of facet product of mux and muxPower#,
   #denotation of facet product of mux and muxPower#,
   #denotation of facet product of regFile and regFilePower#,
   #denotation of cob term#,
   #denotation of p term#]]
>
```

Figure 5.18.: Denotation for cpu*cpuPower

These denotations follow the details of facet product. The observers consist of all observers from each piece of the product. The observers of *completeSystem* include the inputs from *system*, as well as the inputs, outputs, and design parameters of *systemPower*, along with all variables of both. The domain is the least common domain of *HW* and *State_based*, which is *State_based*.

There are several important things to note in constructing the list of term lists of *completeSystem*. The *system* facet has one element in its list of term lists. The *systemPower* also has one element in its list of term lists. Therefore, *completeSystem* will have one element in its list of term lists since it is the cross product of two lists with one element. The *system* and *systemPower* facets have two shared items, *processor* and *mem*. They are facet instantiations, so *completeSystem* will contain the facet products of these items. The denotations of one of these facet products is given above, following the denotation of *completeSystem*. The non-shared items in *system* and *systemPower* – *np*, *ot*, and *p* – will also be part of *completeSystem*.

5.6 Discussion of Results

Our resulting facet, in Figure 5.16, is only useful if it gives us all of the behavior desired from the system design (Figure 5.1, Table 5.2, Figure 5.14, and Figure 5.15). Furthermore, a useful specification is one that can be designed modularly. We argue that facet composition gives us the usefulness of modular design, while maintaining the correct behavior.

5.6.1 Correctness

The denotation of the *completeSystem* facet in Figure 5.17 gives evidence of its desired behavior. The denotations's observers

```
(clk::input bit,rst::input bit,
    memReadData::input word(16),instruction::input word(16),
    we::output bit,memEn::output bit,
    pc::output word(16),aluOut::output word(16),
    memData::output word(16),
    memLoadSelect::bit,regFileWriteEn::bit,immediateSelect::bit,
    aluOp::word(2),aluStatus::word(2),
     addressA::word(4),addressB::word(4),addressC::word(4),
     immediateValue::word(16),A::word(16),B::word(16),C::word(16),
     aluIn::word(16),
     calculation::design real,switch::design real,
     leakage::design real;
     power::output real,
     aluP::real,mux1P::real,mux2P::real,regFileP::real,
     controllerP::real)
```

show the entire state necessary to describe the complete system. Beyond the *bit* inputs and *design* inputs, all of the internal state (for instance, *cpuP* or *nextPower*) of the *completeSystem* is present in its denotation. Note that in the facet definition of *completeSystem* there are no explicit observers listed. Rather, these come from each piece of the facet product used to construct *completeSystem*. Furthermore, inside *completeSystem*'s list of denotation bodies

```
 [[#denotation of facet product of controlUnit and controllerPower#,
   #denotation of facet product of controlUnit and controller#,
   #denotation of facet product of alu and aluPower#,
   #denotation of facet product of mux and muxPower#,
   #denotation of facet product of mux and muxPower#,
   #denotation of facet product of regFile and regFilePower#,
   #denotation of cob term#,
   #denotation of p term#]]
```

is the denotation of any facet instantiation or facet composition at all levels within *completeSystem*. As we push into the denotation of *completeSystem*, we get the constraints at every level of the modular design. Pushing in one level, we've shown in Figure 5.18 that the denotation of *cpu*cpuPower* is part of the denotation of *completeSystem*. This continues for each level of vertical composition in the design.

The denotation of every instantiated component throughout the system is included in the denotation body of the facet that instantiates it. The constraints of each instantiated component are included in the instantiating component. So, *alu*'s denotation is included in the denotation body of *cpu*, *cpu*'s denotation is included in the denotation body of *system*, etc.

When two facets are summed, their constraints are added in separate denotation bodies in the resulting facet, indicating that only one of the bodies must hold. So the *controller* is modularly built up. The denotations for each instruction in the *controller* are in their own denotation body.

```
[[#denotations of all terms in Reset#],
 [#denotations of all terms in AddOp#],
 [#denotations of all terms in SubOp#],
 [#denotations of all terms in AndOp#],
 [#denotations of all terms in OrOp#],
 [#denotations of all terms in LW#],
 [#denotations of all terms in LI#],
 [#denotations of all terms in SW#],
 [#denotations of all terms in BLT#]
 [#denotations of all terms in Jmp#]]
```

This means that only one instruction path must be satisfied for each instruction that comes through the *cpu*.

The product of two facets results in all of the constraints from both facets. So, by taking the product of *cpu* and *cpuPower*, we are requiring a valid implementation to meet all constraints of each. The denotation reflects that all of the terms and facet instantiations of both facets are in the same term denotation body in the resulting facet.

```
[[#denotation of facet product of controlUnit and controllerPower#,
  #denotation of facet product of controlUnit and controller#,
  #denotation of facet product of alu and aluPower#,
  #denotation of facet product of mux and muxPower#,
  #denotation of facet product of mux and muxPower#,
  #denotation of facet product of regFile and regFilePower#,
  #denotation of cob term#,
  #denotation of p term#]]
```

So, we have enforced that a valid implementation meets all of the behavioral constraints of *system* as well as the power constraints of *systemPower*.

5.6.2 Ease of Design

The simplicity of the *completeDesign* facet speaks volumes to the simplicity of design that facet composition enables. The *completeDesign* facet in Figure 5.15 is at the same high level as the block design in Figure 5.16. However, the denotation of *completeDesign* shows how intricate and detailed that facet is under the hood, so to speak. We have the power to fully specify a complex system with the simplicity and elegance of a high-level “black box” feel. Modular design is important for complex systems in that it gives us the benefits of reuse and flexibility. Our coalgebraic semantics gives us the benefits of modularity, while maintaining the power of a detailed design capability. We could have written a course-grained model of power consumption to conjunct with the system design and could easily compare the two. This kind of flexibility allows quick design-space exploration.

Chapter 6

Related Works

6.1 Systems and Logics as Coalgebras

There are several works on applications that are particularly well suited to coalgebras. Two major categories that motivate our use of coalgebras for Rosetta specifications are *modal logics* and *systems*, especially state-based or reactive systems. We describe the uses of coalgebras in these categories and describe why the techniques used are appropriate for Rosetta.

6.1.1 Modal Logics

Modal logics are the family of logics whose operators conditionalize formulas to hold under certain criteria such as “in the future,” “normally,” “necessarily.” There are several works that address coalgebras’ suitability for modal logics. Cirstea et al. [2011] discuss that the more common non-normal modal logics are not amenable to the standard Kripke semantics. Rather, since these modal logics are essentially reactive systems, they are much better suited to coalgebraic semantics.

Cirstea et al. [2011] argue that the first major advantage of using coalgebraic semantics is the generality. A coalgebraic framework is constructed per application, and is therefore applicable to a larger class of modal logics. The second major advantage is the compositionality. The coalgebraic framework allows for integration of different requirements, and many different logics co-exist in the same framework. This allows for the "modular combination of reasoning principles" [Cirstea et al., 2011]. Lastly, the coalgebraic framework lends itself to adaptability. The previous two traits allow for new requirements to be easily added to existing requirements.

Rosetta components are logics in that they express assumptions, definitions – which are essentially assertions – and implications that must hold for the specification to be valid. Furthermore, Rosetta has the ability of expressing temporal concepts, such as current state and next state transitions. For instance, some assertions must hold for the next state of a state-based specification. This makes Rosetta modal. Therefore we can apply the principles of using coalgebras for modal logics to Rosetta domains, components, and facets.

6.1.2 State-based Dynamic Systems

Kurz [2001] describes the theory of systems, and lays out how coalgebras are a natural model of these theories. Systems are understood by their interfaces – how they interact and communicate with other systems. Essentially, they are a set of states and the observable transitions on those states. Systems are reactive in nature, and we look at them as "black boxes." Jacobs and Rutten [1997] also give a thorough tutorial of algebras versus coalgebras, and induction versus coinduction and bisimulation.

With an algebraic definition, the initiality gives us a base to stand on. For instance, when describing a list, we have the base case of an empty list, and all lists can be thought of as the pieces constructed to the base case of an empty list – we have a base case and constructors that build any list. With a coalgebraic definition, we have the dual, finality. Think of a stream. We don't think of streams constructively, but rather we have the entire stream, and we take observations, destructing the stream.

Rather than an inductive principle from the initiality of an algebra, we can utilize a coinductive principle from the finality of a coalgebra. Along with coinduction, we can also exploit bisimulation. Informally, a bisimulation exists between two systems or coalgebras if all of their observations match. So, if A and B are two state machines, then A and B are bisimilar if upon each state transition, their outputs or actions match.

When we specify or design a system in Rosetta, we in a sense do so constructively, i.e. we build up a system using the components that make up that system. However, since we still wish to view the systems we're specifying or designing as black boxes, reasoning about systems is done by observing their behavior. We look at the system as a whole, and the only things we need to know about it are what we are able to observe from its interface. This notion is exactly what coalgebras give us. We start with the structure and take observations of all transitions. To this end, coalgebras are the suitable choice to describe Rosetta specifications. We can then use the notions of coinduction and bisimilarity to reason about and compare behavioral equivalence of systems. This work shows how we we can still build models constructively using composition, while maintaining the coalgebraic structure that is suitable for systems.

6.2 Semantics using Coalgebras

The second area of related works motivating this work is in using coalgebras specifically in semantics. We describe the use of coalgebraic denotation in process calculi, the use of coalgebras in the semantics of Java, as well as address previous semantic work of Rosetta.

6.2.1 Process Calculi

Hausmann et al. [2006] describe the use of a coalgebraic denotation for process calculi. The paper claims coalgebraic semantics add clarity to the calculi as well as allow for comparison and unification of process calculi. It gives the formal denotation of the ambient calculus (for mobile computing) using CoCASL [Mossakowski et al., 2003], an extension of the CASL specification language [Astesiano et al., 2001] that adds built-in coalgebraic structures. Process calculi model concurrent systems. Rosetta is for system specification, and we have shown that behaviors of Rosetta specifications can themselves be considered systems. As such, the approaches used for denoting process calculi with coalgebraic semantics can be applied to our denoting Rosetta with coalgebraic semantics.

Hausmann et al. [2006] first lay out the signature functor for the design of the coalgebraic model of the ambient calculus. This sets up the structure, or type of transition system. Then the paper lays out the transition rules. Essentially, this is the structure and observations of the calculus. The coalgebraic framework gives more structure than the typical approach of a labeled transition system, but also gives the generality to be able to relate it to (via bisimulation, etc.) and combine it with other calculi.

6.2.2 Coalgebras in Java Semantics

Jacobs and Poll [2002] explore the use for a combination of monads and coalgebras in the semantics of sequential Java. The monadic approach gives a clean model of the computational structure of Java, while coalgebras give the program logic based on that monadic structure. The coalgebraic view provides reasoning principles via modal operations and bisimulation. This approach is able to cleanly deal with complexities such as multiple termination patterns in the model of computation.

While Rosetta is significantly different from sequential Java, this work still supports our use of coalgebras in the denotation semantics of Rosetta. We show that the structure of Rosetta is well suited for Rosetta components, and, as for Jacobs and Poll [2002], the use of coalgebras gives us valuable reasoning principles, such as modal logic and bisimulation.

6.2.3 Previous Work on Rosetta Coalgebraic Semantics

Kong et al. [2003] lay down the foundation for the coalgebraic semantics for Rosetta. The work gives the general approach to denoting facets, giving the two-part denotation. It then goes on to give the denotation details within a few specific domains. Kong et al. [2003]'s work is the basis for this continued work. The denotation of components and facets is necessary in developing the denotation of the composition of components and facets. We have refreshed the denotation with necessary updates and details. Section 3.1 describes in detail what Kong et al. [2003]'s work entails, and how this thesis builds on that work.

Chapter 7

Conclusion and Future Work

This work discusses the utility of each composition operator for the system-level designer and the role of the composition operators in the Rosetta semantics, providing examples of the application of these operators to the specification design process. Formalizing the semantics of a specification language gives us assurances as to the validity of the specifications we write with the language. We refreshed the formal coalgebraic denotation of components and facets. We then motivated the need for compositional operators and defined the formal coalgebraic denotation of the resulting components and facets from these compositions.

The extended example shows both the usefulness of the composition operators in developing a real-world system specification as well as the conciseness of the end specification. We showed that a simple, clear, and modular specification had the desired complexity in its denotation. This means the designer can utilize the power of the composition operators, while the complexities of composition lie "under the hood" in the language denotation.

There are several directions to extend this work. One area is in formalizing the coalgebras of interactions. We discussed in Section 4.2.1.1 how, regardless of the operand components' domains, we can compose them by using their least common domain as the resulting domain. However, one of Rosetta's features is its domain interactions. These define translators, functors and combinators that precisely define how information flows between domains. There remains further work on the denotation of interactions, and interactions' impact on the coalgebraic structure of Rosetta components and facets. Furthermore, interactions involve facet combinations, and that needs to be fleshed out in how it compares to the facet composition described in this work.

Another area for future work is in the utilization of tools to model the structure laid out in this thesis. We could use a system such as PVS [FormalWare, 2011] to model and verify domains, components, and facets. We can model facets as coalgebras within PVS. We could then leverage the built-in coinduction proof capabilities to perform bisimulation proofs, etc.

Bibliography

Rosetta wiki: Facet instantiation, August 2008. URL `https://wiki.ittc.ku.edu/rosetta_wiki/index.php/Facet_Instantiation`. Discussion from Rosetta Standards Committee regarding Instantiation and Inclusion.

SystemVerilog 3.0: Accellera's Extensions to Verilog. Accellera, 2002.

P. Alexander. *System Level Design with Rosetta (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 1558607714.

P. Alexander, D. Barton, and C. Kong. *Rosetta Usage Guide*. The University of Kansas / ITTC, 2335 Irving Hill Rd, Lawrence, KS, 2000.

Perry Alexander. *KURM ISA*. University of Kansas, 2009. URL `http://www.ittc.ku.edu/~alex/teaching/eecs443/resources/files/kurm09-isa.pdf`.

Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 1997.

Egidio Astesiano, Michel Bidoit, Helene Kirchner, Bernd Krieg-Bruckner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. Casl: The common algebraic specification language, 2001.

Corina Cirstea, Alexander Kurz, Dirk Pattinson, Lutz Schröder, and Yde Venema. Modal logics are coalgebraic. *The Computer Journal*, 54(1):31–41, 2011. URL `http://comjnl.oxfordjournals.org/cgi/content/abstract/bxp004`. Extends (Cirstea et al. 2008).

SRI FormalWare. *PVS Specification and Verification System*. SRI, 2011. URL `http://pvs.csl.sri.com/`.

Nicolas Frisby, M. Peck, Mark Snyder, and Perry Alexander. Model composition in rosetta. In *ECBS*, pages 140–148. IEEE Computer Society, 2011. ISBN 978-1-4577-0065-1. URL `http://dblp.uni-trier.de/db/conf/ecbs/ecbs2011.html#FrisbyPSA11`.

T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Press, 2002.

Daniel Hausmann, Till Mossakowski, and Lutz Schröder. A coalgebraic approach to the semantics of the ambient calculus. *Theoretical Computer Science*, 366(1-2):121–143, 2006. URL `http://dx.doi.org/10.1016/j.tcs.2006.07.006`. Extends (Hausmann et al. 2005).

Standard Verilog Hardware Description Language Reference Manual. IEEE, New York, NY, 1995.

VHDL Language Reference Manual. Institute of Electrical and Electronics Engineers, Inc., 345 East 47th St., New York, NY 10017, 1994.

Bart Jacobs and Erik Poll. Coalgebras and monads in the semantics of java. *Theoretical Computer Science*, 291:2003, 2002.

Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:62–222, 1997.

Jim Davies Jim Woodcock. *Using Z: Specification, Refinement, and Proof*. University of Oxford, 1995. URL `http://www.cs.cmu.edu/~15819/zedbook.pdf`.

G. Kimmell, E. Komp, G. Minden, J. Evans, and P. Alexander. Synthesis of software defined radios using rosetta. In *Federation of Design Languages (FDL'08*, Stuttgart, Germany, September 2008.

Cindy Kong, Perry Alexander, and Catherine Menon. Defining a formal coalgebraic semantics for the rosetta specification language. 9(11):1322–1349, nov 2003.

Alexander Kurz. *Coalgebras and Modal Logic*. CWI, 2001. URL `http://www.cs.le.ac.uk/people/akurz/CWI/public_html/cml.ps.gz`. Lecture Notes, ESSLLI'01.

Jennifer L. Lohoefener. *A Methodology for Automated Verification of Rosetta Specification Transformations*. PhD thesis, University of Kansas, 2011.

Till Mossakowski, Markus Roggenbach, and Lutz Schröder. CoCASL at work — modelling process algebra. In Hans-Peter Gumm, editor, *Coalgebraic Methods in Computer Science*, volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science; http://www.elsevier.nl/, 2003.

B. Nuseibeh, A. Finkelstein, and J. Kramer. Viewpoints: meaningful relationships are difficult. In *International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon, 2003. ACM Press.

Wesley G. Peck. *Hardware/Software Co-Design via Specification Refinement*. PhD thesis, University of Kansas, 2011.

J. Streb and P. Alexander. Using a lattice of coalgebras for heterogeneous model composition. In *Proceedings of the Multi-Paradigm Modeling Workshop (MPM'06)*, Genova, Italy, October 2006.

J. Streb, G. Kimmell, N. Frisby, and P. Alexander. Domain specific model composition using a lattice of coalgebras. In *Proceedings of the OOPSLA'06 Workshop on Domain Specific Modeling*, Portland, OR, October 22 2006.