Semantic Support for Model Composition*

Cindy Kong[1], Garrin Kimmell[2], Jennifer Streb[2], Perry Alexander[2]

[1] Intel Corporation
cindy.kong@intel.com
[2] The University of Kansas
Department of Electrical Engineering and Computer Science
Information and Telecommunication Technology Center
{kimmell,jenis,alex}@ittc.ku.edu

Submitted to *Integrated Formal Methods 2005*

Abstract. The essence of systems-level design is the need to integrate models representing different system facets to understand the impacts of local decisions on global requirements. Unfortunately, these models may be defined in disparate semantic systems making composition and integrated analysis challenging. As a part of the Rosetta systems-level design effort, a collection of mechanisms based on coalgebraic semantics has been defined to transform and compose models. *Functors* define mechanisms for moving models between specification domains; *coproducts* define mechanisms for composing multiple specifications; and *translator functions* define mechanisms for structurally compose specifications. Together these techniques provide specification composition support for integrating formal, systems-level analysis activities.

1 Introduction

The essence of systems engineering and systems-level design is understanding understanding the impacts of local design decisions on global system properties. *System-level requirements* represent requirements that must be assessed in the whole system and cannot be met by simple budgeting across components. Examples of such requirements include power consumption, security, and cost. Designers cannot simply decompose security and assign elements to system components. Although power can be budgeted, interactions between components complicate meeting local and global power requirements. Like power, cost can be budgeted among components, but integration costs complicate system-level cost calculations.

The distributed and heterogeneous nature of systems-level design complicates systems engineering. Local design decisions frequently have systems-level impacts because

* This material is based upon work supported by the National Science Foundation under Grant Nos. 0209193 and 0350425 and The University of Kansas ITTC under a Technology Transfer Grant.

engineering domains are not orthogonal. For example, how software is written can significantly impact power consumption in the computer system it runs on. Yet, few software engineers are taught to think about power consumption. Engineering domains adopt their own vocabularies and formalisms making communicating information difficult or impossible. Software representations and power modeling representations use radically different vocabularies making it difficult to understand their interaction.

It is neither economically feasible or mathematically sensible to represent all requirements models using the same underlying semantics. Engineers looking at different system facets in different domains necessarily use different formalisms and vocabularies. These intellectual tools are adopted due to their utility within the domain, not the ease of integrating with intellectual tools from other domains. Asking engineers to move their work to a different set of design formalisms will fail for both social and technical reasons. Thus the challenge of systems engineering – bringing together intellectually distant information from across multiple domains during systems design.

The Rosetta systems-level design language [1, 2] attempts to address these issues in providing systems engineers with a systems description language. Sponsored by the Accellera [3] electronic design automation standards body, Rosetta provides designers with mechanisms for representing systems level requirements using multiple domains while supporting composition of heterogeneous models and synthesis of hardware components.

To support the needs of systems-engineers, Rosetta provides explicit support for: (i) composing models from different domains; (ii) moving models between domains; and (iii) passing communicated information between models in different domains. Rosetta uses a co-algebraic semantics for system models. Model composition is achieved using a *pullback* over this co-algebra semantics. Moving models between domains is achieved by defining *functors* that move a model from one domain type to another. Finally, moving information between facets is achieved by defining *translator functions* used at component interfaces to perform translation.

Having provided a modeling language and semantics for systems level modeling, we have accomplished little if we cannot predict behavior from specifications. Rhaskell [4], the Rosetta support environment provides and integrated collection of verification tools and a standard means for integrating new tools and semantics. Current capabilities include a theorem prover [5], static analysis, advanced type checking, and evaluation tools [4], and specification composition tools [6]. Raskell tools provide the beginnings of an integrated formal analysis environment both combining analysis techniques and transforming models between analysis domains.

2 Background

To understand Rosetta's model composition and transformation capabilities, it is necessary to understand the semantics of Rosetta models [7–9] and the domain semi-lattice [2] central to Rosetta's modeling paradigm. It is not necessary to understand the full Rosetta language to make use of its model composition semantics.

A Rosetta model, referred to as a *facet*, uses a coalgebraic semantics to define observations on a component's abstract state. The abstract state is never directly visible, but observed only through facet declarations. In effect, the declaration i :: integer defines an observation i whose values are restricted to the set integer on its associated facet's abstract state.

The Rosetta semi-lattice defines a collection of *domains* that provide vocabulary and model-of-computation semantics for facets. All facets formally extend a domain to define a specific component model. Domains denote facet types by defining the final algebra of a category constructed using extension. Thus, the type associated with a domain is the collection of facets written by extending the domain.

2.1 Co-algebraic Facet Semantics

A Rosetta facet's semantics is defined as a coalgebra over a hidden, abstract state $\mathcal{X}$. We extend the notation from Jacobs [10] to define a coalgebra:

$$F = \langle \boldsymbol{\iota} \rangle : \mathcal{X} \rightarrow \boldsymbol{\tau} \mid \mathcal{T}$$

where $\boldsymbol{\iota}$ are state observers, $\mathcal{X}$ is the hidden abstract state, $\boldsymbol{\tau}$ are the signatures of the observers where $\iota_k : \tau_k$, and $\mathcal{T}$ is the set of terms defining the observers. Facet semantics introduces the restriction that $\mathcal{X}$ cannot appear in $\boldsymbol{\tau}$ – the abstract state is observable only through functions listed in $\boldsymbol{\iota}$.

Thus, the Rosetta facet model:

```
facet andGate(x,y::input bit; z::output bit)::state_based is
begin
  driver: z' = x * y;
end facet andGate;
```

is represented by the coalgebra:

```
⟨x, y, z, s, at⟩ : 𝒳 → bit,bit,bit,state,<*[T1,T2::type](i::T1; s::state)::T2*> |
  z'at(next(s)) = x'at(s) * y'at(s);
```

The transformation from facet specification to coalgebra involves: (i) introducing the abstract state; (ii) expanding the facet domain; and (iii) elaborating definitions. Introducing the abstract state simply defines the abstract state associated with the resulting coalgebra. Expanding the facet domain imports vocabulary and definitions from the facet's domain into the coalgebra as observations of the abstract state. Finally, elaborating definitions reduces abstract definitions to kernel Rosetta using definitions from the domain and other included facets.

The *abstract state*, $\mathcal{X}$, is the hidden abstract state of the coalgebra. Although its value cannot be observed directly, the value of every item defined in a facet is defined with respect to a particular abstract state. Thus, specific properties of the abstract state can be directly observed through variables and functions. Observing the abstract state rather than making it concrete is essential to mechanisms used to compose specifications and build specific definitions from domains.

The *domain* defines the specification vocabulary define for the facet. Domains define everything from the model-of-computation to engineering vocabulary used to define a specification. When writing a specification, the domain is extended to define a specific model embodying properties of interest. This example uses the state_based domain that defines a vocabulary including the concrete state type (state), the current concrete state (s), the next state (next(s)) and how symbols are dereferenced with respect to to concrete state ('at(s)).

Elaboration translates the high-level Rosetta specification into an equivalent kernel Rosetta specification. We will not show the full elaboration here, but simply the first step that involves translating common shorthands into full definitions using the domain. The result of elaborating the single term driver is:

```
z'at(next(s)) = x'at(s) and y'at(s)
```

x'at(s) refers to the value of x in the current state, s, and is written in the specification simply as x. Similarly for y. z'at(next(s)) refers to the value of z in the state following x and can be written using the shorthand z'next(s) or simply z' as in the original specification. The state value, s, is an observation of $\mathcal{X}$ just like any other symbol. This is critical to our composition and refinement mechanisms because it allows the same hidden state to be observed by different concrete state types. Making $\mathcal{X}$ concrete would complicate defining heterogeneous, interacting observations difficult.

2.2 The Domain Semi-Lattice

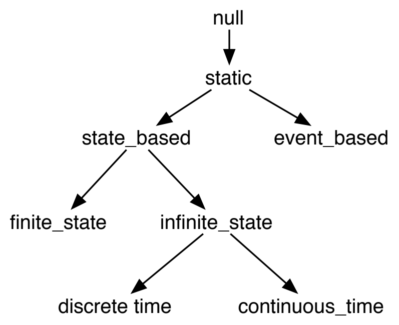
Critical to Rosetta specifications is the concept of a *domain* that provides vocabulary and model-of-computation information to a model. Figure 1 defines a semi-lattice of domains where arrows represent extension. An domain lower in the lattice is a sub-domain while the domain higher is a super-domain. The set of facets written by extending a domain defines the *type* associated with that domain. Kong and Alexander [9, 11] have shown that each domain's associated type is a category of facets with extensions as arrows. Furthermore, the category of facets associated with a domain is a sub-category of its super-domain's category defining a subtype/super-type relationship.

When we write:

```
facet lowpassFilter
        (i :: in real; o :: out real; frequence::design real)::continuous_time is
begin
  ...
end facet lowpassFilter;
```

we are defining a facet that extends continuous_time. Thus, it is of type continuous_time and takes its vocabulary and model-of-computation from the continuous_time domain model.

Fig. 1 The domain semi-lattice with arrows representing extensions.



3 Transforming and Composing Semantics

To achieve our goal of systems engineering support it is not sufficient to simply allow specifications to use different semantics within the same language. To support predictive modeling, we must support model composition and transformation of information from one modeling domain to another.

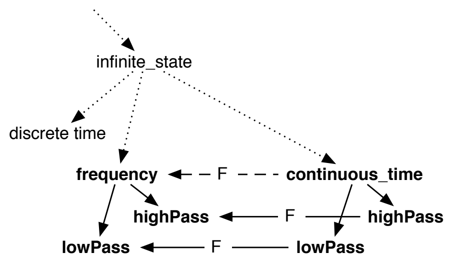
Rosetta supports composition and transformation using an **interaction** construct that defines *functors*, *translators*, and *algebra combinators* and indirectly *product* and *coproduct* operations for composing models. Functors move a facet model from one domain to another. Such operations are simply functions whose signatures specify facet types as domain and range values. Translators enable moving data through facet interfaces between domains. Such operations allow facet models defined in one domain to communicate with facets defined in another. Products and coproducts compose specifications by finding the limit or colimit of two specifications in the domain semi-lattice. Pullback and pushout constructs are used to construct new models. Finally, Algebra combinators generate new specifications from products and coproducts. They are used to convert products into specifications that can be used for analysis or synthesis.x

3.1 Functors

A functor between categories is an operation that maps arrows to arrows and object to objects in different categories. Because objects in the domain semi-lattice are facet types, Rosetta functors transform facets of one type into facets of another. Each domain defines a category of facets that in turn defines a facet type. Associating a functor with two domains provides a function that transforms facets from one type into facets of another.

An excellent example of a functor between facet types corresponding with a common mathematical transformation is using a Fourier transform to move a model from the time domain to the frequency domain. Figure 2 graphically shows a fourier transfer functor and two instances of the transformation. The dashed arrow labeled F between domains frequency and continuous_time defines the functor. Explicit instances of the functor between facets comprise the functor and define fourier transforms on specific models. In this case, fourier transforms exist between time and frequency models of highpass and lowpass filters.

Fig. 2 A functor representing fourier transform and two specific instances of fourier transform between models in the time and frequency domain.



Functor definitions play two important roles in Rosetta modeling. First, they are used to move information between domains to perform analysis. If a domain exists for discrete event simulation, a functor could be written to transform a time-based specification to the simulation domain. The new model can then be simulated to predict behavior. This functor corresponds to the types of analysis performed in engineering design.

Second, functors are used to move information between domains to switch modeling abstractions. The fourier transform is an example of this functor type. We take fourier transforms of time-based models to use a different set of abstractions for modeling and analysis. Examining a filter's transfer function in the frequency domain reveals information about the filter's behavior that is difficult to directly address in the time domain. Such transformations are exceptionally common in engineering design.

Note that all the arrows between domains in the domain semi-lattice are functors. These arrows represent extensions that form more detailed modeling domains from less detailed domains. Because they are defined only over domains and every facet in a domain's category is an extension of that domain, they equally apply to facet's in the domain. These functors are constructed when the semi-lattice is extended, but are no less functors than those written by hand or automatically generated.

3.2 Translator Functions

Translator functions or simply translators are special operations that move data across facet boundaries. Functors transform entire facets into new facets in new domains.

Translators enable communication by transforming data in one domain into data in another while accounting for differences in computation models. Consider the translator function for moving an analog signal into the digital domain:

```
a2bit() from x::real in continuous_time to bit in discrete_time is
  let dt be floor(continuous_time.t) in
    if x@dt =< 2.5 then 0 else 1 end if;
```

The a2bit translator is used to transform analog signals into digital signals in discrete time. The following facet instance transforms analog signals into digital signals using the a2bit translator function.

```
l1: and(x'a2bit,y'a2bit,z'a2bit);
```

It may seem odd that the translator is applied to input parameters as well as output parameters. However, the direction of the parameter is not material. As long as the signal satisfies constraints specified by the translator on both side of the facet interface, the model is consistent. Facet inputs are independent variables that can be driven to any value. Facet outputs are dependent variables that must satisfy constraints placed on them by their associate translator functions.

3.3 Specification Products and Coproducts

Products and coproducts are among the most common mechanisms for defining composition in language semantics. Specifically, defining record and variant structures using these primitives is a standard approach in many semantic systems. The sum operation defines a disjoint union while the product defines a record. By making these first-class operations over models in Rosetta, we provide a mechanism for composing specifications in a similar manner. Facet sum provides a mechanism for defining different specification scenarios while facet product provides a mechanism for defining simultaneous aspects.

The Rosetta product operation corresponds with forming a limit using a pullback. Likewise, the sum operation corresponds with forming a colimit using a pushout. The notation $F_1 * F_2$ signifies a product forming a limit while the notation $F_1 + F_2$ signifies a coproduct forming a colimit. Rosetta forms each construction using the domain semi-lattice and the types of the facets being composed. For illustration, assume the following definitions:

```
facet addFn(x,y,cin::input bit; z::output bit)::state_based is begin
update: z' = x xor y xor cin; end facet addFn;

facet carryFn(x,y,cin::input bit; cout::output bit)::state_based is
begin
  update: c' = (x * y) + (x * cin) + (y * cout);
end facet carryFn;
```

If we wish to define a new model, adder, that embodies properties of both addFn and carryFn we use the product operator:

```
adder :: state_based is addFn * carryFn;
```

The new component, adder, is of type state_based and combines the original models to form a full adder. Specifically, the adder is *both* an add function and a carry function in precisely the same way that a tuple is the collection of its fields. Both are examples of product constructions.

The pullback is formed with the least common facet type involved in the operation as the operation's shared part. Both facets forming adder are from the state_based domain, thus finding the least common type is trivial. The pullback is graphically represented in Figure 3.

Fig. 3 Pullback formed from two specifications from the same domain. The shared part defining a common vocabulary between specifications is the domain itself.

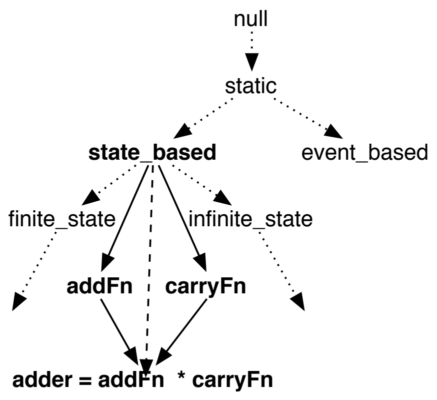
$$\begin{tikzcd}
D' \arrow[dr] \arrow[drr] \arrow[ddr] & & \\
& D \arrow[r, "f"'] \arrow[d, "g"'] & F_1 \arrow[d, "g'"'] \\
& F_2 \arrow[r, "f'"'] & F_1 * F_2
\end{tikzcd}$$

In Figure 3, D represents the shared domain of the facet specifications F_1 and F_2. Because D is shared, declarations from D appear in both F_1 and F_2 and refer to the same specification objects. In keeping with Smith [12, 13], we frequently refer to this as the *shared part* because it is shared between specifications. The shared part is vital because it provides a common frame of reference for the composed specifications. Without it, there is no means for both specifications to simultaneously describe the same observations of the abstract state, $\mathcal{X}$.

f and g represent extensions of D that provide specifics of the actual model. A domain is included in all facet definitions, thus all facets extend a domain and f and g always exist for any facet pair. There is no prescriptive guarantee that these extensions will be conservative, complicating the process of writing specifications. However, Rosetta language design goals place expression above interpretation justifying this design decision.

D' represents the immediate super-type of D. Without modification, the semi-lattice is a tree. Thus, D' is the only direct super-type of D. However, other supertypes can exist in the tree above D'. The formation of the pullback requires that D be final in the

Fig. 4 Pullback formed from two specifications from the same domain in the context of the domain semi-lattice.



category. This is trivially true because f and g are simply extension morphisms. A special case exists when D is the static domain and has no proper supertype. In this case, D must be final because there are no arrows leaving it in the semi-lattice.

If we examine the commutative diagram within the domain semi-lattice, it reinforces the idea that the pullback in fact defines a final object as is required. Figure 4 graphically shows this result. state_based represents the final coalgebra, D. The dashed arrow represents the implicit morphism that exists due to the coproduct. Thus, the resulting adder specification is a subtype of state_based.

Fig. 5 A power consumption specification for a two input device.

```
facet adderPower
  (x,y :: input bit ; p :: output real; pinc :: design real) :: continuous_time is
  consumedPower::real;
begin
  st : consumedPower' = consumedPower +
                      if (event(x) or event(y))
                        then pinc
                        else 0.0
                      end if;
  update: p = consumedPower;
end facet adderPower;
```

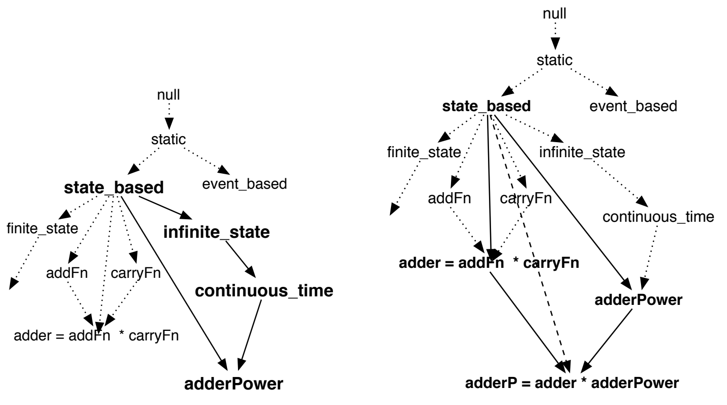
The adder example is trivial because the two facet specifications start in the same domain and their associated operations are orthogonal. What we have done here is similar in nature to the schema conjunction operation in Z [14]. The product and coproduct operations are far more interesting when specifications being composed are of different types.

Consider a new specification of the same adder definition that defines power consumption within the device. Specifically, whenever either of its inputs change the CMOS implementation will consume power due to transistors changing state. One model for this is shown in Figure 5.

The product is again used to define a complete adder as adder∗adderPower. Here the domains are different. adder is of type state_based while adderPower is continuous_time. The least common super-type of state_based and continuous_time is state_based, thus the type associated with the product must also be state_based. Figure 6(a) graphically shows the construction of the morphism from state_based to the adderPower facet verifying the subtype relationship. This allows us to formally define the product:

```
adderP :: state_based is adder∗adderPower
```

Fig. 6 Graphically constructing the pullback to form the integrated power model.



(a) Type relationship constructed between adderP and state_based. (b) Pullback formed from two specifications from different domains.

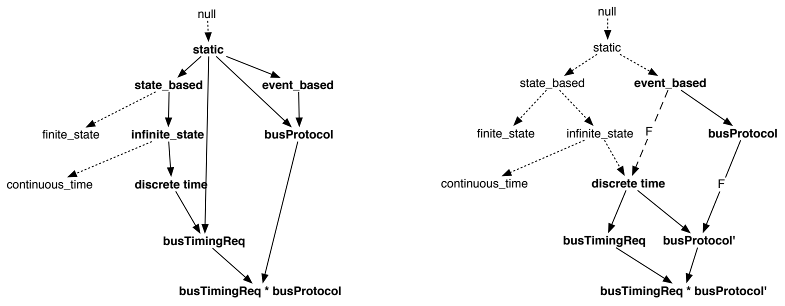
The pullback in the semi-lattice is formed as before with state_based defining the shared part. Figure 6(b) graphically shows the pullback in the context of the semi-lattice. The arrow between state_based and adder is constructed from existing arrow and is one arrow defining the pullback.

The structure of the semi-lattice ensures that any two facets will have a common supertype, even if that supertype is static representing the base Rosetta mathematical system. When composing specifications that share a type or involve composing few errors, engineering abstractions remain relatively intact in the resulting product.

Examining the adder specifications reveals this – no abstractions are lost in forming the product.

When specification composition involves intellectually distant domains, functors can help preserve design abstractions by moving a specification in the semi-lattice. Transforming specifications into a different domain closer to other domains involved in the product avoids moving to the static domain where all design abstractions are lost. Consider Figure 7 where an event_based specification is composed with a discrete_time specification with and without first applying a function.

Fig. 7 Discrete time and signal-based specifications composed with and without a transformation functor. Note the loss of abstractions when the functor is not included.



(a) Composition without functor application

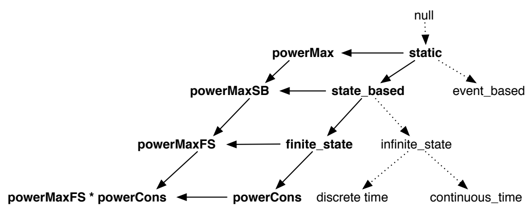
(b) Composition with functor application

Figure 7(a) shows the result of composing specifications without first moving a specification with a functor. The only abstractions that are shared in the resulting specification are those defined in static. Thus, the concepts of time and event are lost in the transformation.

Figure 7(b) results from moving the event_based specification to the discrete_time domain, then forming the product. In this case, the time abstractions remain as well as the stateful nature of the requirements specification. Of course, this assumes the functor, F, can be written. This will not always be the case, but for many domains, functors of this type are well known.

An alternate case involves taking a general specification and using semi-lattice transformations to generate a more specific transformation. Exemplifying this common technique is the task of moving a static power constraint into a temporal domain. Figure 8 shows the construction of a temporal power constraint definition from a static definition. A series of products are generated to incrementally move the static specification through successive refinements to realize a temporal specification that can be combined with the discrete time power model.

Fig. 8 Instantaneous power consumption limit moving down the semi-lattice for composition with power consumption model.



3.4 Algebra Combinators

An *algebra combinator* is an operator that composes two algebras sharing an abstract syntax into a single algebra. The signature of a typical algebra combinator is:

$$(F(a) \rightarrow a) \rightarrow (F(b) \rightarrow b) \rightarrow (F(a \circ b) \rightarrow a \circ b)$$

where a and b are value spaces and F is an abstract syntax parameterized over a value space. $F(a) \rightarrow a$ is thus a valuation function mapping elements of an abstract syntax defined over the value space a to a. It defines requirements for evaluating elements of $F(a)$. Given two instances of F over distinct value spaces, the algebra combinator generates a new algebra over the composition of the original value spaces. Informally, it composes specifications that share abstract syntax elements.

Algebra combinators are used in conjunction with products and functors to generate composite specifications. The product brings together two specifications and identifies their shared abstract syntax. Functors assist this process by allowing the shared abstract syntax to be as rich as possible. The algebra combinators take specifications represented as sums and products and generates an integrated, single specification for analysis.

As a trivial example of an algebra combinator, assume a power constraint written in the static domain in composition with the power consumption model from Figure 5:

```
facet adderPowerConstraint(powerLimit::design real)::static is
begin
  consumedPower =< powerLimit;
end facet adderPowerConstraint;

adderP :: state_based is adderPowerConstraint and adderPower;
```

adderP should limit the instantaneous power consumption modeled by adderPower to the value specified as powerLimit by adderPowerConstraint. A simple functor for composing these specifications is:

```
limit_power(lim :: static ; cons::continuous_time)::continuous_time is
  add_term(forall(t :: time | consumedPower@t < lim.consumedPower),cons);
```

This functor adds a term to the power consumption model limiting consumedPower to the value specified in the power consumption constraint model. Although this is a trivial functor, it does add the specified constraint to the consumption specification and demonstrates capabilities for moving information among domains.

4 Power Analysis – An Example

Rosetta is intended to describe systems in a manner that allows predicting the results of design decisions on system requirements. In particular, we are interested in providing analysis early in the design cycle. To demonstrate Rosetta's capabilities, we have used several example systems including generating test vectors for a radio transceiver [15], parametric modification for a dual spring system [16], power/design trade-off analysis for a hydraulic actuator [1], and power analysis for implementation technology selection in system design [17]. We will overview the latter system to outline Rosetta capabilities on a real-world design problem.

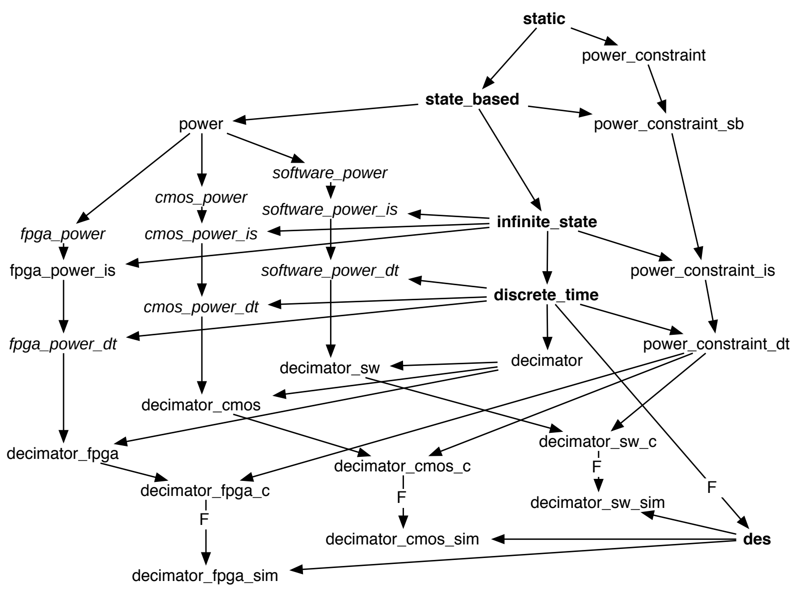
The challenge is to determine whether it is best for a decimator for a TDMA receiver to be implemented in software, FPGA or ASIC before prototyping the component. The approach chosen uses an activity-based power estimation model and simulation to determine activity in the component. We specialized the power model for each implementation technology using a refinement on the basic power model. We specialized the functional model similarly, changing the activity estimation based on the implementation technology. We then composed the power model and the functional model using a coproduct and applied a functor to generate a simulation model. The simulation model was then executed to approximate power consumption. Figure 9 graphically represents this construction for FPGA, CMOS and software implementations.

The model generation process begins with the diagram in Figure 10(a). A segment of the domain semi-lattice is shown with the three original models of the component function, an activity-based power consumption model, and a power constraint model. Each model is define in a domain appropriate for what it represents. The power constraint is constant, the power consumption model is defined over state change, and the functional model is a discrete time system. The remainder of Figure 9 is constructed by defining morphisms on these original models.

To compose the three models, we could simply form to pullbacks using the static domain as the shared part. However, this would eliminate all abstractions in our models and make analysis virtually impossible. Thus, we refine the power consumption and power constraint models so that they share a less abstract domain with the functional model.

The refinement of the power constraint model that appears on the right side of Figure 9 is shown separately in Figure 10(b). Several pullbacks construct the morphism that transforms the static power constraint model into a discrete_time model. This collection of transformations is actually quite trivial as we simply assert that if a

Fig. 9 Full diagram showing refinement and composition of device, power and constraint models.



property is constant, it must hold at any time step. Because it is trivial, this "transformation" is manually performed although the Rhaskell environment could easily automate the task.

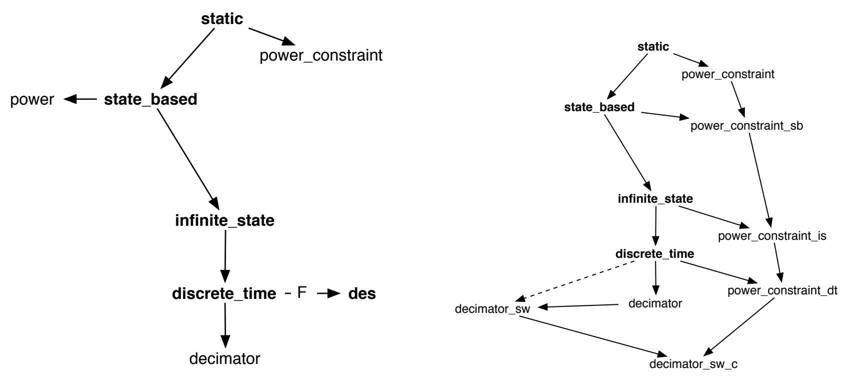
Figure 11(a) shows refinement of the power consumption model from the state_based domain into the discrete_time domain. This one of three constructions that generate FPGA, CMOS and software power consumption models on the left side of Figure 9. Each of these refinements is similar in nature to the power constraint refinement, except they are performed automatically operations written in the Raskell environment.

Following the transformation of each model into the discrete_time domain, pullbacks are used to construct a systems model. The functional model is used to generate activity information for the FPGA, CMOS and software power consumption models while the power constraint model simply asserts a condition that must hold continuously in each model. In both cases, an algebra combinator is used to compose information.

With the products formed, Figure 11(b) shows a functor applied to generate simulations from the final models. This functor is simply a compiler that generates an executable simulation model in the Raskell simulation framework. It should be noted that the algebra combinator used to compose the functional and power consumption model is actually applied during this compilation step.

Looking back at Figure 9 it should now be clear how the diagram is formed. The original models are refined as necessary to generate discrete_time models. These

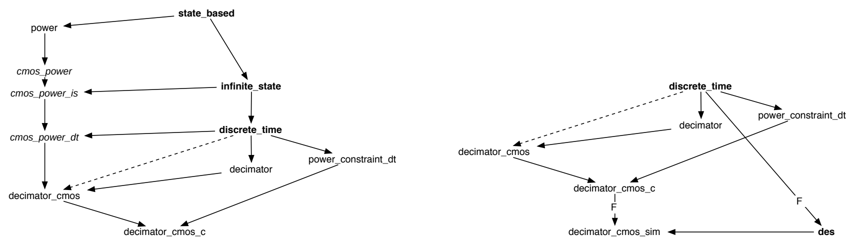
Fig. 10 Original domains and models and refinement of a power constraint model.



(a) Refinement of domains to a functional model.
(b) Refinement of the power constraint to a state-based model.

models are composed using products and a functor is used to generate simulation models. The diagram is busy, but does represent the morphisms necessary to construct the simulation models.

Fig. 11 Refinement of the CMOS power consumption model model and generating a simulation model using a functor.



(a) Refinement of a basic state-based power model to a CMOS power consumption model.
(b) A functor used to generate simulation models from discrete time models.

5 Related Work

The use of category theoretic techniques to describe and manipulate specifications has long history in language semantics. The composition and construction approaches we employ have their origins in the work of Ehrig and Mahr [18] and Smith [13].

Using specification morphisms to generate and manipulate facet specifications is attributable to Smith's KIDS software synthesis system [12]. Approaches initially explored in KIDS are generalized to classify various ways of defining specification morphisms [13, 19]. Although Smith is not the only researcher examining such techniques, the KIDS, PlanWare and SpecWare approaches have proven the most widely useful to date.

Software morphism approaches typically employ theorem provers to derive specification morphisms [20, 21, 12] rather than using constructive proofs to derive actual specifications [22–24]. Such systems represent the larger class of software engineering tools that use theorem provers [25, 26]. The proposed approach could use theorem provers to derive morphisms, however we have found this untenable for our user base. Thus, we use algebra combinators [6]. On particularly promising implementation direction is the use of monad transformers [27–29]. In particular, the work of Lüth [29] using coalgebras to compose monads has provided significant insight.

Institutions [30, 31] represent a formal mechanism for moving information between formal systems. Rosetta functors can implement a type of institution, but are neither as general or as powerful as these formal constructions. We believe that our current facet manipulation semantics is sufficient for our current activities. However, we continue to examine the potential for incorporating institutions formally in the Rosetta system.

Viewpoints [32] represent a less formal mechanism for composing different views of specifications. Quite similar to the domain semi-lattice, viewpoints model hierarchies of alternative specifications [33] Originally from the software engineering domain, Viewpoint analysis is becoming increasingly rigorous supporting examination of inconsistent views [34] and formal analysis through model checking [35].

The Ptolemy [36, 37] system exemplify a simulation approach to heterogeneity. Unlike Rosetta, Ptolemy II does not support model composition. However, it does provide excellent, rigorously defined support for interaction between models from different semantic domains. Unlike Rosetta models, Ptolemy models are executable and thus lend themselves to simulation. Only operational techniques are applied in Ptolemy examples we are aware of.

6 Conclusions and Future Work

This paper presents the specification componsition and transformation techniques use by Rosetta and Rhaskell to analyze heterogeneous specifications. The approach depends composing and transforming specifications rather than composing and transforming analysis results. We have explored this approach in several domains and continue refining semantics and implementing automated tool support.

Our continuing work explores the implementation of algebra combinators and automating more specification morphisms. Algebra combinators are critical to specification composition, yet can be brittle. An effective combinator between two domains may not be at all useful between other domains that appear quite similar. We are working on general frameworks as well as application to new domains such as assurance and security. In addition, we are attempting to use combinators to perform synthesis as well as analysis activities. Specifically, we are beginning to explore hardware/software codesign techniques.

We continue to automate increasing numbers of functor and combinator applications. Rosetta is reflective, supporting such automation. However, manipulating specifications in a semantically sound fashion is known to be a difficult problem. However, we are having success in attacking specific application domains such as embedded systems and telecommunication systems.

The Rosetta language and semantics are currently undergoing standardization with support from the Accellera EDA standards organization. We hope to being IEEE standardization in early 2006. More information on this process can be obtained from the authors.

References

1. Perry Alexander and Cindy Kong. Rosetta: Semantic support for model-centered systems-level design. *IEEE Computer*, 34(11):64–70, November 2001.
2. C. Kong and P. Alexander. Multi-faceted requirements modeling and analysis. In *Proceedings of the IEEE Joint International Requirements Engineering Conference (RE'02)*, Essen, Germany, September 9–13 2002.
3. The accellera homepage. Organization Website. http://www.accellera.org.
4. E. Komp, G. Kimmell, J. Ward, and P Alexander. The Raskell Evaluation Environment. Technical report, The University of Kansas Information and Telecommunications Technology Center, 2335 Irving Hill Rd, Lawrence, KS, USA, November 2003.
5. J. Ward and G. Kimmell. Rosetta Theorem Prover. Technical report, The University of Kansas Information and Telecommunication Technology Center, 2335 Irving Hill Rd, Lawrence, KS, USA, June 2003.
6. Garrin Kimmell, Jennifer Streb, Edward Komp, and Perry Alexander. Composing specifications using algebra combinators. Submitted to *The International Conference on Formal Engineering Methods (ICFEM'05)*, May 2005.
7. C. Kong, P. Alexander, and C. Menon. Defining a Formal Coalgebraic Semantics for the Rosetta Specification Language. *Journal of Universal Computer Science*, 9(11), November 2003. `http://www.jucs.org/jucs_9_11/defining_a_formal_coalgebraic`.
8. C. Kong and P. Alexander. The rosetta meta-model framework. In *Proceedings of the IEEE Engineering of Computer-Based Systems Symposium and Workshop*, Huntsville, AL, April 2003.
9. C. Kong. *Modular Semantics for Model-Oriented Design*. Pdd dissertation, The University of Kansas, Lawrence, KS USA, July 2004.
10. Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. EATCS Bulletin 62, 1997. p.222-259.
11. C. Kong and P. Alexander. Defining a formal semantics for the Rosetta specification language. In *Proceedings of the IFIP Formal Specification of Computer-Based Systems Workshop (FSCBS'03)*, Huntsville, AL, April 2003.

12. Douglas R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.
13. Douglas R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation*, 15:571–606, 1993.
14. J. M. Spivey. *The Z Notation: A reference manual*. International Series in Computer Science. Prentice Hall, New York, NY, 2^{nd} edition, 1992.
15. Srinivas Akkipeddi. Advanced test vector generation from rosetta. Master's thesis, The University of Kansas, Lawrence, KS, 2002.
16. P. Ashenden, P. Alexander, and D. Barton. A dual spring system case-study model in rosetta. In *Proceedings of Federation on Design Languages (FDL'00)*, Tubingen, Germany, September 2000.
17. P. Alexander, G. Kimmell, C. Kong, and B. Morel. A rosetta-based power-aware analysis example. Demonstration at the IEEE Design Automation Conference, June 2002.
18. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. EATCS Mongraphs on Theoretical Computer Science. Springer–Verlag, Berlin, 1985.
19. Douglas R. Smith. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology*, LCNS. Springer Verlag, 1996.
20. Douglas R. Smith. Planware tutorial, 1997. Presented at the 12th International Conference on Automated Software Engineering.
21. Douglas R. Smith. Derived preconditions and their use in program synthesis. In *Proceedings of the Sixth conference on Automated Deduction*, volume 138 of *Lecture Notes in Computer Science*, pages 172–193. Springer-Verlag, 1982.
22. C. Green. Application of Theorem Proving to Problem Solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 219–239, 1969.
23. Zohar Manna and Richard Waldinger. Fundamantals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, Aug 1992.
24. Manfred Broy and Peter Pepper. Program development as a formal activity. *IEEE Transactions on Software Engineering*, 7(1):14–22, jun 1981.
25. Johann M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2001.
26. Michael R. Lowry and Robert D. McCartney, editors. *Automating Software Design*. AAAI Press / The MIT Press, 1991.
27. Mark P. Jones and Luc Duponcheel. Composing monads. Research report YALEU/DCS/RR-1004, Yale University, Yale University, New Haven, Connecticut, Dec 1993.
28. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.
29. C. Luth and N. Neil Ghani. Composing monads using coproducts. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 133–144. ACM Press, 2002.
30. J. A. Goguen and R. M. Burstall. Introducing institutions. *Lecture Notes in Computer Science*, 164:221–255, 1984.
31. A. Kurz and R. Hennicker. On institutions for modular coalgebraic specifications. *Theoretical Computer Science*, 280(1-2):69–103, May 2002.
32. A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992. World Scientific Publishing Co.

33. S. Easterbrook. Domain modeling with hieararchies of alternative viewpoints. In *Proceedings of the First International Symposium on Requiremetns Engineering (RE-93)*, San Diego, CA, January 1993.
34. Steve Easterbrook and Marsha Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *International Conference on Software Engineering*, pages 411–420, 2001.
35. Steve Easterbrook and Mehrdad Sabetzadeh. Analysis of inconsistency in graph-based viewpoints: A category-theoretic approach. In *Proceedgings of The Automated Software Engineering Conference (ASE'03)*, pages 12–21, Montreal, Canada, October 2003.
36. J. Davis. Ptolemy ii - heterogeneous concurrent modeling and design in java, 2000.
37. J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994.