Domain Specific Model Composition Using A Lattice Of Coalgebras

Jennifer Streb, Garrin Kimmell, Nicolas Frisby and Perry Alexander
Information and Telecommunication Technology Center
The University of Kansas
{jenis,kimmell,nfrisby,alex}@ittc.ku.edu

Abstract

This paper presents a semantics for domain-specific modeling in support of system-level design of reactive, embedded systems. At the core of the approach is the use of a lattice of coalgebras to define the semantics of individual models and reusable specification domains. The lattice provides formal support for assessing the correctness of specification transformation. Additionally, using pullbacks and functors within the lattice provides semantic support for specification transformation and composition. Although developed for the Rosetta specification language, the lattice of coalgebras provides general semantic support that could be used to define any system-level design language.

1 Introduction

The Rosetta system-level design language and semantics [1, 2] are designed explicitly to support the needs of system-level design for reactive, embedded systems. To achieve this, Rosetta supports domain-specific, heterogeneous specification by providing a collection of *domains* that define vocabulary and semantics for domain specific design *facets* and a *domain lattice* that organizes domains to support abstract interpretation, transformation and composition of specifications. In this paper we present a semantic infrastructure supporting these system-level design activities.

2 Domain Specific Modeling Semantics

Facets are the fundamental unit of Rosetta specification representing one aspect or view of a multi-aspect system using domain-specific semantics. Information such as component function, performance constraints, and structure are represented using facet

models. Facets use a domain-specific semantic basis appropriate for the information being represented explicitly supporting heterogeneous modeling.

Figure 1 shows two facet models describing power and function for a simple signal processing component. The power model defines a simple activity-based power consumption model that observes changes in the output value. The function model defines the interface of a functional model whose body is omitted for space consideration. The system-level design objective is to define a single model that satisfies both specifications.

```
facet power
  (o::output top;
   leakage::design real;
   switch::design real)::state_based is
  export power;
  power::real;
begin
  power' = power + leakage +
             if event(o) then switch
               else 0
             end if;
end facet power;
```

```
facet interface function
   (i::input real; o::output real;
    clk :: in bit;
    uniqueID::design word(16);
    pktSize::design natural
   )::discrete_time is
  uniqueID :: word(16);
  hit :: boolean;
  bitCounter :: natural;
end facet interface function;
```

Figure 1: Rosetta specification fragments defining power consumption and functional models for a TDMA unique word detector.

2.1 Coalgebraic Semantics

Jacobs [3] observed that coalgebras provide an excellent basis for defining computations that are event driven and non-terminating. Every coalgebra ψ has the form:

$$\psi :: \mathcal{X} \rightarrow \mathcal{F}(\mathcal{X})$$

where $\mathcal{X}$ is the carrier and $\mathcal{F}$ defines constraints on the carrier. Running a coalgebraic system involves unfolding $\mathcal{X}$ with respect to ψ using an anamorphism defined for all such coalgebras. Specifically, given $\mathcal{X}$ the next state is $\psi(\mathcal{X}) = \mathcal{F}(\mathcal{X})$, the next is $\psi(\psi(\mathcal{X})) = \mathcal{F}(\mathcal{F}(\mathcal{X}))$, and so forth. This is precisely the semantics we want for event-driven, continuously operating systems. The application of ψ is associated with a system event, causing a state change when that event occurs.

It is useful to contrast the coalgebra with an algebra having the form:

$$\phi :: G(b) \rightarrow b$$

where b is the carrier and G defines constraints. Here, evaluation is takes the form of a fold of G over b using a catamorphism. Given any *finite* number of applications of G,

the value b can be calculated. Thus, if we use an algebraic model, we are restricted to examining only finite prefixes of potential state sequences.

The semantics of a Rosetta facet is denoted by coalgebra defining observations on changes over an abstract state. The facet's signature defines the coalgebra signature while its terms define its transformation by placing constraints on abstract state observations. For example, when denoting the power facet from figure 1, $\mathcal{F}$ is next observed by facet declarations while a is the facet state. Thus, given a state a, the sequence of states is observed as a, $next(a)$, $next(next(a))$ and so forth as expected. The terms in the facet constrain $next$ providing a definition of the sequential computation. The application of $next$ is constrained to occur only when an event is observed on the system output. This is embodied by the predicate event used in the definition.

The signature of coalgebra denoted by the power facet in figure 1 has the form:

<o,leakage,switch,power,s,next> :: $\mathcal{X} \rightarrow$
(state_time $\rightarrow$ **top**)
$\times$ (state_type $\rightarrow$ real)
$\times$ (state_type $\rightarrow$ real)
$\times$ state_type
$\times$ (state_type $\rightarrow$ state_type)

where o, leakage, switch, and power are observations on $\mathcal{X}$ defined in the facet while s and next are defined by the state_based domain. The corresponding types of those observations comprise the product.

What we are defining in the facet is the observation of the next $\mathcal{X}$ as observed by next. Examining the denotation of the power term reveals the coalgebraic nature of the facet:

power(next(s($\mathcal{X}$)) = power(s($\mathcal{X}$))
+ leakage(s($\mathcal{X}$))
+ **if** event(o(s($\mathcal{X}$))) **then** switch(s($\mathcal{X}$)) **else** 0 **end if**;

Here, the next function is being defined by embedding it in the observer power, commonly called a destructor. As long as $\mathcal{X}$ does not change, the observation remains the same. These characteristics – defining functions in destructors and observing state change – lead us to a coalgebraic semantics over the more traditional algebraic semantics [3].

In a Rosetta facet coalgebra, $\mathcal{X}$ is always held abstract with no associated concrete type. It is never directly visible to the specifier or even to the Rosetta facet that observes it. Instead, a Rosetta facet's state is denoted as an observation of $\mathcal{X}$. For example, if s :: state_type defines a state in some domain, then its value in the coalgebra is denoted $s(\mathcal{X})$ :: state_type – a function over the abstract state. If an item x :: integer is defined in a facet from the domain of s, then it is denoted $x(s(\mathcal{X}))$ :: integer.

Because facet state is simply an observation of $\mathcal{X}$, it is possible to define multiple state observations with multiple semantics. As sequencing of state is the critical element distinguishing models-of-computation, defining different state observations

results in multiple, heterogeneous models-of-computation. When models defining different state semantics observe the same abstract state, then those observations may be related. This is precisely what is needed in system-level design where the distinction between modeling domains is rooted in the underlying computational model.

2.2 Specification Composition

The primary specification composition mechanism in the Rosetta semantics is defined by the category theoretic *pullback* construction. In the traditional specification literature where algebraic specifications dominate presentations, the *coproduct* and *pushout* define specification composition [4]. As Rosetta models are coalgebraic, their duals, the *product* and *pullback*, define model composition. Intuitively, the pushout forms the union of two algebraic specifications defined around a collection of shared declarations. The pullback forms the intersection of two coalgebraic specifications, again defined around a collection of shared declarations. Making $\mathcal{X}$ the minimum shared declaration ensures that specifications involved in the pullback reference the same abstract state. They may observe that state differently, but they observe the same state.

Given two Rosetta models f_1 and f_2 a product is formed as the disjoint combination of f_1 and f_1 much like a record. In contrast, a pullback forms a product around a common, shared specification, d. We say that d is shared between specifications because when properties from f_1 and f_2 refer to declarations in d, they refer to the same element. Properties placed on symbols of d from each specification mutually constrain d implying f_1 and f_2 are no longer orthogonal.

In our power modeling example, we would like to understand how the function being performed impacts power consumption. Thus, a pullback is formed is formed from the power specification and the original function specification. This product model formed by the pullback is defined as a new Rosetta facet in Figure 2.

```
facet power_and_function
    (i :: input real; o :: output top; clk :: in bit; uniqueID::design word(16);
     pktSize :: design natural; leakage,switch::design real)::discrete_time is
  gamma(power(o,leakage,switch))
  * function(i,o,clk,uniqueID,pktSize);
```

Figure 2: Creating the composite specification by forming the product of the functional specification with the application of gamma to the power specification.

The product treats the discrete_time domain as a shared specification among the power and function models. The specification objects that t, delta and next refer to are shared between the specifications. Edges that indicate state change and power consumption are common to both components implying that processing in the functional specification results in power consumption in the power model. Any property defined on these items in one specification must be consistent with definitions in the other –

they are literally shared between the specifications. Other symbols remain orthogonal, but when referenced in properties relating them to shared symbols they are indirectly involved in sharing properties across domains.

2.3 The Domain Lattice

Because Rosetta facets are first-class items, they must have an associated type, called a *domain*. In figure 1 the domain of the function facet is discrete_time while the power facet is of type state_based. Rosetta domains encapsulate vocabulary and semantics for domain specific specification style. Each domain encapsulates units of semantic declarations, a model of computation, and a domain specific modeling vocabulary for reuse among similarly structured specifications.

Domains are simply distinguished facets that represent specifications that are extended. When a new domain is defined, it extends another domain in a manner identical to facet definition. The new domain is aptly call a subtype or subdomain of the original domain. For example, the discrete_time domain is defined as a subtype of state_based. The concepts of state and change present in the state_based domain are inherited and refined within the state_based domain. The distinction between defining a domain and defining a facet is the domain can be further refined to define facets or other domains.

The set of legally defined domains together with the homomorphism relationships resulting from extension define a partially ordered set $(D, \Rightarrow)$ referred to as the *domain lattice*. The domain lattice obeys the formal definition of a lattice requiring the definition of meet ($\sqcap$), join ($\sqcup$), the minimum domain the maximum domain. **null** defines primitive Rosetta semantics including $\mathcal{X}$ and is the least domain in D with all domains inheriting from it. Similarly, **bottom** is the greatest domain and inherits from all domains, making it inconsistent.

For any domain pair D_1 and D_2, $D_1 \sqcap D_2$ and $D_1 \sqcup D_2$ are defined as the least common supertype and greatest common subtype. The existence of **null** and **bottom** ensures that every domain pair will have at least one common superdomain and subdomain. Thus, $(D, \Rightarrow)$ defines a lattice.

2.4 Specification Transformation

A *functor* is a function specifying a mapping from one domain to another. The primary role of functors is to transform a model in one domain into a model in another. Viewing each domain and facets comprising its type as a subcategory of the category of all Rosetta specifications, a functor is simply a mapping from one subcategory to another corresponding to the classic definition of functors in category theory.

When defining domains by extension, two kinds of functors result. Instances of concretization functors, Γ, are defined each time one domain is extended to define another. Abstraction functions, A, are the dual of concretization functions and are known to exist for each Γ due to the multiplicative nature of extension. So, Γ instances move down in abstraction while A instances move up. Each arrow in the domain

lattice moving from one domain down to another defines both an instance of Γ and A. However, A and Γ do not form an isomorphism because A is lossy – some information must be lost or A cannot truly be an abstraction function.

3 Implications of the Domain Lattice

A major application of Rosetta functors is to add or remove detail to support predictive analysis and specification composition. Thus, it is critical to assure that functor application results in correct models. To achieve this, we view functor application from the perspective of abstract interpretation [5] where programs and specifications are statically analyzed by focusing only on necessary details. An abstraction is *safe* when the model resulting from it is faithful to the original model.

Because Rosetta focuses on domain-specific specification composition, we need to verify the safety of functors moving specifications within the lattice. More specifically, we want to verify that by moving a specification or model between Rosetta domains we do not sacrifice correctness. One technique common in the abstract interpretation community is establishing a *Galois connection* [6] between domains in the lattice.

A Galois connection (C, α, γ, A) exists between two complete lattices (C, $\sqsubseteq$) and (A, $\sqsubseteq$) if and only if $\alpha : C \rightarrow A \wedge \gamma : C \leftarrow A$ are monotone functions that satisfy $\gamma \circ \alpha \sqsupseteq \lambda c.c$ and $\alpha \circ \gamma \sqsubseteq \lambda a.a$. Typically, α is an abstraction while γ is an associated concretization. Within the Rosetta domain lattice, the initial focus on the functors A and Γ formed when one domain is extended to define another. This gives the Galois connection we are initially interested in the form (D, A, Γ, D).

The extension of one domain to form another gives us a concretization functor, Γ, that defines a homomorphism between domains. Because Γ is multiplicative, we know from lattice theory that an inverse abstraction functor, A, exists and can be derived from it. With A and Γ and the homomorphism, we can define a Galois connection between any Rosetta domain, D_0, and any of its subdomains, D_1, as $(D_0, A_1, \Gamma_1, D_1)$. Knowing the Galois connection exists we are guaranteed any transformation between D_0 and D_1 using Γ of A is safe. We are also guaranteed that the original model is an instance of the abstract model and the abstract model is truly an abstraction.

We also know that the *functional composition* of two Galois connections is also a Galois connection [6]. Formally, if $(D_0, A_1, \Gamma_1, D_1)$ and $(D_1, A_2, \Gamma_2, D_2)$ are Galois connections then $(D_0, A_2 \circ A_1, \Gamma_1 \circ \Gamma_2, D_2)$ is also a Galois connection. Not only can we assure safety between any domain and its subdomain, but we are also guaranteed safety of any transformation within the entire Rosetta domain lattice that follows abstraction links.

4 Related Work

The use of products for specification composition is well established in the literature for both specification [4] and synthesis [7]. Although the coalgebra and pullback are less frequently used, there has been work using coalgebras to define composable specification systems [8] and the use of coalgebraic specification is seeing acceptance in the specification community [9]. Brevity prevents complete discussion of coalgebraic techniques – see Jacobs' and Rutten's excellent tutorial for more details [3].

UML meta-models define semantics that has been exploited for domain specific tool development and model-integrated design [10]. The model-integrated approach reflects our approach to model refinement and abstraction as the central features in design synthesis and analysis respectively. The model-integrated approach uses UML as its modeling language, although like the coalgebraic semantics presented here it should not be limited to UML models.

Viewpoints are a software specification technique applied to domain specific modeling [11]. Viewpoints are less formal than Rosetta and focus primarily on software systems. However, interaction between models searching for inconsistencies has been explored extensively giving Viewpoints a similar system-level focus[12].

An alternative approach using operational modeling is the Ptolemy [13] project. Ptolemy (now Ptolemy Classic) and Ptolemy II successfully compose models using multiple computation domains into executable simulations and software systems. Ptolemy II introduces the concept of a system-level type [14] that provides temporal information as well as traditional type information. Like Rosetta, Ptolemy II uses a formal semantic model for system-level types. Unlike Rosetta, Ptolemy models are executable and frequently used as software components.

5 Discussion

This paper provides an overview of the approach to domain specific model composition embodied in the Rosetta specification system. With a formal semantics for heterogeneous models supporting composition and transformation functions, it becomes possible to define heterogeneous models, compose them and generate abstract analysis models. Further details are available in *System-Level Design with Rosetta* [2] one of several overview papers [15].

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] Perry Alexander. *System-Level Design with Rosetta*. Morgan Kaufmann Publishers, Inc., 2006.

[3] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. EATCS Bulletin 62, 1997. p.222-259.

[4] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. EATCS Mongraphs on Theoretical Computer Science. Springer–Verlag, Berlin, 1985.

[5] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 100.

[6] Flemming Nielson, Hanne RIIS Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 2005.

[7] Douglas R. Smith. Constructing specification morphisms. *Journal of Symbolic Computation*, 15:571–606, 1993.

[8] A. Kurz and R. Hennicker. On institutions for modular coalgebraic specifications. *Theoretical Computer Science*, 280(1-2):69–103, May 2002.

[9] J. Rothe, H. Tews, and B. Jacobs. The coalgebraic class specification language CCSL. *Journal of Universal Computer Science*, 7(2):175–193, March 2001.

[10] A. Misra, G. Karsai, J. Sztipanovits, A. Ledeczi, and M. Moore. A model-integrated infomration system for increasing throughput in discrete manufacturing. In *Proceedings of The 1997 Conference and Workshop on Engineering of Computer Based Systems*, pages 203–210, Montery, CA, March 1997. IEEE Press.

[11] S. Easterbrook. Domain modeling with hiearchies of alternative viewpoints. In *Proceedings of the First International Symposium on Requiremetns Engineering (RE-93)*, San Diego, CA, January 1993.

[12] 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.

[13] 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.

[14] Edward A. Lee and Yuhong Xiong. System-level types for component-based design. Technical report, University of California at Berkeley, February 2000.

[15] 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.