

Using a Lattice of Coalgebras For Heterogeneous Model Composition

Jennifer Streb and Perry Alexander

*Information and Telecommunication Technology Center
The University of Kansas
{jenis,alex}@ittc.ku.edu*

Abstract

System-level design is characterized by a need to bring together concurrent information from numerous domains to assess the impact of local decisions on global properties. As such, to support system-level design a language must minimally address heterogeneous specification, specification transformation and specification composition. We propose a semantics for defining these operations based on a lattice of coalgebraic system models. Within this lattice we can provide formal definitions for soundness using Galois connections, specification transformation using functors, and specification composition using the classical sum and product operations from category theory. Embodied in the Rosetta specification language, this semantics has proved useful in assessing system level properties such as power consumption and security. This paper overviews the semantics and provides a simple example of its use.

Keywords: Heterogeneous Specification, Coalgebraic Semantics, Specification Composition

1 Introduction

The essence of system-level design is bringing together information from multiple, concurrent domains to assess global effects of local decisions. Thus, for any language or semantic system to address system-level design needs it must *support the representation of heterogeneous information and support composition of information across domains*. The Rosetta language and semantics [1,2] are designed explicitly to support the needs of system-level design. It supports heterogeneous specification by providing a collection of *domains* that provide vocabulary and semantics for writing specifications and the *domain lattice* that supports transforming and composing information across domains. In this paper, we will eschew discussion of the main body of the Rosetta syntax, instead concentrating on the semantic infrastructure required to construct the domain lattice, transform, and compose models.

2 Background

Models in Rosetta are referred to as *facets* and are the fundamental unit of specification. Each facet represents one aspect or view of a multi-aspect system. In-

*This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

formation such as component function, performance constraints, and structure are all represented using facet models. The key is that each facet represents a system from one perspective using a semantic basis appropriate for the information being represented. A complete system model composes facets representing multiple perspectives into a composite system model. The domain lattice provides support for these operations while coalgebras form the semantic basis for individual models.

2.1 The Domain Lattice

Vocabulary and semantics for defining facets are provided by *domains*. Each domain provides to varying degrees units of semantic representation, a model of computation, and a domain specific modeling vocabulary. Ideally, a domain defines a collection of definitions that characterize a particular computation or modeling style. This may vary from simply unit-of-semantic definitions like the `state_based` domain that defines a simple stateful computation model to complex engineering domains like the `digital` domain that provides a complete semantics for writing digital system models.

When a facet is defined, it is declared as an element of a domain and inherits all of that domain's declarations. Formally, the new facet *extends* a domain to define a new model. For example, if a `register` facet is defined of type `state_based`, then the concepts of state, change and event are available as a built-in part of the specification vocabulary. In this way, a facet's associated domain defines its type and the type associated with a domain is the collection of consistent facets that can be defined by extending it.

When a new domain is defined, it is declared as a subdomain of an existing domain. Like facets, the new domain extends the original domain and inherits all of that domain's declarations. If a new `discrete_time` domain is defined as a subtype of `state_based`, then the notions state and change are inherited and refined within the new domain. The distinction between defining a domain and defining a facet is the domain can be further refined to define facets or other domains. When a facet is defined, it cannot be extended and defines a leaf in the domain.

The collection of domains and the extensions used to define them define a tree that is referred to in Rosetta as the *domain lattice*. The set of domains, D , together with the homomorphism relationships resulting from extension define a partially ordered set (D, \Rightarrow) . Join (\sqcup) and meet (\sqcap) can subsequently be defined as the least common supertype and greatest common subtype of any pair of domains. It can easily be proved that any domain pair will in fact have a least common supertype and a greatest common subtype. The `null` domain is the least domain in the collection and all domains inherit from it. `bottom` is the greatest domain and inherits from all domains making it inconsistent. Specifically:

$$\forall f :: \text{facet} \cdot \text{bottom} \Rightarrow f \wedge f \Rightarrow \text{null}$$

Including `null` and `bottom` with the partially ordered set (D, \Rightarrow) defines a lattice whose top and bottom elements are `null` and `bottom` respectively:

$$(D, \Rightarrow, \sqcup, \sqcap, \text{null}, \text{bottom})$$

2.2 Coalgebraic Semantics

The domain lattice organizes domains but says nothing about the semantics of domains and facets. A facet’s underlying semantics are denoted by a coalgebra [3] defining observations on an abstract state, \mathcal{X} . The signature for a general coalgebra is:

$$\langle x, y, z, s \rangle :: \mathcal{X} \rightarrow T_x \times T_y \times T_z \times T_s$$

where x , y , z , and s are observations on \mathcal{X} and T_x through T_s are the types of those observations. When s is treated as state, this signature has the form of a classic Rosetta facet coalgebra. For any observation, x , made relative to state, the associated type will be:

$$T_s \rightarrow T_x$$

a functional mapping from a state value to a value of the type associated with the observation. One particularly important observation is the next state given by $\text{next}(s)$:

$$T_s \rightarrow T_s$$

mapping one system state observation to another.

A facet’s signature defines its associated coalgebra signature and its terms define the coalgebra function by placing constraints on individual observations. This denotation is relatively straightforward and will not be discussed in detail here. It suffices to understand that the parameters and declarations defined in a facet signature define observations on \mathcal{X} .

We choose coalgebras over their better known duals, algebras, due to the non-terminating and heterogeneous nature of the types of systems we model. Coalgebras are more natural than algebras for representing non-terminating systems. The inductive proof theory associated with algebras requires a base case or initial state that may not exist in many embedded systems. As stream transformers, coalgebras and their associated proof techniques are well equipped to deal with reactive, non-terminating embedded systems.

The heterogeneous nature of system-level specifications requires that multiple computation models be considered during modeling and analysis. In the coalgebra, \mathcal{X} can be held abstract with no associated concrete type. In this case, Rosetta states simply become observations of the abstract state making multiple simultaneous state observations possible. Furthermore, by defining relationships between states in different domains, one can relate information associated with one state observation to information associated with other state observations. This critical feature allows determination of when information observed in one domain impacts information observed in another.

3 Lattice of Coalgebras

The result of the domain lattice definition and the coalgebraic semantics of facets is a *lattice of coalgebras* that serves as the underlying Rosetta semantics. With this semantic basis, we can now put the domain lattice to work defining specification transformation and composition. Additionally, the lattice facilitates establishing the

soundness of such operations using Galois connections. Each of these is critical to supporting model heterogeneity and composition necessary for system-level design. Figure 1 shows a part of the lattice of domains defined for traditional Rosetta specifications.

Functors and products discussed in this section and homomorphisms discussed earlier are examples of reflective Rosetta operations making up the *facet algebra* used to compose specifications. Products compose specifications and functors transform specifications to define new specifications. Homomorphism is a relation over specification pairs. Other important facet algebra operations include equivalence (isomorphism), relabeling and instantiation.

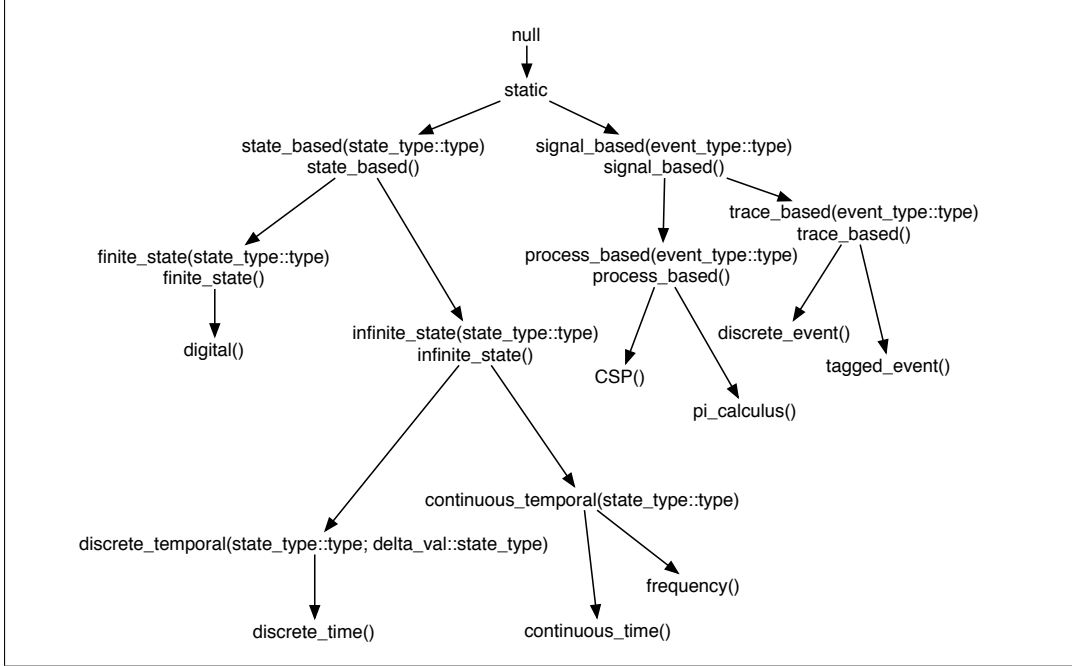


Fig. 1: The Rosetta Domain Lattice

3.1 Functors and Specification Transformation

A *functor* in the domain lattice is a function specifying a mapping from one domain to another. The primary role of functors in the domain lattice 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. Any model in the original category can be transformed into a model in the second. This corresponds 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. In Figure 1 Each arrow 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.

3.2 Safety and Galois Connections

Abstract interpretation [4] provides a capability for focusing analysis by eliminating unneeded detail from a specification. Among the most challenging problems in abstract interpretation is assuring that once the abstraction is performed the resulting model is faithful to the original. This is the notion of *safety* – assuring that when an abstraction is performed, the information retained is correct.

In the case of the Rosetta domain lattice, we need to verify the safety of functors moving specifications up and down the lattice. More specifically, we want to verify that by moving a specification or model between Rosetta domains we do not sacrifice correctness. Establishing a *Galois connection* [5] between domains in the lattice provides exactly this assurance.

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:

- (1) $\gamma \circ \alpha \sqsubseteq \lambda c.c$
- (2) $\alpha \circ \gamma \sqsubseteq \lambda a.a$

The two conditions above express that we do not sacrifice safety by going back and forth between the two domains although we may lose precision. For our purposes the notion of precision isn't important. We simply want to assure that by moving back and forth between domains we maintain a safe approximation of the original model.

Condition 1 states that abstraction (α) followed by concretisation (γ) of a specification or model results in either the same specification or model, or one *more abstract* than the original yet still safe. Condition 2 states that concretisation followed by abstraction of a specification or model will result in either that same specification or model, or one *less abstract* than it.

We have stated that extension of one domain to form another gives us a concretisation function, Γ , that defines a homomorphism between domains. Because Γ is multiplicative, we are assured by nature of the lattice that an inverse, A , exists and can be derived from it. Thus, for any domain pair that is ordered by the lattice, we can define functors that move a specification between them.

With A and Γ and the homomorphism, we can now define a Galois connection between any Rosetta domain, D_0 , and any of its subdomains, D_1 , as $(D_0, A_1, \Gamma_1, D_1)$. With the existence of the Galois connection we can now assure safety of any transformation between these two domains. Furthermore, the “functional composition” of two Galois connections is also a Galois connection [5]. 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. This is important because not only can we assure

safety between any domain and its subdomain, but we can also assure safety of any transformation throughout the entire domain lattice.

The existence of the Galois connections are extremely advantageous. They allow us to perform abstract interpretation with the certainty of the entire system still functioning as expected. They also allow multiple perspectives of a specification with the assurance of safety throughout. Additionally, we are guaranteed safety of any transformation within the entire Rosetta domain lattice due to the ability to functionally compose Galois connections.

3.3 Specification Composition

Although useful by itself, heterogeneous specification truly becomes useful only when specifications can be combined to understand interactions. The domain lattice as presented thus far supports writing specifications using multiple models of computation and satisfies our goal of heterogeneity. Looking at the domain lattice from a categorical perspective enables using standard product and sum operations to perform composition.

The primary specification composition mechanism in the Rosetta semantics is the category theoretic concepts of *product* and *pullback*. A specification product is simply a pair of specifications that simultaneously describe a system. Because the specifications simultaneously hold, they must be mutually consistent. Mutual consistency between specifications in different domains implies consistency among heterogeneous specifications – precisely a goal of system-level design.

The question becomes how two specifications from different domains can ever be inconsistent. Specifically, if two domains are distinct, then properties defined in one domain cannot reference the other because there are no shared symbols. The answer lies in functors used to move information from one domain to another and in the pullback used to define the product.

In the traditional specification literature where algebraic specifications dominate presentations, the *coproduct* is used for specification composition. Certainly the discussion of specification product mimics that discussion as should be expected. We use the product here because the Rosetta semantics is coalgebraic rather than algebraic. Duality between algebras and coalgebras suggest the product is more appropriate. Analytic work verifies this suggestion.

Formally, Given two models A and B the product is formed from the disjoint combination of A and B . As the composition is disjoint, there is no possibility of interaction. A pullback is a special construction for forming a product where each element is derived from a common specification, C that is frequently referred to as the “shared part”. As the name implies, the elements of C are shared between specifications in that when properties from A and B refer to elements of C , they are the same element. Properties placed on symbols of C from each specification mutually constrain C and A and B are no longer orthogonal. Thus we have heterogeneous specifications interacting.

4 Application Methodology

With semantic elements in place, we can outline a methodology for their application in a specification process. We start by defining specifications for system facets of interest. Functors defined by homomorphisms in the domain lattice are then used to move specifications to appropriate domains for composition and analysis. We then compose specifications using facet product operations. Finally, we verify the consistency of resulting specifications using simulation, theorem proving and model checking, and synthesize specifications into implementations using synthesis and compilation techniques. The resulting methodology is both effective and mathematically sound due to the Galois connections defined over the domain lattice.

4.1 Define Facet Specifications

The initial specification task is selecting specification domains for each relevant system requirements model and defining facet specifications for those models. A classic example from the Rosetta literature we use here is power-aware design where implementation fabric selection is a design decision made by understanding the interaction between functional requirements and power constraints. Example specifications are shown in Figure 2.

To effectively select domains and write specifications, the goals of the system analysis processes must be known. One must begin with the end in mind regardless of the system analysis methodology. Rarely does any system design process, specification or otherwise, yield anything useful that was not planned from the beginning.

<pre> facet power (o::output top; leakage,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; </pre>	<pre> 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; </pre>
--	---

Fig. 2: Rosetta specification fragments defining power consumption and functional models for a TDMA unique word detector. Due to space constraints, only the interface is shown for the functional model.

4.2 Transform Specifications for Analysis

Moving specifications using functors accomplishes two basic tasks: (i) moving a specification to an analysis domain; or (ii) moving a specification to a domain for composition with another specification. In the former case, specifiers move definitions from descriptive domains to new domains better equipped for analysis. In the latter case, specifiers move definitions to new domains where specification composition yields new, more detailed specifications. The Galois connections established over the Rosetta domain lattice guarantee the safety of abstraction and concretization functors.

Specifications for our example component exist in different domains that could be composed immediately using the product operation. In this case however, a more accurate performance prediction can be made if the specifications are composed in the same domain. Thus, we have three options: (i) Move the power specification to the `discrete_time` domain; (ii) move the functional specification to the `state_based` domain; or (iii) move both specifications to a common, intermediate domain. For this example, we choose to move the power specification into the `discrete_time` domain by applying a concretization functor. This decision is motivated by the existence of a `discrete_time` simulation environment exists that can be used to analyze the resulting specifications. The built-in concretization function for moving `state_based` specifications to `discrete_time`, `gamma`, is used for this task:

```
gamma(power());
```

The beauty of using `gamma` functions defined by the domain lattice is that if the extension between domains used to form the concretization function is consistent, `gamma` exists, `alpha` exists and the Galois connection assures their soundness. Thus, when moving the `power` specification above, we are certain that the resulting specification in its new domain will be sound with respect to the original specification.

4.3 Compose Specifications

After transforming specifications into appropriate domains, the facet product is used to compose specifications. The specification product is formed from the power specification in the `discrete_time` domain and the original function specification. This new product facet is defined in Figure 3 using the application of `gamma` and the product operation.

```
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);
```

Fig. 3: Creating the composite specification by forming the product of the functional specification with the application of `gamma` to the power specification.

The product does far more than simply pair the specifications. Both specifications inherit a definition of time from the `discrete_time` domain. Specifically, a time value (`t`), quanta (`delta`) and next time function (`next`) are defined in `discrete_time`.

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 shared properties across domains.

4.4 Verification and Synthesis

With the composite model constructed, verification and synthesis are performed to predict system behavior and generate system components. Any system with a semantics compatible with the resulting Rosetta model can be used for analysis. In this case, simulation is performed by defining a domain associated with the simulator and using a functor to make the transformation. To preserve soundness, a verification obligation remains to show that simulator behavior is consistent with the domain describing it. Although non-trivial, this analysis is performed once per tool.

5 Related Work

Possibly the most visible work in heterogeneous modeling is the Universal Modeling Language (UML) [6,7,8]. Initially developed for object-oriented software systems, UML has expanded to digital hardware, embedded software and most recently system-level modeling. UML employs a collection of diagramming techniques whose semantics are customised using *profiles* specific to different domains.

UML meta-models provide additional semantics for heterogeneous systems that has been exploited for domain specific tool development and model-integrated design [9]. 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 [10,11,12,13] are a software specification technique where multiple perspectives of a software system are recorded. 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 [14,15,16].

An alternative approach using operational modeling is the Ptolemy [17,18] project. Ptolemy (now Ptolemy Classic) and Ptolemy II successfully compose models using multiple computation domains into executable simulations and actual software systems. Ptolemy II introduces the concept of a system-level type that provides temporal information as well as traditional type information. Specifically, the temporal characteristics of a type become a part of its description. 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.

6 Discussion

This paper overviews the approach to multi-paradigm specification embodied in the Rosetta specification system. We began with the assertion that Rosetta facets would be denoted as coalgebras, organized around domains situated in a lattice defined by homomorphism. Using the lattice as a basis, we discussed how homomorphisms define a Galois connection that assures the safety of information across specification transformations. Using the lattice and coalgebra semantics, we discussed how func-

tors are used to move specifications between domains and how products are used to compose specifications in a well-defined, controlled manner.

Although space prevents discussing details of Rosetta or the Rosetta specification system, this overview should motivate further study of the lattice-of-coalgebras approach. We assert that the approach has potential beyond the Rosetta specification system and our initial results in digital design, power-aware design and security suggest broad applicability.

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] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.
- [5] Flemming Nielson, Hanne RIIS Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 2005.
- [6] The UML Group. *UML Metamodel*. Rational Software Corporation, Santa Clara, CA, 1.1 edition, September 1997. <http://www.rational.com>.
- [7] The UML Group. *UML Semantics*. Rational Software Corporation, Santa Clara, CA, 1.1 edition, July 1997. <http://www.rational.com>.
- [8] A. Evans and S. Kent. Core meta-modelling semantics of UML: The pUML approach. In *Proceedings of UML 99*, October 1999.
- [9] A. Misra, G. Karsai, J. Sztipanovits, A. Ledeczi, and M. Moore. A model-integrated information system for increasing throughput in discrete manufacturing. In *Proceedings of The 1997 Conference and Workshop on Engineering of Computer Based Systems*, pages 203–210, Monterey, CA, March 1997. IEEE Press.
- [10] Anthony Finkelstein, Steve Easterbrook, Jeff Kramer, and Bashar Nuseibeh. Requirements engineering through viewpoints. Technical report, Imperial College, Department of Computing, 180 Queen's Gate, London SW7 2BZ, 1992. <http://citeseer.nj.nec.com/finkelstein92requirements.html>.
- [11] 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.
- [12] S. Easterbrook. Domain modeling with hierarchies of alternative viewpoints. In *Proceedings of the First International Symposium on Requirements Engineering (RE-93)*, San Diego, CA, January 1993.
- [13] Julio Cesar Sampaio do Prado Leite. Viewpoints on viewpoints. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 285–288, 1996.
- [14] Steve Easterbrook and Mehrdad Sabetzadeh. Analysis of inconsistency in graph-based viewpoints: A category-theoretic approach. In *Proceedings of The Automated Software Engineering Conference (ASE'03)*, pages 12–21, Montreal, Canada, October 2003.
- [15] Steve Easterbrook and Marsha Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *International Conference on Software Engineering*, pages 411–420, 2001.
- [16] S. Easterbrook and B. Nuseibeh. Managing inconsistencies in evolving specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE-95)*, pages 48–55, York, UK, April 1995. IEEE Press.
- [17] 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.
- [18] J. Davis. Ptolemy ii - heterogeneous concurrent modeling and design in java, 2000.