Title: **Heterogeneous Modeling using a Lattice of Coalgebras**

Technical Area: **TA2**

Organizations: University of Cincinnati
University of Kansas
University of California @ Riverside

Technical Point of Contact: Philip A. Wilsey
Dept of EECS, PO Box 210030
Cincinnati, OH 45221–0030
(513) 556–4779 (voice)
(513) 556–7326 (fax)
philip.wilsey@uc.edu

1 Introduction

This document addresses **TA2** and presents ideas for the multi-resolution modeling of complex heterogeneous networked multi-scale systems-of-systems (SoS). The proposed approach uses a *Lattice of Coalgebras* to provide a framework that enables the design and quantitative assessments of SoS systems. The use of a Lattice of Coalgebras promotes the ability to bring together heterogeneous multi-domain models into a common framework to access the impact of local decisions on global properties. By using a lattice-based approach we can use Galois connections and functors to develop *safe* transformations between disparate domains. These safe transforms then allow the alignment of multi-domain system models into a unifying SoS model for analysis. Furthermore, we propose using coalgebras (over algebras) due to the non-terminating and heterogeneous nature of SoS. 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.

1.1 A Lattice of Domains

Vocabulary and semantics for defining models in a specific domain can be thought of as a *modeling domain* or simply *domain*. Each domain provides to varying degrees units of semantic representation, (i) a model of computation, and (ii) a domain specific modeling vocabulary. Ideally, a domain defines a collection of definitions that characterize a particular computation or modeling style.

When a new model is written, it *extends* a domain using that domain as a semantic basis an to define vocabulary. In this way, the domain defines the type of a model. For example, if a simulator is defined of type $\mathsf{state\_based}$, then the concepts of state, change and event are available as a built-in part of the specification vocabulary.

When a new modeling domain is defined, it is typically a subdomain of some existing modeling domain. Like a model, the new domain extends the original domain and inherits all of that domain's declarations. For example, if a new $\mathsf{discrete\_time}$ domain is defined as a subtype of $\mathsf{state\_based}$, then the notions state and change are inherited and refined within the new domain.

Modeling domains and their associated extensions define a lattice we will refer to as a *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:

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

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

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

1.2 Coalgebraic Semantics

The semantics of both domains and models is denoted by a coalgebra [**?**] defining observations on an abstract state, $\mathcal{X}$. The signature for a 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 $\mathsf{next(s)}$:

$$T_s \rightarrow T_s$$

mapping one system state observation to another.

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. Specific 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.

2 Lattice of Coalgebras

Using this semantic basis, we can use the domain lattice to define specification transformation and composition. Additionally, the lattice facilitates establishing the safety of such operations using Galois connections. Each of these is critical to supporting model heterogeneity and composition necessary for system-level design.

Figure 1: The Rosetta Domain Lattice

2.1 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. Γ instances move down in abstraction while A instances move up. Each extension between domains 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.

2.2 Safety and Galois Connections

Abstract interpretation [**?**] 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. Establishing a *Galois connection* [**?**] 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:

$$\gamma \circ \alpha \sqsupseteq \lambda c.c \tag{1}$$

$$\alpha \circ \gamma \sqsubseteq \lambda a.a \tag{2}$$

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 **??** states that abstraction (α) followed by concretization (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 **??** states that concretization 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 concretization 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 the domain lattice, A, Γ and the homomorphism, we can now define a Galois connection between any 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 [**?**]. 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.

2.3 Specification Composition

The primary specification composition mechanisms in the domains lattice are the *product* and *pullback* constructions [**?**]. 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.

In the traditional formal specification literature where algebraic semantics dominate, the *coproduct* and *pushout* are the dominant specification composition constructions [**?**, **?**, **?**]. Traditionally, a pushout of specifications forms the union of two specifications where shared specification that is jointly constrained in both specifications. With coalgebras, the product is the appropriate composition operator as we are looking for an interaction.

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. The elements of C are shared between specifications – 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.