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

Technical Area: **TA2**

Organizations: University of Cincinnati, Philip A. Wilsey
University of Kansas, Perry Alexander
University of California, Riverside, Nael Abu-Ghazaleh

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

We propose a framework and associated tools for multi-resolution modeling and analysis of complex heterogeneous networked multi-scale systems-of-systems (SoS) targeting Technical Area **TA2**. The proposed approach uses a *Lattice of Coalgebras* to provide a framework that enables the design and quantitative assessments of SoS systems. The Lattice of Coalgebras enables the bringing together of heterogeneous multi-domain models into a common framework to assess the impact of local decisions on global system properties. In particular, we propose to use Galois connections and functors to develop *safe* transformations between disparate domains and scales. These safe transformations enable the alignment of multi-domain system models into a unifying SoS domain for analysis. Furthermore, we propose using coalgebras (over algebras) due to the non-terminating and heterogeneous nature of SoS since they are more natural than algebras for representing such 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

The vocabulary and semantics for defining models in a specific domain can be thought of as a *modeling domain* or simply *domain*. Each domain provides 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 to define vocabulary. In this way, the domain defines the type of a model. For example, if a simulator 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.

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 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.

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:

$$\forall m : model \cdot \mathsf{bottom} \Rightarrow m \wedge m \Rightarrow \mathsf{null}$$

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 [1] 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 coalgebraic model. 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 $\mathsf{next(s)}$ of type $T_s \rightarrow T_s$ 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.

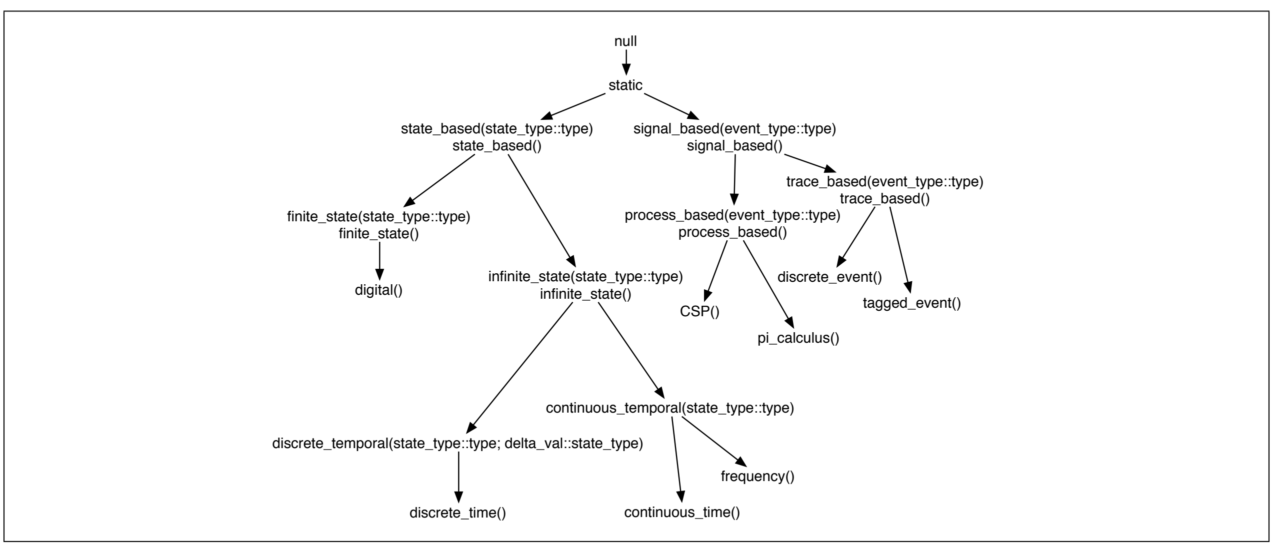

Figure 1: Example modeling domain Lattice

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 comprising its type as a subcategory of the category of all models, 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, namely: *concretization functions* and *abstraction functions*. 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 [2] 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* [3] 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:

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

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

These conditions state 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 is not 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 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 2 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 [3]. 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 [4]. 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 *co-product* and *pushout* are the dominant specification composition constructions [4, 5, 6]. 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.

3 Systems of Systems for EW and ISR

A mathematics for transforming and composing models with formal guarantees is precisely the semantic system needed for modeling systems-of-systems. Specifically, we can address three important problems: (i) interfacing domains with different semantics; (ii) moving models to a new semantic domains; and (iii) reifying or synthesizing simulation frameworks from coalgebraic specifications. Using formal tools, we can implement these capabilities in models whose semantics can be fully or partially verified.

3.1 Abstraction and Concretization

Defining abstraction and concretization morphisms among modeling domains allows moving data across semantic boundaries. In the context of systems-of-systems, this corresponds to directly connecting or otherwise allowing interactions among models that may have quite different underlying semantics. In an intelligence surveillance and reconnaisance (ISR) domain this allows modeling the transformation of continuous sensor data into discrete events useful for an analyst; understanding interactions between seemingly isolated system components; and modeling interacting systems whose timing characteristics differ radically.

The multiplicative nature of functors means that connections can be established between literally any set of models in the domain lattice. The domain lattice formalism allows formal modeling of data fusion in ISR systems where multiple information flows occur at different abstraction levels with disparate underlying semantics. Similarly, we can treat information loss due to communication errors or countermeasures as an abstraction. Systems see the lossy, abstracted signal. Then a concretization function is used to explore potential implications of information loss over a space of potential outcomes. As a result, an analyst can understand the risks and costs associated with decision in the context of information loss.

3.2 Model Transformation and Composition

Functors among domains are frequently used to transform models themselves. Instead of communicating among models in different domains, we can transform models themselves to different domains. For example, one can transform an analog component model into a digital model of the same component. That model may then be simulated in a traditional digital simulator with full knowledge that the abstraction is sound. The same model could be concretized back to the analog domain allowing exploration of the consistency of the transformation. by determining if the abstraction removes critical information.

Model composition is a specific instance of model transformation. If two models exist in two different domains, transforming both into a common domain forming a pullback in the domain lattice results in a single model with properties of both original models. This approach allows modeling a system using a collection of models representing different modeling perspectives. In an ISR domain involving a software defined radio or network, a security model defining confidentiality and integrity properties can be written and composed with a system model for analysis [7]. Similarly, an EW system may be modeled as a collection of separate operational modes that are then composed for analysis or for a complete system definition. Finally, in an ISR domain different models of the same adversary can be composed to form a more complete representation.

3.3 Coalgebra Reification

Among the more exciting applications is reification or synthesis of simulators and systems from coalgebraic models. Monads have become a common programming pattern in functional languages implementing options, state, IO and domain specific languages. Monads are algebraic structures with specific properties that allow execution, thus writing a algebraic specification can result in an executable monadic specification [8, 9].

Comonads are to monads as coalgebras are to algebras. They are executable structures that can be synthesized from coalgebras. We have explored simulation comonads and their synthesis from coalgebras with some initial success. While further investigation is needed, we believe that synthesizing comonadic simulation engines from coalgebras is feasible. Thus, a domain in the domain lattice can be synthesized into a simulator for models in that domain. One need not synthesize each model, but only the underlying domain and specification morphisms associated with it.

We see applications in both the ISR and EW domains. In ISR synthesis of MAC policies from security models; synthesis of security protocol implementations from analyzed models; and filters from models of information abstraction. In EW synthesis of implementations and simulations from waveforms and adversary models in a manner similar to software defined radios.

References

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

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

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

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

[6] Douglas R. Smith. KIDS: A Semiautomatic Program Development System. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[7] Garrin Kimmell. *System Synthesis from a Monadic Functional Language*. PhD thesis, The University of Kansas, Lawrence, KS USA, December 2008.

[8] G. Kimmell, E. Komp, G. Minden, and P. Alexander. Synthesizing Software Defined Radios from Rosetta Specifications. In *Proc. of the Forum on specification and Design Languages (FDL'08)*, pages 23–25, Stuttgart, Germany, September 2008.

[9] Wesley Peck. *Hardware/Software Co-Design via Specification Refinement*. PhD thesis, The University of Kansas, Lawrence, KS, December 2011.