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

August 11, 2006

Abstract

System-level design is characterized by a need to bring together concurrent information from numerous domains to assess the cross-domain impact of local decisions. To support system-level design a language must address domain-specific, heterogeneous specification. However, specification and specification transformation play equally important roles. We propose a semantics for denoting transformation and composition operations based on a lattice of coalgebraic system models. Within this lattice we can provide formal definitions of model transformation using functors, transformation safety using Galois connections, and model composition using the classical sum and product operations from category theory. Embodied by the Rosetta specification language, we have used this semantic system to specify and verify system-level properties such as power consumption and security.

1 Introduction

For any language or semantic system to address system-level design needs it must support *representation of heterogeneous information* and *composition of information from multiple domains*. The Rosetta system-level design language and semantics [1, 2] are designed explicitly to support the needs of system-level design. Specifically, Rosetta provides support for domain-specific, heterogeneous specification. It achieves this by providing a collection of *domains* that define vocabulary and semantics for domain specific specification languages and the *domain lattice* that supports metamodeling along with transformation and composition of specifications across domains. In this paper we concentrate on the semantic infrastructure required to construct the domain lattice, transform, and compose models rather than the semantic details of the Rosetta language.

2 Domain Specific Modeling Semantics

Facets are the fundamental unit of Rosetta specification representing domain-specific models. Each facet represents one aspect or view of a multi-aspect system. Information such as component function, performance constraints, and structure are represented using facet models. Each facet represents a system from one perspective using a domain-specific semantic basis appropriate for the information being represented. A complete system model composes facets representing multiple perspectives into a composite system model.

Coalgebras provide the semantic basis for Rosetta models. They support the definition of non-terminating systems common in Rosetta's primary embedded systems application domain. The domain lattice provides support for transformation of facets by organizing modeling domains and defining transformation between them. As we shall see, the domain lattice provides a basis for defining Galois connections that in turn can assure the correctness of transformations between domains.

2.1 Coalgebraic Semantics

A facet's underlying semantics is 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.

A facet's signature defines its associated coalgebra signature while its terms define the coalgebra's transformation 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 $\mathcal{X}$ and $\mathcal{X}$ is never directly exposed to the modeler.

We choose coalgebras over algebras due to the heterogeneous and non-terminating nature of the types of systems Rosetta models. 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 domain that defines it. Instead, a Rosetta facet state is denoted as an observation of $\mathcal{X}$ like any other declared item. For example, if `s :: state_type` defines a state in some domain, then its value in the coalgebra is denoted $s(\mathcal{X})$ – 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 $x(s(\mathcal{X}))$.

If state is simply an observation of $\mathcal{X}$, it is possible to define multiple state observations with multiple semantics and thus multiple computation models by simply defining different state observations. This is precisely the need in system-level design where the true distinction between modeling domains is the unerlying computational model. 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.

2.2 Domains

Vocabulary and semantics for defining domain specific facets are provided by *domains*, frequenly called *facet types*. 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 items that characterize a particular computation or modeling style in a modeling domain. This may vary from simple 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.

Unit-of-semantics information provided by a domain defines the vocabulary used to define a model-of-computation. For example, the `state_based` Rosetta domain declares an abstract state type, a current state variable, an abstract next state function, and defines how observations are made with respect to these declarations. The `state_based` domain does not define properties, but simply provides the declarations. Any domain that purports to be `state_based` provides specific properties for these common stateful abstractions. Domains that provide exclusively unit-of-semantics information define metamodels in the Rosetta domain lattice.

Model-of-computation information provided by a domain defines a model-of-computation for the domain. We view a model-of-computation as a description of how computation proceeds. Thus, defining model-of-computation information for a `state_based` domain requires constraining the definitions of state type, state and next state. In the embedded systems domain Rosetta targets, the defining characteristic for a modeling domain is its model-of-computation. Thus, the use of domains to provide multiple models-of-computation is at the heart of Rosetta's domain specific modeling capability.

Engineering domain information provided by a domain defines a vocabulary for engineering specification. A good way to visualize what engineering domains define is by considering an analogy from engineering education. All engineers take roughly the same mathematics. Yet different engineering domains use those mathematics differently by defining domain-specific vocabularies. Rosetta's engineering domains provide the

same capabilities by extending common models-of-computation with declarations for one specific discipline.

2.3 The Domain Lattice

Formally, any new facet *extends* its 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.

Similarly, 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, $\forall f :: facet \cdot \mathsf{bottom} \Rightarrow f \wedge f \Rightarrow \mathsf{null}$ Including **null** and **bottom** with the partially ordered set $(D, \Rightarrow)$ defines a lattice whose top and bottom elements are **null** and **bottom** respectively: $(D, \Rightarrow, \sqcup, \sqcap, \mathsf{null}, \mathsf{bottom})$

3 Modeling in the Domain Lattice

The domain lattice definition along with the coalgebraic semantics of facets serve 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 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 shows a part of the lattice of domains defined for traditional Rosetta specifications.

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.

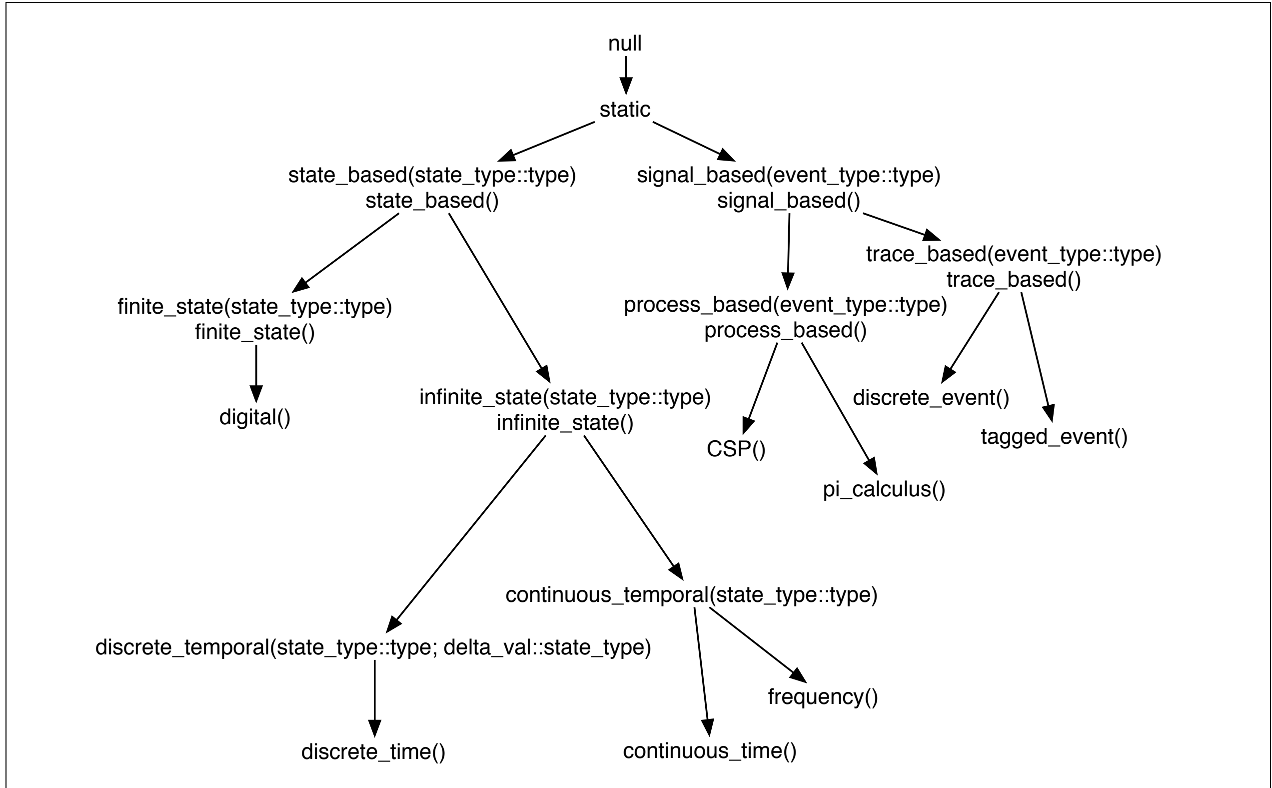

Figure 1: The Rosetta Domain Lattice

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 $\gamma \circ \alpha \sqsupseteq \lambda c.c$ and $\alpha \circ \gamma \sqsubseteq \lambda a.a$. These conditions 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.

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. The functor, A, takes a model and moves it up a level of abstraction transforming it into a meta-model. To verify the meta-model is a safe abstraction of the original model, we use the Galois connection to assure safety of the transformation.

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 is safe. We are also guaranteed that the original model is an instance of the meta-model, and the meta-model is truly a correct abstraction. 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 are also guaranteed safety of any transformation within the entire Rosetta domain lattice.

3.3 Specification Composition

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 model composition.

The primary specification composition mechanism in the Rosetta semantics are 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 *co-product* 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 for domain-specific modeling 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 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.

```
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;
```

```
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 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

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 safety. 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 );
```

Figure 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 safety, 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, domain-specific 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. Although UML is not always thought of as a domain specific language, its profiles can certainly be viewed as providing this capability.

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] 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[13].

An alternative approach using operational modeling is the Ptolemy [14, 15] 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 provides an overview of the approach to domain specific model composition embodied in the Rosetta specification system. We began with the assertion that Rosetta facets represent models, and these models would be denoted as coalgebras, organized around domains situated in a lattice defined by homomorphism. Using the lattice as a basis, we described how homomorphisms define a Galois connection that

assures the safety of models, meta-models, and specification transformations. Using the lattice and coalgebra semantics, we discussed how functors 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 100.

[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 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.

[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 hieararchies of alternative viewpoints. In *Proceedings of the First International Symposium on Requiremetns Engineering (RE-93)*, San Diego, CA, January 1993.

[13] 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.

[14] 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.

[15] J. Davis. Ptolemy ii - heterogeneous concurrent modeling and design in java, 2000.