Techniques for Metamodel Composition

Matthew Emerson and Janos Sztipanovits
Institute for Software Integrated Systems
Vanderbilt University
mjemerson@isis.vanderbilt.edu

Abstract

The process of specifying an embedded system involves capturing complex interrelationships between the hardware domain, the software domain, and the engineering domain used to describe the environment in which the system will be embedded. Developers increasingly turn to domain-specific modeling techniques to manage this complexity, through such approaches as Model Integrated Computing and Model Driven Architecture. However, the specification of domain-specific modeling language syntax and semantics remains more of an art than a science. Typically, the syntax of a DSML is captured using a metamodel; however, there are few best-practices for metamodeling and no public collection of reusable metamodel b to address common language specification requirements. There is a need for an advanced, comprehensive language design environment that offers tool support for a wide range of metamodel reuse strategies and the preservation of metamodeling best-practices. We outline existing techniques for the reuse and composition of metamodels, and propose a new metamodel composition technique we call Template Instantiation.

1 Introduction

In its current state, the definition of new domain-specific modeling languages via metamodeling is more of an art than a science. Metamodeling best practices are not well understood or documented. Most DSMLs are built completely from scratch, using only the metamodeler's personal experience in language definition to guide the process. Consequently, we see a need for libraries of reusable metamodel fragments or patterns to serve as DSML building-blocks. Metamodel reuse will bring to the realm of DSML specification benefits analogous to those that software reuse brought to the realm of software engineering:

- The avoidance of duplication of effort
- The emergence of high-quality reusable metamodel fragments
- The recognition of key metamodeling patterns and best practices
- A significant reduction in the time-to-market for new DSMLs

The cornerstone for metamodel reuse is tool support for a variety of metamodel composition techniques.

We envision a unified, comprehensive modeling language design environment which provides direct support for every language design task. The full scope of language design extends far beyond abstract syntax specification through metamodeling. Especially for embedded systems modeling, there is a strong need for domain-specific modeling languages with unambiguous semantics and built-in analysis capabilities. Consequently, a comprehensive language design environment will also fully support the specification of formal structural and behavioral semantics, semantic mappings, property-preserving model transformations, and links to external formal analysis tools. Wherever possible this environment should ease the language design process by enabling at least partial automatic generation or composition for each of the aforementioned elements. Strong support for the reuse of syntactic metamodel fragments and patterns will greatly help in this effort, because each composable metamodel fragment can have associated structural semantics, semantic mappings, and model transformations that can also be composed and reused. Of course, reusable syntactic patterns will also help language designers rapidly construct new languages including time-tested, quality modeling styles and patterns of expression.

The Model Integrated Computing toolsuite [8] includes tools that can be used to address the capabilities mentioned above. The Generic Modeling Environment (GME) includes a metamodeling environment for the specification of DSML abstract syntax, and also has some support for the composition and reuse of metamodels [3]. The Graph Rewriting And Transformation tool (GReAT) is used to build model transformers [2]. The current MIC solution for the formal specification of DSML behavioral semantics is called *Semantic Anchoring* [9][10]. Semantic Anchoring mandates the use of GME for defining language syntax by formal metamodels, GReAT for creating semantic mappings using formal model transformations, formal models of computation represented as *Semantic Units* for capturing behavioral semantics, and the formal Abstract State Machine [5] framework for expressing and executing the formal semantics specifications built using the Semantic Units. Recent research advocates the use of Horne Logic to specify DSML structural semantics, and describes the *4ml* toolsuite for the generation and analysis of structural semantics from GME metamodels [6][1].

This paper focuses on the reuse of DSML syntax via metamodel composition. In Section 2 we review existing metamodel compo-

sition techniques, each of which should be supported in a comprehensive language design tool. In Section 3 we propose a new metamodel composition technique we call Template Instantiation. We also include a proof-of-concept demonstration of this technique using a metamodeling template instantiation tool built for use with GME. In Section 4 we discuss some ideas for future investigation regarding the relationship between the reuse of metamodels and the reuse of semantic specifications.

2 Current Metamodel Composition Methods

In this section, we describe three general techniques for metamodel composition and discuss when each may be best applied. We also provide illustrative examples of the different techniques using primarily the GME metamodeling language, MetaGME [4].

2.1 Metamodel Merge

A metamodel may be thought of as a namespace for a set of modeling constructs. When composing two modeling languages together, name collisions between the two composed metamodels need to be dealt with in an intelligent way. Typically, name collisions between the elements of two metamodels implies that the domains of those modeling languages intersect in some way; however, this "concept collision" may occur even between modeling constructs with different names. Any time two DSMLs include modeling constructs that capture a shared set of real-world entities, those concepts can be used as "join points" to stitch the two languages together into a unified whole. We refer to this metamodel composition technique as Metamodel Merge.

MOF 2.0, the OMG standardized metamodeling language [7], dictates an algorithm for merging metamodels that fuses together the meta-objects with the same name and metatype from each metamodel. MOF terms this operation a *Package Merge* because it operates at the Package level and recursively impacts all of the elements contained within the merging packages. The new metamodel resulting from a MOF Package Merge maintains no dependency relationships, such as inheritance, importation, redefinition, or type conformance, with the metamodels merged to create it – each element belonging to the new metamodel is newly-created according to the Package Merge algorithm. The MOF specification provides specific rules for merging each of the different MOF metatypes, which we will not review in depth here. However, Package Merge can be generally understood as a recursive unioning of model elements matched by name and metatype.

MetaGME enables Metamodel Merge through the use of three types of class inheritance and a special *Class Equivalence* operator. Class Equivalence is used to show a full union between two classes. The unioned classes cease to exist as distinct metamodel elements, instead fusing into a single class. This union encompasses all the attributes and associations, including generalization, specialization, and containment, of each of the fused classes. The union process is very similar to merging classes through MOF Package Merge, except that the operation takes place at the class level instead of the package (or metamodel) level, the two merged classes do not need to have the same name, and the use of the Class Equivalence operator does not produce a new derivative metamodel.

It is possible simply to use inheritance as a weaker mechanism for merging metamodels - the "is-a-kind-of" specialization relationship is similar to, but weaker than, the "is-equivalent-to" Class Equivalence relationship. In addition to the regular notion of class inheritance supported in both MetaGME and MOF 2.0, MetaGME also defines two special inheritance operators, implementation inheritance and interface inheritance. In implementation inheritance, the child inherits all of the parent's attributes, but only the containment associations where the parent functions as the container. No other associations are inherited. Interface inheritance allows no attribute inheritance but does allow full association inheritance, with one exception: containment associations where the parent functions as the container are not inherited.

Figures 1-3 provide an example of the metamodel merge with MetaGME. Figure 1 shows part of a DMSL for modeling the properties of software components and their deployment onto abstracted hardware components for execution, and Figure 2 shows part of a DSML for modeling the details of hardware components. It might make sense to merge these two DSMLs and get a combined language that can capture details about both the hardware and the software of a particular system. In Figure 3, the Class Equivalence operator (the hollow diamond) is used to fuse the top-level concepts from each language dealing with hardware. The new class resulting from this fusion will have a new name (specified on the Class Equivalence object) and include the SecurityEnclave attribute from the ProcessingModule class.

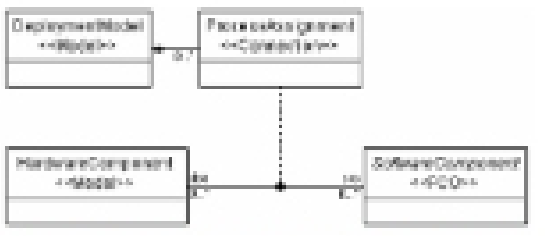

Figure 1. Software deployment metamodel fragment

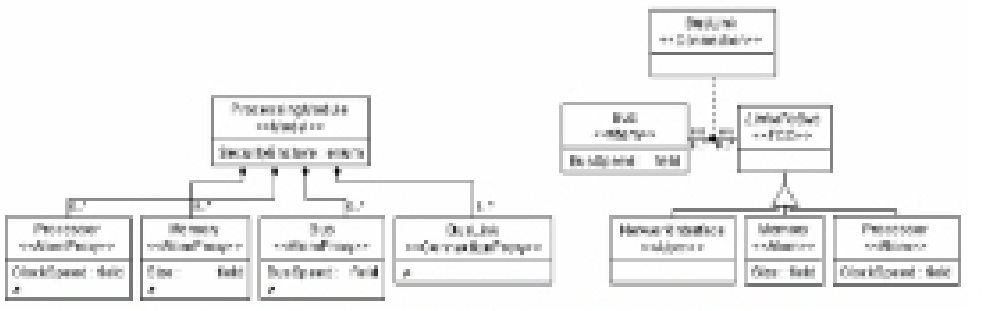

Figure 2. Hardware component metamodel fragment

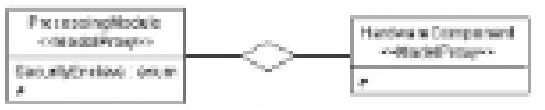

Figure 3. Class merge

2.2 Metamodel Interfacing

The metamodels of two DSMLs capturing two conceptually distinct but related domains may be composed to explore the interactions between the two domains. Performing this composition requires the delineation of an interface between the two modeling languages consisting of new modeling entities and relationships that do not strictly belong to either of the composed modeling languages.

For an example use case, imagine that we had a language for modeling system requirements and a separate language for modeling hardware and software. It would be useful to interface these two languages to enable the traceability of requirements to the hardware and software components that satisfy them. Figure 4 below displays the additional metamodel fragment needed to interface the

two DSMLs. In this figure, Requirement originates from one metamodel while HardwareComponent and SoftwareComponent originate from a second metamodel. The interface between the two metamodels when composed consists of a new model view, TraceModel, which can contain references to Requirements, SoftwareComponents, and HardwareComponents. A new type of connection, RequirementTrace, is also defined to relate the requirements to the components that satisfy them.

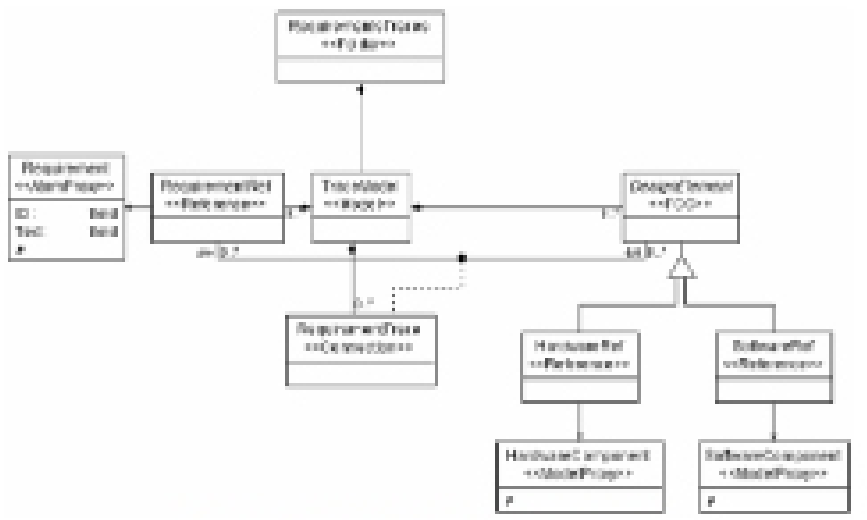

Figure 4. Metamodel Interfacing

2.3 Class Refinement

Class refinement may be utilized when one DSML captures in detail a modeling concept that exists only as a "black box" in a second DSML. This technique is employed similarly to metamodel interfacing, but the relationship between the two composed modeling languages is given by the hierarchical containment of the constructs of one metamodel within a single construct of another metamodel.

For example, consider a simple DSML for modeling the topologies of electronic components of automobiles as depicted in Figure 5. Eventually, the details or behaviors of the components may need to be elaborated, so it may become necessary to further refine the Component language construct. Suppose we want to view the automobile model as a system of concurrent communicating finite state machines. In that case, we would need to refine the Component model using the constructs from the FSM metamodel shown in Figure 6. The composition of the two metamodels is shown in Figure 7.

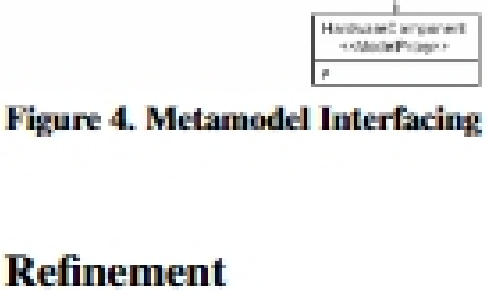

Figure 5. Automobile electronic component metamodel

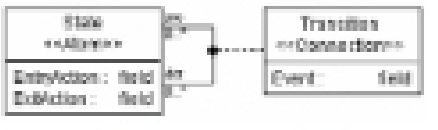

Figure 6. FSM metamodel for capturing component behavior

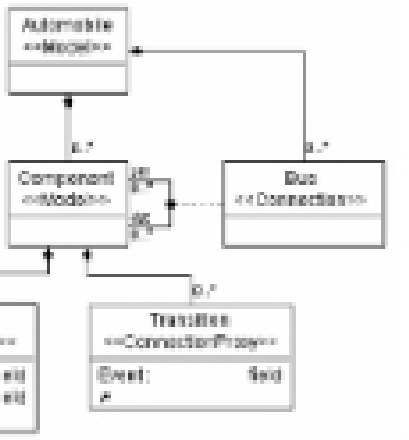

Figure 7. Automobile electronic component refinement

3 Template Instantiation: A New Metamodel Composition Method

Template Instantiation is a new metamodel composition technique we propose to overcome a limitation of the previously-discussed techniques. It is intended to support the multiple reuse of common metamodeling patterns or styles in a single composite metamodel. In Template Instantiation, we record common metamodeling patterns as abstract metamodel templates, then instantiate (replicate and concretize) those templates in domain-specific metamodels. Our previous experience with the definition of modeling languages has resulted in the discovery of a number of commonly-occuring metamodeling patterns. These candidate metamodeling templates include (but certainly are not limited to):

- Composition hierarchies of composite and atomic objects
- Modular interconnection, in which composite objects are associated through exposed ports (a specific incarnation of this pattern would be component-based modeling)
- StateCharts-style modeling
- Data Flow graphs
- The proxy metamodeling pattern, in which a reference type is defined for an class with the same attributes, composition roles, and associations as the class itself

Each of the above patterns repeatedly occur in new DSML specifications, and some may potentially be used multiple times in the same metamodel, so they are good candidates for metamodeling templates.

The previously-discussed metamodel composition techniques are not suited toward the multiple reuse of metamodel fragments within the same composite metamodel. To demonstrate this limitation, we can consider a simple example: using Metamodel Merge to incorporate hierarchy into the metamodel shown in Figure 8. This metamodel captures a Data Flow language where the internal behavior of the Data Flow Actors is captured using FSMs. Now, suppose we want to be able to define hierarchical Actors that contain whole Data Flow graphs, and we want the FSMs in this language to be hierarchical as well. We will attempt to do this by merging in the abstract Hierarchy metamodel depicted in Figure 9. The composition using Metamodel Merge (inheritance, in this case) is shown in Figure 10.

Unfortunately, in the metamodel resulting from the merge the composition has had an undesired side-effect: according to Figure 10, FSM States can contain Data Flow Actors! In fact, any time Metamodel Merge is used to merge in the same metamodel in multiple