Design Patterns for Metamodels

Hyun Cho and Jeff Gray
University of Alabama
Department of Computer Science
Box 870290
Tuscaloosa, AL, 35216
hcho7@ua.edu, gray@cs.ua.edu

ABSTRACT

A metamodel is used to define the abstract syntax (i.e., entities, attributes, and relations) of a Domain-Specific Modeling Language (DSML). In addition, a metamodel also defines constraints and static semantics that provide additional information about the modeling language beyond the abstract syntax. In many cases, the specification of a new metamodel is highly dependent on the designer's background and experiences. Thus, metamodel designs often differ from designer to designer, even for recurring design problems (i.e., there is more than one way to specify a modeling language with a metamodel). The quality of a metamodel design may also vary according to the designer's domain knowledge and modeling language expertise. To provide consistent solutions for recurring metamodel design issues, design patterns applied to metamodels may offer key insights, especially to new language designers who have less experience. In this paper, we motivate the need for design patterns for metamodels and provide a few examples of the concept.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages.

Keywords

Metamodel, Design Pattern, Visual Modeling, Domain-Specific Modeling Languages.

1. INTRODUCTION

A language is generally developed by designing and implementing three language elements: concrete syntax, abstract syntax, and semantics. When developing Domain-Specific Modeling Languages (DSMLs), especially visual modeling languages, the concrete syntax corresponds to modeling elements that symbolize the concepts of the domain, and abstract syntax is defined as the inter-relationships of modeling concepts as defined within the metamodel. Semantics, which govern structural and behavioral properties of DSMLs, are also often associated with a metamodel. Thus, to develop a quality DSML, language designers are required to have deep understanding of both a target domain and language development expertise.

General-Purpose Modeling Languages (GPMLs) are designed to model a wide range of domains, such as business process modeling and manufacturing, in addition to software design**.** Generally, GPMLs like the UML consist of a large set of language constructs to be used in many contexts, with the understanding that not all of the provided modeling concepts are needed for each domain. However, DSMLs are designed and implemented to be used for a specific domain. Because DSMLs can offer several benefits due to their conciseness and expressiveness, many DSMLs have been developed and applied for various domains.

Despite numerous successful case studies, there are several challenges that may contribute to the lack of widespread adoption of DSMLs in industry: (1) domain knowledge and language development expertise are required when developing DSMLs, but few experts have such expertise, (2) lack of methods and guidelines to develop and manage quality DSMLs. In addition, DSMLs are often developed from scratch because many are designed and implemented for internal use within a specific domain, which makes it difficult to publicly share development artifacts. Due to these issues, DSMLs are sometimes developed only when they are absolutely necessary.

We believe that one way to resolve these issues is to reuse previous DSML designs, especially reuse of recurring concepts in the design of metamodels. Although DSMLs are developed to be used for a specific domain, some design decisions commonly occur across modeling language creation, regardless of the domains of interest. For example, classifiers, which represent domain notations and their relationships (e.g., association, aggregation, and inheritance), are typically present in every DSML.

In this paper, we consider the application of design patterns in metamodel design. The goals of this paper are (1) identification of common design problems of DSMLs by analyzing the concrete syntax of DSML examples, (2) the proposition of metamodel design patterns based on the results of the concrete syntax analysis, and (3) metamodel design guidelines.

The paper is organized as follows. Section 2 describes our approach for identifying metamodel design patterns and then lists a set of questions that can be used to find other metamodel design patterns. To identify a set of questions, we analyze commonality of DSMLs and present a feature model as the result of the analysis. Based on the analysis, basic metamodel designs are elicited and elaborated. Section 3 discusses the possible issues to rationalize our approach and Section 4 concludes with future work.

2. APPROACH OF METAMODEL DESIGN PATTERN IDENTIFICATION

Since the notion of patterns in urban design and the architecture of their construction were introduced into the software community [2], design patterns [11] have been widely adopted in both research and the software industry during the past decades.

As design patterns describe mature solutions for particular software design problems that recur in a given context [7], software architects and designers can leverage the experience of master designers. The solution can be structured into either desirable patterns [11] or undesirable patterns (e.g., "anti-patterns) [5]. Desirable patterns represent reusable elements at a higher level of abstraction, and undesirable patterns describe patterns that protect a design from common and expensive mistakes, which should be identified and avoided at an early design stage. As a result, design patterns can help capture domain knowledge and improve the quality of software products. Design patterns are applied to a wide range of software development areas, such as software architecture design [6], user interface design, information visualization, and business modeling.

In this paper, we consider the notion of design patterns in metamodel design to capture experience that could be applied across a broad base of metamodels. To mine design patterns in metamodels, we took the following steps:

- *Context setting*: To identify issues of metamodel design, we reviewed the concrete syntax of several DSMLs and modeled their commonality and variability. Because complete DSMLs are challenging to obtain from industrial settings, we include GPMLs such as UML diagrams, assuming that each diagram can be tailored for a specific domain. A feature model [14] was used to summarize our understanding of commonality and variability in the DSML examples that we analyzed.
- *Identification of metamodel design problems*: Based on the feature model that was created from the analysis of DSML concrete syntax, we observed several metamodel design challenges. To derive the recurring metamodel design idioms, we focused on the commonalities in the DSML feature model. These commonalities represent a few of the common features at the core of metamodeling.
- *Metamodel design pattern proposal*: Based on the identified problems from step 2, we searched and analyzed relevant metamodels and proposed a design pattern for each problem identified.

2.1 Context Setting

To identify the commonly recurring metamodel design problems, we examined the concrete syntax of several DSMLs (please see the Appendix for example domains), with specific focus on classifiers and relationships. To generalize the concrete syntax of DSMLs, we assume that most modeling languages commonly use a Box-and-Line style, even though there is some disagreement in the community on how to interpret and understand the syntax and semantics of graphical languages. Typically, Boxes represent the instances of the domain concepts such as key functionalities or behaviors, and Lines that connect Boxes describe how the connected Boxes communicate or are related to each other syntactically and semantically. The key benefit of using the Box-and-Line style is its simplicity, and thus, many modeling languages inherently contain the notion of Box-and-Line even though they are realized with different concrete syntax. For example, Petri Nets define four basic symbols (i.e., Places, Transitions, Directed arcs, and Marks) to model and analyze reachability, liveness, and boundedness of concurrent discrete event systems. Places and Transitions, denoted by circles and rectanglrd (or bars), correspond to Boxes; Directed arcs, represented by arrows, correspond to Lines.

Based on this observation, prior to identifying metamodel design patterns, the concrete syntax of DSMLs should be identified and generalized from model instances. In particular, we paid close attention to what and how many modeling entities are used in DSMLs, and what and how the relationships link modeling elements both syntactically and semantically.

2.2 Identification of Metamodel Design Problems

Based on the analysis of existing DSMLs, we identified the commonality among several DSMLs and derived a feature model as shown in Figure 1.

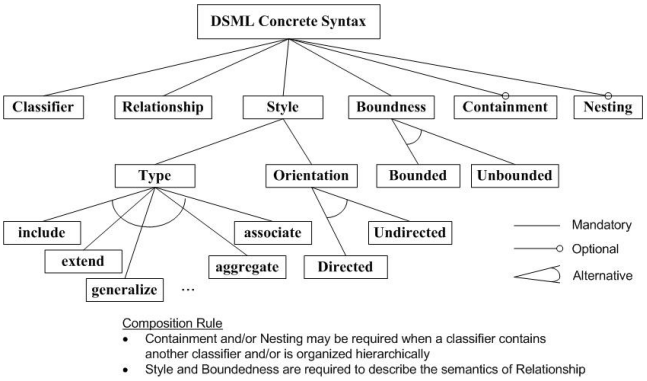

Figure 1. Feature Model of DSML Concrete Syntax

Four major features (i.e., Classifier, Relationship, Style, and Boundedness) are defined as mandatory features. There are two other features (i.e., Containment and Nesting), which describe characteristics of a classifier, that are defined as optional features. In addition, Sub features of Type, Orientation, and Boundedness are defined as an Alternative feature because a relationship can have only one kind of Type, Orientation, and Boundedness. Based on the feature model in Figure 1, we derived the following questions that relate to metamodel design challenges:

- How to design a metamodel if the concrete syntax of the DSML consists of simple boxes and lines? This question will examine how to design a metamodel for a very primitive concrete syntax, which consists of classifiers and association relationships. Thus, the solution for this problem will be the base metamodel, and metamodels for complex DSMLs will be designed by extending this base metamodel.
- How to design or evolve a base metamodel if the concrete syntax is more complex (e.g., classifiers are linked with several different types of relationships)? This is generally required for both GPMLs and DSMLs. For example, in a UseCase diagram, a use case can be linked with other use cases that include or extend the relation. This question may also be important in the design of DSMLs, which heavily depend on relationships between classifiers to describe domain knowledge.
- How to represent boundedness of a relationship? Generally, most DSMLs implicitly enforce that both ends of a relationship are bounded to classifiers to represent which classifier drives a behavior and which classifier reacts to the action. In some cases, one end of the relation can be open. A DSML for representing chemical structure [3] can be a good

example for this case because some chemical structures have lone pairs of electrons, which are not involved in chemical bond formation, as well as bonding pairs.

- How to design a metamodel to represent *containment* and *nesting*? Some DSMLs may contain one or more types. Petri Nets and Activity Diagrams are examples of languages that have containment. As mentioned above, Petri Nets are defined with four modeling elements (i.e., Places, Transitions, Directed Arcs, and Marks). Places represent the pre- and post-state of a system by transition, and transition shows the place where events occurred. Directed arcs show the direction of a transition. Transitions between places are determined by the contained number of tokens in a place and are fired when one or more start places, linked to the same transition, contain enough tokens to satisfy the firing condition. Nesting can be a special case of containment and used to control the level of abstraction by organizing classifiers hierarchically.

2.3 Metamodel Design Patterns

Based on the questions described in Section 2.2, we propose an initial set of solutions in this section. The solutions are proposed by investigating several metamodels, including UML. We use object-oriented notations, such as those used in class diagrams, to represent metamodel designs.

2.3.1 Design for Base Metamodel

Extension to a base metamodel is proposed as a candidate solution for the first question related to metamodel design when the concrete syntax consists of boxes and lines. Consideration of metamodel design for the simple box-and-line style DSMLs is important because this style may be used when requirements of a DSML are captured at an initial sketch level, which may occur at the early stage of DSML development. This issue emerges when a domain needs to be modeled with a very high level of abstraction.

In the Box-and-Line style, boxes are generalized as a set of *Classifiers* and lines are mapped to *Relationships*. As a *Relationship* normally links two *Classifiers*, one for the source *Classifier* and the other for the target *Classifier*, the *Classifier* and *Relationship* are linked with two association relationships, source and target.

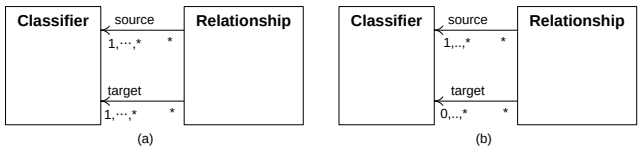

Figure 2. Base Metamodel

Multiplicity is assigned to the association in order to specify the number of participating instances. In addition, it can also be used to describe the boundedness of a relationship. For example, Figure 2(a) shows the relationship links for two classifiers with source and target, which denotes the situation where at least one source and target exist due to both the multiplicity of source and target being specified as one-to-many. On the contrary, in Figure 2(b), the multiplicity of source and target is set to one-to-many and zero-to-many, respectively. This means that there exists at least one source, but the target may or may not exist in the relationship.

2.3.2 Metamodel for Typed Relationships

Associations represent a common relationship type in DSMLs. However, several types of relationships may exist to enrich the semantics between linked model elements. For example, a UseCase diagram has two typed relationships, such as include and extend. A class diagram has three typed relationships (i.e., inheritance, aggregation, and composition) in addition to association.

Several metamodel designs for typed relationships and classifiers have been presented in the literature. Figure 3(a) is excerpted from the UML Superstructure Specification v2.3 [24], and Figure 3(b) is excerpted and simplified from Ouardani et al. [20]. Although the number of participating elements is equal, the two metamodel designs are different in two key ways: linked elements and a typed relationship to linked metamodel elements. First, when looking at the linked elements, both metamodels are designed to inherit typed relationships (i.e., include and extend) from the common parent relationship. However, in Figure 3(a), each typed relationship is linked with a classifier, but in Figure 3(b) the classifier and relationship are linked to each other instead of linking the typed relationships.

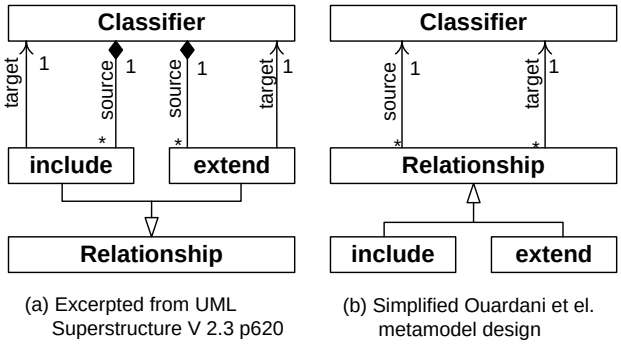

Figure 3. Metamodel Design for Typed Relationship (adapted from [20] and [24]

The two metamodel designs are both acceptable, but Figure 3(a) is a more preferable design than Figure 3(b). To rationalize the preference, we need to consider the notion of upcasting and downcasting in object-oriented programming. Upcasting and downcasting occur in inheritance hierarchies. Upcasting is "*part of expressing the is-a relationship and converts a derived class reference or pointer to a base class reference or pointer and is allowed for public inheritance without the need for an explicit type cast.*" Downcasting is "*the opposite process of upcasting and converts a base class pointer or reference to a derived class pointer or reference.*" [21]. Normally, downcasting is avoided because *is-a* relationships are not reversible. This means that the base class cannot access the data members and methods defined in a derived class. As a result of downcasting, a system may show unsafe system behavior and break abstraction and encapsulation.

As another point of differentiation, the two metamodels use different typed relationships between the classifier and relationship. In Figure 3(a), two different relationships, composition and association, are used to link between classifiers and typed relationships (i.e., include and extend). In this metamodel, a composition relationship may be introduced to describe that the source classifier is strongly dependent on the target classifier. But in Figure 3(b), the association relationship is used for both source and target links. Typically, association is used to link classifiers weakly, and composition is used to

describe a part-whole relationship. However, because the two relationships are relevant to each other and the semantics of the two are defined slightly differently among OO modeling approaches [1], it is difficult to say which one is more appropriate. In general, we believe that association is to be preferred to composition if there is no clear part-whole relationship.

2.3.3 Metamodel for Containment

Containment represents a part-whole hierarchy and is used to raise the level of abstraction by grouping large and complex model elements with a simple element. The Composite design pattern [11] is commonly used for designing containment needs, but containment also can be designed without using the Composite design pattern. Three different containment metamodel designs are shown in Figure 4.

Figure 4(a) uses the Composite design pattern to design a containment metamodel. The design leverages the benefits of the design pattern such as facilitating the addition of new kinds of classifiers and recursive composition. Figure 4(b) represents containment with a unary composition relationship. Although Figure 4(b) represents a viable design option for containment, the design is only applied for containing the same type of classifiers and may violate the open/closed principle when a container needs to include new kinds of classifiers. In Figure 4(c), the classifier is inherited from Composite Classifier, which represents an abstract classifier that can have sub-classifiers. In addition, a classifier is linked with the Container through an association relationship.

The intent of the design is to treat the container differently from the contents by introducing *Container*, which may have different characteristics than other classifiers. For example, a deployment diagram (or allocation diagram) may be used to illustrate how physical resources (i.e., storages, processors, network interfaces) are allocated onto execution environment nodes. Physical resources are often composed of the same or other physical resource (i.e., a network interface may have a processor and storage to manage network packets) to provide their own functionality, but the instance of the composed physical resources are treated as composite physical resources. However, nodes are composed of physical resources to offer services, but they can be designated as a container rather than composite entities because they are grouped logically. The advantage of the design is that containers can specify classifiers to be contents of the container through the links between classifiers and container.

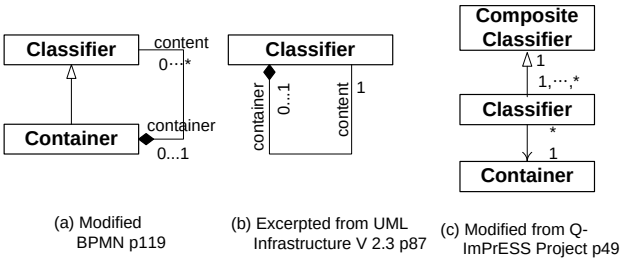

Figure 4. Metamodel Design for Containment (adapted from [23], [24], and [26])

As described above, each metamodel design has its own intent, but the metamodel design for containment can be unified into Figure 4(a), which is an overly general design for containment that has the flexibility of extension for adding new classifiers or other properties. For example, if a container should have different characteristics that are abstract (or logical) from classifier, an attribute may be added to the classifier class to represent that need. Moreover, Figure 4(a) can represent nested containers without additional descriptions.

3. APPLICATION OF METAMODEL DESIGN PATTERNS

Describing the applicability of a design pattern is an important factor in characterizing its usefulness and promoting its understanding. The metamodel design patterns introduced in Section 2.3 can form the basis for designing the metamodel for a DSML. The first metamodel design pattern can be applied to design a simple box-and-line style DSMLs. The second and third patterns can be used to describe metamodels that support typed relationships and containment, respectively. In addition to these applications, metamodel design patterns can be used for composing and inferring metamodels.

Metamodel composition [8][15][16][18][19] is a technique that creates a new metamodel by reusing all or part of existing metamodels. To make metamodels reusable and/or composable, the metamodels are refined to abstract metamodels that are not designed for specific DSMLs, but capture general structures and behaviors of DSMLs. The proposed metamodel design patterns are elicited from commonality analysis and can represent general characteristics of DSMLs, much like abstract design patterns.

Metamodel inference is the other application area of metamodel design patterns. Metamodel inference has recently been considered as an application of grammar inference [4][10][12] and used to recover metamodels from existing model instances [9][13]. To infer a metamodel accurately, a metamodel inference engine may require a large set of training data [17]. However, having a large set of existing training data may not be practical in many cases. To complement the lack of training data, metamodel design patterns can be used as a supplementary aid to generate representative instances for metamodel inference through the commonality provided by DSMLs for recurring metamodel design problems.

4. DISCUSSION

In this paper, we proposed three metamodel design patterns based on the common characteristics of DSMLs, which we think are important and offer the potential for high impact for metamodel design. Although the notion of "patterns" does not fit the generally ascribed parts that are contained in a pattern language, the design guidelines for each concept will be expanded into a more complete description following a pattern language.

The main threat for the generality of the approach is the selection of DSMLs that we used to derive our feature model. Because DSMLs are designed for different domains and sometimes used in closed groups, it is difficult to collect all kinds of DSMLs. Thus, our feature model may not reflect features that are mandatory for all types of DSMLs. However, considering DSML development tools such as GEMS [22] and MetaCase [25] and reviewing the literature, we believe that the proposed approach and metamodel design patterns have general applicability.

Another concern regarding the generality of our proposal emerges when one considers that DSMLs are often used to model system behaviors. However, most of the DSMLs that we analyzed are used to model the structural aspects of a system. However, reviewing several available behavioral modeling languages, the approach may be applied equally if new patterns can be

uncovered. For example, sequence diagrams, activity diagrams, and state transition diagrams are the typical examples of behavioral modeling languages, even though they are classified as general-purpose modeling languages. They also can be analyzed to determine characteristics of behavioral modeling that may be of use also for DSML designers.

5. CONCLUSIONS AND FUTURE WORKS

In this paper, we analyzed the commonality and variability of DSML concrete syntax based on the properties of classifiers and their relationships. From this analysis, we derived recurring metamodel design problems from a DSML feature model. In addition, we proposed three metamodel design patterns for the identified problems. As the proposed metamodel design patterns are the basic and/or core of the metamodel design, they can be commonly applied across different metamodel designs. We believe that the proposed metamodel design guidelines will help to design quality metamodels.

With the proposed metamodel design patterns, our future work will apply these patterns to metamodel composition. The patterns introduced in this paper form the basis for metamodel construction, but they need to be composed with other metamodel elements to construct a complete metamodel for a DSML. For this, we will introduce the notion of component, whereby each metamodel design pattern and metamodel element will be treated as components. Generating a metamodel through metamodel composition can leverage the benefits of both design patterns and Component-Based Development, while minimizing manual tasks of language design for recurring situations.

Our focus has been on metamodels that are defined in a graphical manner. There are also many popular metamodeling environments and languages that focus on a textual description of a metamodel. We plan to perform a similar analysis on textual metamodels.

In addition, we will research how metamodel design patterns can be used effectively for metamodel inference. We have observed occasions when a metamodel needs to be reconstructed from legacy instances (e.g., when DSMLs have evolved and no development documents exist). Normally, reconstructing through inference requires a large set of model instances to train an inference engine. Although preparing a quality training set affects the inference accuracy, it is challenging and considered mundane and error-prone to prepare such training sets. Thus, we expect that the use of metamodel design patterns can resolve the issues of training set preparation and computation complexity.

6. ACKNOWLEDGMENTS

This work is supported by NSF CAREER award CCF-1052616.

7. REFERENCES

[1] Albert, M., Pelechano, V., Fons, J., Ruiz, M., & Pastor, O. 2003. Implementing UML association, aggregation, and composition: a particular interpretation based on a multidimensional framework. In Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE'03), Johann Eder and Michele Missikoff (Eds.). pp. 143-158, Springer-Verlag, Berlin, Heidelberg.

[2] Alexander, C., Ishikawa, S., & Silverstein, M. 1977. *A Pattern Language*. Oxford University Press, Oxford.

[3] Ash, S., Cline, M. A, Homer, R. W., Hurst, T. & Smith, G. B. 1997. SYBYL Line Notation (SLN): A Versatile Language for Chemical Structure Representation, *Journal of Chemical Information and Computer Sciences*, vol. 37, no. 1, pp. 71-79.

[4] Berwick, R. C., & Pilato, S. 1987. Learning Syntax by Automata Induction. *Machine Learning*, vol. 2, no. 1, Mar 1987, pp. 9-38.

[5] Brown, W., McCormick, H., Mowbray, T., & Malveau, RC. 1998. *Anti-patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons.

[6] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley.

[7] Coplien, J. O. 2003. Software design patterns. *In: Encyclopedia of Computer Science (4th ed.), Ralston, A., Reilly, E. D., Hemmendinger, D. (Eds.)*. pp. 1604--1606. John Wiley and Sons Ltd., Chichester, UK.

[8] Emerson, M. & Sztipanovits, J. 2006. Techniques for metamodel composition. *In The 6th OOPSLA Workshop on Domain-Specific Modeling*, pp. 123-139, Oct 2006, Portland, OR, USA.

[9] Favre, J-M. 2004. CacOphoNy: Metamodel driven architecture reconstruction, *In Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004)*, Delft, The Netherlands, 2004, pp. 204-213.

[10] Fu, K-S. & Booth, T. L.1986. Grammatical Inference: Introduction and Survey-Part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 3, pp. 343-359, May 1986.

[11] Gamma, E., Helm, R. Johnson, R., & Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, MA, USA.

[12] Gold, E. M. 1967. Language identification in the limit. *Information and Control*, vol. 10, pp. 447–474, 1967.

[13] Javed, F., Mernik, M., Gray, J., & Bryant. B. R. 2008. MARS: A metamodel recovery system using grammar inference. *Information and Software Technology*, vol. 50, no. 9-10 (Aug 2008), pp. 948-968.

[14] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., & Peterson, A.S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Technical Report CMU/SEI-90-TR-21*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[15] Karagiannis, D. & Höfferer. P. 2008. Metamodeling as an Integration Concept. *Software and Data Technologies* (2008), vol. 10, pp. 37-50.

[16] Karsai, G., Maroti, M., Ledeczi, A., Gray, J., & Sztipanovits, J. 2004. Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology*, vol. 12, no. 2, pp. 263-278.

[17] Kirsopp, C., & Shepperd, M. 2002. *IEE Proceedings: Software*, vol. 149, no. 5, pp 123-130.

[18] Ledeczi, A., Nordstrom, G., Karsai, G., Volgyesi, P., & Maroti, M. 2001. On metamodel composition. *In Proceedings of the 2001 IEEE International Conference on Control Applications, 2001. (CCA '01)*, pp.756-760, Sep 2001, Mexico City, Mexico.

[19] Mapelsden, D., Hosking, J., & Grundy, J. 2002. Design pattern modelling and instantiation using DPML. *In Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded*

applications (CRPIT '02), pp. 3-11, Feb 2002, Sydney, Australia.
[20] Ouardani, A., Esteban, P., Paludetto, M., & Pascal, J. C. 2006. A Meta-modeling Approach for Sequence Diagrams to Petri Nets Transformation within the requirements validation process. *In Proceedings of the European Simulation and Modeling Conference*, pp. 345-349, Toulouse, France.
[21] Prata, S. 2004. C++ Primer Plus, 5th edition. Sams
[22] GEMS. http://www.eclipse.org/gmt/gems/
[23] OMG Business Process Model And Notation (BPMN) Ver. 2.0, http://www.omg.org/spec/BPMN/2.0/
[24] OMG UML 2 Superstructure, http://www.omg.org/spec/UML/2.3/Superstructure/PDF
[25] MetaCase. http://www.metacase.com/
[26] QImPrESS Service Architecture Meta-Model, http://www.q-impress.eu/wordpress/wp-content/uploads/2009/05/d21-service_architecture_meta-model.pdf.

Appendix: Listing of Example Domains for Representative DSMLs

Domain	Diagrams	Brief Description	Key Modeling Elements	Containment/ Nesting		Relationship	
						Style/ Boundedness	
Concurrent Discrete Event System Modeling	Petri net	Modeling systems with concurrency and resource sharing	Place, Transition (C), Directed Arc (R)	A	N	Directed	Closed
Data Modeling	ERD	Model the logical structure of database	Entity(C), Relation(R)	N	N	Directed	Closed
Project Management	Gantt Chart	Model project activities with relevant information (i.e., duration, cost, …)	Task(C), Predecessor (R)	N	N	Directed	Open
	PERT Chart	Identify the critical path of the project by modeling the sequence of tasks	Task(C), Directed arcs (R)	N	N	Directed	Closed
Electronic Circuit Design	Schematic Diagram	Represent how electronic components are connected with others	Component (C), Line(R)	N	A	Undirected	Closed
	PCB Layout	Show the placement of electronic components on printed circuit board	Hole (C), Line (R)	N	N	Undirected	Closed
Molecular Modeling	-	Model the structures and reactions of molecules	Atom (C), Bond (R)	N	N	Undirected	Open
SW Design	Flowchart	Model process or algorithm	Symbols (C), Connector(R)	N	N	Directed	Closed
	Component Diagram	Represent static structure of components and their relations	Component, Interface, Port (C), Connector (R)	A	A	(Un)Directed	Both
	UseCase Diagram	Describe system functionalities or behaviors with UseCase and Actor	UseCase, Actor (C), Relation (R)	N	A	(Un)Directed Typed	Closed
	Class Diagram	Describe the static structure of the system in terms of classes	Class (C), Relation (R)	N	N	(Un)Directed Typed	Closed