A Generative Approach to Model Interpreter Evolution

Jing Zhang, Jeff Gray, and Yuehua Lin

Department of Computer and Information Sciences
University of Alabama at Birmingham
{zhangj, gray, liny} @ cis.uab.edu

Abstract. *Domain-specific modeling techniques are being adopted with more frequency in the engineering of computer based systems. In the presence of new stakeholder requirements, it is possible that a meta-model undergoes numerous changes during periods of evolution. There is a fundamental problem in maintaining the model interpreters in terms of such meta-model schema changes. This position paper outlines the technical challenges involved in providing evolution of model interpreters. The paper proposes an approach that is based on a mature program transformation engine to automate the evolution of interpreters in the presence of meta-schema changes.*

1. Introduction

Domain-specific modeling (DSM) techniques are frequently adopted in the development of computer-based systems (CBS), especially in the domain of embedded control software (e.g., avionics and automotive control systems). Meta-configurable domain-specific modeling environments [8, 10] provide support for customization of modeling tools that enable domain experts to construct models in notations that are familiar to them. Such tools typically offer the ability to generate, or synthesize various artifacts from domain models (e.g., synthesis of input to analysis tools, or generation of source code from the models). The ability to describe properties of a system at a higher abstraction level, and in a technology-independent notation, can protect key intellectual assets from technology obsolescence. DSM also supports rapid evolution of computer-based systems when the hardware and software configuration is tightly coupled, but must frequently evolve to a new configuration schema (e.g., retooling in an automotive factory or reconfiguration of an avionics product-line).

In DSM terminology, a meta-model defines the valid concepts and rules for constructing a model in a specific domain and can be instantiated to provide a domain-specific modeling language (DSML) that is customized to the visual representation and semantics appropriate for that domain. A DSML may have multiple *model interpreters* associated with it that permit synthesis of different types of software artifacts from the domain models. For example, one interpretation may synthesize to C++ program source code, whereas a different interpretation may synthesize to a simulation engine or analysis tool [12]. Generally speaking, a meta-model is relatively stable and seldom changes during the process of the system evolution; however, the meta-model may need to be modified or extended if it cannot represent concepts described in new system requirements. Under such circumstances, each evolution of the meta-model will typically break the interpreter that was defined on the previous version. Consequently, there is a need to develop methods and tools to automate the evolution process of model interpreters in the presence of meta-model schema changes. This position paper outlines the motivation and technical challenges involved in providing evolution of model interpreters, and investigates a generative approach [6] that uses a mature program transformation system to assist in the automation of interpreter evolution.

2. Motivation and Technical Challenges

Changing stakeholder requirements often necessitate the need for evolution of the modeling language associated with a domain. The evolution of a domain requires that changes be made to the underlying meta-model. As shown in Figure 1, with the evolution of the meta-model, the models and model interpreters that were defined under the previous meta-model are often made invalid under the new meta-model. There exists some difference, Δ_{MM} (termed a *maintenance delta* in [4]) between the old meta-model and the new one, which captures the evolving features of the domain. Δ_{MM} must reflect the difference between the old and new instance models (annotated as Δ_{M}) such that the new models can preserve the original semantics under the new meta-model definition. Additionally, Δ_{MM} should also correspond to the difference between the old and new interpreters (represented by Δ_{I}) such that the new interpreters can be adapted based on the new meta-model schemas.

The problem of schema evolution is common across many software development activities (e.g., database schema evolution). To understand this phenomenon better, consider the evolution of a programming language and a compiler defined for a specific definition of the language. If the language were to evolve (e.g., Ada 83 to Ada 95) to a new syntax and semantics, the previous programs may no longer be valid and the previous compilers will not work under the new definition. When a programming language definition changes, it is necessary to evolve the previous programs defined by the language. Modifications to the compiler must also be made.

Regarding the notion of *model evolution* in the presence of meta-model schema changes (i.e., the automatic mapping from Δ_{MM} to Δ_{M}), work has already been done by others to address this problem [2, 13]. The typical approach is to perform model transformations from one representation schema to another using a model transformation engine. However, no research has been conducted to address the more difficult problem of *model interpreter evolution* (automatic mapping from Δ_{MM} to Δ_{I}). Model interpreters are often written in a general programming language and traverse the internal data structure of a model in order to synthesize a new artifact based on the model properties. Interpreters may invoke an API that

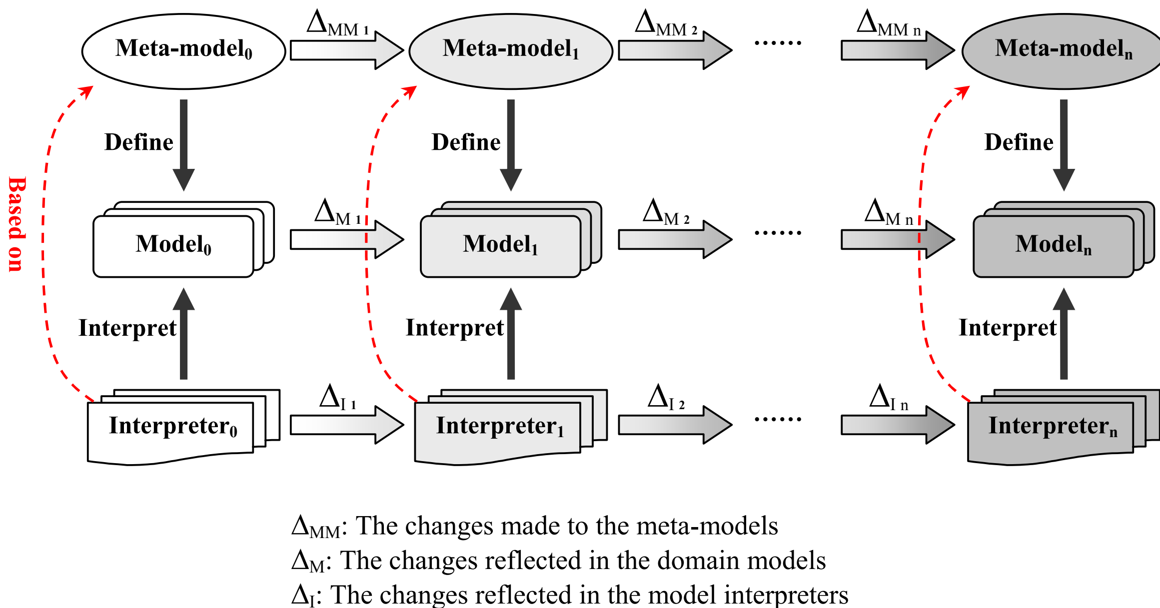

Figure 1. The evolution of the models and interpreters in terms of meta-model changes

the modeling tool exposes to provide access to the model data structure. Alternatively, some modeling tools may have a proprietary scripting language for defining the behavior of a model interpreter. Regardless of the underlying approach for writing an interpreter (i.e., a general programming language, or a proprietary scripting language), the problem of interpreter evolution still remains. Current practice requires each model interpreter to be modified manually after each meta-model schema change. This can be a time-consuming and error prone task for complex model interpreters of considerable size.

A classic example of meta-model evolution is the specialization of a domain concept into two or more derivations. Figure 2 illustrates two different meta-models (top of figure) and their corresponding instance models (middle of figure). Pseudo-code is also provided at the bottom of the figure as an example of the type of code written in a model interpreter. A simple meta-model of a finite state machine (FSM) is shown in the top-left of the figure, which defines a state diagram as a sequence of states that connect to each other by transitions. However, as the domain evolves, a new requirement (right-hand side of Figure 2) may necessitate three distinct kinds of states, including the start state, end state, and inner state. Furthermore, the new meta-model may specify that there exist only one start state (with no input transitions into it), as well as a constraint that the end-state have no output transitions.

As the meta-model of Figure 2 evolves, not only will the instance models be modified, but also the corresponding model interpreters. For example, in order to start simulating the FSM, the original interpreter (left-hand side) has to perform the following initialization procedures:

1. Search all of the states and iterate through them one by one.
2. For each state, if it has no incoming transitions, mark it as the start state.
3. If no start state exists, or if there is more than one start state, an error message will be given and the interpreter will stop executing.
4. If a start-state was found, the simulation transfers execution control from the marked state to the next state.

Comparatively, the interpreter for the new meta-model (right-hand side of Figure 2) simply searches for the model atom named “StartState” and then begins the simulation. Current practice would require that the former interpreter be re-written and transformed manually. Because very few domains are completely stable, evolution of the meta-model will inevitably break the connection to a pre-existing interpreter. This example illustrates the potential advantage of automation for model interpreter evolution.

However, support for model interpreter evolution automation from higher-level models is not well-represented in the research literature. This is because of the following four key challenges:

1. Model interpreters are typically written by hand without any formal specification. Various modelers may have different ways to interpret a model. Consequently, different developers may program interpreters in various ways in order to achieve the same behavior. It is hard to maintain and evolve such subjective realizations of an interpreter.

2. The formal specification that indicates each step of the meta-model transformation (as represented from $\Delta_{MM\ 1}$ to $\Delta_{MM\ n}$ in Figure 1) is also required in order to capture the evolving domain designs. Moreover, such meta-model transformation specifications must include the *entire* knowledge for the underlying interpreter evolution. It is an onerous task to map the high-level abstract model elements to the lower-level concrete details of the corresponding source code.

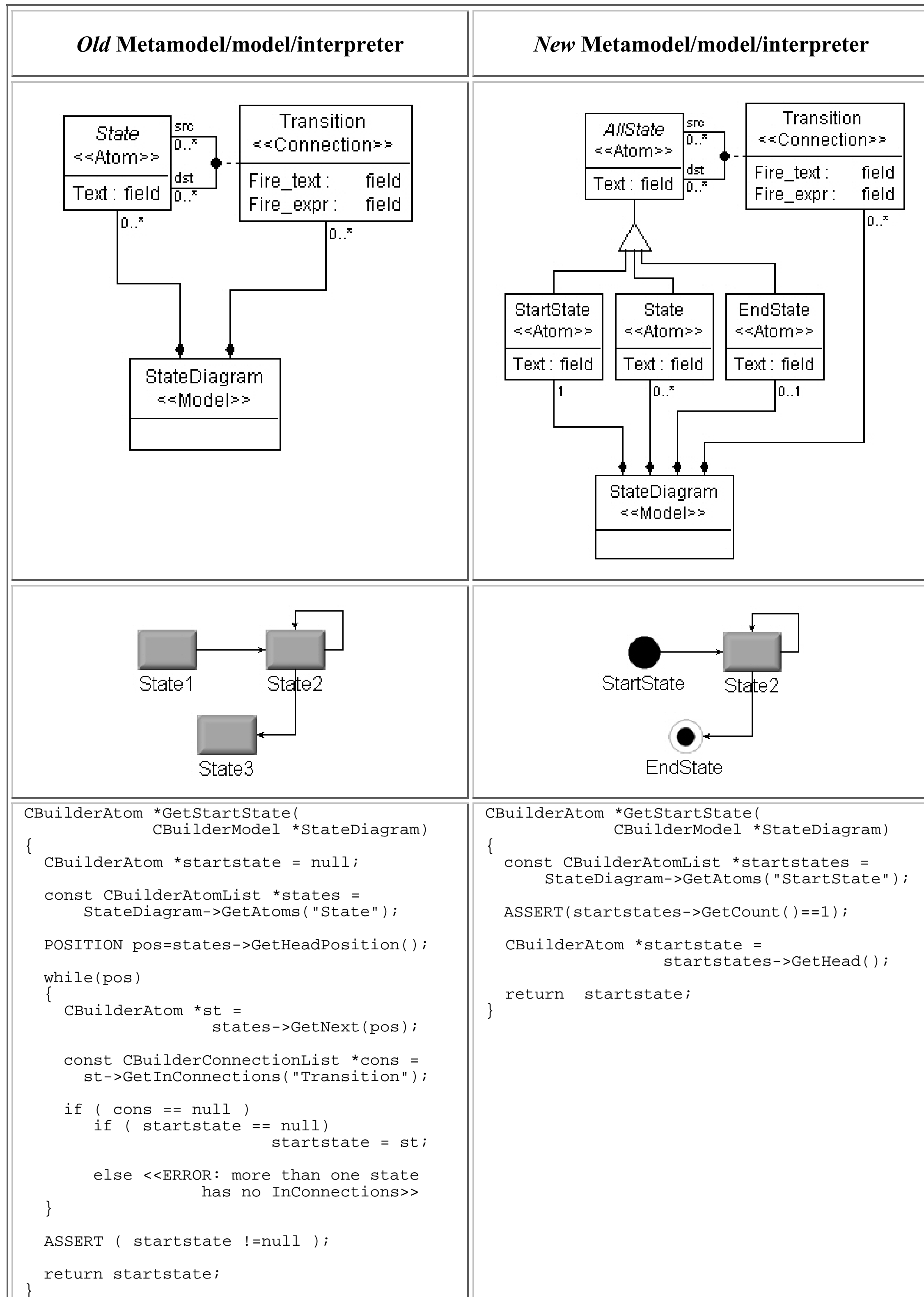

Figure 2. The original meta-model/model/interpreter (left) and the evolved meta-model/model/interpreter (right)

3. If the formal specification is unavailable for the meta-model transformation (e.g., only meta-model$_0$, meta-model$_n$, model$_0$, model$_n$ and interpreter$_0$ in Figure 1 are available, and all of the intermediate transformation specification deltas are absent), a rigorous algorithm has to be developed for the meta-model difference comparison between meta-model$_0$ and meta-model$_n$. More importantly, this algorithm must have the capability to embody enough knowledge to infer the low-level interpreter difference (i.e., Δ_I in Figure 1) that is used for the code transformation. Developing such a super intelligent algorithm is very arduous and time-consuming, if not unfeasible.
4. From our previous experience, complex model interpreters can reach several thousand lines of source code. The need to make the interpreters applicable to the new meta-model requires the invasive capability to alter the interpreter source code. Thus, a mature parser must be constructed to parse the underlying interpreter source. In addition, a powerful program transformation engine is also required to perform the large-scale adaptations to the interpreters according to the model transformation specifications that are described at the model level.

3. A Generative Solution: Model Interpreter Evolution Architecture

With respect to the challenges enumerated in the previous section, we are investigating the feasibility of utilizing the Design Maintenance System (DMS) [5] to support parsing and source-level transformation of the interpreter implementations. The core component of DMS is an Abstract Syntax Tree (AST) term rewriting engine that supports powerful capabilities for pattern matching and source transformation. An important feature of DMS is the source-to-source transformation rules that can be applied to modify a large cross-section of a code base. In our approach, domain models are represented in the Generic Modeling Environment (GME) [11]. We provide a generative approach [6] to enforce a mapping from meta-model transformations to the corresponding interpreter transformation through a model-driven

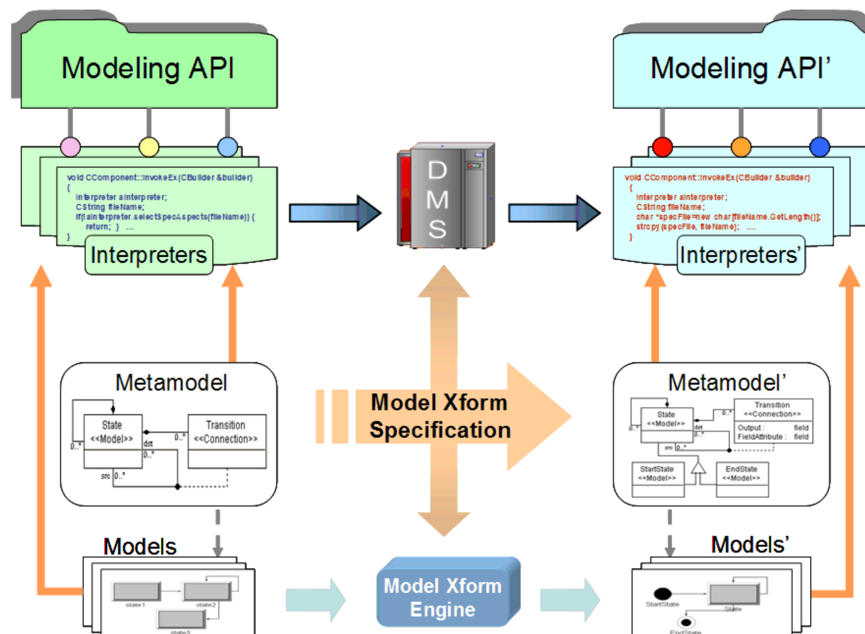

Figure 3. Overview of Model Interpreter Evolution Architecture

program transformation (MDPT) technique. In previous work, we have applied MDPT to evolve a large avionics application based on properties of models [9], which is realized by generating the DMS rewriting rules from high-level modeling specifications. This position paper describes an approach that applies MDPT to model interpreters, not the underlying computer-based system.

To implement and evaluate this approach, a Model Interpreter Evolution Architecture (MIEA) is under investigation (see Figure 3). As the meta-model evolves, a formal specification is needed to represent each step of the model transformation. The specification, either represented in plain text (e.g., C-SAW [7]) or in a visual/modeling language (e.g., GReAT [2]), is then fed into the specific model transformation engine that will update the corresponding models. In addition, this model transformation specification should contain the information for transforming the underlying model interpreters. Such information can be divided into two parts: 1) a pattern description of the interpreter signature, and 2) the replacement rule to perform the transformation. The transformation specification will be used to generate the DMS rewriting rules to transform the original interpreter into a new one that matches the changes to the meta-model. In fact, allowing domain developers to specify explicitly the DMS rewriting rules in the process of model transformation is a simple and straight-forward approach. However, the domain experts/developers would most likely not possess the knowledge to understand the complicated, low-level DMS rules. Such a reality necessitates a high-level specification from which the DMS rules can be generated.

The combination of the model transformation specification and the DMS program transformation engine, as illustrated in Figure 3, could provide a potential solution for challenges 2 and 4 that were identified in Section 2. With respect to the third challenge, there are several researchers investigating model difference comparison algorithms, such as [3]. Incorporating an existing well-developed algorithm can help to automate a set of DMS rewriting rules based on model differences. Several simple examples are listed below:

1. *Differences of names for any model entities, relationships and attributes*. For instance, a model named as "Model1" is now modified as "Model2." The corresponding DMS rules can be generated as:

```
rule ChangeName (id:identifier):
          expression_statement -> expression_statement =
"\id -> GetModels(\"Model1\");" -> "\id -> GetModels(\"Model2\");".
```

   As a consequence, after applying this "ChangeName" rewriting rule to the DMS engine, every statement like **GetModels("Model1")** would be replaced by **GetModels("Model2")** in the interpreters (assuming the name "Model1" is unique in the meta-model diagram).

2. *Differences of model types.* There are several types involved in the GME model entities, such as model, atom, reference and set [1]. For example, within a "Networking" domain, there may be an atom entity named "Router." During the domain evolution, perhaps the "Router" *atom* is converted to a "Router" *model* (a model can contain other atoms and models, but an atom represents a final entity). The DMS rule fragment is:

```
rule ChangeModelType (id:identifier):
          expression_statement   ->  expression_statement =
 "\id -> GetAtoms(\"Router\");"  ->  "\id -> GetModels(\"Router\");".
```

   After applying the "ChangeModelType" rewriting rule to the DMS engine, every statement like **GetAtoms("Router")** would be transformed into **GetModels("Router")** in the interpreters.

3. *Differences of attribute types*. In GME, an attribute type can be of type String, Integer, Double, Boolean, or Enumeration. The following DMS rule presents an example for changing a model entity Router's attribute "Speed" from an Integer type to a Double:

```
rule ChangeAttrType ():
            declaration_statement ->  declaration _statement  =
            "int Router_speed;"   ->  "double Router_speed;".
```

   After applying this "ChangeAttrType" rewriting rule to the DMS engine, every declaration statement like "**int Router_speed;**" would be transformed into "**double Router_speed;**" in the transformed interpreter. A rigorous naming rule should be applied to the identifier used in the interpreters to indicate the attribute with the changed type. In this example, the identifier ("Router_speed") is named as the entity name ("Router") followed by an underscore and the attribute name ("speed"). This example emphasizes the need to specify formally the effect of a model interpreter.

Nevertheless, only a few changes (i.e., those restricted to the modeling tool API invocation, such as **GetModels** and **GetAtoms** in the above examples) can often be automated easily. Other types of changes (e.g., OCL constraint changes) to the meta-model may only require modifications to the models and will not affect the interpreters. However, most of the general changes (especially those drastic semantic variations) made to the meta-model can hardly be captured and reflected in the interpreters through the meta-model difference comparison algorithm alone. This problem again triggers the necessity for the explicit formal specification of the meta-model transformation.

4. Conclusion and Future Work

This paper outlines several challenges involved in model interpreter evolution and proposes a generative approach to implement the automatic evolution of the model interpreters by the transformation rules that are generated from high-level specifications. The initial architecture for model interpreter evolution is being investigated to verify the feasibility of this approach and its limitations.

There is another external factor that may force interpreter evolution. Although less frequent than meta-model evolution, it is possible that the API provided by the modeling tool (i.e., the interface coming out of the modeling API at the top of Figure 3) will also change. As an example, a modeling tool may change its underlying internal data structures (e.g., a modeling tool that moves from Microsoft COM-based containers to a C++ STL style of containers). The future work will also investigate the evolution of model interpreters that are hard-coded to a specific API in the presence of changes to the host modeling tool. Furthermore, experimental studies will be conducted to evaluate the results of this research using well-defined metrics to compare manual efforts with the proposed automated transformation approach.

5. Acknowledgement

This work is supported by the DARPA Information Exploitation Office (DARPA/IXO), under the Program Composition for Embedded Systems (PCES) program.

6. References

[1] *The Generic Modeling Environment: GME 4 User's Manual*, Institute for Software Integrated Systems, Vanderbilt University, 2004.

[2] Aditya Agrawal, Gábor Karsai, and Ákos Lédeczi, "An End-to-End Domain-Driven Software Development Framework," *Domain-Driven Development track, 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, 2003, pp. 8-15.

[3] Marcus Alanen and Ivan Porres, "Difference and Union of Models," *Proceedings of the UML 2003 Conference*, Springer-Verlag LNCS 2863, San Francisco, California, October, 2003, pp. 2-17.

[4] Ira D. Baxter, "Design Maintenance Systems," *Communications of the ACM*, 1992, pp. 73-89.

[5] Ira Baxter, Christopher Pidgeon, and Michael Mehlich, "DMS: Program Transformations for Practical Scalable Software Evolution," *26th International Conference on Software Engineering*, Scotland, UK, May, 2004, pp. 625-634.

[6] Krzysztof Czarnecki and Ulrich Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.

[7] Jeff Gray, Janos Sztipanovits, Douglas C. Schmidt, Ted Bapty, Sandeep Neema, and Aniruddha Gokhale, "Two-level Aspect Weaving to Support Evolution of Model-Driven Synthesis," in *Aspect-Oriented Software Development*, (Robert Filman, Tzilla Elrad, Mehmet Aksit and Siobhán Clarke, eds.), Addison-Wesley, 2004, Chapter 30.

[8] Jeff Gray, Juha-Pekka Tolvanen, Matti Rossi, and guest editors, "Special Issue: Domain-Specific Modeling with Visual Languages," *Journal of Visual Languages and Computing*, Volume 15, Issues 3-4, 2004, pp. 207-209.

[9] Jeff Gray, Jing Zhang, Yuehua Lin, Suman Roychoudhury, Hui Wu, Rajesh Sudarsan, Aniruddha Gokhale, Sandeep Neema, Feng Shi, and Ted Bapty, "Model-Driven Program Transformation of a Large Avionics Framework," *Generative Programming and Component Engineering (GPCE 2004)*, Springer-Verlag LNCS, Vancouver, BC, October, 2004.

[10] Gábor Karsai, Miklos Maroti, Ákos Lédeczi, Jeff Gray, and Janos Sztipanovits, "Composition and Cloning in Modeling and Meta-Modeling," *IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling)*, 2004, pp. 263-278.

[11] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai, "Composing Domain-Specific Design Environments," *IEEE Computer*, 2001, pp. 44-51.

[12] Sandeep Neema, Ted Bapty, Jeff Gray, and Aniruddha Gokhale, "Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems," *First ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*, Springer-Verlag LNCS 2487, Pittsburgh, PA, October 6-8, 2002, pp. 236-251.

[13] Jonathan Sprinkle and Gábor Karsai, "A Domain-Specific Visual Language For Domain Model Evolution," *Journal of Visual Languages and Computing*, vol. 15, no. 2, 2004.