Isomorphism in Model Tools and Editors

George Edwards
Blue Cell Software
Los Angeles, CA, USA
george@bluecellsoftware.com

Yuriy Brun
University of Washington
Seattle, WA, USA
brun@cs.washington.edu

Nenad Medvidovic
University of Southern California
Los Angeles, CA, USA
neno@usc.edu

***Abstract*—Domain-specific languages (DSLs) are modeling languages that are customized for a specific context or project. DSLs allow for fast and precise modeling because the language features and constructs can be precisely tailored based on the needs of the modeling effort. There exist highly customizable model-editing tools that can be easily configured to support DSLs defined by end-users (e.g., system architects, engineers, and analysts). However, to leverage models created using these tools for automated analysis, simulation, and code generation, end-users must build custom analysis tools and code generators. In contrast to model editors, the implementation and maintenance of these analysis and code generation tools can be tedious and hampers the utility of DSLs. In this paper, we posit that analysis and code generation tools for DSLs are, in fact, isomorphic to model editing tools. The implication of this insight is that model editors, analysis tools, and code generators can be treated as analogs conceptually and architecturally, and highly customizable analysis and code generation tools for DSLs can be built using the same approach that has already proven successful for the construction of DSL model editors.**

I. Introduction

Domain-specific models are becoming increasingly utilized in consumer electronics, distributed robotics, different classes of embedded systems, and many other complex, software-intensive systems. Modeling allows for both rigorous design analysis, which can improve overall system quality, and automated code generation, which can accelerate development. In particular, *domain-specific* models can be tailored to a relevant class of applications, allowing architects to concentrate only on those decisions that are of importance in the domain. In the context of a given domain, domain-specific languages (DSLs) result in more concise and intuitive models than standardized modeling languages such as UML.

However, because a DSL may contain arbitrary constructs, today, model analysis and code generation tools must be customized to work with each individual DSL. This places an undue burden on the architects who wish to use DSLs: they are forced to spend time building intricate, complex model analysis and generation tools, rather than reuse existing ones. In contrast, model-driven engineering (MDE) platforms, such as the Generic Modeling Environment (GME) [1] and the Eclipse Graphical Modeling Framework (GMF) [2], ease the creation of customized *model editors* to support a given DSL. Figure 1 shows that software architects can construct domain-specific model editors using these MDE platforms quickly and easily by defining a *metamodel* — a formal specification of a DSL.

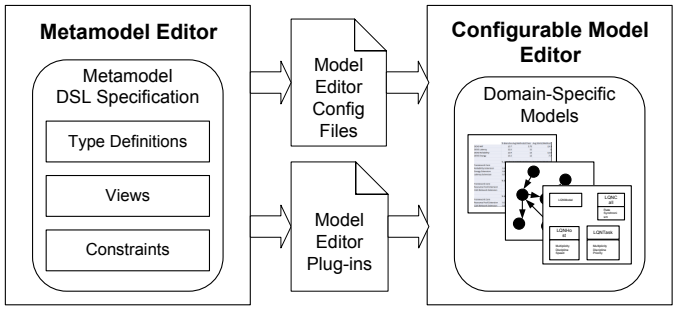

Fig. 1. MDE platforms use a metamodel to generate and configure a model editor.

In this paper, we argue that existing MDE platforms make an implicit, yet crucial, distinction between model editors and model interpreters (automated analysis, simulation, and code-generation tools). These platforms expect editors and interpreters to be built in vastly differing ways — editors through configuration of an off-the-shelf tool while interpreters through manual, time-consuming, and error-prone end-user development [4]. As a result, while DSLs have experienced widespread industry adoption in large-scale defense and aerospace programs, such adoption in small business and desktop application development [6] has remained relatively limited. We posit that the distinction at the heart of MDE is a false dichotomy, i.e., that model editors and interpreters are, in fact, isomorphic. This isomorphism becomes apparent if one regards the process of rendering models within an editor as just one possible form of model interpretation.

The implication of this insight is that model editors, analysis tools, and code generators can be treated as analogs conceptually and architecturally. More importantly, model analysis and code generation tools can be constructed using the same method that has already proven successful for the construction of model editors. This paper presents an argument in support of this guiding insight. The paper then outlines a strategy for constructing an MDE tool-chain that exploits the isomorphism of model editors and interpreters, treats interpreters in the same manner that editors have been treated in the past, and thereby allows their automated construction and reuse.

The rest of this paper is organized as follows. Section II overviews important concepts. Section III explains how model editors and interpreters can be treated isomorphically. Finally, Section IV reviews related work.

978-1-4577-1639-3/11/$26.00 © ASE 2011, Lawrence, KS, USA

II. Domain-Specific Modeling

To understand and appreciate the shortcomings of existing MDE platforms, it is necessary to understand the processes required to leverage domain-specific models for automated quality analysis and code generation. This section summarizes the key concepts and processes in domain-specific modeling.

A. Model Transformation

Analysis and code generation using domain-specific models is achieved through *model transformation*, which is a process that maps an input model to an output model. Model transformation is one of the central activities in model-driven development paradigms such as model-driven architecture (MDA) and model-driven engineering (MDE) because it allows a single model to be used for a variety of purposes.

In a domain-specific modeling project, model transformations are used to map high-level, domain-specific design models to other types of models that can be directly analyzed and executed. Model transformations are commonly used to automatically generate the type of models required by a specific analysis tool or implementation code for a specific run-time platform. For example, a transformation might map an architecture description language (ADL) to an executable C++ program. In practical terms, a model transformation is usually implemented by a plug-in to the model editor, which we refer to as a *model interpreter*, as depicted in Figure 2.

B. Metamodeling

Recall that current MDE platforms use a DSL specification called a metamodel to automatically synthesize a custom model editor that supports the DSL. Software engineers specify the metamodel using a metamodel editor and metamodeling language (or *metalanguage*) provided by the modeling platform. The metalanguage provides a set of types, or *metatypes*, that an engineer instantiates to define the domain-specific modeling types. These metamodeling concepts are depicted in Figure 3.

Once a metamodel has been defined, the engineer invokes a special built-in metamodel interpreter, or *metainterpreter*, to generate configuration files and/or plug-ins for a configurable, pluggable model editor framework, also provided by the MDE platform. The model editor framework uses the configuration files and plug-ins to visually render the domain-specific model

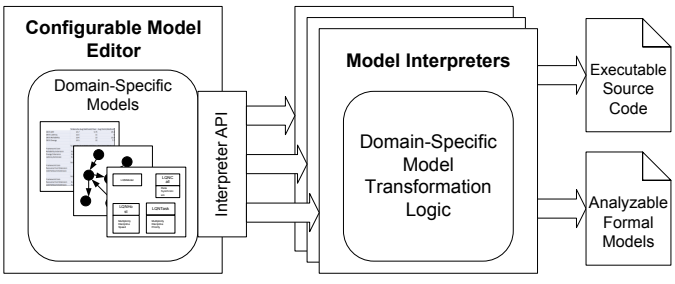

Fig. 2. MDE platforms provide pluggable interfaces and APIs for model interpreters to extract and manipulate the information contained in models and generate other types of artifacts.

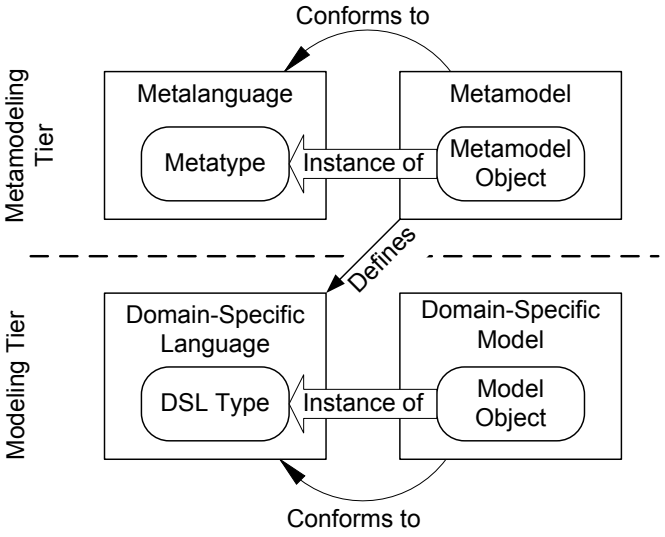

Fig. 3. Typing and instantiation relationships among metamodels and domain-specific models.

types, manage an internal representation of the model, and enforce constraints on model well-formedness, according to metamodel specification (recall Figure 1).

C. Modeling Semantics

When domain-specific modeling with metamodeling is used, one set of semantics are applied to the metatypes defined in the metamodel, and a second set of semantics are applied to the types defined in the domain-specific models.

First, the metainterpreter implemented by an MDE platform applies a set of semantics to the metatypes when it performs a model transformation from a metamodel to a set of configuration files for a model editor. These metatype semantics capture how the domain-specific types will be rendered and manipulated in the domain-specific model editor. In other words, these semantics, which we will term *presentation semantics*, define the behavior of the domain-specific types within the run-time environment of the model editor.

Second, model interpreters built by end-users apply a set of semantics to their models when they perform design analysis and code generation. These interpreters define the consequences of the use of domain-specific types within a given context, or *semantic domain*, such as a simulation, analysis tool, or run-time environment. In other words, these semantics, which we will term *analysis semantics* or *code generation semantics*, define the behavior of domain-specific types within run-time environments other than the model editor.

Using existing MDE platforms, presentation semantics are defined in a totally different way than analysis and code generation semantics. Presentation semantics are defined through properties (attributes and associations) attached to the metatypes. For example, a metatype may have attributes that define its shape and color, where it appears on the screen, or what happens when it is clicked in the editor. Since the available metatype properties are fixed by the developer of

the MDE platform *a priori*, there are a finite number of options available to end-users to define the behavior of their domain-specific types within the editing tool. This limitation means that end-users have flexibility to change the presentation semantics of their types, but only within certain constraints. However, it also means that end-users can very quickly and easily create customized model editors for their DSLs, without having to write any code — all they have to do is set a few properties in their metamodels.

In constrast, analysis and code generation semantics are defined by the transformations implemented in model interpreters. Since end-users are free to implement any transformation they wish, these semantics are totally unconstrained. However, defining analysis and code generation semantics this way leads to burdensome interpreter development and maintenance, as well as other negative side effects. For example, semantic definitions become buried in low-level source code, making it difficult for a new engineer joining a project to determine what the semantics of a specific type are.

Since today's MDE platforms only generate a custom model editor (and not other tools), the metatype properties provided are intentionally limited to definition of visualization, presentation, and editing semantics. In the next section, we discuss how analysis and code generation semantics could also be defined through metatype properties, leading to some limitations in the semantic options available to end-users, but also completely avoiding the painful interpreter development and maintenance required by existing MDE platforms.

III. Isomorphism in Tools and Editors

Existing MDE platforms automatically synthesize only one of the tools needed for an end-to-end toolset — the domain-specific model editor. This paper proposes that the same mechanism used by current MDE platforms to synthesize domain-specific model editors can be used to synthesize the other tools needed for an end-to-end toolset — the domain-specific analysis tools and code generators. Current MDE platforms cannot provide built-in model interpreters for analysis and code generation because their metatypes lack sufficient semantics for their instances to be automatically mapped to other forms, such as executable code. Rather, the metatypes only contain sufficient semantics to automatically map metatypes to graphical display elements.

Recall that existing MDE platforms synthesize domain-specific model editors by (1) incorporating into metamodels information about how model objects should be presented in the synthesized domain-specific editor, (2) implementing a highly flexible, configurable model editor that allows model objects to be presented in a variety of ways, and (3) implementing a metainterpreter that generates configuration files or plug-ins for the configurable model editor that specify how each object should be presented, according to the information in the metamodel. We propose synthesizing domain-specific model interpreters in an analogous way:

1) Define metatypes and metatype properties that capture the semantics of domain-specific types for a particular semantic domain. The semantic options need to be defined *a priori* by the MDE platform developers, and properties that allow metamodelers to select from among those options need to be included in the MDE platform's metalanguage.
2) Implement a configurable model interpreter that allows model objects to adopt a variety of semantics. Conceptually, the configurable model interpreter can be viewed as a library of model transformation operations that can be flexibly applied to model elements in multiple ways to account for semantic variability.
3) Implement a corresponding metainterpreter that generates configuration files or plug-ins for the configurable model interpreter that specify the semantics of each domain-specific type, according to the metatype property values assigned to the object's type definition in the metamodel.

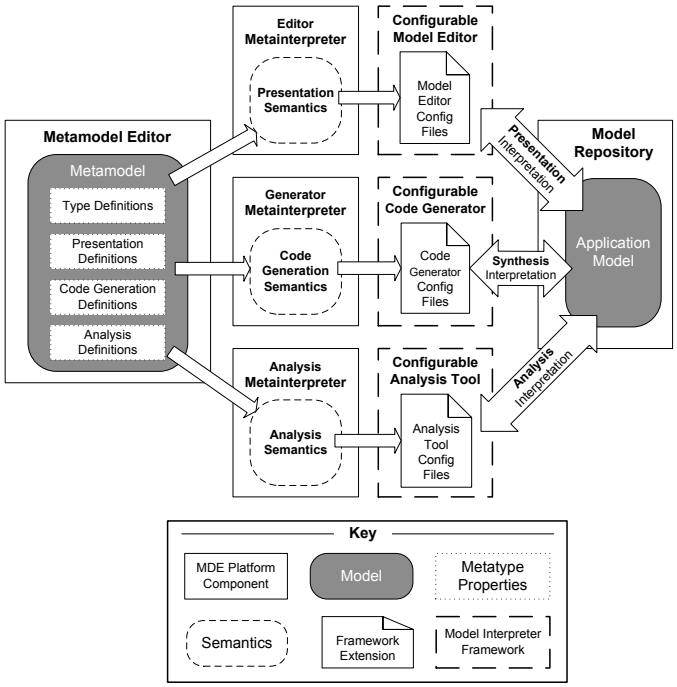

Fig. 4. Using Domain-specific models for system analysis, code generation, and model editing.

Figure 4 depicts the similarity of the ways tools and editors use models and illustrates their isomorphism.

In this approach, the primary functions of a metainterpreter are (1) deriving the semantics of each domain-specific type, and (2) realizing the semantics of each type in a set of transformation rules. The semantics of each domain-specific type are selected from the set of possible semantics permitted by the MDE platform. Each possible semantics corresponds to a different usage of the configurable interpreter's model transformation functionality. Therefore, a metainterpreter includes (1) a function that takes as parameters the property values of a metatype instance and returns a semantic definition from the set of possible semantics, and (2) a function that takes a semantic definition as a parameter and returns a set of

operations to invoke in the configurable interpreter.

Notionally, the configurable intepreter can be thought of as a virtual machine whose instruction set is the set of implemented transformation operations. In this analogy, the metainterpreter is akin to a compiler whose function is to generate programs to be executed by the virtual machine. Transformation steps that depend on built-in semantic assumptions (*i.e.*, not metatype properties) are "hard-coded" into the configurable interpreter and are protected from modification. Transformation steps that vary based on metatype properties are "programmable" via the metamodel.

Alternatively, the combination of the metainterpreter and configurable interpreter can be viewed as a compiler-compiler that generates compilers for DSLs. Both our approach and compiler-compilers generate programs for translating input models or programs specified in one language to a different target language. However, we propose a very different approach to semantic specification, which is less powerful in theory (arbitrary semantics cannot be captured), but is also much more practical. By utilizing a pre-decided set of properties to define semantics, metamodelers do not have to develop complex semantic definitions using denotational semantics, structural operational semantics, or some other formalism. This makes semantic specification much simpler and easier.

IV. Related Work

Other research efforts in the area of domain-specific modeling, such as Cadena and PACC, are also developing meta-configurable toolchains (although different names are used, such as "reasoning framework" or "analysis framework"). These related projects have developed toolchains with powerful capabilities and useful features, but have not fully recognized the broad applicability and significant ramifications of this approach, and therefore they have not created nor implemented a generalized architecture for creating meta-configurable toolchains, as we do in this paper.

Cadena [3] is a model-driven toolchain for component-based systems that uses CALM as its metamodeling language. CALM is based on a three-tiered typing system: a *style* tier, a *module* tier, and a *scenario* tier. Bogor is a highly extensible model checking framework that can be applied to Cadena models. Bogor allows engineers to reuse the analysis infrastructure while customizing selected constructs and behaviors. Bogor allows engineers to extend the input language to the model checking framework with domain-specific abstractions. Cadena and Bogor elegantly integrate metamodeling facilities with model checking. However, some of the mechanisms that enable extension and reuse in Bogor are specific to model checking, making them difficult to generalize and apply to other types of analysis and code generation.

The DUALLY framework [5] supports interoperability among architecture DSLs. DUALLY leverages engineer-defined mappings between DSL metamodels and a core set of architectural concepts codified in a metamodel, called A_0. Thus, the analysis and code generation tools available for any DSL with a defined mapping can be applied to architectural models specified in other DSLs. Although DUALLY eliminates the need to manually program model transformations, it still requires defining the mappings between languages. Therefore, DUALLY automates some MDE activities as the expense of others.

The ALFAMA workbench [7] automates the construction of DSLs for application frameworks. ALFAMA leverages an aspect-oriented domain-specific modeling layer that defines *specialization aspects* that modularize framework hot-spots (extension points) and associates those aspects with DSL concepts. The DSM layer allows engineers to bypass metamodel development, resulting in an approach that is, in some ways, the reverse of ours: rather than enhancing metamodels to eliminate interpreter development, ALFAMA enhances interpreters to eliminate metamodel development. This has many merits but also some drawbacks. First, high-level metamodels are more maintainable than low-level source code. Therefore, it may be advantageous to expend some additional effort on metamodel development rather than DSM layer development. Second, inferring DSLs from framework hot-spots tightly couples the DSL to a particular framework.

Model-driven development tools produced by the Predictable Assembly from Certifiable Components (PACC) Initiative, such as the PACC Starter Kit, include limited forms of meta-configurability. The Construction and Composition Language (CCL) is used within PACC for specifying component behaviors and assemblies. CCL also includes constructs for specifying connectors as services of a component execution environment. CCL has a strong emphasis on precise behavioral semantics for rigorous analysis, and is less concerned with domain-specific extensibility. The PACC Starter Kit (PSK) is a model-driven development environment that includes a performance analysis framework and a model-checking framework. The performance analysis framework transforms component-based architectural models into a form that supports rate monotonic analysis for prediction of worst-case response times. The model-checking framework transforms architectural models into the input language of a software model checker that verifies safety and security properties.

References

[1] "The Generic Modeling Environment," www.isis.vanderbilt.edu/Projects/gme/.

[2] "The Eclipse Graphical Modeling Framework," www.eclipse.org/modeling/gmf/.

[3] G. Jung *et al.*, "A Type-centric Framework for Specifying Heterogeneous, Large-scale, Component-oriented, Architectures," in *Generative Programming and Component Engineering*. ACM, 2007, p. 42.

[4] G. Karsai *et al.*, "On the Use of Graph Transformation in the Formal Specification of Model Interpreters," *J. Universal Computer Science*, vol. 9, no. 11, pp. 1296–1321, 2003.

[5] I. Malavolta *et al.*, "Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies," *IEEE Transactions on Software Engineering*, 2009.

[6] P. Mohagheghi *et al.*, "Where Is the Proof? - A Review of Experiences from Applying MDE in Industry," in *4th European Conf. on MDA Foundations and Applications*, Berlin, Germany, June 2008, pp. 432–443.

[7] A. L. Santos *et al.*, "Automating the Construction of Domain-Specific Modeling Languages for Object-Oriented Frameworks," *Journal of Software and Systems*, 2010.