

Customizable Output Generation in Modeling Environments Using Pipelined Domains

Dragan Milicev

School of Electrical Engineering, Dept. Comp. Sc. & Eng.

University of Belgrade

emiliced@etf.bg.ac.yu

Abstract

Domain-specific modeling and metamodeling environments most often base their output generation capability on wizards, output templates, grammar-based transformers, or hard-coded output generators. The complexity of the specification process for such generators, and their dependence on the domain do not encourage customization, flexibility, and reuse. This paper proposes a solution to this problem. In the proposed approach, the domains are (meta) modeled using the standard object-oriented paradigm. Second, the generation of a model in the target domain from a model in the source domain is specified using extended UML object diagrams that allow specification of conditional, repetitive, and sequential creation of instances of the target domain's abstractions. Finally, the transformation of models may be performed in a pipelined fashion, where each domain model and mapping may be either created from the scratch or reused from the repository. This approach allows more efficient, incremental building of more abstract domains and their mapping into less abstract domains, because each transformation step is much less complicated to specify, maintain, and reuse. Furthermore, by simple choosing another pipeline, different versions of the ultimate implementation from the same initial high-level, user-defined model may be obtained automatically. A prototypal supporting tool has been implemented and briefly presented in the paper.

Keywords: Domain-specific modeling, metamodeling, object-oriented modeling, model transformation, The Unified Modeling Language (UML)

1 Introduction

Apart from the important roles of domain-specific modeling and metamodeling environments (included, but not limited to CASE tools) in specifying, documenting, and visualizing systems, the purpose of modeling tools is most often system construction [4]. 'Construction' means producing output from the system specification that may be interpreted by a certain external environment to provide the desired system's behavior. The examples include, but are not limited to: documentation, source code in a certain programming language, database scheme, hardware description, specification of implementation interpretable by a runtime environment, or any other formal structure. Most often, the existing tools base their output generation capability on wizards, output templates, grammar-based transformers, or hard-coded generators. The complexity of the specification process for such generators, and their dependence on the domain do not encourage customization, flexibility, and reuse, as briefly explained in this paper. Therefore, a solution to this problem is proposed here.

In the proposed approach, the domains are (meta) modeled using the standard object-oriented paradigm. Second, the generation of a model in the target domain from a model in the source domain is

specified using extended UML object diagrams that allow specification of conditional, repetitive, and sequential creation of instances of the target domain's abstractions. Finally, the transformation of models may be performed in a pipelined fashion, where each domain model and mapping may be either created from the scratch or reused from the repository. This approach allows more efficient, incremental building of more abstract domains and their customizable mapping into less abstract domains, because each transformation step is much less complicated to specify, maintain, and reuse.

The paper continues as follows. Section 2 is a brief overview of the existing approaches and their weaknesses. Section 3 presents the core ideas of the approach. Section 4 contains a short description of the prototypal supporting tool. The paper ends with a conclusion. More details and comments may be found in [12].

2 Overview of the related work

A broadly accepted approach to output generation is the use of wizards. Wizards allow the user to specify parameters of the generation in a sequence of interactions, and then produce the output. However, wizards are only a manifestation of the background transformational process specified at the tool's development time. The subject of this paper is the method of developing transformations and reusing them. A sound assumption is that the existing wizards are implemented using some of the approaches discussed below.

A first approach is the development of a hard-coded transformer, where the developer has to program the output generation in a scripting language. Some of the drawbacks of this approach are evident: (1) The process of specifying may be extremely tedious, time-consuming, and error-prone. (2) The developer must deal with the complexity of the target domain (e.g., target language syntax and semantics). (3) Possibly available general-purpose and reusable code generators from some common domains are not reused. (4) The transformer is not reusable. Various variants exist that partially cure the complexity and maintenance problem. A first one is using the Visitor design pattern [6] to de-couple the structural model specification from the transformer's code. Another approach uses output templates that represent textual specifications of the desired output, parameterized with references to source model variables that are replaced at the transformation time [21]. However, these approaches do not support a more formal, preferably visual mapping specification, and do not promote model pipelining proposed here.

It is similar with a number of approaches that use structural transformations of grammar-based models and various rule-based techniques [7, 8, 9]. Although their goal is the same as here, there are a number of differences. First, they primarily deal with textual models (or, more generally, with strings of entities). Second, their 'metamodels' are expressed with grammars, where the entities are

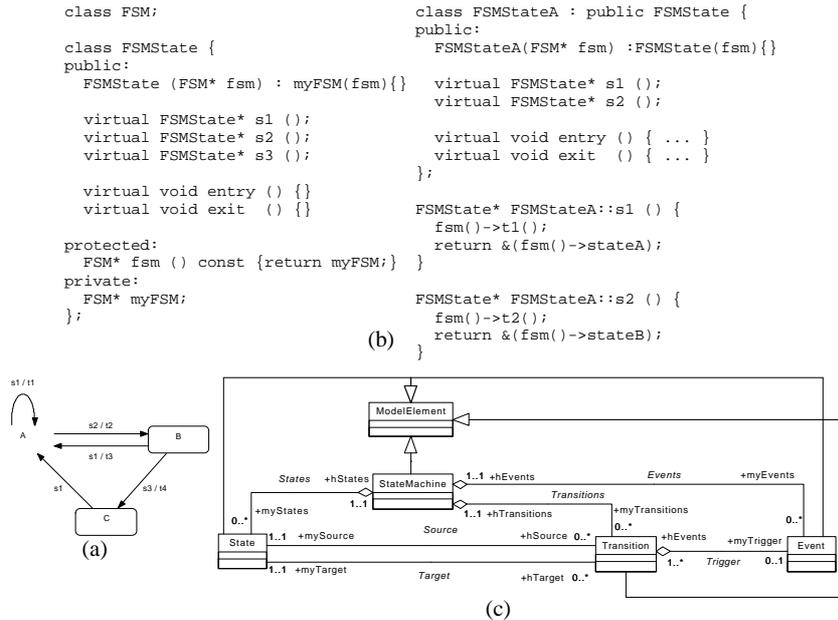


Figure 1: Demonstrational example: Code generation for state machines. (a) A sample state machine. (b) An excerpt of the generated code. (c) The metamodel.

defined hierarchically (using sub-entities), and where recursion is the main difficulty, instead of the object-oriented paradigm that is used here. The main purpose of the supporting environments in that case is to build an internal representation (derivation tree) from the user-defined model (textual program) by parsing it, and then to transform this internal representation into the target internal representation. Thus, the internal structure of the model is inherently a tree. In the modeling environments that use object-oriented paradigm for metamodeling as the one proposed here, there is no need for the parsing phase, because the user explicitly creates the instances of abstractions and their links. Therefore, the model representation is a graph of objects (instances of classes) connected with links (instances of associations). This is why the approach presented here may be considered as a more general structural transformation. Finally, defining a grammar for a certain domain and specifying the mapping between the grammars may be a difficult task because it requires more sophisticated work than defining or just understanding the metamodels specified in object-oriented terms.

A research field also related to metamodeling is the field of visual programming languages (VPL) [1, 5, 17]. However, the underlying metamodels of VPLs are also grammars [5] or other formal models. Consequently, VPL metaenvironments have the same characteristics as the grammar-based environments described previously. Besides, the problem of the model transformation, which is the subject of this paper, is not considered as an important one in the field of VPLs.

Automatic generation of tools and applications has been an attractive discipline for years, and a lot of customizable CASE and meta-CASE tools, both commercial and academic ones, are available at the moment [18, 19, 20, 21, 22, 23, 24, 25, 26]. All these tools provide programming interfaces to their metamodels through which the user may access the models in the generated CASE tools to produce the output. However, output generation is always specified using the described hard-coded approach. As a conclusion, to the best of our knowledge, we are not aware of any other approach that is closely related to the one presented here.

3 Idea of the proposed approach

A simple example that will be used here is shown in Figure 1. It deals with customizable generation of C++ code out from the models that represent state machines. The code is generated using the State design pattern [6]. It is assumed that the user has specified a state machine named FSM shown in Figure 1a. For this example, several classes should be generated in the output C++ code. The first is named FSM. It contains operations that correspond to the events of the state machine. The second class is abstract and is named FSMState. It contains one polymorphic operation for each event. Finally, one class derived from FSMState is generated for each state. It overrides the operations that represent those events on which the state reacts. These operations perform transitional actions and return the target state.

The first assumption of the proposed approach is that the domains are (meta) modeled using the standard object-oriented paradigm. The most important abstractions used for metamodeling are Class, Attribute, Operation, Generalization, and Association. For the definition of the semantics of these abstractions, a core (structure-oriented) subset of the UML metamodel [14] is used (it is referred to as the UMLCore domain). A conceptual model of a domain (the metamodel) is defined in terms of instances of the UMLCore abstractions (Class, Association, etc.) and links between them, as in Figure 1c.

A model in a certain domain is again a set of instances of the abstractions and links between them (as instances of associations). 'Abstractions' are instances of Class, and 'associations' are instances of Association in the corresponding metamodel. Each model is organized as a hierarchy of packages that own instances and other nested packages. Figure 2 shows the metamodeling (a) and modeling (b) phases for the example. In the metamodeling phase, (meta-)metamodels (left browser) are the domains UMLCore (contains Class, Association, etc.) and DomainMapping (the metamodel of the mapping specifications, to be described later). (Meta-)Models (right browser) are the (meta-)model of StateMachines from Figure 1c, and OOP, which is a reusable

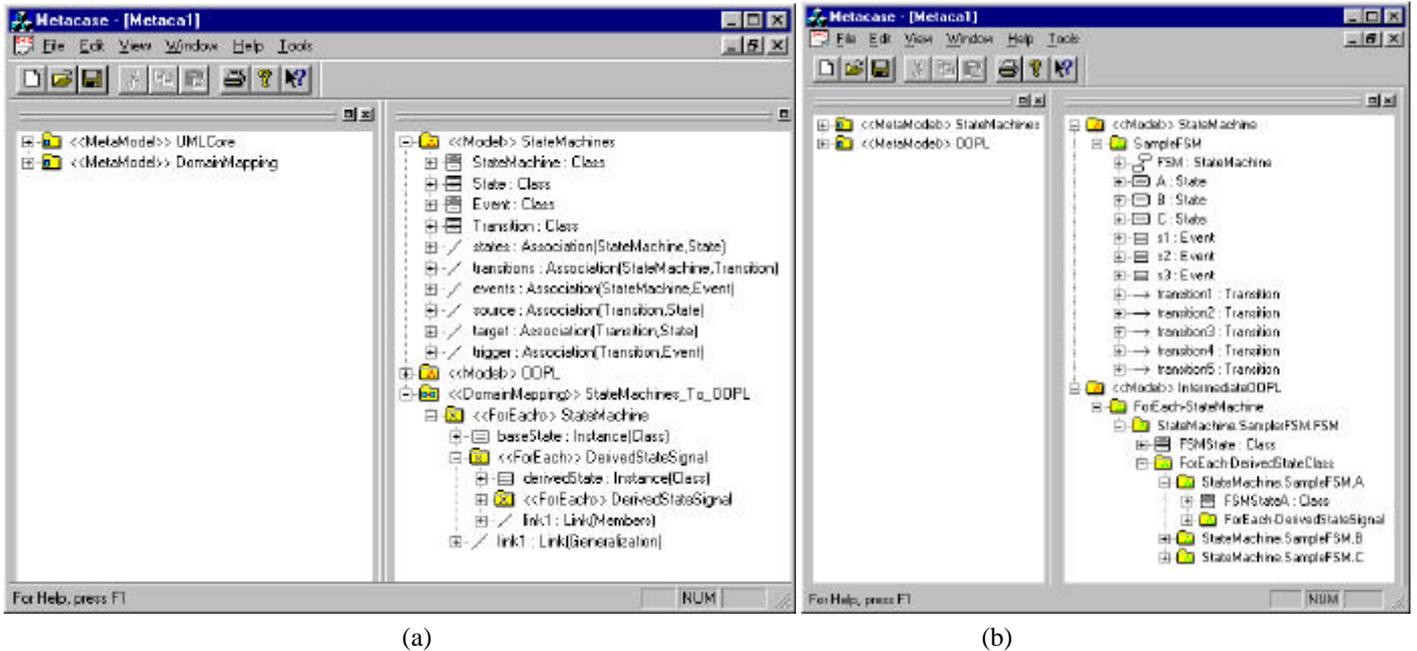


Figure 2: Demonstrational example in the supporting tool. (a) Metamodeling phase. (b) Modeling phase. In both cases the left browser shows the metamodels (domains), and the right one shows the models in the corresponding phase. Each model in the right browser is an instance of exactly one metamodel from the left browser.

domain that is used to generalize target OO programming languages. It contains abstractions that may be mapped directly into the target code (Class, Derivation, MemberFunction, DataMember, FunctionBody, etc., but not Association as in UMLCore). In the modeling phase, the metamodels are StateMachines and OOPL, and the models are concrete examples (instances) of them. For example, the model "StateMachine" is from the "StateMachines" domain and contains a specification of the example from Figure 1a.

Second, the generation of a model in the target domain from a model in the source domain is specified using extended UML object diagrams that allow specification of conditional, repetitive, and sequential creation of instances of the target domain's abstractions. The details may be found in [12]. An example is shown in Figure 3, and its representation in Figure 2a. The packages stereotyped with "ForEach" specify iteration through collections of elements in the source model, and creation of instances and links in the destination model for each iterated element. The specifications are represented again as instances and links from the "DomainMapping" domain. From that specification, the tool generates executable code that may be used as a transformer from a source model ("StateMachine" in Figure 2b) into a destination model ("IntermediateOOPL" in Figure 2b). The intermediate model is then transformed into the source code by a built-in and reusable code generator.

Therefore, the proposed approach introduces one or more intermediate domains in the output generation process. In other words, it simplifies complex and cumbersome transformations of a model into another representation by doing the transformation in multiple steps. This has the advantage that each step becomes simpler and that existing transformation can be reused. Finally, the transformation of models may be performed in a pipelined fashion, where each domain model and mapping may be either created from the scratch or reused from the repository. Thus, by simple choosing another pipeline, different versions of the ultimate

implementation from the same initial high-level, user-defined model may be obtained automatically.

An example is shown in Figure 4. In the metamodeling phase, the developer defines domain (meta)models, and domain-mapping specifications. Because the transformational process is pipelined, the domain models and mapping specifications may be reused from a repository. For the example in Figure 4, code generators from the OOPL domain are reusable. In the modeling phase, the user defines the source model. The desired transformation is chosen with specifying the pipeline. The example in Figure 4 shows that there may exist more transformations from the source (StateMachine) into the intermediate (OOPL) domain, depending on the desired concept for implementing state machines. Both of them may be mapped into several target programming languages, by simply choosing different transformation pipelines.

4 Supporting tool

A prototypal (meta) modeling tool that supports the described approach has been implemented (Figure 2). The tool manipulates with models and generates C++ code that is linked to the tool's core (fixed) part. The same organization of the tool, shown in Figure 5, is used in both phases (metamodeling and modeling). It consists of two parts: the M2Level part manipulates with metamodels, and the M1Level part manipulates with models. Each of the parts has a fixed core part that is used for all domains. The fixed parts provide the organization of the models (hierarchy of packages). The generated parts (M2Gen and M1Gen) are domain-specific. The M2Gen part contains instances of the UMLCore abstractions (which are defined as C++ classes in M2Fix) that define the domain metamodels. The M1Gen part contains C++ code for the classes that implement abstractions from the corresponding domains, along with model transformers. If a domain model is defined using the UMLCore metamodel, the tool generates (using a built-in generator) M2Gen and M1Gen parts that are linked with the Fix parts to produce the modeling tool for that domain.

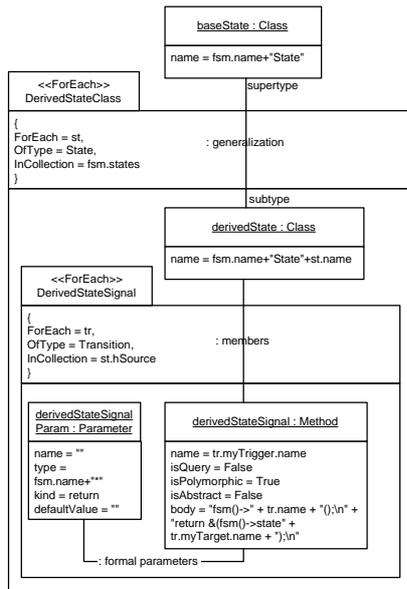


Figure 3: An example of a domain-mapping specification. The diagram shows a part of the specification for the derived state classes and their member functions for the events.

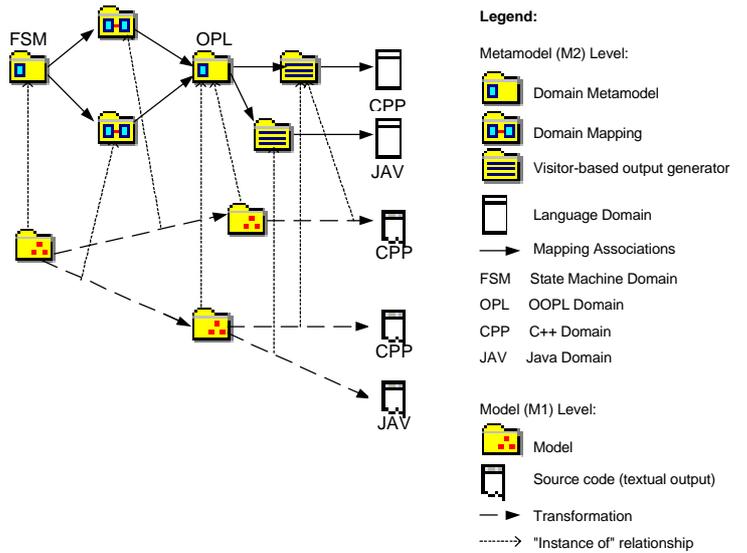


Figure 4: An example of model pipelining. In the metamodeling phase, the domain metamodels and their mapping are specified. In the modeling phase, a user-defined model from the source domain may be automatically transformed into several target models, by choosing a pipeline.

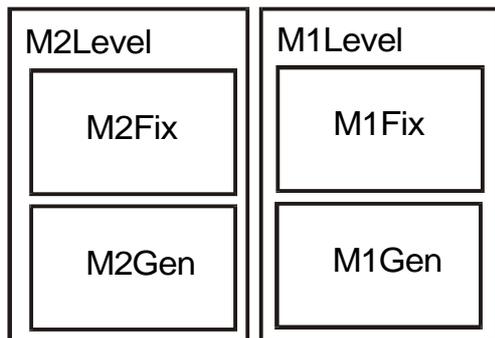


Figure 5: Organization of the tool. The M2Level part manipulates with metamodels. The M1Level part manipulates with models.

The tool supports three different strategies for specifying output generation: domain mapping, Visitor-based hard coded generation [6], and template-based generation. When a domain model is defined, the developer may specify arbitrarily many domain-mapping (represented as models in the DomainMapping domain) or Visitor-based (coded in C++) transformers. If the desired output is a text, templates may be used, too. The tool generates code for the transformers that is a part of the generated domain-specific tool. It is expected that these three techniques may satisfy most of the needs in practice.

Since the tool generates its own extensions as C++ code that is compiled and linked with the Fix parts to produce a modeling environment, the tool is completely flexible and extensible, because it allows arbitrary user-defined extensions written in C++. Its user interface is also modifiable, so the developer may redefine the default generic specification dialogs for the domain abstractions.

Several reusable metamodels (UMLCore, OOPL) and Visitor-based code generators for them have been implemented, too. The future work will concentrate on developing a repository of other reusable domains and transformers, such as relational database, entity-relationship, dataflow, workflow, state-transition, etc.

5 Conclusion

The proposed approach has been successfully used in several complex examples that will be reported elsewhere. It has proved the expectations that it allows more efficient, incremental building of more abstract domains and their mapping into less abstract domains, because each transformation step is much less complicated to specify, maintain, and reuse. Furthermore, it provides customizable application generation in different versions of the ultimate implementation from the same initial high-level user-defined model.

Acknowledgements

The author is grateful to D. Marjanovic, P. Nikolic, M. Ljeskovic, M. Zaric, and Lj. Lazarevic who contributed to the tool implementation and examples.

References

- [1] Anlauff, M., Kutter, P. W., Pierantonio, A., "Montages/Gem-Mex: A Meta Visual Programming Generator," *Proc. 14th IEEE Symp. Visual Languages*, Sept. 1998
- [2] Artsy, S., "Meta-modeling the OO Methods, Tools, and Interoperability Facilities," *OOPSLA'95 Workshop in Metamodeling in OO*, Oct. 1995
- [3] Booch, G., *Object-Oriented Analysis and Design with Applications*, 2nd ed., Benjamin/Cummings, 1994
- [4] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley Longman, 1999
- [5] Costagliola, G., Tortora, G., Orefice, S., De Lucia, A., "Automatic Generation of Visual Programming Environments," *IEEE Computer*, Vol. 28, No. 3, March 1995, pp. 56-66
- [6] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley Longman, 1995
- [7] Garlan, D., Cai, L., Nord, R. L., "A Transformational Approach to Generating Application-Specific Environments," *Proc. Fifth ACM SIGSOFT Symp. Softw. Development Environments*, Dec. 1992, pp. 68-77
- [8] Garlan, D., Krueger, C. W., Staudt, B. J., "A Structural Approach to the Evolution of Structure-Oriented Environments," *Proc. ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. Practical Softw. Development*

Environments, Dec. 1986

- [9] Habermann, A. N., Notkin, D. S., "Gandalf: Software Development Environments," *IEEE Trans. Software Engineering*, Vol. 12, No. 12, Dec. 1986, pp. 1117-1127
- [10] Karrer, A. S., Scacchi, W., "Meta-Environments for Software Production," *Report from the ATRIUM Project*, Univ. of Southern California, Los Angeles, CA, Dec. 1994, <http://www2.umassd.edu/SWPI/Atrium/localmat.html>
- [11] MetaModel.com, *Metamodeling Glossary*, <http://www.metamodel.com>
- [12] Milicev, D., "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments," submitted for publication, available from the author on request
- [13] Nordstrom, G., Sztipanovits, J., Karsai, G., Ledeczi, A., "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments," *Proc. IEEE ECBS'98 Conf.*, 1998
- [14] Rational Software Corp. et al., *UML Semantics*, Ver. 1.1, Sept. 1997
- [15] Rational Software Corp. et al., *Object Constraint Language Specification*, Ver. 1.1, Sept. 1997
- [16] Sztipanovits, J. et al. "MULTIGRAPH: An Architecture for Model-Integrated Computing," *Proc. IEEE ICECCS'95*, Nov. 1995, pp. 361-368
- [17] Zhang, D.-Q., Zhang, K., "VisPro: A Visual Language Generation Toolset," *Proc. 14th IEEE Symp. Visual Languages*, Sept. 1998
- Customizable CASE and meta-CASE tools**
- [18] Advanced Software Technologies, Inc., *Graphical Designer*, <http://www.advancedsw.com>
- [19] Lincoln Software Ltd., *IPSYS ToolBuilder*, <http://www.ipsys.com>
- [20] MetaCase Consulting, *MetaEdit+ Method Workbench*, <http://www.metacase.com>
- [21] MicroGold Software Inc., *WithClass Scripting Tool*, <http://www.microgold.com>
- [22] mip GmbH, *Alfabet*, <http://www.alfabet.de>
- [23] Platinum Technology, *Paradigm Plus*, <http://www.platinum.com/clearlake>
- [24] Rational Software Corporation, *Rational Rose*, <http://www.rational.com>
- [25] Univ. of Alberta, *MetaView*, <http://www.cs.ualberta.ca/news/CS/1998/research/>
- [26] Vanderbilt University, *Multigraph Architecture*, <http://www.isis.vanderbilt>

