

Extended Object Diagrams for Transformational Specifications in Modeling Environments

Dragan Milicev

University of Belgrade

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

POB 35-54, 11120 Belgrade, Serbia, Yugoslavia

emiliced@etf.bg.ac.yu

Abstract

One of the most important features of software tools for domain-specific modeling is automatic output generation. Since the existing techniques for specifying output generation in customizable modeling and metamodeling environments suffer from some weaknesses analyzed in this paper, a new approach is proposed. The analysis is based on the observation that the output generation is a process of transformation of a model from the source domain into the model from the target domain. If the domains are at distant levels of abstraction, the mapping is difficult to specify, maintain, and reuse. Therefore, the proposed approach introduces one or more intermediate domains. Assuming that the source, target, and intermediate domains are conceptually modeled (metamodeled) using the object-oriented paradigm, the proposed approach uses extended UML object diagrams for specifying the mapping between them. The diagrams specify instances and links that should be created by the transformational process. The proposed extensions are the concepts of conditional, repetitive, and sequential creation. These concepts are implemented using the standard UML extensibility mechanisms. Several examples from different software engineering domains are presented in the paper. They prove some important benefits of the approach: the specifications are clear and concise, easy to maintain and modify. Besides, the approach leads to better reuse of domain models and to remarkably shorter production time.

Keywords: object-oriented modeling, Unified Modeling Language (UML), object diagram, metamodeling, model transformations

1 Introduction

Modeling is a central part of all the activities that lead up to the deployment of good software, as of any other engineering system [4]. Each modeling domain lies upon another model that defines (1) abstractions of the domain; (2) their properties and relationships; (3) their semantics and behavior in the model; (4) their visual appearance (notation) and behavior in the supporting tool. The latter, underlying model is called the *metamodel* of the considered modeling domain. Therefore, metamodeling is the process of defining the metamodel of the considered modeling domain. 'Meta' should be treated as a relative reference, not as an absolute qualification: each modeling domain has its underlying metamodel, which is specified by abstractions of another meta-metamodel, etc. [11]. This paper is focused to the modeling domains that can be metamodeled using the usual object-oriented paradigm [4], as opposed to some other paradigms, such as grammar-based specifications.

Apart from their important roles in specifying, documenting, and visualizing systems, the purpose of modeling tools is most often system construction [4], where '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 of output include, but are not limited to: documentation, source code in a certain programming language, database scheme, hardware

description, or any other formally defined structure. It may be observed that the output generation is actually a transformation of the user-specified model from the domain of his interest into the model from another target domain. (Precisely, this is actually *generation* of another model, but the term *transformation* is used in this context more often.) The problem of specifying output generation may exist in three different contexts. (1) In fixed, non-customizable domain-specific modeling tools, where the source and target domain metamodels, along with the mapping between them are fixed at the time of the tool development (as the problem of designing the output generation feature). (2) In customizable modeling tools, where the metamodels are fixed, but the mapping is customizable by the user. For example, a modeling tool such as a CASE tool may offer interfaces to the built-in metamodels (e.g., the UML metamodel, a metamodel of the target programming language, the relational metamodel, etc.), and the user may specify the mapping. (3) In fully featured metamodeling tools, where the user can specify both the metamodels and the mapping.

This paper discusses the problems of the techniques for specifying output generation implemented so far in the existing (meta-) modeling tools, and proposes a new approach that deals with the problems. The approach has two major contributions.

First, very often the source and the target domains are at distant levels of abstraction, and the mapping is difficult to specify, maintain, and reuse. Therefore, the

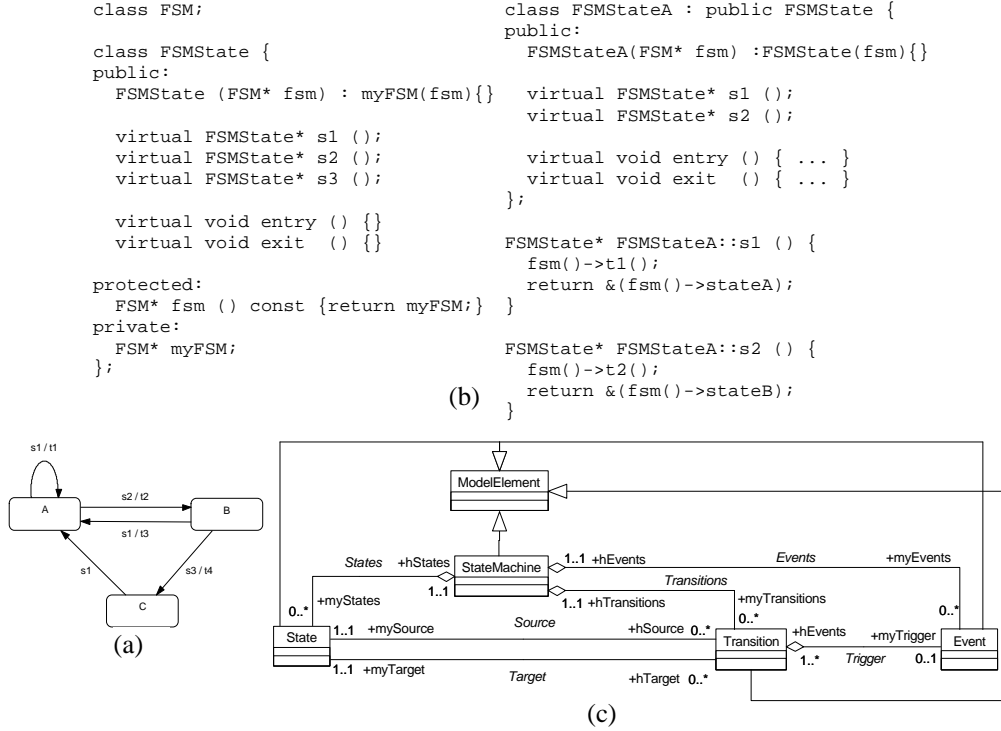


Figure 1: Demonstrational example: Code generation for state machines. (a) A sample state machine. (b) An excerpt from the generated code. (c) The metamodel.

proposed approach introduces one or more intermediate domains. 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.

Second, it uses extended UML object diagrams to specify visually the mapping between the domains. The diagrams specify instances and links that should be created by the transformational process. The proposed extensions are the concepts of conditional, repetitive, and sequential creation. These concepts are implemented using the standard UML extensibility mechanisms. Consequently, the specifications are clear and concise, easy to maintain and modify, and lead to shorter production time.

The paper continues as follows. Section 2 reveals the motivation for this work and defines the problem precisely using a simple demonstrative example. Section 3 briefly discusses the related work. The idea of our approach is presented in Section 4. Section 5 shows several examples that illustrate the applicability and efficiency of the approach. The paper ends with conclusions.

2 Motivation and problem statement

The problem and the proposed solution will be demonstrated using a simple example from the field of telecommunication software development. The goal is to develop a simple modeling tool that generates C++ code for state-machine models. The code generation for state

machines should be completely customizable: the user should be able to change the code generated from the same model if he needs another execution model due to performance, concurrency, distribution, or other requirements.

The example is shown in Figure 1. It is assumed that the desired code is obtained using the *State* design pattern [6]. It is also assumed that the user has specified a state machine named *FSM* as shown in Figure 1a. For this example, several classes are generated in the output C++ code. The first is named *FSM* and is the interface class whose behavior is specified in the model by the given state machine. 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. Other details may be found in [6]. The metamodel of the domain (state machines) is shown in Figure 1c. (This is a simplified version of the metamodel for state machines from [14].)

Now, the code generation strategy to be applied to each state machine should be specified. Let us consider two possible approaches. A straightforward one is to hard code the output generation scheme in an operation (e.g., a member function of the class *StateMachine* that implements the state machine abstraction in the modeling tool). The operation should read the data from the model

instances (i.e. to navigate through the model and read attribute values) and produce the textual output following the C++ syntax and semantics. An excerpt of such operation that generates the beginning of the declaration for the class FSMState may be:

```
// Generate base state class:
output<<"class "<<(this->name+"State")<<"{\n";
output<<"public:\n";
output<<"    "<<(this->name+"State")<<"(" ;
output<<(this->name)<<"* fsm):myFSM(fsm){}\n";
//...
```

The drawbacks of this approach are obvious:

- (1) The process of specifying is extremely tedious, time-consuming, and error-prone.
- (2) The user must deal with the complexity of the target domain (C++ syntax and semantics).
- (3) The built-in general-purpose and reusable C++ code generator is not used at all.
- (4) Any modification is very difficult to apply because the code is not clear and comprehensible.
- (5) The code is not reusable.
- (6) The user must deal with the technical details such as the correctness of the output stream, opening files (.h and .cpp files must be created in C++), etc.

The core reasons for the listed drawbacks may be revealed by the following observation. The output generation process may be viewed as a creation of a target model from the source model. The source model is the model explicitly specified by the user in the modeling tool and consists of instances of state machines, states, events, and other abstractions from the source domain. The target model is the textual output, i.e. the generated C++ source code whose metamodel is implicitly assumed by the user (C++ syntax and semantics). The code of the given operation is actually a specification of the mapping between the two domains. Since the two domains are at distant levels of abstraction, their direct mapping by the hard-coded special-purpose generator has all these drawbacks.

This mapping between two distant domains has the same disadvantages as the process of object-oriented programming in the target programming language (e.g. C++) without previous modeling at a higher level of abstraction (e.g. with UML). This is because the programming language level of abstraction is too far from the level of abstraction that is suitable for the developer's way of thinking. For our example, instead of directly generating the textual output, it may be reasonable to create an intermediate model based on a metamodel of a higher level of abstraction, such as a subset of UML, which includes abstractions supported directly by a common object-oriented programming language (class, operation, attribute, etc.). Because the general-purpose C++ code generator from UML models may be built in the tool, it may be reused for the generated intermediate model. Hence, the idea is to create needed instances from the intermediate domain using the built-in UML metamodel, and then to invoke the built-in code generator to produce the output:

```
void StateMachine::generateCode () {
// Temporary package for the intermediate model:
Package& pck = Package::create();

// Intermediate model:
// Base state class:
Class& baseState = Class::create(pck);
baseState.name = this->name+"State";

// Base state class constructor:
Method& baseStateConstr =
    Method::create(pck);
baseStateConstr.name = this->name+"State";
Link::create(Members::Instance(),
             baseState,baseStateConstr);

//...
// Code generation:
pck.generateCode();
}
```

This code excerpt shows the creation of instances for the class FSMState and its constructor. It creates instances of UML abstractions Class and Method, using the built-in UML metamodel interface. Then, it sets the values of their attributes. Finally, it creates links between these instances. All these instances are packed into a temporary package for which the output is generated in the end.

This approach remedies most of the drawbacks of the first approach. In the first place, it eliminates the impedance-matching problem between the source and target domains by introducing an intermediate level. By doing this, the process of output generation is split into two steps, where each step is much easier to specify than before. Besides, the second step is supported by the built-in and reusable code generator. Thus, the first-step mapping specification is completely reusable for other target languages, provided that general-purpose code generators from UML are available. However, the specification of the operation body is still tedious and error-prone. Besides, the code may be very complex and difficult to manage. Since it is actually a specification of the process of creating instances from the intermediate domain, where both source and intermediate domains may be formally defined by their metamodels, this specification may be provided in another formal way. The idea is to use a visual specification, preferably one that is compatible with the UML standard. This is the subject of this paper.

3 Overview of the Related Work

Due to the fact that the process of domain-specific metamodeling can be formalized, the need for tool support of this process has been recognized for long [2, 11, 13]. This need was first met in the domain of automatic programming environment generation [10]. By the maturation of numerous software-engineering methodologies and notations, especially of object-oriented ones, which all have been developed with the perspective of CASE tools support, the field of meta-CASE research has evolved [2, 11]. However, we do not constrain our discussion here on the field of software modeling, CASE, and meta-CASE tools, although it is

our major field of interest with a strong research background. The results of our work may be applied to metamodeling domains other than software systems. That is why we use the term "metamodeling environment" rather than the term "meta-CASE tool."

There are a number of approaches addressing a similar problem using structural transformations of grammar-based models and various rule-based techniques [7, 8, 9]. Their goal is to transform a user-defined structural model written in a domain-specific language into another structural model in another target language. Although the goal is similar to the one presented here (transformation of models), there are a number of differences. First, although their principles may be generalized to more abstract terms, they primarily deal with textual models (or, more generally, with strings of entities). Second, their 'metamodels' are expressed with grammars, where the entities are 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, 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.

The rule-based approaches allow the user to specify the differences between the source and the target grammars ('metamodels') and a supporting tool may help in generating the model transformer but with some intervention of the user [7]. The approach presented here allows the user to specify the mapping, and the transformer is generated without any intervention of the user. Furthermore, 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 (in meta-environments) or just understanding (in customizable modeling environments) the metamodels specified in object-oriented terms. It is evident that some domains may be metamodeled with much less effort using the object-oriented paradigm instead of grammars. This includes most modeling methods with visual notations. For such cases, the proposed approach is definitely superior. Consequently, the proposed approach may be treated as a complement to the grammar-based structural transformations, more suitable for object-oriented metamodels.

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. In an automatically generated VPL environment, the user chooses a graphical element and puts it onto a diagram rather arbitrarily. The task of the tool is to check the correctness of the diagram when the translation operation is explicitly invoked, considering the underlying grammar. Then, it should parse the grammar elements and develop an internal representation analogous to the derivation tree in classical compilers. On the other side, in object-oriented modeling environments, the user is usually explicitly constrained in designing diagrams, and the contents of the diagram is determined at the time of its construction. The user creates and manages explicitly *model* (semantic) elements, while visual elements are only views to them. 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 CASE tools has been an attractive discipline for years, and a lot of extensible CASE and meta-CASE tools, both commercial and academic ones, are available at the moment [18, 19, 20, 21, 22, 23, 24, 25]. A major commonality (and a weakness also) of all existing meta-CASE tools that is of greatest interest to our work is the output generation facility. 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 a scripting language that is proprietary and vendor-specific. Hence, the first hard-coded output generator strategy described in the previous section is available to the user. As they often offer a flexible interface to their metamodels, the user may create an intermediate model as described in the second approach in the previous section. Nevertheless, this intermediate model may be created only using the same scripting language, and there is no other opportunity for doing this at a higher level of abstraction (e.g., visually). None of these tools promotes domain mapping as an explicitly supported strategy available to the user. 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 in this paper.

4 Domain mapping specification

The idea of the domain mapping (Figure 3) is to create an intermediate metamodel and a specification of the mapping from the source to the intermediate domain. A model transformer is automatically generated from the mapping specification. It is used to create the intermediate model from the user-defined source model. Finally, the built-in code generator produces the ultimate output. The benefit is because each of the transformations

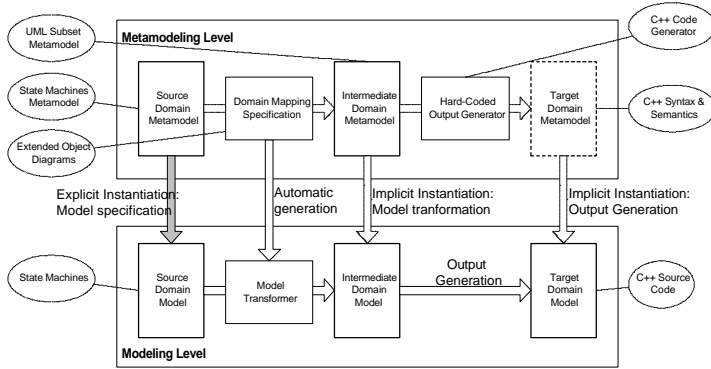


Figure 3: The idea of the domain-mapping strategy in the context of the demonstrational example. The transformation from the source into the target domain is split into two (or generally more) steps in order to cope with the complexity of the mapping specification.

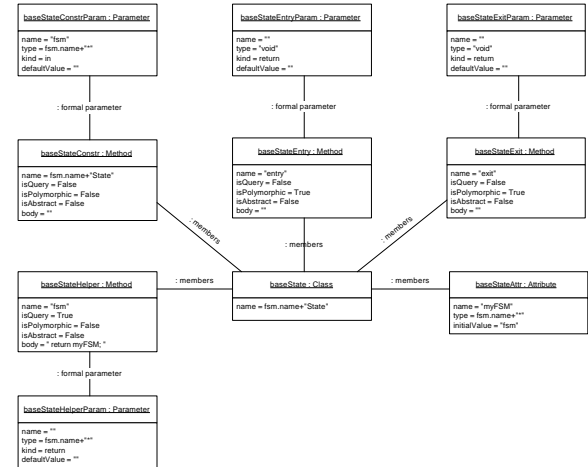


Figure 4: A simple part of the object diagram for the domain mapping specification of the demonstrational example. The diagram shows only the specifications for the base class FSMState and its members that are generated by default. The diagram belongs to the context of the state machine accessible through the fsm identifier.

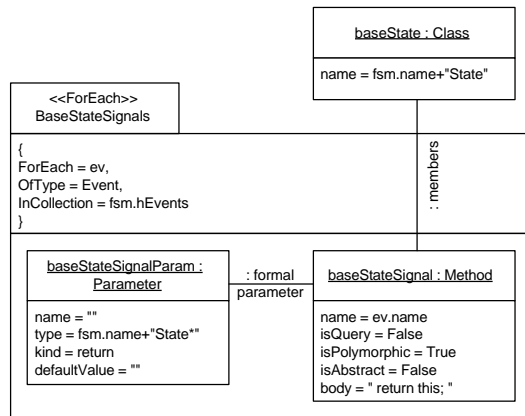


Figure 5: "ForEach" concept for repetitive object specification. The diagram shows only the specification for the base class FSMState and its member functions generated for the state machine's events. It belongs to the context of the state machine accessible through the fsm identifier.

is much less complex than the direct transformation, and is thus easier to specify, maintain, and reuse.

The specification of the domain mapping should be formal and preferably graphical. Since it is actually a specification of a set of instances of the abstractions (classes) from the intermediate domain that should be created, UML object diagrams may be used. An excerpt for our example is shown in Figure 4. It is assumed that the diagram is defined for one instance from the source model, which is referred to by a certain identifier in the diagram. For this example, it is an instance of the type StateMachine, referred to by the identifier fsm. The diagram specifies the set of instances of classes from the intermediate metamodel that should be created for each StateMachine instance fsm from the source model. The diagram specifies also the values of their attributes, along with the links between them. The attribute values are defined as expressions that refer to the instances from the

source model and their attribute values, using the navigation through the source model. The links are instances of associations from the intermediate metamodel.

A standard object diagram is not sufficient for the mapping purposes. There is also a need for repetitive object creation. For our example, one method in the base state class should be created for each event that the machine reacts upon (see Figure 1). For this purpose, we use a stereotyped package with the stereotype ForEach. The example is shown in Figure 5. ForEach package represents iteration through a collection of instances from the source model and creation of a set of intermediate domain instances for each of them. It contains three tagged values:

- ForEach: An identifier that is introduced into the scope of this package. It may be used inside the scope of the package to refer to the current element of the iteration.
- OfType: The type of the current element. The iteration is type-sensitive, in the sense that only the elements of the specified type from the collection are processed, and the others are ignored (in the case that the elements are polymorphic). The type is from the source metamodel.
- InCollection: An expression that evaluates to a collection of the instances from the source model to iterate.

When a link connects an instance inside a package and another outside that package, then each repetitive instance created by the iteration will be linked to the outer instance. For the expressions that are used to define attribute values or collection in a ForEach package, any formal language for navigation through the source model

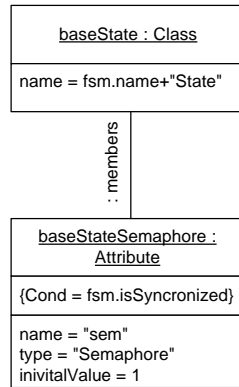


Figure 6: Conditional object creation. The diagram shows only the specification for the base class `FSMState` and its data member (a semaphore) generated for synchronization, only if the state machine is "synchronized."

may be used. For example, Object Constraint Language (OCL) may be used [15] if the tool is capable of parsing these expressions or the programming interface of the model is OCL-compliant. The other option is the scripting language used in the tool.

Another needed concept is conditional creation. An instance, a link, or a `ForEach` package may be tagged with a condition that is a Boolean expression again in the scope of the source model. If the expression evaluates to `False` when the intermediate model is being created, the conditional instance or link is not created, or the package is ignored. A simple example is shown in Figure 6. The example assumes that the `StateMachine` type in the source metamodel has a Boolean attribute named "isSynchronized." If the value of this attribute is `True`, the generated state machine code should be mutually exclusive in a concurrent environment. This is achieved by an attribute of type `Semaphore` that is generated in the base state class and the corresponding wait/signal operations in all publicly accessible operations (not shown in the picture).

Since `ForEach` packages actually represent loops in the process of intermediate model generation, they may be nested. An example is shown in Figure 7. For our example, a derived class should be created for each state. This is specified with the outer `ForEach` package. For each of the events this state reacts upon, an operation should be generated in this class (specified with the nested package).

A `ForEach` package introduces a scope of the expressions. The rules for the scope nesting are identical as in the traditional procedural programming languages. An expression may use identifiers from the scope in which it is defined, as well as from its enclosing scopes. A `ForEach` identifier is local for its package, and hides the same identifiers from the enclosing scopes.

It has been mentioned that the presented specifications belong to the context of one instance from the source model. A certain identifier (`fsm` in our example) refers to this instance. However, we generalize

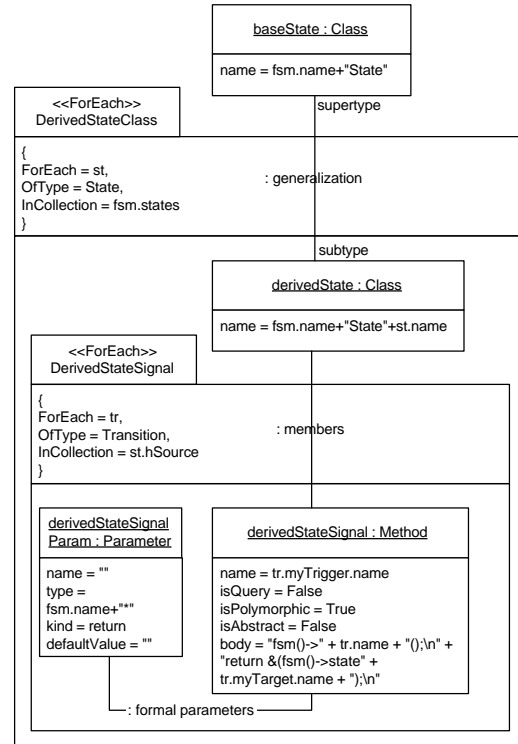


Figure 7: Nesting of "ForEach" packages. The diagram shows a part of the specification for the derived state classes and their member functions for the events.

this context in the following way. The whole mapping is specified following the UML style of hierarchically organizing models in packages. Thus, the mapping specification is actually another model, represented with a package hierarchy, where each package may, but need not be a `ForEach` one, and may own instances, links, and other packages. (Ordinary packages serve as grouping elements only and map into the same grouping of the elements of the generated model.) Besides, following the UML diagrammatic style, it is allowed that the contents of one package are defined by several diagrams to enhance readability and clearance. Therefore, all the diagrams shown in figures 4 to 7 belong to a `ForEach` package with the `InCollection` value referring to a tool-manipulated collection of all instances of the given type in the source model (for our example, something like: `StateMachine::getAllInstances()`).

The generated model is organized as a hierarchy of packages, where each package is an unordered collection of the elements it owns by default. More precisely, the ordering of the elements in a package is implicitly determined by the order of their creation; by default, the ordering of creation is not defined. Sometimes, however, an explicit ordering of the elements is needed. This ordering may ensure a proper sequential traversal through the model elements; for example, if a sequential structure (e.g., text) is to be further generated from that model. If an element x is to be created after an element y , it may be considered dependent on y . This relationship is specified

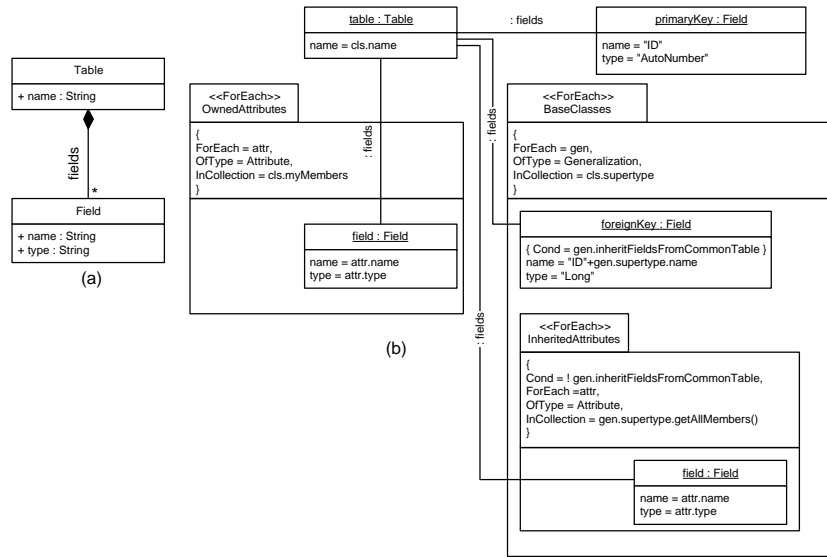


Figure 8: Example: Generation of the relational database scheme from a UML class model. This example focuses on inheritance. The source domain metamodel is UML (not shown here). (a) The target domain metamodel (relational). (b) The domain mapping specification. Operation `getAllMembers()` returns the collection of all owned *and* inherited members of a GeneralizableElement (Type in this case).

in the mapping diagram with a dependency from x to y , stereotyped as `<<sequence>>` [4]. Consequently, y will precede x in a traversal of the elements of their enclosing generated package.

From the diagrams formally specified as shown above, the source or the scripting code for the model transformer used at the modeling level may be generated automatically. For our example, the code is shown in the appendix, and the details are reported elsewhere [12]. The algorithm for generation of such code is as follows. For a package, the algorithm is: first introduce implicit sequence dependencies from links to the instances they connect, then sort topologically the owned elements according to the sequence dependencies, and then generate code for each of the elements (recursively for its nested packages). If the package is a `ForEach` one, the specified iteration will be performed, and one package in the generated model will be created for each iterated element. Each instance generates statements that will first create an object of the specified type and then set its attributes to the specified values, using the programming interface of the modeling tool.

The approach of the domain mapping may be generalized to arbitrarily many intermediate domains. The idea is that a tool may generate several intermediate models as different levels of modeling abstraction, using the domain mapping specifications. The process of creation of intermediate models may be viewed as a descent down the abstraction levels. The tool may allow the user to make changes in each intermediate model, prior to generating the next one, if the user is not satisfied with the automatically generated model. By using different domains for intermediate models, it may be expected that a better understanding of the problem and more complete modeling may be achieved. On the other

side, other more abstract domains may be built on top of already designed domains, and the transformation may be easily specified using the mapping from the new domain into the already implemented lower-level one. This is one of the directions for the future work.

5 Case study and evaluation

The example of the modeling tool for state machines has been implemented as a final project for the B.Sc. degree at the University of Belgrade. The specification had about 30 instances and seven `ForEach` packages. The implementation of the code generation part, using domain mapping, and a built-in C++ code generator, took about ten hours, including testing.

Apart from this example, two more are presented here (these are just small excerpts of much more complex examples from practice). The second example is the problem of transforming object-oriented class model into the relational database model. This is a common task in object-oriented programming when persistence of objects is accomplished by a relational database. Here, the source domain is UML. The target domain is the code that may be used to define database tables and fields, e.g., SQL declarations. However, the direct mapping from the class model into the textual SQL declarations is difficult to specify. Therefore, an intermediate domain is introduced, with the metamodel shown in Figure 8a. It is a simplified version that encompasses tables and fields only. It is now easy to specify generation of SQL declarations from this intermediate domain, because it is almost (if not completely) one-to-one mapping. In this example, the accent is on inheritance, as the most difficult task in this process. It is assumed that the user is offered two strategies of implementing inheritance in relational tables. The first one assumes that a derived class has its own

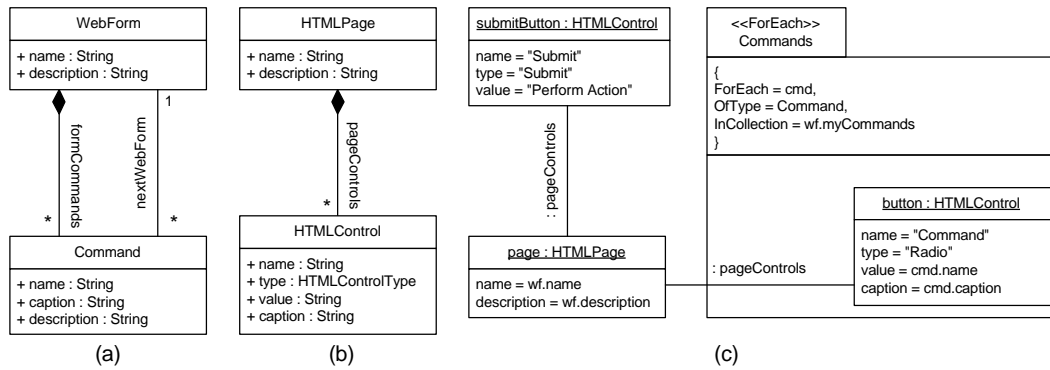


Figure 9: Example: Web design tool. (a) An excerpt from the source domain metamodel. (b) An excerpt from the target domain metamodel. (c) The domain mapping specification.

independent table, with all inherited attributes copied into its own table. In this approach, an object is represented with a single record in the table that represents its class. In the second approach, a derived class has a table without inherited attributes, but its records are dependent on the records from the table that represents its base class. In this second approach, an object is represented by a set of records in the tables that represent its own class and its base class. We assume that the user may chose one of the approaches for each generalization in the class model, by setting the Boolean attribute of the generalization named "inheritFieldsFromCommonTable." This attribute should be added to the UML metamodel as a tagged value of generalization. If this field is set to True, the second approach is chosen. In both approaches, the table should have a primary key (of type "AutoNumber" and named "ID"), and the set of the fields for the attributes of the class. In the first approach, the table should have the fields for all attributes from the base class, for each inheritance relationship tagged with `inheritFieldsFromCommonTable = False`. In the second approach, the table should have only a foreign key (of type "Long" and named "ID"+<baseClassName>) to link it to the base class table. The corresponding mapping scheme is shown in Figure 8b.

The third example shows a case when UML is not used as any of the domains. It is taken from one of our projects with database-centric web application development. A method and infrastructure for rapid application development have been developed. A very small part of the idea is presented here, just to illustrate the usage of metamodeling and domain mapping. In this approach, application is modeled by the navigation through *web forms*. From one web form, the user can choose a *command*, which performs some actions in the database on the server and displays another web form. The commands are implemented as radio button options in the web form, and a "Submit" button that posts the data from the form to the server. A very small part of the source domain metamodel is shown in Figure 9a. This domain should be mapped into the standard HTML textual output. However, this mapping is complex because the source domain has other concepts not shown

here. Therefore, an intermediate model is introduced that may be mapped one-to-one to the target domain. It contains abstractions such as an **HTMLPage** or an **HTMLControl** (text box, list box, radio button, etc.). This metamodel is shown in Figure 9b. As in the previous example, generation of HTML from the intermediate domain is straightforward. The mapping scheme for this example is shown in Figure 9c. The author implemented the complete prototype tool in only three days, including metamodeling, code generation, and testing.

In practice, the following method for defining intermediate domains and mapping specifications is proposed. After the source domain is defined and well understood, the most important task is the design of its metamodel. All common principles of object-oriented analysis and design may be applied to this process [3]. Then, the desired target output is informally specified and supported by an example. For this purpose, a simple yet descriptive example from the source domain is developed. Then, the desired code for this example is generated manually. The result of this process corresponds to the example shown in Figure 1. Afterwards, an intermediate domain that will make the output generation less complex is found. It should be very close to the target domain, so that the desired output may be easily generated from it. If it is still conceptually far from the source domain, other intermediate domains should be built upon it, etc. We have successfully found such a domain in all the cases. Reuse of already developed domain models is of much help. For example, if the target output is C++ or any other object-oriented programming language code, we use a UML subset as the intermediate domain. Another useful and reusable example is the relational domain. The metamodel of the intermediate domain should be built, too, if it is not already available. Finally, the domain mapping is specified using the following procedure. The developer goes through the sample output, and tries to find out of which element in the intermediate model that part of the code is an outcome. It is then specified in the mapping object diagram. The procedure is applied iteratively and incrementally. This procedure is much easier than the hard-code approaches, because the elements of the target

output that originate from the same source model element may be spread all over the target model. For instance, in our first example, the events of a state machine produce operation declarations in many separate classes. Therefore, it is easier to go sequentially through the generated output and build incrementally the domain mapping object diagram as the need for each of its elements arises. Other possible heuristics and a more formal approach to this process will be investigated in the future work.

The research team from the University of Belgrade has successfully used the described approach in several other large projects. All the examples confirmed the expectations on possible benefits of the strategy. The specifications of output generation are clear and concise, easy to maintain, modify, and reuse. They are hierarchically organized, visually presented (using multiple consistent diagrams), and thus cope well with a potential complexity of the mapping. It is possible to build the mapping specifications incrementally and iteratively, and to test them using only partially developed object diagrams. (Such incremental testing of partially defined mappings is not available in other techniques.) The process of specification is less tedious and error-prone. As the most important benefit, the development of output generator is shortened a lot. For instance, the first example (state machines) was started by using the conventional hard-coded approaches. It took us several weeks only to specify, without testing and debugging that were extremely difficult. By using the domain mapping strategy, we have reduced the working time to the order of hours. Production time will be shortened even more when a considerable repository of domain models and their transformers to various versions of the target implementation is created. In that case, user-defined models and transformers may be reused for different versions of the target implementation by using different transformers of the intermediate domains from the repository. Besides, as already stated, the mapping from the higher-level domains into the reusable intermediate domains may be defined with less effort than before.

Nevertheless, there are some weaknesses of our approach recognized so far. Although the specification supports conditional, sequential, and repetitive instance creation, it does not support recursion. Namely, one of the most important features of the traditional approaches that traverse the model structure and invoke operations for the model elements is that these operations may be recursive. This issue is particularly important when generating recursive structures, what is sometimes needed in textual output. In the examples we have studied so far, we have not encountered the need for recursion. However, the solution exists, but the future work will investigate this issue more deeply and will try to find a way for specifying recursion that best fits the definition of the existing concepts.

Another issue that may be improved is the visual specification. It is very often the case that a lot of instances and links must be specified in the domain mapping model, in order to describe formally the creation of an instance of a composite abstraction (e.g., a class and a set of its members in Figure 4). If that abstraction has a compound symbol defined in the accompanying notation, it may be much easier to use that symbol instead of the set of instances and links. It is possible to incorporate this feature in our approach, while completely preserving the described semantics.

6 Conclusions

The problem of specifying output generation in the context of modeling environments has been studied in this paper, and a new approach, called domain mapping, has been proposed. The approach is based on the observation that the automatic output generation is a process of creating a model in the target domain from the model in the source domain. If the domains are at distant levels of abstraction, the mapping is difficult to specify, maintain, and reuse. This is why one or more intermediate domains are introduced. The mapping is specified using UML object diagrams that show the instances from the intermediate domain that should be created by mapping. The diagrams are extended with the concepts of conditional, repetitive, and sequential creation. These concepts are implemented using the standard UML extensibility mechanisms.

Several case studies from different software engineering domains have been presented. All the examples have proved the major benefits of the approach. The specifications are clear and concise, thus easy to maintain and modify. The domain mapping strategy leads to a better reuse of domain models and to a remarkably shorter production time.

Acknowledgements

The author is grateful to D. Marjanovic, P. Nikolic, M. Ljeskovac, M. Zaric, and Lj. Lazarevic who contributed to the implementation of a supporting tool and the case study.

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 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] mip GmbH, *Alfabet*, <http://www.alfabet.de>
- [22] Platinum Technology, *Paradigm Plus*,
<http://www.platinum.com/clearlake>
- [23] Rational Software Corporation, *Rational Rose*,
<http://www.rational.com>
- [24] Univ. of Alberta, *MetaView*, <http://www.cs.ualberta.ca/news/CS/1998/research/>
- [25] Vanderbilt University, *Multigraph Architecture*,
<http://www.isis.vanderbilt.edu>

Appendix

The generated C++ code for the model transformer (an excerpt for the diagram in Figure 7). ForEach/EndForEach are C++ macros that implement type-sensitive iteration.

```
// Temporary package for the intermediate model:
Package& pck = Package::create();
// Intermediate model:
ForEach(fsm,StateMachine,StateMachine::getAllInstances())
    // Generated for objects:
    // Object: baseState
    Class& baseState = Class::create(pck);
    baseState.name = fsm.name+"State";

    // Generated for ForEach packages:
    // Package: DerivedStateClass
    ForEach(st,State,fsm.states)
        // Generated for objects:
        // Object: derivedState
        Class& derivedState = Class::create(pck);
        derivedState.name = fsm.name+"State"+st.name;

        // Generated for ForEach packages:
        // Package: DerivedStateSignal
        ForEach(tr,Transition,st.hSource)
            // Generated for objects:
            // Object: derivedStateSignal
            Method& derivedStateSignal = Method::create(pck);
            derivedStateSignal.name = tr.myTrigger.name;
            derivedStateSignal.isQuery = False;
            derivedStateSignal.isPolymorphic = True;
            derivedStateSignal.isAbstract = False;
            derivedStateSignal.body = "fsm()->" + tr.name + "();\n" +
                "return &(fsm()->state" + tr.myTarget.name + ");\n";

            // Object: derivedStateSignalParam
            Parameter& derivedStateSignalParam = Parameter::create(pck);
            derivedStateSignalParam.name = "";
            derivedStateSignalParam.type = fsm.name+"*";
            derivedStateSignalParam.kind = Return;
            derivedStateSignalParam.defaultValue = "";

            // Generated for ForEach packages:

            // Generated for links:
            // Link: <unnamed> of Association: members
            Link& link02 = Link::create(Members::Instance(),derivedState,derivedStateSignal);
            // Link: <unnamed> of Association: formal parameters
            Link& link03 = Link::create(FormalParameters::Instance(),
                derivedStateSignal,derivedStateSignalParam);
        EndForEach(tr)

    // Generated for links:
    // Link: <unnamed> of Association: generalization
    Link& link01 = Link::create(Generalization::Instance(),derivedState,baseState);
EndForEach(st)

// Generated for links:
EndForEach(fsm)
```