

Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments

Dragan Milicev

University of Belgrade

School of Electrical Engineering, Department of Computer Science

P.O. Box 35-54, 11120 Belgrade, Serbia, Yugoslavia

dmilicev@rcub.bg.ac.yu

Abstract

One of the most important features of modeling tools is generation of output. The output may be documentation, source code, net list, or any other presentation of the system being constructed. The process of output generation may be considered as automatic creation of a target model from a model in the source modeling domain. This translation does not need to be accomplished in a single step. Instead, a tool may generate multiple intermediate models as other views to the system. These models may be used either as better descriptions of the system, or as a descent down the abstraction levels of the user-defined model, gradually leading to the desired implementation. If the modeling domains have their metamodels defined in terms of object-oriented concepts, the models consist of instances of the abstractions from the metamodels and links between them. A new technique for specifying the mapping between different modeling domains is proposed in the paper. It uses UML object diagrams that show the instances and links of the target model that should be created during automatic translations. The diagrams are extended with the proposed concepts of conditional, repetitive, parameterized, and polymorphic model creation, implemented by the standard UML extensibility mechanisms. Several examples from different engineering domains are provided, illustrating the applicability and benefits of the approach. The first experimental results show that the specifications may lead to better reuse and shorter production time when developing customized output generators.

Keywords

Object-oriented modeling, domain-specific modeling, Unified Modeling Language (UML), model-based output generation, automatic model transformation, model-based translation, object diagram, metamodeling, modeling tool

1 INTRODUCTION

Modeling is a central part of all the activities that lead up to the deployment of a good engineering system [4]. There are software tools supporting the process of modeling in many engineering domains. A modeling tool provides an environment for applying the underlying method and notation, along with model consistency checking, navigability, and visualization, thus making the modeling process less time-consuming and error-prone. One of the most important features of modeling tools is automatic *output generation*. The output may be documentation, source code (for software systems), net list (for on-chip hardware), or any model other than that created by the user. The process of output generation may be viewed as an automatic translation of the source model into the target model. This mapping does not need to be a single-step generation of an ultimate output from the tool. The tool may generate multiple intermediate models as other views that describe the system better, or that decrease the abstraction level of the user-defined model, leading to the implementation gradually. Moreover, for

more complex applications, it may be necessary to customize the output generator or build a new one. The problem is how to specify this automatic model translation in a simple, yet formal way.

This paper discusses the drawbacks of some approaches in existing environments and proposes a solution. The solution is based on the idea that the process of creation may be specified using the Unified Modeling Language (UML) object diagrams, extended with the proposed concepts of conditional, repetitive, parameterized, and polymorphic creation. This is called the *domain mapping* specification. Some examples that demonstrate the potential of this approach are presented, too.

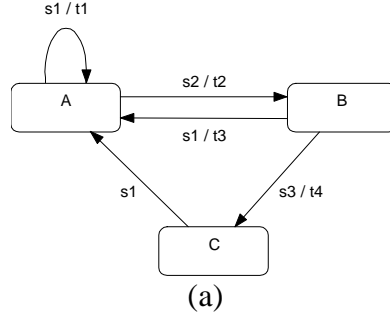
The paper is organized as follows. In the next section, the motivation for this work and the specific context in which the proposed approach may be used are discussed. An overview of the related work is given in Section 3. The idea and the purpose of the approach are described in Section 4. A more detailed presentation of the proposed concepts for the domain mapping specification is given in Section 5. Several examples and the discussion of a preliminary case study are presented in Section 6. The paper ends with conclusions.

2 MOTIVATION AND PROBLEM STATEMENT

Although the term "domain" is already used in different contexts and with different meanings in various software-engineering disciplines, it will be used here with a specific meaning. First, we will distinguish between the terms "problem domain," which refers to the conceptual space of the particular problem being solved by a concrete application or system, and the term "modeling domain." A modeling domain, in this paper, is a conceptual space of modeling various systems from similar problem domains using a certain modeling language, with the defined syntax (notation) and semantics for that language. If otherwise stated, the unqualified term "domain" refers to a modeling domain.

A modeling domain may be defined in terms of its abstractions, their semantics, properties, relationships, and behavior. The conceptual model of a modeling domain is called the *metamodel* of the considered domain. Therefore, metamodeling is the process of defining a metamodel for a domain. "Meta" should be treated as a relative reference, not as an absolute qualification: each modeling domain has its underlying metamodel, which may be specified in terms of abstractions of another meta-metamodel, etc. [10]. This paper is focused on the modeling domains the metamodels of which may be defined using basic object-oriented structural concepts (classes and attributes, associations, and generalization) [4, 6], as opposed to some other paradigms, such as grammar-based specifications.

The problem and the proposed solution will be demonstrated using a simple example from the field of telecommunication software development. In this example, the task is to develop a simple modeling environment that may generate C++ code for state-machine models [1, 2, 3, 6]. The code generation for state machines should be customizable. In case the user needs different execution models due to performance, concurrency, distribution, or other issues, the user should be able to change the manner in which the code is generated from the initial state-machine model.



```

class FSM {
public:
    FSM ();

    void s1 ();
    void s2 ();
    void s3 ();

protected:
    friend class FSMStateA;
    friend class FSMStateB;
    friend class FSMStateC;

    void t1() {...}
    void t2() {...}
    void t3() {...}

    FSMStateA stateA;
    FSMStateB stateB;
    FSMStateC stateC;

    FSMState* curSt;
};

FSM::FSM () :
    stateA(this), stateB(this), stateC(this),
    curSt(&stateA) { curSt->entry(); }

void FSM::s1 () {
    curSt->exit();
    curSt = curSt->s1();
    curSt->entry();
}
...

class FSMState {
public:
    FSMState (FSM* fsm) : myFSM(fsm) {}

    virtual FSMState* s1 () {return this;}
    virtual FSMState* s2 () {return this;}
    virtual FSMState* s3 () {return this;}

    virtual void entry () {}
    virtual void exit () {}

protected:
    FSM* fsm () const { return myFSM; }
private:
    FSM* myFSM;
};

class FSMStateA : public FSMState {
public:
    FSMStateA (FSM* fsm) : FSMState(fsm) {}

    virtual FSMState* s1 ();
    virtual FSMState* s2 ();

    virtual void entry () { ... }
    virtual void exit () { ... }
};

FSMState* FSMStateA::s1 () {
    fsm()->t1();
    return &(fsm()->stateA);
}

FSMState* FSMStateA::s2 () {
    fsm()->t2();
    return &(fsm()->stateB);
}
  
```

(b)

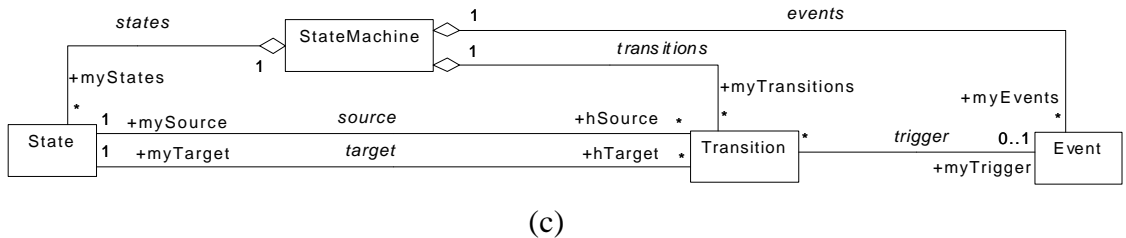


Figure 1: Demonstrational example: code generation for state machines. (a) A sample state machine. (b) A fragment of the generated code. (c) The metamodel.

The example is shown in Figure 1. It is assumed that the user wants to obtain the code shown in Figure 1b, which is an implementation of the *State* design pattern [5]. For this example and the given

assumption, several classes should be generated in the output code. The first is named `FSM`. It contains the methods that correspond to the events on which the state machine reacts. The second class is abstract and is named `FSMState`. It has 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 reference to the target state. The class `FSM` contains one member for each state, and a reference to the abstract class `FSMState` that refers to the current state. All the methods of this class are implemented in the same way: they invoke the corresponding operation of the current state which, by means of virtual mechanism, performs the transitional action and returns the new current state. Therefore, an `FSM` object reacts to the events depending on its current state [5].

A simple metamodel of the considered domain is shown in Figure 1c. It defines the abstractions of the domain as the classes `State`, `Transition`, etc., as well as their relationships. For example, a transition is connected with its source and destination states. This is represented by the associations `source` and `target` in the metamodel. A concrete model defined by the user in this domain, such as the one in Figure 1a, consists of object instances of the classes from the metamodel, connected by the links as instances of the corresponding associations [6]. For example, the state A in Figure 1a is a notational symbol for an instance of `State`. Similarly, the transition edge from A to B is a notational symbol for an instance of `Transition`, linked by an instance of association `source` with the instance A, etc.

The user may want to specify the code generation strategy that should be applied to each instance of `StateMachine` created in the model. A straightforward approach, by far mostly used in existing modeling tools, is to hard-code the output generation in a procedure. The procedure should navigate through the instances in the model, read their attribute values, and produce the textual output following the C++ syntax and semantics. For this specification, C++ will be used here for generality, instead of any vendor-specific scripting language that is usually available for these purposes in the existing environments. A small fragment that generates the very beginning of the declaration for the class `FSMState` may be:

```
void generateStateMachineCode (StateMachine* this) {
    ... // File opening and other preparation actions
    // 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 specification 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 user must deal with all technical details, such as correctness of the output stream, opening files (.h and .cpp files must be created), etc.
- (4) Modifications are difficult to apply because the code is not clear and comprehensible.
- (5) The code generator is not easily reusable for other target languages such as Java.
- (6) It may be likely the case that this problem exists in a broader context of a more general modeling tool, such as a tool for modeling in UML, which already has a C++ code generator. This built-in general-purpose and reusable code generator (e.g., from UML models to C++) is not used at all.

The output generation may be viewed as a creation of a new target model from a source model. The source model is the model explicitly created by the user in the modeling tool. The target model is the generated code whose metamodel is implicit—this is the C++ syntax and semantics. The presented code of the code generator is actually a specification of the mapping between these domains. The direct

mapping between these two domains, using the hard-coded special-purpose generator, has all the listed drawbacks because the domains are conceptually very distant.

The direct mapping between two distant domains has the same disadvantages as the process of object-oriented programming in a target programming language (e.g., C++), without previous modeling using a language of a higher level of abstraction (e.g., UML). For the demonstrational example, instead of directly generating the textual output, it may be reasonable to create an intermediate model from a domain of a higher level of abstraction. (This does not mean that the intermediate domain is at a different level in the metamodeling hierarchy, but it is conceptually more abstract and closer to the source domain.) This domain may consist of the basic object-oriented concepts from UML, such as class, method, attribute, etc., so it can be easily mapped into the target C++ domain. Since the general-purpose C++ code generator from the intermediate UML models may exist in the tool, it may be reused for the generated intermediate model. Hence, the idea of the generation process is to create the needed instances of the intermediate model using the built-in UML metamodel first, and then to invoke the built-in code generator to produce the output:

```
// 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";
createLink("members",baseState,baseStateConstr);

//... and much, much more ...

// Code generation using the built-in code generator, and destruction:
pck->generateCode();
deleteFromModel(pck);
```

This code fragment creates instances of the abstractions `Class` and `Method`, using the programming interface of the tool's implementation of the UML metamodel. These instances represent the class `FSMState` and its constructor in the resulting code. Then it sets the values of their attributes. After that, it creates links between these instances. All these instances are packed into a temporary package for which the code is finally generated.

This approach eliminates some of the drawbacks of the first approach. First, it eliminates the "impedance-mismatching" problem between the source and the target domains by introducing an intermediate domain. By doing this, the process of output generation is split into two steps, where the second one is supported by the built-in code generator. The first step does not deal with the specialties of the C++ syntax and semantics, nor with the technical details (output files). Moreover, the mapping from the intermediate domain into the C++ source code is more-or-less standardized, and many tools provide code generators that may be reused in this approach. On the other hand, the mappings between arbitrary user-defined higher level domains should be customizable. The state-machine modeling is only one of such examples. The Case Study section will provide some more.

These are reasons why many commercial modeling tools use intermediate domains and reusable code generators, although they are often not directly visible to the user. The advantages of introducing an intermediate model and a mapping from the source domain, which is independent of the target domain, are explained well in [31].

However, for more complex applications, when the mapping from the source domain into the target output is more sophisticated, the specification may still be very complex and difficult to maintain. Besides, the described opportunities are in most commercial tools solely available. Moreover, the metamodels are often not object-oriented (i.e., do not incorporate inheritance of abstractions). This makes the programming of customized code generators more difficult than necessary. For example, one of the most popular UML modeling tools, Rational Rose [18], has a metamodel in which the concepts of a class attribute and operation do not have a common base abstraction of a class member. This enforces the user, for example, to make iterations through all class attributes and operations separately, in order to provide an output for all members of a class. Furthermore, the scripting languages themselves are often vendor-specific and not object-oriented. For instance, in Rational Rose, which uses a variation of Basic for scripting, the user may not easily take a major part of the built-in code generators, and simply just override some parts of interest. Even more difficult is to make a complex generator from the scratch, because the possibilities for decomposition are weak (procedures and modules only, without classes and inheritance). A more in-depth discussion on the results of the related research in output generation is given in the next section.

Since the previous specification is actually a specification of the process of creating instances of abstractions from the intermediate domain, where both domains may be 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. Such a specification may be easier to build and maintain, and less error-prone. Consequently, it may cure the drawbacks mentioned previously. Finally, the specification of output generation in customizable modeling and metamodeling tools is only one potential field of application of the solution proposed here. The approach may be generalized as a method for formal specification of automatic transformations of models, possibly from different domains, with the prospects of benefits further discussed later in the paper.

3 OVERVIEW OF THE RELATED WORK

Since domain-specific metamodels may be formalized, the need for tool support to this process has been recognized for a long time [8, 10, 11]. This need was first met in the field of automatic programming environment generation [9]. With the maturation of numerous software-engineering methodologies and notations, especially of object-oriented ones, which have been developed with the perspective of Computer Aided Software Engineering (CASE) tools support, the field of meta CASE research has evolved [8, 10]. However, the discussion in this paper is not constrained to the field of software modeling, CASE, and meta CASE tools. The results of this work may be applied to domains other than software systems. That is why the qualification "CASE" is not used in the term "(meta)modeling tool."

A lot of customizable modeling and metamodeling tools, both commercial and academic ones, are available at present. For the purposes of this work, three kinds of modeling tools may be identified, according to the level of their customizability:

1. Simple, non-customizable modeling tools, where the metamodels and the mapping between them are fixed at the time of the tool design; the user may modify neither of them.
2. Customizable modeling tools, where the metamodels are fixed at the tool design time, but the user may customize the mapping to some extent [15, 18].
3. True metamodeling tools, where the user may specify the metamodels and the mapping [12, 13, 14, 16, 17, 19, 20].

A common property and weakness of all existing tools, which is most important to this work, is their output generation facility. All these tools provide a programming interface to the implementation of their metamodels through which the user may access the model elements in the modeling tool to produce the desired output. However, output generation is usually specified using a scripting language

that is proprietary and vendor-specific. Hence, the first hard-coded output generation strategy described in the previous section is available to the user. As they often offer a flexible interface to their metamodel, 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 by using the same scripting language, and there is no other opportunity for doing this at a higher level of abstraction and/or visually.

In order to cope with the complexity of the output generation specifications, the tools offer different possibilities for their decomposition, either procedural or object-oriented. In both approaches, the structure of the model elements is traversed in a certain order, and an operation responsible for output generation for each kind of element is invoked. The operation generates output for the visited element and invokes operations for the elements with which it is connected. The problem of decoupling the domain's type structure and the traversal strategy is often solved using the *Visitor* design pattern [5].

A recently proposed technique [25, 26] exploits the conveniences of the *Visitor* design pattern and uses concise and abstract, but textual specifications for output generation. In this approach, the user must provide *traversal specifications* and *visitor specifications*. A traversal specification determines the navigation strategy, i.e. the set of model elements that should be visited before or after an element of a certain type. A visitor specification defines the operations that should be performed at the visit of an element of a certain type. These operations may include specific, user-defined output generation and/or further navigation, as defined by the traversal specification. From these specifications, the C++ code of the output generator is obtained automatically. However, this approach does not focus on the process of creation of the target model; this is completely left to the user-defined C++ code. In case of the textual output, this approach is very efficient. However, if the target domain's metamodel is object-oriented, it is not appropriate. In this case, it suffers from the same shortcomings as the approaches described previously.

Another approach may be described as a template-directed output generation [15]. It is also known as the concept of *archetypes* [31]. The output scheme is defined as a text written in the target language, representing the desired output with parameters (tags) that refer to the source model elements. The tags will be replaced with the concrete values (strings) obtained from the referred model elements at the time of output generation. For the demonstrational example, a part of the template for the base state class may look like (tags are enclosed in brackets):

```
class [fsm.name]State {
public:
    [fsm.name]State ([fsm.name]* fsm) : myFSM(fsm) {}

    [ForEach ev:Event in fsm.myEvents]
    virtual [fsm.name]* [ev.name] () {return this;}
    [EndForEach ev]

    virtual void entry () {}
    virtual void exit  () {}
    ...
}
```

Semantically, this approach is equivalent to the first solution described in the previous section, because it directly specifies the desired textual output, although in a slightly different manner. It may be superior since the specification is clearer, because it does not include output commands and formatting characters—they are implicitly defined by the template. When formatting is complex and important, this may be significant, as in the given simple example. However, this approach has to define control structures for output generation other than the simple sequence, which is implicit in the template text, e.g., conditions and repetitions. In the case when these structures are often needed, this approach loses

its clarity. Furthermore, this approach is applicable to textual (sequential) output only. The specification is language-dependent and is hardly reusable for other target languages. Finally, the impedance-mismatching problem remains—the source and the target domains are still conceptually far apart.

There are a number of approaches addressing a similar problem using structural transformations of grammar-based models and various rule-based techniques [22, 23, 24, 32]. Their goal is to transform a user-defined model written in a domain-specific language into another model in another target language. Although the goal is similar to the one presented here (transformation of models), there are also many differences. First, although their principles may be generalized to more abstract terms, they primarily deal with textual models (or, more generally, with sequential structures of elements). 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 for metamodeling. The main purpose of the supporting environments of that kind 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 that is subject to transformations is inherently a tree. In the modeling environments that use the object-oriented paradigm for metamodeling, there is no need for the parsing phase, because the user explicitly creates instances of abstractions and their links. In these environments, therefore, the model representation is a typed graph of objects (instances of classes) as nodes, connected with links (instances of associations) as edges. This is why the approach presented here may be considered 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 [23]. The approach proposed 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 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 object-oriented metamodels. It is evident that some domains may be modeled with much less effort using the object-oriented paradigm instead of grammars. This includes most modeling methods with visual notations. For such cases, the approach proposed here 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.

Another issue that is proposed in this paper is the idea of highly abstract, domain-specific modeling, and gradual, automatic model refinement until the desired system implementation is reached. The potentials of this idea have been noticed in the Draco approach [29], but in a completely different context. The Draco approach supports program construction in domain-specific textual languages, and source-to-source transformations into programs from other domains. Therefore, the concepts of domains, domain mappings, and model transformations (called "program refinements" in Draco) proposed here have their counterparts in Draco. The Draco approach advocates also the reuse of mappings and domains from repositories, in order to quickly and easily customize the system implementation. However, Draco is based on textual languages, so it can be considered as one of the grammar-based approaches. Therefore, it suffers from the same drawbacks as the other approaches, because the transformations are often difficult to specify, understand, and maintain.

On the other hand, the approach proposed in this paper is based on a completely different, object-oriented paradigm. It supports generalization/specialization and polymorphism of metamodels and mappings, too [27]. It also uses a visual notation and hierarchical organization of specifications. As a conclusion, to the best of his knowledge, the author is not aware of any other approach that is closely related to the one presented in this paper.

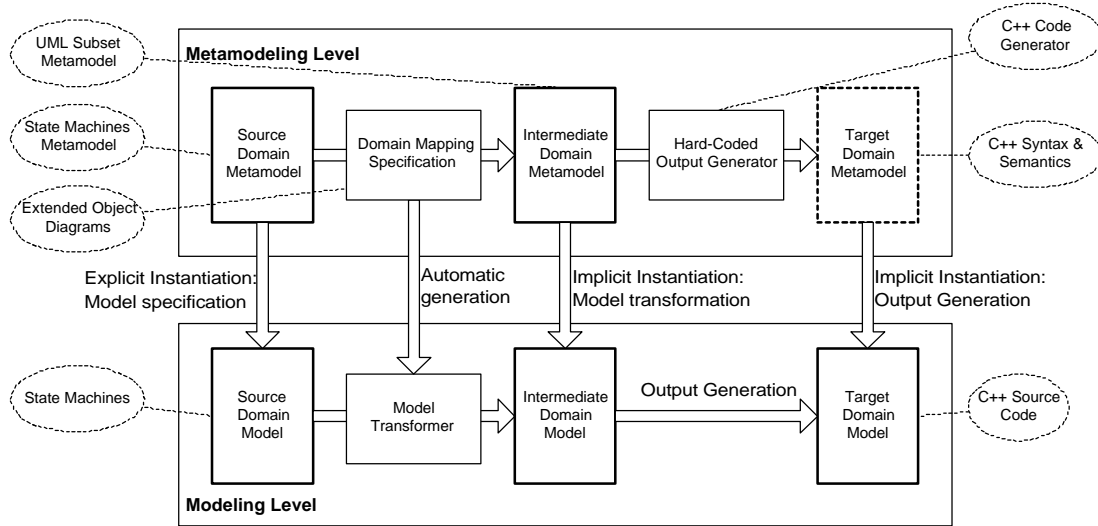


Figure 2: The idea of the domain mapping strategy in the context of the demonstrational example. The transformation from the source into the target domain is split into two (or generally more) steps in order to cope with the complexity of the mapping specification.

4 IDEA AND PURPOSE OF DOMAIN MAPPING

The idea of the domain mapping strategy is shown in Figure 2. The genesis of the idea was described in Section 2, where the intermediate model was introduced. The principle is in introducing an intermediate metamodel at the metamodeling level, and an intermediate model at the modeling level. For the demonstrational example, the intermediate metamodel is a subset of the UML's metamodel. The advantage is because each of the transformations is much less complex than the direct transformation, and is thus easier to specify and maintain.

The domain mapping specification should be formal and preferably graphical. It should specify the set of instances of the abstractions from the intermediate metamodel that are to be created for an element from the source model, along with the links (instances of associations) between them. Consequently, it is best represented with a UML object diagram [6]. However, a standard object diagram is not sufficient for mapping purposes. The concepts of repetitive and conditional object creation are also needed. These concepts will be described in the next section.

Consequently, the source and the intermediate models consist of instances of the abstractions from the corresponding metamodels and links between them (Figure 3). The source model (Figure 3a) is created by the user. The intermediate model (Figure 3c) is generated automatically by a model transformer. The transformer is automatically generated from the domain mapping specification (Figure 3b). The elements of the generated model are hierarchically grouped into packages according to the repetitive elements of the mapping specification. This yields to a better navigability through the generated model and a capability of tracing to the originating elements of the source model.

The contents of the models may be viewed as typed graphs of instances (nodes) connected with links (edges). This structure is fundamentally different from the sequential organization of a textual output. That is why it requires methods for specification of mappings other than those described in the previous sections. The hierarchical organization of models in packages as in Figure 3 is only conceptual, aimed to make the complexity of the graphs tractable.

Figure 4a shows the UML definition of the relationships between the models and their metamodels. There is also an important generalization of the approach depicted in Figure 4b: the models may be generated from each other in a pipelined fashion, where each transformation is performed using a certain domain mapping specification. Note that a domain mapping specification, as

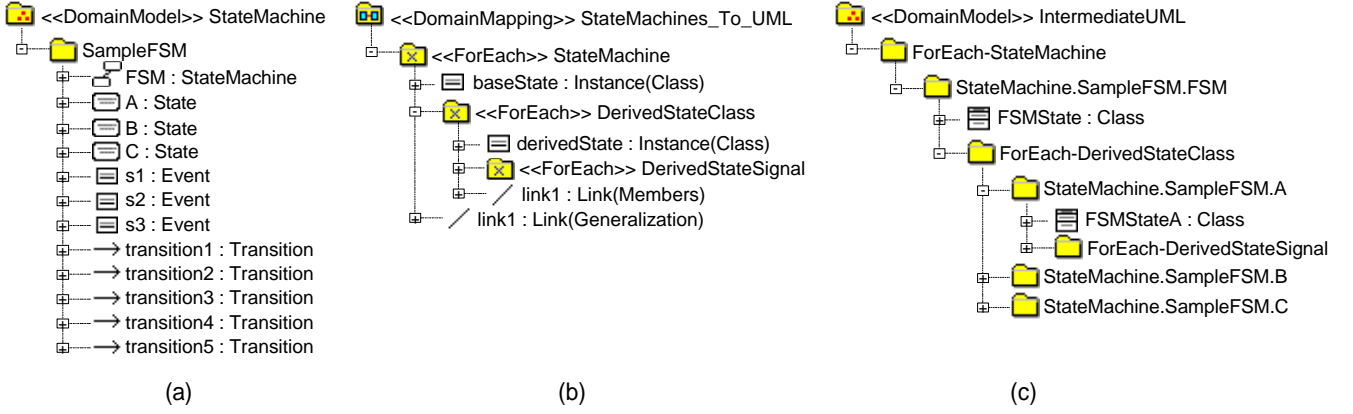


Figure 3: The source model (a), the mapping specification (b), and the generated (intermediate) model (c). The source and the generated models consist of instances of the abstractions from the corresponding metamodels, linked by instances of associations. The source model is created by the user. The intermediate model is generated automatically, using the domain mapping specification. The elements of the generated model are hierarchically grouped into packages according to the repetitive elements of the mapping specification. The origin of a package generated by repetition is encoded in its name.

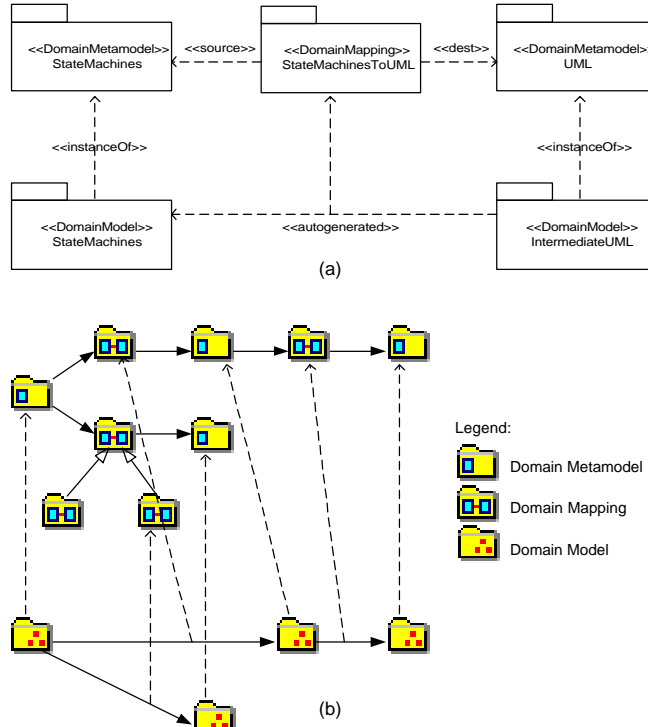


Figure 4: (a) Domain mapping scheme. The packages in the upper row represent the metamodeling level. The metamodels are in the packages stereotyped with <<DomainMetamodel>>. The domain mapping specification is in the package stereotyped with <<DomainMapping>>. The packages stereotyped with <<DomainModel>> in the bottom row represent the modeling level. The intermediate model is generated automatically from the source model, using the given mapping specification. (b) Model pipelining as a generalization of the approach. The models may be generated from each other in a pipelined fashion, where each transformation is performed using a certain domain mapping specification.

a kind of a package, may be specialized by several other mappings, e.g., for variants of the mapping into the same domain [27]. Of course, only one specific mapping may be used for generating one model.

The applicability of the proposed approach is manifold. First, it may be used for output specification in modeling tools of all degrees of customizability. For non-customizable modeling tools,

it can be used by developers as a method for designing output generation. A customizable modeling tool, such as a CASE tool, may uncover its metamodels (e.g., the UML metamodel, a metamodel of the target programming language, the relational metamodel, etc.), and the user may specify the mappings. It can be also featured by metamodeling tools, where the user can customize both the metamodels and the mapping schemes.

Second, by generating models from different domains for the same system, a better understanding of the system and its more complete modeling may be achieved. In other words, complementary models from other domains may be automatically generated from the user-defined model in a consistent manner, in order to improve the system's specification.

Finally, models of different levels of abstraction may be obtained automatically by model pipelining. The process of creation of intermediate models may be viewed as a descent down the levels of abstraction, reaching the ultimate implementation gradually and consistently. Besides, other more abstract domains may be built on top of the already designed domains, and their implementations may be easily specified by the mappings from the new domains into the already implemented, conceptually close ones. Consequently, a repository of reusable domains and mappings may significantly contribute to efficient modeling and customizable system construction. An example will be given in Section 6.

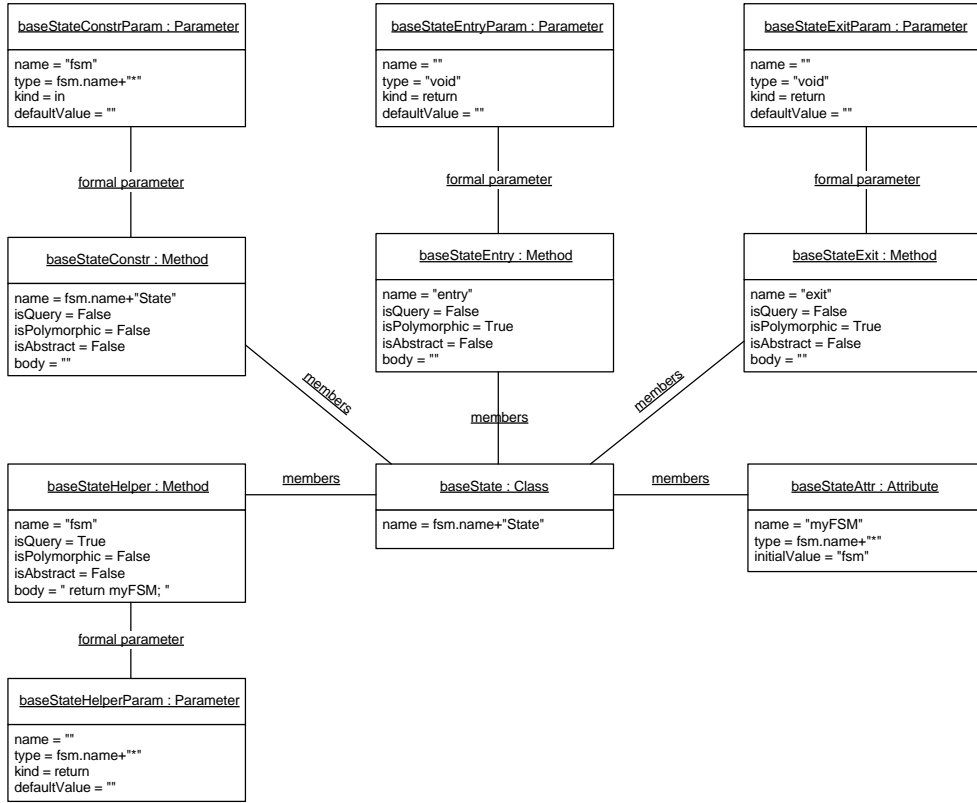


Figure 5: An object diagram from the domain mapping specification of the demonstrational example. The diagram shows the specifications for the base class `FSMState` and its members that are generated unconditionally. The diagram belongs to the context of the state machine accessible through the `fsm` identifier.

5 DOMAIN MAPPING SPECIFICATIONS

A mapping between two domains is specified in a hierarchy of packages, rooted with a `<<DomainMapping>>` package (Figure 3b). The packages contain instances and links. The specification is depicted with UML object diagrams. From now on, one mapping is considered, and its input and output domains are referred to as the source and the target domains, even though the output domain may be an intermediate one, as in the demonstrational example. The formal definitions of the semantics of the domain mapping specifications, along with the description of the procedure for generating the transformer based on these specifications may be found in [27, 28].

Instances, Attributes, and Links

An object diagram that specifies generation of a part of the target model for the demonstrational example is shown in Figure 5. It is assumed that the diagram is defined for one instance of `StateMachine` from the source model, referred to by the identifier `fsm`. The diagram specifies the set of instances of the target metamodel types that should be created for each `StateMachine` instance from the source model. The diagram specifies also the values of their attributes, along with the links between the instances.

The attribute values are defined by expressions that refer to the source model instances and their attribute values, using the navigation through the source model. The expressions may also use the results of user-defined functions that have access to the source model. These functions may construct

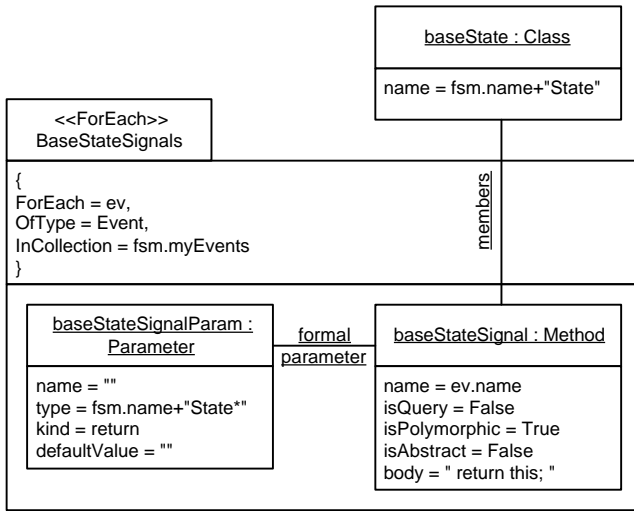


Figure 6: ForEach concept for repetitive element creation. The diagram shows only the specification for the base class `FSMState` and its members generated for the state machine's events. The diagram belongs to the context of the state machine accessible through the `fsm` identifier.

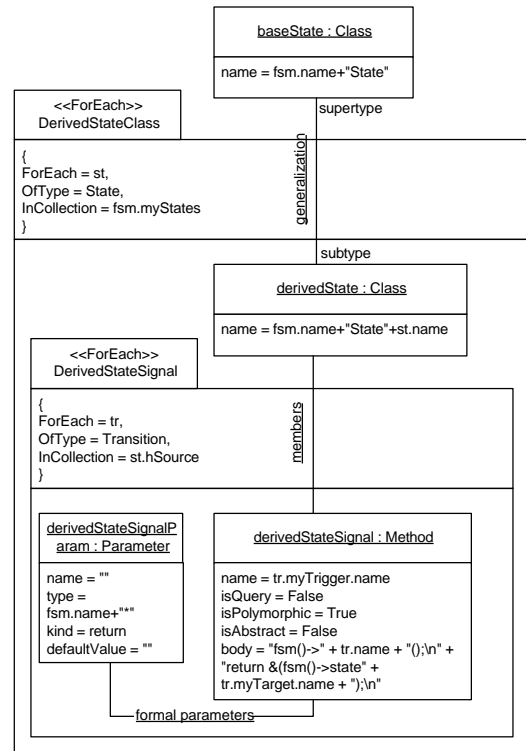


Figure 7: Nesting of ForEach packages. The diagram shows a part of the specification for the derived state classes and their members for the events.

attribute values in a manner not expressible through the object diagrams. This feature is particularly useful when structures other than graphs are to be created. For example, textual (i.e., sequential) structures are needed as bodies of methods in the UML domain, often parameterized with the elements of the source model and constructed in a complex manner.

Repetitions

The standard UML object diagrams are not sufficient for mapping purposes. There is also a need for repetitive element creation. For the demonstrational example, one operation in the base state class `FSMState` should be created for each event that the machine reacts upon (see Figure 1). For this purpose, a stereotyped [6] package with the stereotype `ForEach` is used. The example is shown in Figure 6. A `ForEach` package represents iteration through a collection of elements from the source model and creation of a set of target model elements for each of them. It has three tagged values [6]:

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

A link may connect two instances *i1* and *i2* from different `ForEach` packages. This means that a link will be created in the target model for each pair of instances created from *i1* and *i2* in the first common package that encloses instances *i1* and *i2*. In particular, when a link connects an instance inside a `ForEach` package and another outside that package, then each repetitive instance created by the iteration will be linked to the outer instance. (Formal definitions are given in [27, 28].) For the

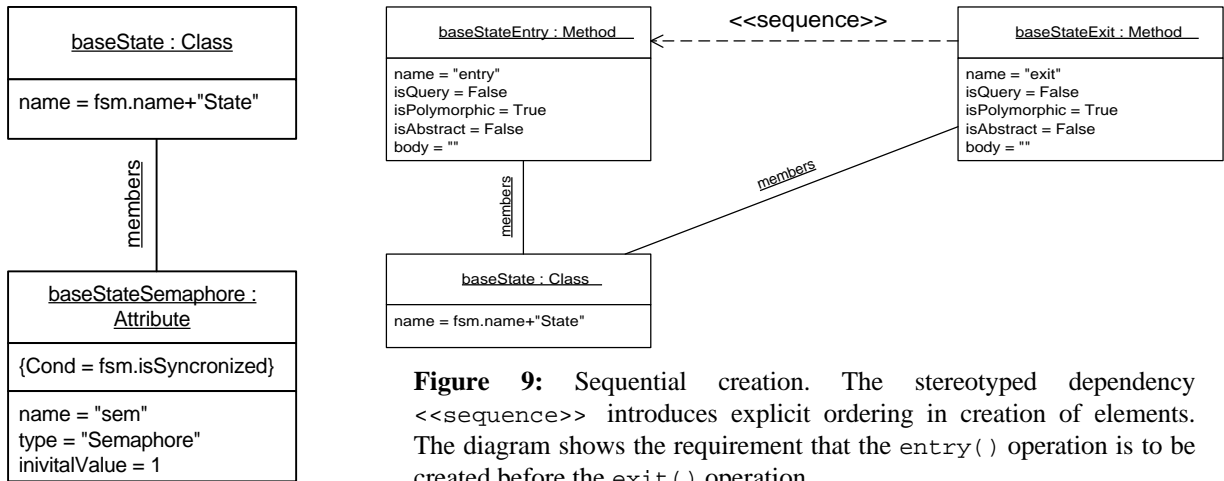


Figure 8: Conditional creation. The diagram shows the specification for the base class `FSMState` and its member (a semaphore) generated for synchronization purposes, only if the state machine is declared as "synchronized."

Figure 9: Sequential creation. The stereotyped dependency `<<sequence>>` introduces explicit ordering in creation of elements. The diagram shows the requirement that the `entry()` operation is to be created before the `exit()` operation.

expressions that specify attribute values of instances or the collection in a `ForEach` package, any formal language for navigation through the source model may be used. For example, the Object Constraint Language (OCL) [6] may be used if the programming interface of the tool is OCL-compliant. Another option is the programming language that is used in the tool for scripting (C++ is used in here).

`ForEach` packages actually represent loops in the process of the target model generation. They may be nested. A fragment for the demonstrational example is shown in Figure 7. Here, 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, as specified with the inner package.

A `ForEach` package introduces a scope for the expressions. The rules for the scope nesting are identical to those in procedural programming languages. An expression may use identifiers from its own scope, as well as from its enclosing scopes. A `ForEach` tagged value identifier is local to its package, and hides the same identifiers from the enclosing scopes.

Following the UML style, since a `ForEach` is actually a kind of a package, it is allowed that the contents of one `ForEach` package are defined by several diagrams to enhance readability. The whole domain mapping specification is thus organized into a hierarchy of packages rooted by a `<<DomainMapping>>` package, where each package owns a set of elements depicted in a set of diagrams. The owned elements may be instances, links, or nested simple or `ForEach` packages. `ForEach` packages may iterate through all instances of a source domain abstraction, using a built-in operation for that purpose (e.g., `StateMachine::getExtent()` [7]). Consequently, the diagrams shown in figures 5 to 7 belong to the same `ForEach` package that iterates through all instances of `StateMachine`.

Conditions

The concept of conditional creation is also needed. Instances, links, and packages may be tagged with conditions that are Boolean expressions defined in the context of the source model (the `Cond` tagged value). If the expression evaluates to `False` when the target model is being created, the element is not

created. A simple example is shown in Figure 8. The example assumes that the `StateMachine` type in the source metamodel has a Boolean attribute `isSynchronized`. If the value of this attribute is set to `True` for an instance of `StateMachine` in the source model, the generated state machine code should be mutually exclusive in a concurrent environment (monitor). This may be achieved with an attribute of the librarian type `Semaphore` that is generated in the base state class `FSMState` and the corresponding wait/signal operations in all publicly accessible operations (not shown in the diagram).

Sequences

The generated target model is a hierarchy of packages (Figure 3c), where each package is an unordered collection of its elements 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 (except for some special cases described in [27, 28]).

Sometimes, however, an explicit ordering of elements is needed. This ordering may ensure a proper sequential traversal through the target 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 with a dependency from x to y , stereotyped with `<<sequence>>`. Consequently, y will precede x in a traversal of the elements of their enclosing package.

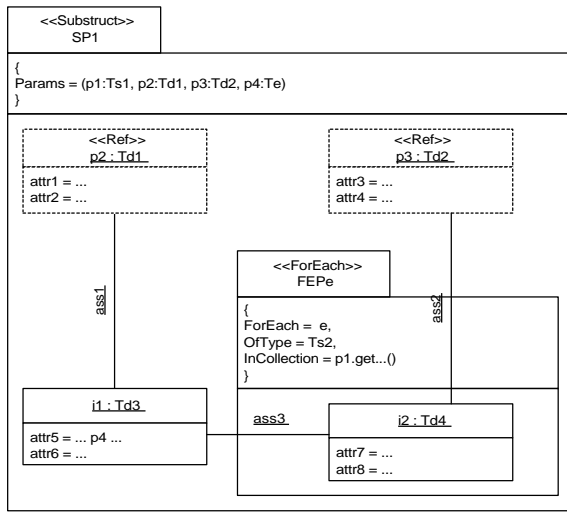
Figure 9 shows an example where the requirement that the operation `FSMState::entry()` should precede `FSMState::exit()` is defined by a sequence dependency. Therefore, the C++ code generator will encounter the `entry()` operation in the intermediate model first, and will generate it prior to the `exit()` operation.

Parameterized Substructures and Recursion

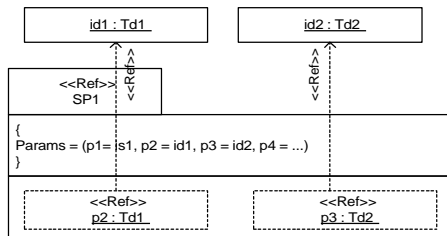
It is often the case that the same or a slightly different substructure should be generated at different places in the target model. In order to avoid redundancies and improve the organization of the mappings, the proposed technique supports definitions of parameterized substructures. They correspond to the concept of procedures in procedural languages. An example of a substructure definition is shown in Figure 10a. A parameterized substructure definition is given in a package stereotyped with `<<Substruct>>`. Such a package may contain usual mapping specifications, as described before. However, it cannot contain other `<<Substruct>>` packages. This constraint has been introduced due to implementation reasons, because some languages that may be used to implement the transformer do not support static nesting of procedure definitions. A substructure definition may have its formal parameters, defined in the `Params` tagged value. The parameters may be:

- (a) References to the elements of the source model, which may be used in expressions for navigation through the source model.
- (b) References to the instances of the target model already created at the time of substructure generation. These references may be used for creating links between the instances from the generated substructure and those from its environment.
- (c) Other parameters interpreted by the supporting environment and used for implementation details, e.g. for attribute setting expressions.

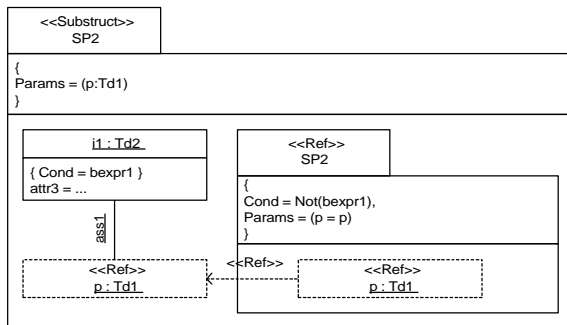
Inside a substructure definition, the parameters that are references to target model instances may be depicted as instances with the stereotype `<<Ref>>`. Those instances may have attribute settings and links towards other instances. All such specifications affect the referenced instances. Links may not surpass the borders of a substructure definition. Expressions in a substructure definition may use all local identifiers, including the formal parameters.



(a)

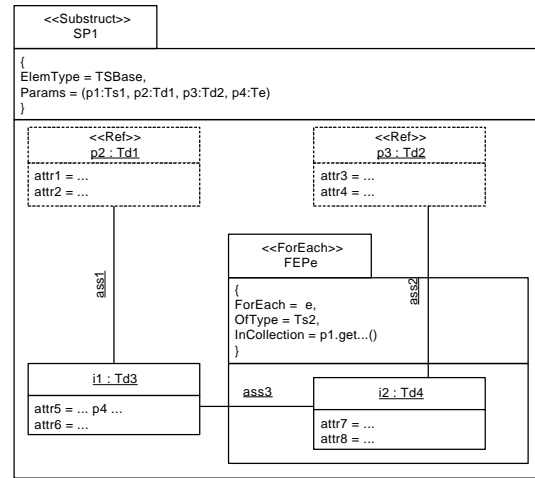


(b)

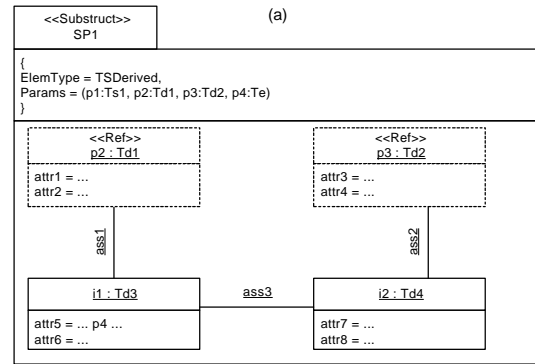


(c)

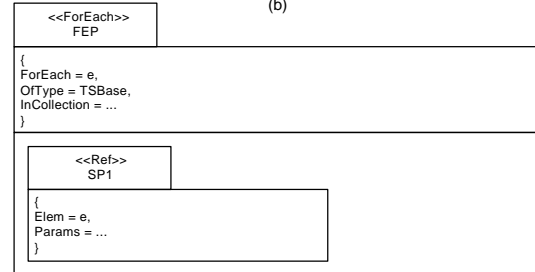
Figure 10: Parameterized substructures and recursion. (a) Substructure definition is in a `<<Substruct>>` package. (b) Substructure reference (`<<Ref>>` package) is a request for generating substructure at the place of "invocation." (c) A substructure definition may directly or indirectly refer to the same substructure (recursion).



(a)



(b)



(c)

Figure 11: Polymorphism. (a) A substructure definition may be assigned to a source metamodel type `TSBase` using the `ElemType` tagged value. (b) The substructure definition may be overridden for a derived type `TSDerived`. (c) The referenced substructure is generated polymorphically, depending on the concrete type of the source model element referred to by the `Elem` tagged value.

The request for generating a substructure is specified with a *substructure reference* (or *invocation*), depicted as a `<<Ref>>` package (Figure 10b). This concept is analogous to procedure invocation in procedural languages. The actual parameters of a substructure reference are defined through the `Params` tagged value. The actual parameters that are references to instances from the target model may be depicted as `<<Ref>>` instances, connected by `<<Ref>>` dependencies to the instances from the invocation environment. A `<<Ref>>` package may not contain other elements.

A substructure reference may exist in any package, including a definition of another or the same substructure. Therefore, a substructure definition may refer, directly or indirectly, to the same substructure. In other words, recursion is also supported, as shown in Figure 10c.

Polymorphic Substructures

A substructure definition `SP1` (Figure 11) may be assigned to a type `TSBase` from the source metamodel. This is specified by the `ElemType` tagged value of the substructure definition (Figure 11a). Let `TSDerived` be a type derived from `TSBase`. The substructure `SP1` may be redefined (overridden) for `TSDerived` (Figure 11b), provided that the redefinition has the same signature (the number and the types of the formal parameters).

When such substructure is referenced (Figure 11c), a reference to a source model element is provided by the `Elem` tagged value. The substructure is generated polymorphically, depending on the concrete type of the referenced element. For the example in Figure 11c, the iteration `<<ForEach>>FEP` uses the reference `e` to the current element of the collection. The reference is of type `TSBase`, but it can refer to objects of the class `TSBase`, as well as to objects of the derived class `TSDerived`. If `e` refers to an object of `TSBase`, the substructure defined in Figure 11a will be generated. If, however, `e` refers to an object of `TSDerived`, the substructure defined in Figure 11b will be generated.

This mechanism is completely analogous to the same concept of polymorphism in traditional object-oriented programming languages. Consequently, it has the same potentials of usage. It should be noticed, however, that the definitions of substructures are indirectly related with the source metamodel types, hence the domain's type hierarchy is not "spoiled" with the definitions of transformations.

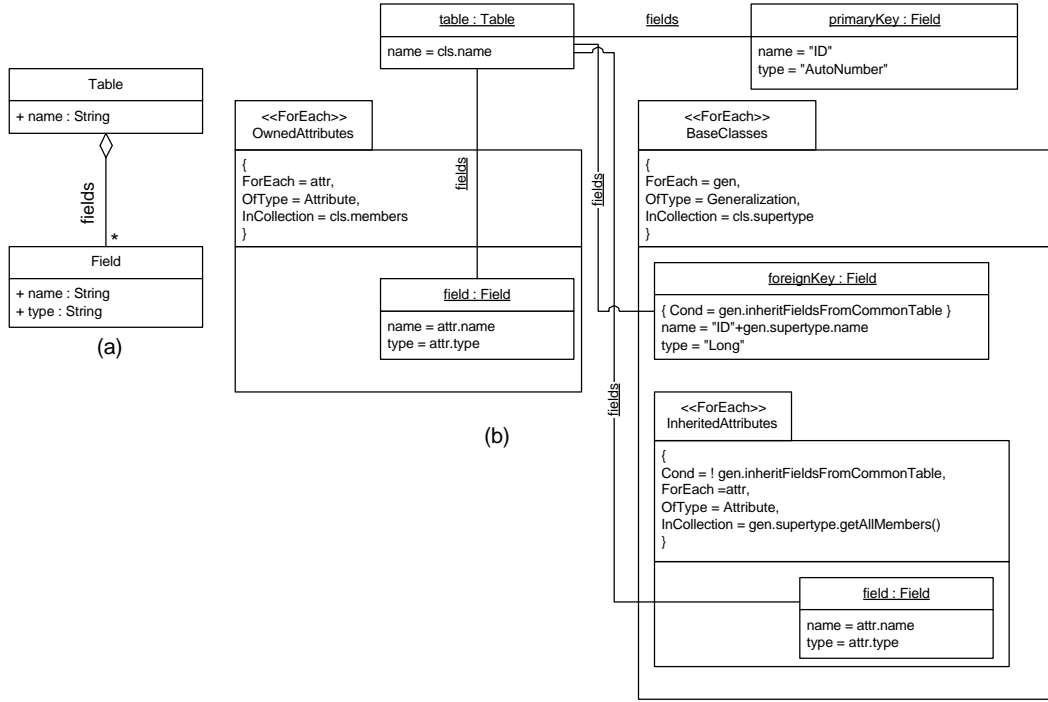


Figure 12: Example: Generation of the relational database scheme from a UML class model. This example focuses on inheritance. The source domain metamodel is the metamodel of UML (not shown here). (a) The target metamodel (relational). (b) The domain mapping specification. Operation `getAllMembers()` returns the collection of all owned and inherited members of a class.

6 EXAMPLES AND CASE STUDY

This section presents three simple examples that illustrate the applicability of the approach, and short descriptions of several more complex systems that have been developed using the proposed approach. The experiences from these projects are briefly reported and discussed.

Example 1: Object-Oriented to Relational Transformation

This example deals with the problem of transforming the object-oriented structural (class) model into the relational database model. This is a common task when persistence of objects is accomplished with a relational database. The source domain in this case 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 12a. It is a simplified version encompassing tables and fields only. It is now easy to specify the generation of SQL declarations from this intermediate domain, because it is almost (if not completely) a 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 independent table, with all inherited attributes placed 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 with a set of records in the tables that represent its own class and its base class. We assume that the user may select one of the approaches for each generalization in the class model, by setting a Boolean property of the generalization named `inheritFieldsFromCommonTable`. If this property is set

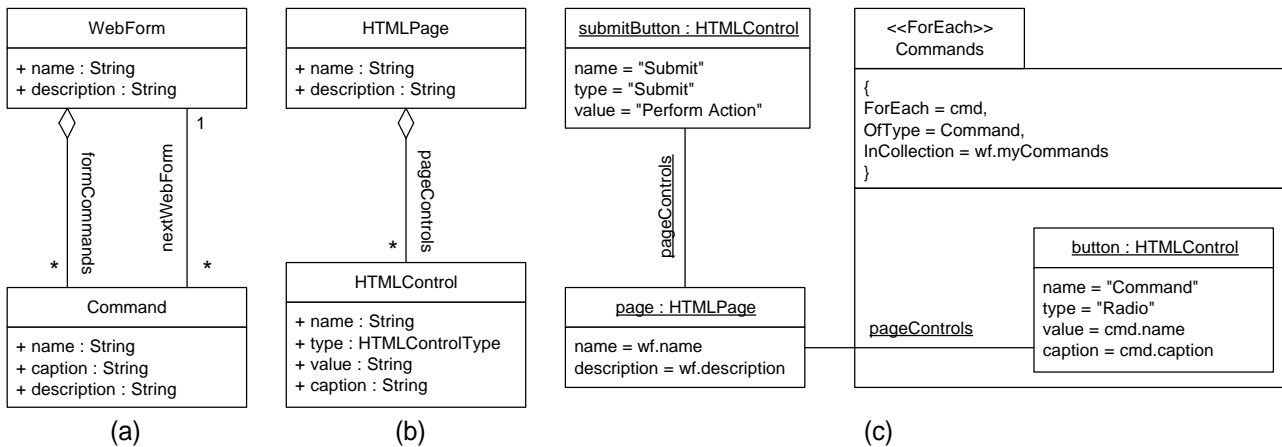


Figure 13: Example: Web application modeling. (a) An excerpt from the source domain metamodel. (b) An excerpt from the target domain metamodel. (c) The domain mapping specification.

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 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 mapping specification is shown in Figure 12b.

Example 2: Web Application Development

This example shows a case when UML is not used as any of the domains. It is taken from one of the author's projects on database-centric web application development [27]. The author has designed a method and infrastructure for rapid web application development. A very small part of the approach is presented hereby, just to illustrate the usage of domain mapping. In this approach, the application is modeled as navigation through *web forms*. From one web form, the user can choose a *command*, which performs some actions in the database on the server (i.e., queries assigned to that command) and displays another web form. The commands are implemented as radio button options in the web form, and one "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 13a. 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 domain is introduced, which may be mapped directly to the target domain. It contains abstractions such as **HTMLPage** or **HTMLControl** (text box, list box, radio button, etc.). The intermediate metamodel is shown in Figure 13b, and the mapping scheme in Figure 13c. The complete prototype modeling tool was implemented by the author in only three days, including metamodeling, code generation, and testing.

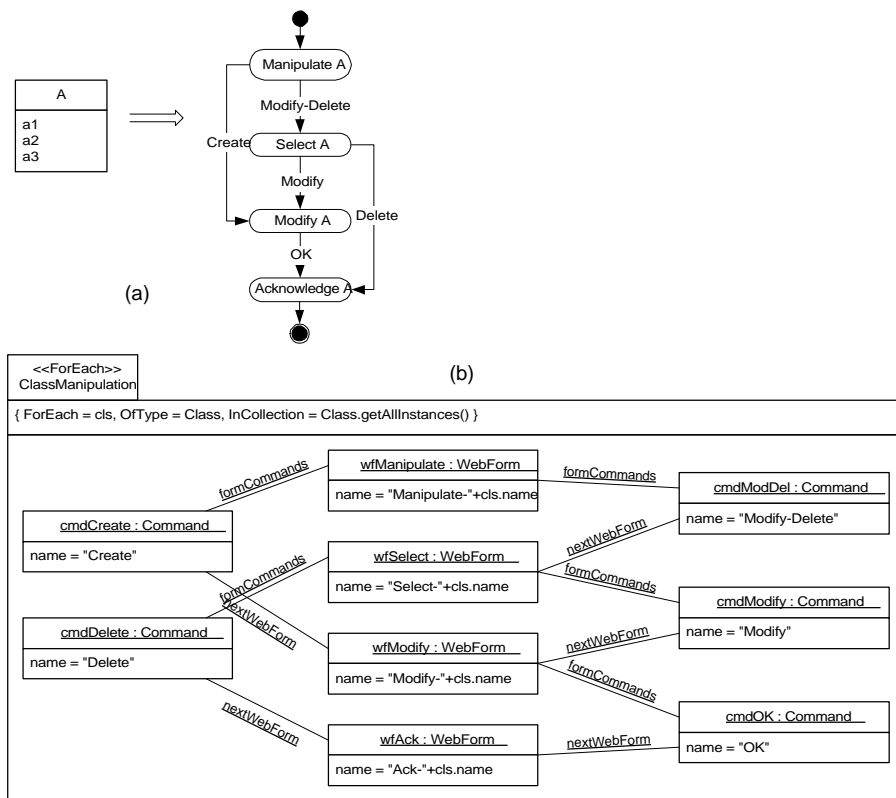


Figure 14: Example: Automatic implementation of structural use-cases. (a) A schematic representation of the mapping: for a class defined in the source model, a navigational pattern represented by the state-transition diagram may be generated. The states represent web forms, and the transitions represent commands (navigation). (b) The domain mapping specification.

Example 3: Automatic Implementation of Structural Use-Cases

The third example illustrates possibly the greatest potential of the proposed strategy, model pipelining. It is an upgrade of the previous example of the database-centric web application development. Namely, the structural model of a problem domain may be specified by the conceptual class model with abstractions (classes) from that problem domain and relationships between them (most notably, associations and generalizations). On the other hand, the behavioral aspect of the problem is specified by use cases [4, 6]. However, it may be noticed that the majority of use cases that have to be implemented in a typical application may be characterized as "structural." These are the use cases by which the user is able to simply create, modify (attribute values of), or delete instances of the abstractions and their links (as instances of associations). In other words, these use cases simply manipulate with the application's object structure. Often very few use cases may be qualified as "behavioral," meaning that they navigate through the application's object structure (instances and links) and provide the functionality that is specific for the application.

UML does not specify any formal semantics of use cases, nor any format for use case scenarios that implement use cases. Moreover, most formats that analysts use to express use case scenarios are very informal, and could not be used to develop implementation automatically. That is why use case diagrams are generally useless for generating code. Instead, developers have to make much effort to implement every structural use case from the scratch, unless the implementation framework supports it generically. On the other side, while implementing structural use cases, the developers may use various patterns that are based on the semantics of the structural relationships between abstractions. In that case, the implementation may be supported automatically, as it will be indicated here.

To illustrate this, a very simple example is shown in Figure 14a. For each class *A* defined in the structural model, a set of web forms may be generated that are used to create, modify, or delete instances of *A*. For example, creation of an instance of *A* incorporates creation of an instance with default values of attributes, and then modification of these values using the "Modify *A*" form. Furthermore, modification of an existing instance of *A* includes selection of the instance first, and modification of its attribute values then, etc.

Following this strategy, an application development environment may offer the developer to specify whether he/she wants to implement the usual structural use cases for a class in the defined conceptual model automatically or manually. The developer may also select a design pattern for the automatic implementation. A domain mapping specification from the UML domain, i.e. its structural part (class model), into the domain of web forms and navigation as in this example (Figure 14b) may be selected for the automatic implementation using the selected pattern. The mapping may be defined by the developer or chosen from a repository.

Of course, the necessary condition to achieve the automatic implementation of a use case is to generate the intermediate model, such as the scenario definition in Figure 14a, which has formal execution semantics. In other words, a formal modeling domain should be used for specifying scenarios. An example of such a domain is the state-transition specification of web forms and commands from the previous example, and another, very promising one is the proposal of Action Semantics for UML [7].

The described approach does not tend to propose any general formal semantics of the notion of use case, but only to indicate possibilities to automatically generate implementations (scenarios) of structural use cases, using various design patterns. The proposed domain mapping strategy is a means to specify formally the automatic implementation of use cases using patterns, i.e., to define the semantics of particular use case categories. The pattern and the intermediate modeling domain with formal execution semantics that is used in each particular situation depend on the problem domain, implementation issues, and other aspects, and may be customizable.

On the other side, the database definition can be obtained from the class model, too, as shown in the first example. Thus, by using model pipelining, the complete implementation of structural use cases can be obtained from the very abstract conceptual model of the problem domain by decreasing the level of abstraction gradually, in a sequence of automatically generated models.

Of course, it is wrong to expect that this strategy may lead up to a complete desired implementation. However, the strategy is still open for the user to incorporate additional functionality at each level of abstraction in the model sequence. Besides, with a considerable amount of customizability of the mapping, it is reasonable to expect that the user will be satisfied with the greatest part of the generated models, but will still want to make some slight modifications. The user can still do this in each generated model [27]. This method seems to be potential, but further investigation of its applicability, potentials, and possible weaknesses is needed.

Case Study and Discussion

Apart from the systems developed by the author and described in the previous examples, three other more complex and real-world projects have been accomplished, and several others are in progress. They have been developed by three different undergraduate students of computer science, as the final graduation projects at the University of Belgrade. Their subjects were three modeling techniques for different domains, and their goal was to define the metamodels of the domains, and to specify the generators for C++ code that may serve as executable implementation of the models. Following the proposed approach, they used an intermediate domain that is a subset of UML. This domain was defined to be conceptually close to common object-oriented programming languages (C++ and Java), so the C++ code generator has been constructed easily for this domain. This was accomplished by

defining the abstractions that are directly supported by these languages, such as class, operation, attribute, inheritance, etc., but not association for example (because it does not have a direct counterpart in the languages, but is implemented through attributes). Hereby, this domain is referred to as the OOPL (OO programming language) domain. The metamodel of this domain has a dozen of classes and a dozen of associations.

The first project dealt with state machine (SM) modeling. Since it was the pilot project, its task was to check the correctness of the proposed concepts and the developed prototype tool. The source domain metamodel had four classes and six associations. The specification had about thirty instances and seven `ForEach` packages. It took the student about three days to implement and test the code generator, although the transformer from the source domain into OOPL was created manually from the mapping specifications (the supporting tool was not ready at that time).

The next project dealt with metamodeling of the structural part of the ROOM method (Real-Time Object-Oriented Modeling) [3]. The source domain metamodel had 19 classes and 26 associations. The domain mapping specifications had 5 diagrams, about 40 instances, and 5 `ForEach` packages, and took the student about five days to develop.

The third project dealt with hardware logic design (LD), as usually supported by common digital circuit modeling tools. The goal was to support modeling the circuit structure and behavior, and their simulation using the generated C++ code. In this project, hardware blocks could be either hierarchically decomposed by the defined substructure, or their behavior could be specified using the C++ code for their functional simulation. The source domain metamodel had 12 classes and 8 associations. The mapping specifications consisted of 3 diagrams, about 20 instances, and 6 `ForEach` packages, and took the student about three days to develop.

The case study has addressed the issues and indicated the properties of the approach as follows:

(a) Applicability. The examples were chosen to cover different aspects: structural modeling (structural ROOM), behavioral modeling (SM), and combined (LD), as well as software (ROOM and SM), and non-software (LD) domains.

(b) Acceptability and learning effort. The projects have been accomplished by three different undergraduate students without any professional experience, but only with the educational software development training at the level of a standard CS curriculum. They had good education in procedural and OO programming, and introductory knowledge of UML modeling, but no knowledge or experience at all in conceptual and domain-specific modeling, metamodeling, and automatic code generation. They have all perceived the technique easily, without any help of the author, just by reading technical reports and studying available examples. They have all reported that they needed a short time to learn to apply the method, just the time it took them to read the available materials.

(c) Production time. To avoid the possibility of the author's subjectivity and the effects of the learning curve, the three projects have been accomplished by three different students without any help of the author. The time needed to specify the output generation was always short (a couple of days). The duration of this task depended predominantly on the complexity of the modeling domain and its metamodel. Besides, in the first project (SM), the production time for the proposed approach was compared with the traditional scripting approach. It took the student about ten days to implement the output generator directly in C++, and only three days with the help of the proposed method. Precisely, in this particular case, there might have existed a small learning curve effect, because the same student completed the code generator using the traditional approach first, and then applied the proposed method. The effect was, however, likely to be very bound to the knowledge of the particular modeling domain and the form of the desired code, which existed also before applying the traditional method. Besides, the counter-effect of having to learn and apply the newly proposed method by the student for the first time may have made the balance. Finally, if it existed at all, we really believe it was not significant, because our previous rich experience in developing code generators of much greater complexity make us confident in the stated conclusion.

(d) Maintainability. In all three projects, the students had tasks to make two or more different versions of the output generation, by changing the initial mapping. They have reported no difficulties in accomplishing these tasks, because they have found their specifications clear, concise, and easy to modify and extend. This was due to the UML's feature of grouping model elements into diagrams, which is completely followed by the proposed approach. A diagram may show only parts of the complete mapping, i.e. only some elements that are relevant to the particular context of the diagram. The consistency of the mapping is preserved by the supporting tool, although one element of the mapping may exist in several diagrams. Therefore, the diagrams may be organized to reflect the coherent, loosely coupled parts of the desired target model. Consequently, a modification that is needed in the target model is easily applied to the mappings because it is often reflected to a small number of mapping diagrams. This advantage of the proposed technique over the traditional scripting techniques is actually the inherited advantage of the visual software modeling approaches (such as UML) over the textual programming languages, when maintainability is concerned. It is because the proposed technique uses the same concepts and tools for presentation and decomposition as UML (diagrams and packages instead of text and files).

(e) Reusability. The OOPL domain, i.e. its metamodel, and its C++ code generator has been reused in all three projects. Otherwise, if the traditional scripting approach had been used, the students would have had to develop separate code generators, completely independent and specific for their tasks, and to deal with all subtleties of the C++ syntax and semantics. Using the proposed approach, the students did not have to deal with any specific OO programming language, and the majority of the mapping specifications, except for the definitions of operation bodies, may be reused for generating Java code.

(f) Process. These projects have contributed to the proposal of a process of developing modeling environments and output generators. After a modeling domain is chosen and informally described, its metamodel should be defined. Our experience shows that this is the most difficult task and takes the longest time. However, this is the problem of conceptual modeling in general, and has nothing to do with the proposed approach. Secondly, one or more examples of models from this domain and the desired output for those models should be constructed. The examples should be simple enough to be tractable, but rich enough to cover all important concepts of the domain. This tends to be the second time-consuming task. Finally, the mapping specification is constructed by analyzing the parts of the sample output, and defining the mapping elements that produce them. The students have reported that this was the easiest task in the whole process, if the examples and the metamodel are constructed correctly.

The case study has shown some weaknesses of the approach, too. Namely, the concepts of `ForEach` packages, conditional creation, sequential dependencies, substructures, and recursion cover all fundamental control-flow structures. That is why even generation of sequential structures (e.g., text) may be specified using the proposed concepts and diagrams. However, the experience shows that textual structures are more easily generated using the traditional scripting or template-oriented approaches. This is especially the case when the bodies of operations of the output code are specified. Besides, the OOPL domain is not reusable enough when the bodies of operations are specified, because they are language-dependent. Furthermore, apart from sequences, conditionals, and iterations, control flow may generally incorporate concurrent paths or activations. Concurrency allows partially ordered sequences as well, not just the strict orderings that sequences require. This concept has not been addressed yet in the proposed method, and its applicability and potentials should be studied. All these issues may be addressed in the future work. As a conclusion, the proposed approach is suitable for bridging the gap between conceptually distant domains. The transformation of a domain that may be mapped directly and easily into a textual form is more suitably specified using the described traditional methods.

7 CONCLUSIONS

This work has studied the problem of customizing and specifying output generation in the context of modeling environments. Some weaknesses of the existing approaches have been analyzed, 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 a model in the source domain. If the modeling domains are conceptually at distant levels of abstraction, the mapping is specified, maintained, and reused with difficulties. This is why one or more intermediate domains should be introduced. The domains are metamodeled using the object-oriented paradigm, and the models consist of instances of the domain abstractions and links between them. The mapping is specified using the UML object diagrams that show instances of the target model to be created by the transformer. The UML object diagrams are extended with the concepts of conditional, repetitive, sequential, parameterized, and polymorphic element creation. These concepts are implemented by the standard UML extensibility mechanisms.

Several examples from different engineering domains have been provided. The preliminary case study have demonstrated that the proposed approach may be applied to different modeling domains, and that it appears to improve productivity and maintainability.

A metamodeling tool based on UML that will support the described approach is being developed at present. The architecture of the tool and the results of its use will be reported elsewhere. A further study of the applicability of the approach by building a repository of metamodels and mappings is needed, too.

Acknowledgements

The author is grateful to D. Marjanovic, Lj. Lazarevic, M. Zaric, and I. Djordjevic for their contributions to this research and the case study, D. Bojic, I. Tartalja, and Z. Jovanovic from the University of Belgrade, J.-P. Tolvanen from MetaCase Consulting, B. Selic from Rational, and anonymous reviewers for their invaluable comments on the paper.

REFERENCES

Note: The references are classified here for the sake of clearness in the review process. The list may be resorted in the final form of the paper on request.

Software Engineering and General

- [1] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* 8, 1987, pp. 231-274
- [2] SDL Forum Society, *SDL-2000*, <http://www.sdl-forum.org>, 2000
- [3] Selic, B., Gullekson, G., Ward, P., *Real-Time Object-Oriented Modeling*, John Wiley & Sons, Inc., 1994

Object-Oriented Modeling and UML

- [4] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modeling Language User Guide*, Addison-Wesley Longman, Reading, Mass., 1999
- [5] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns*, Addison-Wesley Longman, Reading, Mass., 1995
- [6] The Object Management Group, *OMG Unified Modeling Language Specification*, Ver. 1.3, June 1999, <http://www.omg.org>

- [7] "Response to OMG RFP ad/98-11-01: Action Semantics for the UML," Ver. 16, revised Sept. 5, 2000, <http://www.omg.org>

Metamodeling

- [8] Artsy, S., "Meta-modeling the OO Methods, Tools, and Interoperability Facilities," *OOPSLA'95 Workshop in Metamodeling in OO*, October 1995
- [9] Karrer, A. S., Scacchi, W., "Meta-Environments for Software Production," *Int'l J. Soft. Eng. and Know. Eng.*, Vol. 3, No. 2, pp. 139-162, May 1993. Revised and reprinted in *Advances in Software Engineering and Knowledge Engineering*, Hurley, D. (ed.), Vol. 4, pp. 37-70, 1995
- [10] MetaModel.com, *Metamodeling Glossary*, <http://www.metamodel.com>
- [11] Nordstrom, G., Sztipanovits, J., Karsai, G., Ledeczi, A., "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments," *Proc. IEEE ECBS'98 Conf.*, 1998

Customizable CASE and Meta CASE Tools

- [12] Advanced Software Technologies, Inc., *Graphical Designer*, <http://www.advancedsw.com>
- [13] Lincoln Software Ltd., *IPSYS ToolBuilder*, <http://www.ipsys.com>
- [14] MetaCase Consulting, *MetaEdit+ Method Workbench*, <http://www.metacase.com>
- [15] MicroGold Software Inc., *WithClass Scripting Tool*, <http://www.microgold.com>
- [16] mip GmbH, *Alfabet*, <http://www.alfabet.de>
- [17] Platinum Technology, *Paradigm Plus*, <http://www.platinum.com/clearlake>
- [18] Rational Software Corporation, *Rational Rose*, <http://www.rational.com>
- [19] University of Alberta, *MetaView*, <http://www.cs.ualberta.ca/news/CS/1998/research/software.html>
- [20] Vanderbilt University, *Multigraph Architecture*, <http://www.isis.vanderbilt.edu>

Model Transformations

- [21] Aitken, W., Dickens, B., Kwiatkowski, P., De Moor, O., Richter, D., Simonyi, C., "Transformation in Intentional Programming," <http://research.microsoft.com/ip>
- [22] 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*, December 1986
- [23] Garlan, D., Cai, L., Nord, R. L., "A Transformational Approach to Generating Application-Specific Environments," *Proc. Fifth ACM SIGSOFT Symp. Softw. Development Environments*, December 1992, pp. 68-77
- [24] Habermann, A. N., Notkin, D. S., "Gandalf: Software Development Environments," *IEEE Trans. Software Engineering*, Vol. 12, No. 12, December 1986, pp. 1117-1127
- [25] Karsai, G., Sztipanovits, J., Franke, H. "Towards Specification of Program Synthesis in Model-Integrated Computing," *Proc. IEEE ECBS'98 Conf.*, 1998, pp. 226-233
- [26] Karsai, G. "Structured Specification of Model Interpreters," *Proc. IEEE ECBS'99 Conf.*, March 1999, pp. 84-91

- [27] Milicev, D., *Automatic Model Transformations in Modeling Environments* (in Serbian), Ph.D. thesis, University of Belgrade, School of Electrical Engineering, March 2001. Also available at: <http://www.rcub.bg.ac.yu/~dmilicev>
- [28] Milicev, D., "Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments," tech. rep. TI-ETF-RTI-00-0042, Univ. Belgrade, School of Elec. Eng., October 2000. Also available at: <http://www.rcub.bg.ac.yu/~dmilicev>
- [29] Neighbors, J. M., "The Draco Approach to Constructing Software from Reusable Components," *IEEE Trans. Software Engineering*, Vol. 10, No. 5, September 1984, pp. 564-574
- [30] Papazoglou, M. P., Russell, N., "A Semantic Meta-Modelling Approach to Schema Transformation," *Proc. ACM Conf. Information and Knowledge Management (CIKM'95)*, 1995
- [31] Shlaer, S., Mellor, S. J., "Recursive Design of an Application-Independent Architecture," *IEEE Software*, January 1997
- [32] Simonyi, C., "Intentional Programming - Innovation in the Legacy Age," presented at IFIP WG 2.1 meeting, June 1996, <http://research.microsoft.com/ip>

APPENDIX

The generated C++ code of the transformer specified by the diagram in Figure 7. `ForEach` and `EndForEach` are C++ macros that implement type-sensitive iteration.

```
void DomainMapping_StateMachines_To_UML (Package& pckTarget) {

    // ForEach Package: StateMachine
    Package& pckForEachStateMachine = Package::create(pckTarget);
    pckForEachStateMachine.dmOrgName = "StateMachine";
    pckForEachStateMachine.name = "ForEach-StateMachine";
    ForEach(fsm, StateMachine, StateMachine::getAllInstances())

        Package& pckStateMachine = Package::create(pckForEachStateMachine);
        pckStateMachine.dmOrgName = "StateMachine";
        pckStateMachine.name = fsm.getFullPathName();

        // Instance: baseState
        Class& baseState = Class::create(pckStateMachine);
        baseState.dmOrgName = "baseState";
        baseState.name = fsm.name+"State";

        // ForEach Package: DerivedStateClass
        Package& pckForEachDerivedStateClass = Package::create(pckStateMachine);
        pckForEachDerivedStateClass.dmOrgName = "DerivedStateClass";
        pckForEachDerivedStateClass.name = "ForEach-DerivedStateClass";
        ForEach(st, State, fsm.states)

            Package& pckDerivedStateClass = Package::create(pckForEachDerivedStateClass);
            pckDerivedStateClass.dmOrgName = "DerivedStateClass";
            pckDerivedStateClass.name = st.getFullPathName();

            // Instance: derivedState
            Class& derivedState = Class::create(pckDerivedStateClass);
            derivedState.dmOrgName = "derivedState";
            derivedState.name = fsm.name+"State"+st.name;

            // ForEach Package: DerivedStateSignal
            Package& pckForEachDerivedStateSignal = Package::create(pckDerivedStateClass);
            pckForEachDerivedStateSignal.dmOrgName = "DerivedStateSignal";
            pckForEachDerivedStateSignal.name = "ForEach-DerivedStateSignal";
            ForEach(tr, Transition, st.hSource)

                Package& pckDerivedStateSignal = Package::create(pckForEachDerivedStateSignal);
                pckDerivedStateSignal.dmOrgName = "DerivedStateSignal";
                pckDerivedStateSignal.name = tr.getFullPathName();
```

```

// Instance: derivedStateSignal
Method& derivedStateSignal = Method::create(pckDerivedStateSignal);
derivedStateSignal.dmOrgName = "derivedStateSignal";
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";

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

// Link: link1 : FormalParameters
{ // First side:
    Parameter* ptrDerivedStateSignalParam = (Parameter*)
        (pckDerivedStateSignal.getElement("Parameter","derivedStateSignalParam"));
    // Second side:
    Method* ptrDerivedStateSignal = (Method*)
        (pckDerivedStateSignal.getElement("Method","derivedStateSignal"));
    // Link:
    if (ptrDerivedStateSignal && ptrDerivedStateSignalParam) {
        Method& derivedStateSignal = *ptrDerivedStateSignal;
        Parameter& derivedStateSignalParam = *ptrDerivedStateSignalParam;
        MMLink& link1 = MMLink::create(FormalParameters::Instance(),
            derivedStateSignal,derivedStateSignalParam);
        link1.dmOrgName = "link1";
    }
}

EndForEach(tr)

// Link: link1 : Members
{ // First side:
    Class* ptrDerivedState =
        (Class*)(pckDerivedStateClass.getElement("Class","derivedState"));
    // Second side:
    Package& pckForEachDerivedStateSignal =
        *pckDerivedStateClass.getPackage("ForEach-DerivedStateSignal");
    ForEach(pckDerivedStateSignal,Package,pckForEachDerivedStateSignal.getAllPackages())
        Method* ptrDerivedStateSignal =
            (Method*)(pckDerivedStateSignal.getElement("Method","derivedStateSignal"));
    // Link:
    if (ptrDerivedState && ptrDerivedStateSignal) {
        Class& derivedState = *ptrDerivedState;
        Method& derivedStateSignal = *ptrDerivedStateSignal;
        MMLink& link1 = MMLink::create(Members::Instance(),
            derivedState,derivedStateSignal);
        link1.dmOrgName = "link1";
    }
}
EndForEach(pckDerivedStateSignal)
}

EndForEach(st)

// Link: link1 : Generalization
{ // First side:
    Class* ptrBaseState =
        (Class*)(pckStateMachine.getElement("Class","baseState"));
    // Second side:
    Package& pckForEachDerivedStateClass =
        *pckStateMachine.getPackage("ForEach-DerivedStateClass");
    ForEach(pckDerivedStateClass,Package,pckForEachDerivedStateClass.getAllPackages())
        Class* ptrDerivedState =
            (Class*)(pckDerivedStateClass.getElement("Class","derivedState"));
    // Link:
    if (ptrDerivedState && ptrBaseState) {
        Class& derivedState = *ptrDerivedState;
        Class& baseState = *ptrBaseState;
    }
}

```

```

        MMLink& link1 = MMLink::create(Generalization::Instance(),
                                     derivedState,baseState);
        link1.dmOrgName = "link1";
    }
    EndForEach(pckDerivedStateClass)
}

EndForEach(fsm)
}

```

FIGURE CAPTIONS

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

Figure 2: 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 3: The source model (a), the mapping specification (b), and the generated (intermediate) model (c). The source and the generated models consist of instances of the abstractions from the corresponding metamodels, linked by instances of associations. The source model is created by the user. The intermediate model is generated automatically, using the domain mapping specification. The elements of the generated model are hierarchically grouped into packages according to the repetitive elements of the mapping specification. The origin of a package generated by repetition is encoded in its name.

Figure 4: (a) Domain mapping scheme. The packages in the upper row represent the metamodeling level. The metamodels are in the packages stereotyped with <<DomainMetamodel>>. The domain mapping specification is in the package stereotyped with <<DomainMapping>>. The packages stereotyped with <<DomainModel>> in the bottom row represent the modeling level. The intermediate model is generated automatically from the source model, using the given mapping specification. (b) Model pipelining as a generalization of the approach. The models may be generated from each other in a pipelined fashion, where each transformation is performed using a certain domain mapping specification.

Figure 5: An object diagram from the domain mapping specification of the demonstrational example. The diagram shows the specifications for the base class `FSMState` and its members that are generated unconditionally. The diagram belongs to the context of the state machine accessible through the `fsm` identifier.

Figure 6: `ForEach` concept for repetitive element creation. The diagram shows only the specification for the base class `FSMState` and its members generated for the state machine's events. The diagram belongs to the context of the state machine accessible through the `fsm` identifier.

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

Figure 8: Conditional creation. The diagram shows the specification for the base class `FSMState` and its member (a semaphore) generated for synchronization purposes, only if the state machine is declared as "synchronized."

Figure 9: Sequential creation. The stereotyped dependency <<sequence>> introduces explicit ordering in creation of elements. The diagram shows the requirement that the `entry()` operation is to be created before the `exit()` operation.

Figure 10: Parameterized substructures and recursion. (a) Substructure definition is in a <<Substruct>> package. (b) Substructure reference (<<Ref>> package) is a request for generating substructure at the place of "invocation." (c) A substructure definition may directly or indirectly refer to the same substructure (recursion).

Figure 11: Polymorphism. (a) A substructure definition may be assigned to a source metamodel type `TSBase` using the `ElemType` tagged value. (b) The substructure definition may be overridden for a derived type `TSDerived`. (c) The referenced substructure is generated polymorphically, depending on the concrete type of the source model element referred to by the `Elem` tagged value.

Figure 12: Example: Generation of the relational database scheme from a UML class model. This example focuses on inheritance. The source domain metamodel is the metamodel of UML (not shown here). (a) The target metamodel (relational). (b) The domain mapping specification. Operation `getAllMembers()` returns the collection of all owned *and* inherited members of a class.

Figure 13: Example: Web application modeling. (a) An excerpt from the source domain metamodel. (b) An excerpt from the target domain metamodel. (c) The domain mapping specification.

Figure 14: Example: Automatic implementation of structural use-cases. (a) A schematic representation of the mapping: for a class defined in the source model, a navigational pattern represented by the state-transition diagram may be generated. The states represent web forms, and the transitions represent commands (navigation). (b) The domain mapping specification.

Author's Biography

Dragan Milicev received the diploma degree in 1993, MSc in 1995, and PhD in 2001, all in Computer Science at the University of Belgrade, School of Electrical Engineering. He is an assistant professor at the Department of Computer Science of the same school. He has been serving as a researcher, project manager, or consultant in a number of academic and industrial projects. He authored or co-authored three books on object-oriented programming and modeling (in Serbian), and several papers in international journals and conferences. His research interests include object-oriented software engineering, model-based software development, metamodeling, and information systems. His personal web page with information on his projects and activities can be found at <http://www.rcub.bg.ac.yu/~dmilicev>.

