

**TEHNIČKI IZVEŠTAJ**  
**Elektrotehnički fakultet**  
**Univerzitet u Beogradu**

**TECHNICAL REPORT**  
**Faculty of Electrical Engineering**  
**University of Belgrade**

**Dragan Milicev**

**Elektrotehnički fakultet, Univerzitet u Beogradu**  
**Faculty of Electrical Engineering, University of Belgrade**

# **Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments**

Broj: TI-ETF-RTI-00-  
Datum: Oktobar 2000.

Number: TI-ETF-RTI-00-  
Date: October 2000



# Automatic Model Transformations Using Extended UML Object Diagrams in Modeling Environments

**Dragan Milicev**

Faculty of Electrical Engineering, Dept. Comp. Eng. & Sc.  
University of Belgrade  
POB 35-54, 11120 Belgrade, Serbia, Yugoslavia  
emiliced@etf.bg.ac.yu

## Abstract

One of the most important features of modeling tools is output generation. The output may be documentation, source code, net list, or any other model of the system being constructed. The process of output generation may be considered as the automatic creation of the model in the target domain from the model in the source domain. In this paper, it is recognized that this mapping 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 to describe the system better, or to decrease the level of abstraction of the user-defined model, gradually leading to the desired implementation. If the domains are metamodeled using the object-oriented paradigm, the models consist of instances of the domain abstractions and links between them. A new technique for specifying the mapping between different domains is proposed here. It uses the UML object diagrams that show the instances and links of the target model that should be created automatically. The diagrams are extended by the concepts of conditional, repetitive, parameterized, and polymorphic model creation, which are implemented using the standard UML extensibility mechanisms. Several examples from different software engineering domains are provided, illustrating the applicability and benefits of the approach. The first experimental results show that the specifications tend to be clear and concise, easy to maintain and modify, and lead to better reuse and to shorter production time.

## Keywords

Object-oriented modeling, Unified Modeling Language (UML), object diagram, metamodeling, domain-specific modeling, metaenvironment, meta CASE tool, automatic model transformation, automatic code generation, automatic output generation

## 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 the 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 the automatic generation of the model in the target domain from the model in the source domain. This mapping does not need to be a single-step generation of the ultimate output from the tool. The tool may generate multiple intermediate models as other views that describe the system better, or that decrease the level of abstraction of the user-defined model, leading to the implementation

systematically. The problem here is how to specify this automatic model transformation easily yet formally.

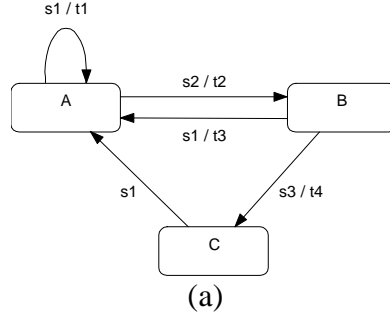
This paper discusses the drawbacks of some approaches in existing environments and proposes a solution. The solution is based on the idea that process of creation may be specified with the Unified Modeling Language (UML) object diagrams, extended with the concepts of conditional, repetitive, parameterized,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

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 purposes 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. Section 6 contains formal definitions of the transformation semantics and some other implementation issues. Several examples that demonstrate the benefits of the approach are given in Section 7. The paper ends with some conclusions.

## 2 MOTIVATION AND PROBLEM STATEMENT

A modeling domain may be defined in terms of the abstractions from that domain, their properties and relationships, their semantics and behavior, and their visual appearance (notation) and behavior in the supporting tool. This kind of specification of a modeling domain is called the *metamodel* of the considered modeling domain. Therefore, metamodeling is the process of defining the metamodel of the 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 by the abstractions of another meta-metamodel, etc. [9]. This paper is focused to 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 tool that generates C++ code for state-machine models [1, 2, 3, 6]. The code generation for state machines should be customizable. The user should be able to change the manner in which the code is generated from the same model, if the user needs different execution models due to performance, concurrency, distribution, or other issues.



```

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 ();

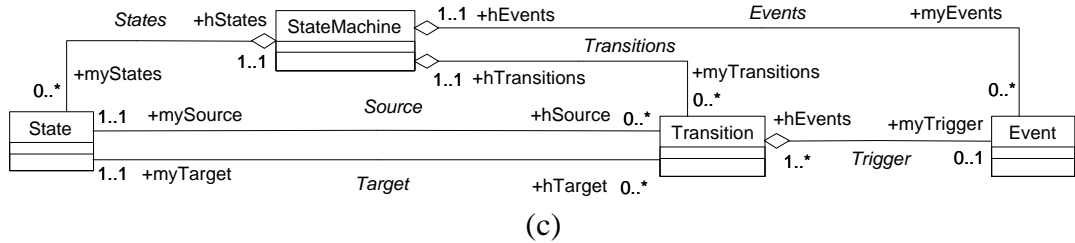
    virtual void entry () { ... }
    virtual void exit () { ... }
};

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) An excerpt from 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, as 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` here. It contains methods that correspond to the events of the state machine. 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. This way, an `FSM` object reacts depending on its current state [5].

The metamodel of the domain (state machines) is shown in Figure 1c. The user may now want to specify the code generation strategy that should be applied for each instance of the abstraction `StateMachine` defined in the model. It may be assumed that the tool provides the operation `StateMachine::generateCode()` that will be invoked for each `StateMachine` instance to generate the code. The problem is to define this operation.

A straightforward approach, most often used in existing modeling tools, is to hard code the output generation scheme in this operation. The operation should navigate through the model instances, read their attribute values, and produce the textual output following the C++ syntax and semantics. For this specification, C++ will be used here, instead of any vendor-specific scripting language that is usually available for these purposes in the existing environments. A small excerpt that generates the very 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 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) Any modification is very difficult to apply because the code is not clear and comprehensible.
- (5) The code is not reusable.
- (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 process may be viewed as a creation of another 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, and other abstractions from the source domain. The target model is the generated source code whose metamodel is implicitly assumed (C++ syntax and semantics). The code of the operation `StateMachine::generateCode()` is actually a specification of the mapping between these domains. Since these domains are at very distant levels of abstraction, their direct mapping by the hard-coded special-purpose generator has all these drawbacks.

The direct 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., 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 that it is conceptually more abstract and closer to the source domain.) This domain may consist of the basic object concepts from UML, such as class, method, attribute, etc., so it can be easily mapped into the target C++ domain. Because 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:

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

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

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

This code excerpt creates instances of the UML concepts `Class` and `Method`, using the programming interface 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 remedies some of the drawbacks of the first approach. First, it eliminates the impedance-mismatching problem between the source and 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 UML 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 side, mappings between arbitrary user-defined higher level domains may not and should not be standardized. The example of the state machines is only one of them. The Case Study section will provide some more.

However, the specification is still tedious and error-prone. Besides, the code may be very complex and difficult to maintain. Since it is actually a specification of the process of creating instances from the target domain, where both 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. Such a specification may be easier to build and maintain, and less error-prone. Consequently, it will cure all 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

Because the process of domain-specific metamodeling may be formalized, the need for tool support to this process has been recognized for a long time [7, 9, 10]. This need was first met in the domain of automatic programming environment generation [8]. By 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 [7, 9]. 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 their customizability:

1. Simple, non-customizable modeling tools, where the domain 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 [14, 17].
3. True metamodeling tools, where the user may specify the metamodels and the mapping [11, 12, 13, 15, 16, 18, 19].

A major commonality, and a weakness, of all existing tools that is of the greatest interest to this work is their output generation facility. All these tools provide a programming interface to 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 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 of 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 the 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 [24, 25] 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, in any order. From these specifications, the C++ code of the output generator is obtained automatically. However, this approach does not focus to the process of creation of the target model; this is completely left to the user-defined C++ code. In the case of the textual output, this approach is very efficient. If the target model is object-oriented,

however, it is not appropriate. In that case, it suffers from the same shortcomings as the approaches described previously.

Another approach may be described as a template-directed output specification [14]. 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]

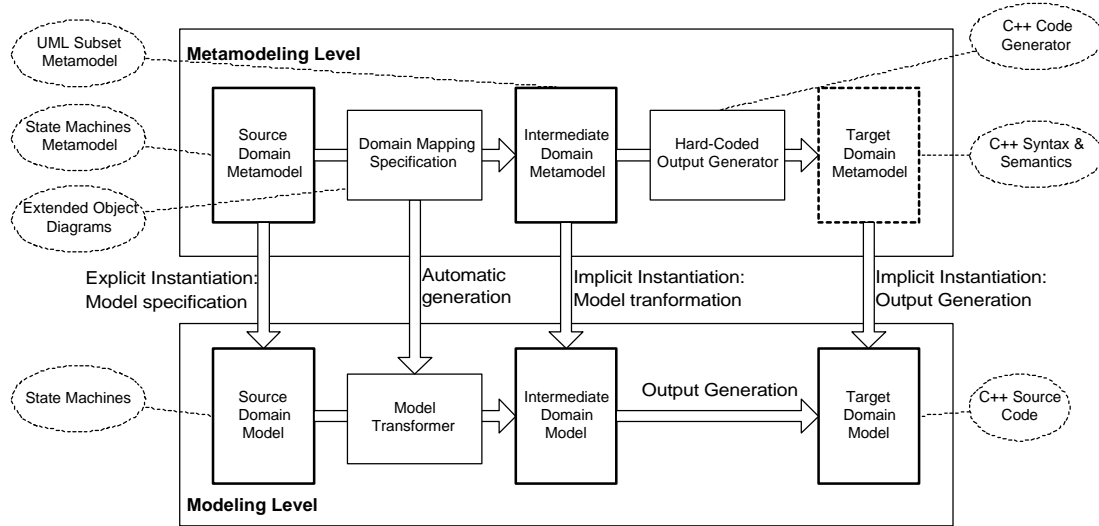
    virtual void entry () {}
    virtual void exit  () {}
    ...
}
```

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. In the case 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 distant.

There are a number of approaches addressing a similar problem using structural transformations of grammar-based models and various rule-based techniques [21, 22, 23, 29]. 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 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 the 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 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 [22]. The approach presented here allows the user to specify the





**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.

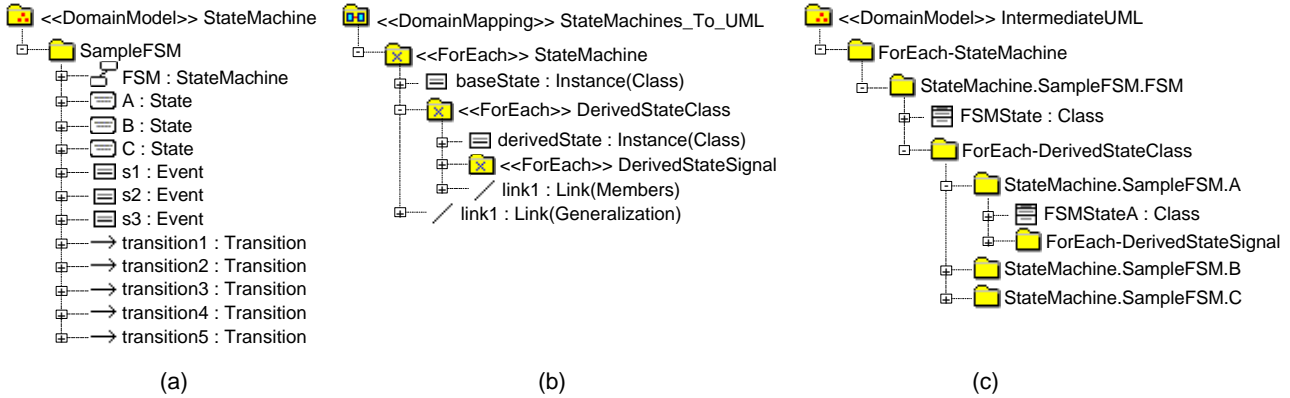
mapping, and the transformer is generated without any intervention of the user. Furthermore, defining a grammar for a certain domain and specifying the mapping between grammars may be a difficult task, because it requires more sophisticated work than defining (in meta-environments) or just understanding (in customizable modeling environments) the object-oriented metamodels. It is evident that some domains may be metamodelled 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 [27], 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 side, the approach proposed in this paper is based on a completely different, object-oriented paradigm. It supports generalization/specialization and polymorphism of domains and mappings, too. 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 tool or approach that is closely related to the one presented in this paper.

## 4 IDEA AND PURPOSE OF DOMAIN MAPPING

The idea of the domain mapping strategy is shown in Figure 2. The principle is in creating 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 that of the UML. The



**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 domains, linked by the 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.

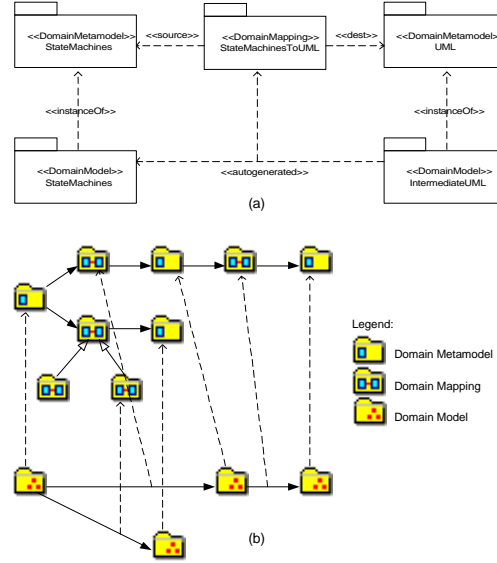
advantage is because each of the transformations is much less complex than the direct transformation, and is thus easier to specify and maintain.

The domain mapping specification should be formal and preferably graphical. It should specify the set of instances of the intermediate metamodel types 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 the 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 domains (Figure 3). They are mutually linked by the instances of associations from their metamodels. 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 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 one from the other in a pipelined fashion, where each generation is performed using a certain domain mapping specification. Note that a domain mapping specification, as a kind of a package, may be specialized by several other specifications, e.g., for variants of the mapping into the same domain. Of course, only one specific mapping may be used for generation of one model.



**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 general applicability of the approach. The models may be generated one from the other in a pipelined fashion, where each generation is performed using a certain domain mapping specification.

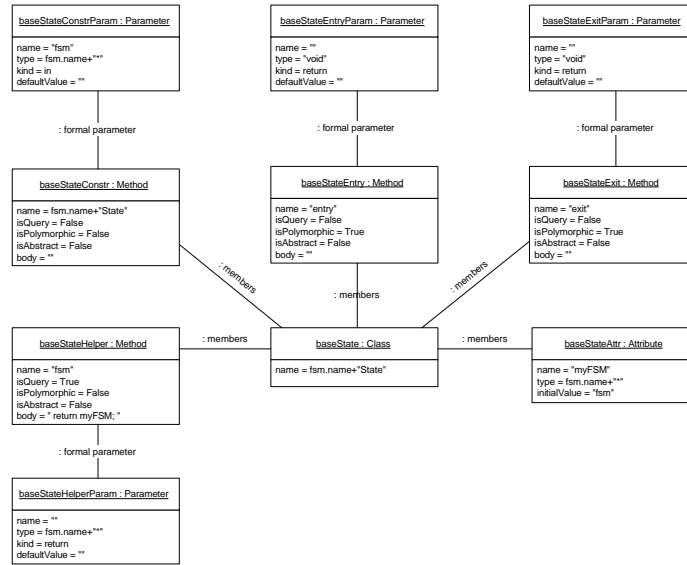
The applicability of the proposed approach is manifold. First, it may be used for output specification in modeling tools of all degrees of customizability. For non-customizable modeling tools, it can be used by developers as a method for designing output generation. A customizable modeling tool, such as a CASE tool, may offer interfaces to 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 a metamodeling tool, where the user can customize both the metamodels and the mapping scheme.

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 7.

## 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



**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.

of the domain mapping specifications, along with the description of the procedure for generating the transformer based on these specifications are given in the next section and in [26].

## Instances, Attributes, and Links

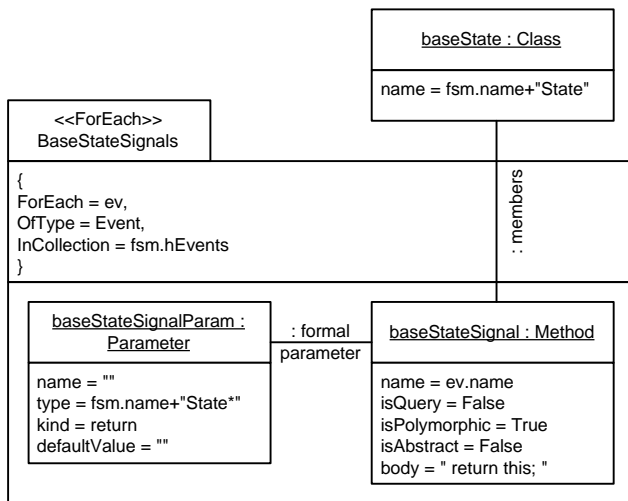
An object diagram that specifies a part of the target model for the demonstrational example is shown in Figure 5. It is assumed that the diagram is defined for one instance from the source model of the type `StateMachine`, referred to by the identifier `fsm`. The diagram specifies the set of instances 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 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 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.

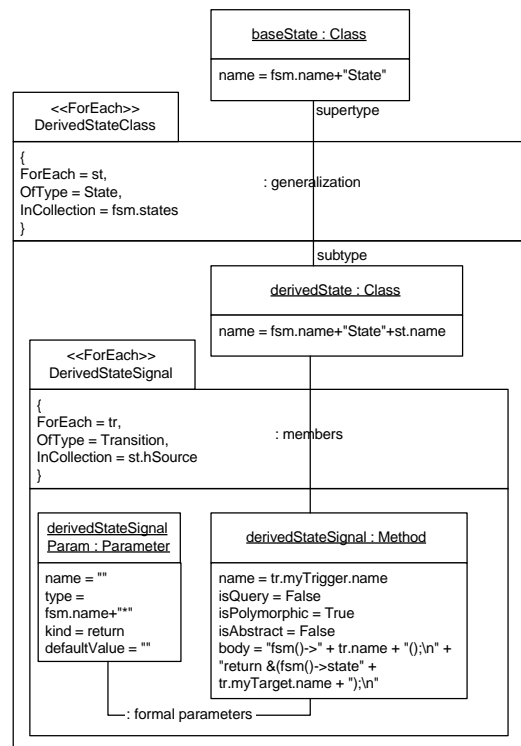
## Repetitions

The standard UML object diagrams are not sufficient for the 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.



**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.

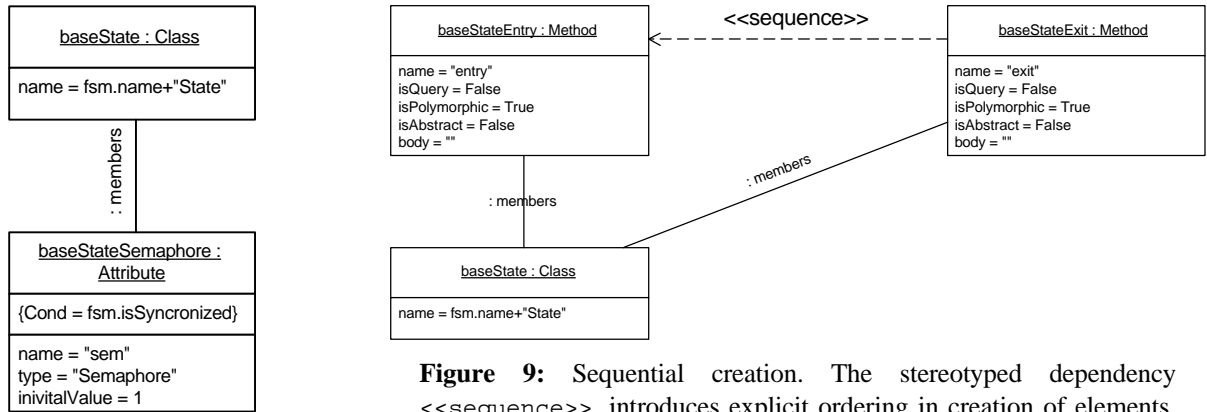
- `OfType`: The type of the current element. The iteration is type-sensitive: only the elements of the specified type from the collection are processed, and the others are ignored (in the case that the elements are polymorphic). The type is from the source metamodel.
- `InCollection`: An expression that evaluates to a collection of the source model elements to iterate.

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 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 the next section and in [26].) For the expressions that specify attribute values of instances or the collection in a `ForEach` package, any 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 metamodel 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. An 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 (specified with the nested 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



**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.

`<<DomainMapping>>` package, where each package owns a set of elements depicted in a set of diagrams. The owned elements may be instances and 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::getAllInstances()`). Consequently, the diagrams shown in figures 5 to 7 belong to the same `ForEach` package that iterates through all instances of `StateMachine`.

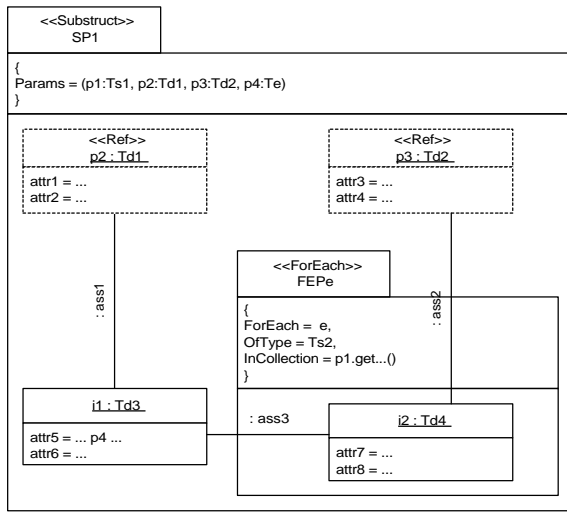
## Conditions

Another needed concept is conditional creation. 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 a source model instance of `StateMachine`, 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).

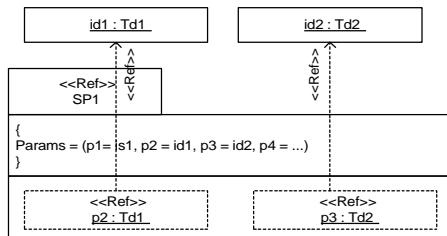
## Sequences

The generated target model is a hierarchy of packages (Figure 3c), where each package is an unordered collection of the elements it owns (instances and other packages) 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 later).

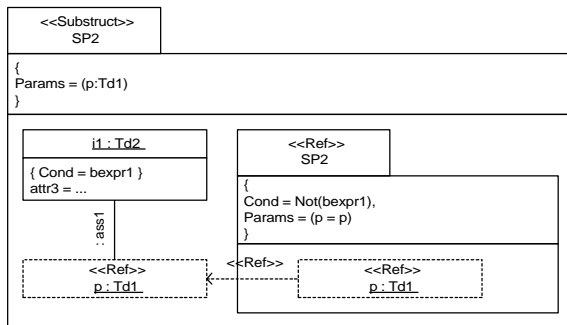
Sometimes, however, an explicit ordering of the elements is needed. This ordering may ensure a proper sequential traversal through the target model elements; for example, if a sequential structure (e.g., text) is to be further generated from that model. If an element  $x$  is to be created after an element  $y$ , 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. The precise consistency rules and semantics of sequence dependencies will be given in the next section.



(a)

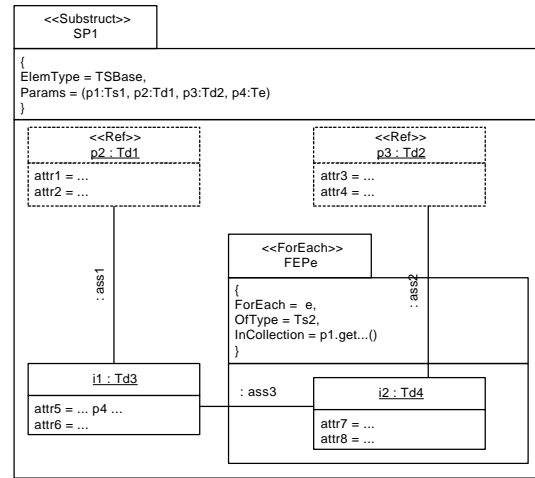


(b)

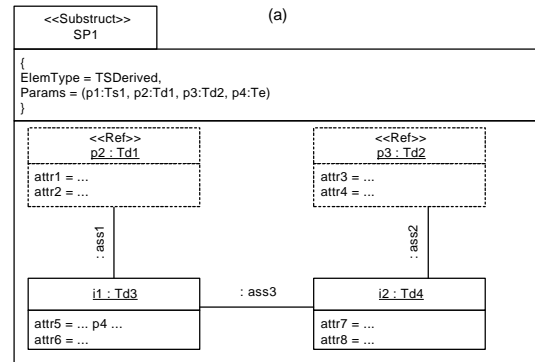


(c)

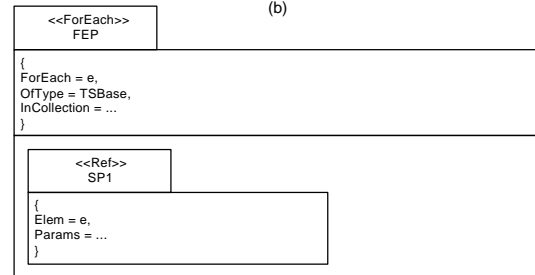
**Figure 10:** Parameterized substructures and recursion. (a) Substructure definition is in a `<<Substruct>>` package. (b) Substructure reference (`<<Ref>>` package) is a request for generating substructure at the place of "invocation." (c) A substructure definition may directly or indirectly refer to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



(a)



(b)



(c)

**Figure 11:** Polymorphism. (a) A substructure definition may be assigned to a source domain type `TSBase` using the `ElemType` tagged value. (b) A 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 9 shows an example where the requirement that the operation `FSMState::entry()` should precede `FSMState::exit()` is defined by a sequence dependency. This way, 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, which is correspondent 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 for 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 generated 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.

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's 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 domain. 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 is generated. If, however, `e` refers to an object of `TSDerived`, the substructure defined in Figure 11b is generated.

This mechanism is completely analogous to the same concept 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 model types, hence the domain's type hierarchy is not "spoiled" with the definitions of transformations.



## 6 FORMAL DEFINITIONS

In this section, the semantics and consistency rules of the domain mapping specifications are defined precisely. Then, the implementation is discussed. Finally, it is presented how manual modifications of the target model may be logged and preserved.

### Basic Definitions

A *domain-mapping element* is a model element [6] that exists in a domain mapping specification (i.e., in a package hierarchy rooted with a package stereotyped with <<DomainMapping>>). In particular, a `ForEach` package is a kind of a domain-mapping package.

Following the UML semantics, each domain-mapping or target model element *me* is owned by exactly one package, except for the root package. When a package *p* owns *me*, it is said also that *me* belongs to *p*; the notation is:  $owns(p, me) \Leftrightarrow belongs(me, p)$ .

The "belongs-to" relationship is extended to the whole package hierarchy into the *is-in* relationship: a (domain-mapping or target) model element *me* is in a package *p*, denoted with  $isIn(me, p)$ , iff there is a sequence of packages  $p_0, p_1, \dots, p_n = p, n \geq 0$ , such that  $belongs(me, p_0)$ , and  $belongs(p_i, p_{i+1}), 0 \leq i < n$ . The *first common enclosing package* of two elements *me1* and *me2* ( $me1 \neq me2$ ) is defined as:  $fcep(me1, me2) = p$  iff  $isIn(me1, p)$  and  $isIn(me2, p)$ , and there is no other package *p'* such that  $isIn(me1, p')$  and  $isIn(me2, p')$  and  $isIn(p', p)$ . Since the package hierarchy is a tree, the  $fcep(me1, me2)$  is unique for each *me1* and *me2*, and  $fcep(me1, me2) = fcep(me2, me1)$ . The first common enclosing `ForEach` package of two domain-mapping elements *dme1* and *dme2*, if exists, is denoted with  $fcefep(dme1, dme2)$ .

We introduce also the notion of the *first enclosing sibling elements* of two model elements *me1* and *me2*. The first enclosing sibling element of *me1* and *me2* in that order, denoted with  $fese(me1, me2)$ , is *me1* itself if  $belongs(me1, fcep(me1, me2))$ , or the package *p1* such that  $isIn(me1, p1)$  and  $belongs(p1, fcep(me1, me2))$ , otherwise;  $fesp(me2, me1)$  is defined symmetrically. It is easy to see that  $fese(me1, me2)$  and  $fese(me2, me1)$  belong to the same package  $fcep(me1, me2)$ , i.e., there are siblings in the package hierarchy.

The "belongs-to" relationship is inherited from the UML metamodel [6] without any changes of semantics. In particular, a domain-mapping link that connects two instances *dmi1* and *dmi2* belongs to the first common enclosing package of *dmi1* and *dmi2*.

There is an implicit tracing relationship between each target model element *tme* and a domain-mapping element *dme* of which *tme* is a consequence; in other words, *tme* is *generated* (or *created*) from *dme* in the process of target model generation.

### Sequence Dependencies

Apart from the explicit user-defined sequence dependencies in domain mapping specifications, some dependencies are implicitly introduced for the purpose of proper model creation. Thus, a link is implicitly sequence-dependent on the instances it connects. Moreover, an explicit sequence dependency is meaningful only for the elements that belong to the same package. This is because the generation algorithm traverses the domain-mapping package hierarchy in the depth-first order. Therefore, if a domain mapping package *dmpn* belongs to the package *dmp*, then all the target model elements generated from the elements contained in the hierarchy starting from *dmpn* are created before the next element of *dmp* is processed. Although a sequence dependency may connect two elements from arbitrary packages, it is satisfied if there is a sequence dependency from the first enclosing sibling elements of these two elements. Formally, an explicit sequence dependency from *me1* to *me2* is satisfied if there is a sequence dependency from  $fese(me1, me2)$  to  $fese(me2, me1)$ .

Therefore, the system should introduce two kinds of implicit sequence dependencies prior to the target model generation:

- 1) from each link to the instances it connects, and then
- 2) for each sequence dependency (explicit or implicitly created in the previous step) from  $me1$  to  $me2$ , a sequence dependency from  $fese(me1, me2)$  to  $fese(me2, me1)$  should be created.

The obtained graph structure of domain mapping elements belonging to one package (as nodes) and sequence dependencies between them (as directed edges) should produce a directed acyclic graph. If there is a cyclic sequence dependency in a package, the domain mapping specification is not correct.

## Model Generation Semantics

If the specification is correct after the implicit sequence dependency generation, i.e., if there are no cyclic sequence dependencies between elements in one package, the system may generate the target model. The generation procedure traverses the domain mapping package hierarchy in the depth-first order. For each package, its elements are processed in the topological order according to their sequence dependencies.

Here we define the precise semantics of the domain mapping specification, in terms of the elements of the target model to be created from each kind of mapping specification element.

### Package

For a domain mapping (simple) package  $dmp$ , named  $pName$ , a target model package  $tmp$ , named  $pName$ , will be created only if the expression specified in the `Cond` tagged value of  $dmp$  evaluates to True. If a domain mapping element  $dme$  belongs to a domain mapping package  $dmp$ , then each target element  $tme$  generated from  $dme$  will belong to the package  $tmp$  generated from  $dmp$ .

### ForEach Package

For a domain mapping ForEach package  $dmp$ , named  $pName$ , a target model package  $tmp$ , named "ForEach-" +  $pName$ , will be created. Furthermore, for each element  $smei$  of the source model, iterated by the iteration specified in  $dmp$ , and with a full path name  $smeiFullName$ , a target model package  $tmpi$ , named  $smeiFullName$  will be created, only if the expression specified in the `Cond` tagged value of  $dmp$  evaluates to True for this  $smei$ ;  $tmpi$  will belong to  $tmp$ . Finally, if a mapping element  $dme$  belongs to  $dmp$ , a corresponding target element  $tmei$  will be generated from  $dme$  for each  $smei$ , and it will belong to  $tmpi$ , only if  $tmpi$  has been created.

### Instance

For a domain mapping instance  $dmi$ , referring to the target model type  $T$  and named  $dmiName$ , a target model instance  $tmi$  of type  $T$ , named  $dmiName$ , will be created, only if the expression specified in the `Cond` tagged value of  $dmi$  evaluates to True. The attributes of  $tmi$  will be set to the values specified in  $dmi$ .

### Link

Let  $dml$  be a domain mapping link that connects two domain mapping instances  $dmi1$  and  $dmi2$ , and refers to the target domain association  $A$ . Let  $dmp$  be the first common enclosing package of  $dmi1$  and  $dmi2$  ( $dml$  belongs to  $dmp$ ). Let  $tmp$  be a certain target model package generated from  $dmp$ . For each target model instance  $tmi1$  generated from  $dmi1$  in  $tmp$ , and for each target model instance  $tmi2$  generated from  $dmi2$  in  $tmp$ , a target model link  $tml$  of  $A$  will be created, connecting  $tmi1$  and  $tmi2$ , only if the expression specified in the `Cond` tagged value of  $dml$  evaluates to True.

## Code Generation

From the described specifications, the source or scripting code that is used to create the target model may be generated automatically. For our example, this code is shown in the Appendix. For each

domain mapping package, the elements it owns are visited and the code is generated as follows. Formatting issues, such as indentation of the generated code or capitalization of identifiers, are skipped. The variable `parentPckName` is a global variable that stores the name of the package that is currently used as the owner of the newly created target model elements. For creation of links, the details concerning roles of the instances that are linked (if they are instances of the same type) are omitted.

#### **DMPackage::generateCode()**

```
output: "if (" + cond + ") {";

String pckName = "pck" + name;
output: "Package& " + pckName + " = Package::create(" + parentPckName + ");";
output: pckName + ".dmOrgName = \"" + name + "\"";";
output: pckName + ".name = \"" + name + "\"";";

String parentPckNameSaved = parentPckName;
parentPckName = pckName;

Topologically sort the owned elements, regarding to the sequence dependences;
Traversing the owned elements in the topological order, for each element e do
    e.generateCode();

output: "}";

parentPckName = parentPckNameSaved;
```

#### **DMForEachPackage::generateCode()**

```
String pckName = "pckForEach" + name;
output: "Package& " + pckName + " = Package::create(" + parentPckName + ");";
output: pckName + ".dmOrgName = \"" + name + "\"";";
output: pckName + ".name = \"ForEach-" + name + "\"";";

output: "ForEach(" + forEach + "," + ofType + "," + inCollection + ")";

output: "if (" + cond + ") {";

String pckNestedName = "pck" + name;
output: "Package& " + pckNestedName + " = Package::create(" + pckName + ");";
output: pckNestedName + ".dmOrgName = \"" + name + "\"";";
output: pckNestedName + ".name = forEach + \"getFullParthName()\"";";

String parentPckNameSaved = parentPckName;
parentPckName = pckNestedName;

Topologically sort the owned elements, regarding to the sequence dependences;
Traversing the owned elements in the topological order, for each element e do
    e.generateCode();

output: "}";

output: "EndForEach(" + forEach + ")";

parentPckName = parentPckNameSaved;
```

#### **DMInstance::generateCode()**

```
output: "if (" + cond + ") {";
```

```

String tpName = type.name;
output: tpName + "&" + name + " = " + tpName + "::create(" + parentPckName + ");";
output: name + ".dmOrgName = \"\" + name + \"\"";

ForEach(attr,AttributeSetting,getAttributeSettings())
    output: name + "." + attr.name + "=" + attr.value + ";";
EndForEach(attr)

output: "}";

```

```

// Helper function:
generateInstanceAccess(Package p, String pckName)
if (p.isForEach()) {
    // ForEach package
    String pckForEachName = "pckForEach" + p.name;
    String pckNewName = "pck" + p.name;
    output: "Package& " + pckForEachName + " = *" + pckName +
        ".getPackage(\"ForEach-\" + p.name + \"\");";
    output: "ForEach(" + pckNewName + ",Package,\" +
        pckForEachName + ".getAllPackages())";
    pckName = pckNewName;
} else {
    // Simple package
    String pckNewName = "pck" + p.name;
    output: "Package& " + pckNewName + " = *" + pckName + ".getPackage(\"\" +
        pckNewName + \"\");";
    pckName = pckNewName;
}

DMLink:generateCode()
Let dmil = sideA, dmi2 = sideB;
Let tpName1 = dmil.type.name, tpName2 = dmi2.type.name;
Let dmp = owningPackage;
Let pList1 = listOfPackages(dmp,dmil.owningPackage); // The list excludes dmp
Let pList2 = listOfPackages(dmp,dmi2.owningPackage); // The list excludes dmp
output: "{";

// Access to the first side:
String pckName1 = dmp.name;
ForEach (p1,Package,pList1)
    generateInstanceAccess(p1,pckName1);
EndForEach(p1)

String ptrName1 = "ptr" + dmil.name;
output: tpName1 + "*" + ptrName1 + " = (" + tpName1 + "*)(\" +
    pckName1 + ".getElement(\"\" + tpName1 + "\",\"\" + dmil.name + "\");";

// Access to the second side:
String pckName2 = dmp.name;
ForEach (p2,Package,pList2)
    generateInstanceAccess(p2,pckName2);
EndForEach(p2)

String ptrName2 = "ptr" + dmi2.name;
output: tpName2 + "*" + ptrName2 + " = (" + tpName2 + "*)(\" +

```

```

        pckName2 + ".getElement(\"" + tpName2 + "\",\"" + dmi2.name + "\");";

// Link creation:
output: "if ( " + ptrName1 + " && " + ptrName2 + " ) {";
output: tpName1 + "& " + dmi1.name + " = *" + ptrName1 + ";";
output: tpName2 + "& " + dmi2.name + " = *" + ptrName2 + ";";
output: "if ( " + cond + " ) {";
output: "MMLink& " + name + " = MMLink::create(" + type + "::Instance()," +
        dmi1.name + "," + dmi2.name + ");";
output: name + ".dmOrgName = \"" + name + "\";";
output: "}";
output: "}";

Generate "EndForEach" clauses for ForEach packages in pList2 (reverse order);
Generate "EndForEach" clauses for ForEach packages in pList1 (reverse order);
output: "}";

```

## Target Model Consistency

In UML, a model element of one kind must have a unique name in the context of the package to which it belongs. The same holds for domain mapping elements. Moreover, the same requirement for the generated models is easily met by the following rules. Each generated package or instance is named as the corresponding domain mapping element from which it originates. For the packages generated for a certain iteration of a `ForEach` package, a unique name is constructed from the full name (including the path) of the source model element that is current for that iteration. Since the full path name of a source model element of certain kind is unique in the whole source model, the generated name is also unique in the context of the enclosing package. In order to differentiate a generated name from full path names in UML which are built using `::` as separators, we replace `::` with `.'.`  Other domain-specific consistency rules are defined for each domain, either by the user or by the tool.

## Manual Modifications

When the target model is generated, the system allows the user to perform modifications on it, and logs all these modifications. The history of all modifications is logged from the very first generation of the model cumulatively. When the user requires model regeneration, the system will first generate a new target model from the scratch, using the possibly changed source model and the domain mapping specification. Then, the system will apply all logged modifications, in chronological order. When the precondition of a logged modification is not satisfied, the modification is simply ignored (skipped). This may happen, for example, if a modification refers to a model element that is no longer generated due to the changed source model or mapping specification.

The logging of modifications is based on the fact that each target model package or instance is uniquely identified by its full path name. A link is identified by its type (the association of which it is an instance) and the instances it connects; if a link is an instance of an association class, it is identified as an instance of that class. (Since multiple links of the same type connecting the same instances are rarely needed or even seamless, unless they are instances of an association class, we neglect this case.) Each modification is treated as an operation on a certain model element, with some arguments. In the following tables, the operations that are allowed and logged are listed, along with their arguments and preconditions; *ID* represents the unique identification of a package or instance by its full path name.

### Operations with Packages

Operation	Meaning	Arguments	Preconditions
<i>create</i>	Create a package	<i>parentPckID</i> : ID of the package to which the created package will	The path defined with <i>parentPckID</i> is correct.

		belong <i>pckName</i> : name of the created package	<i>pckName</i> is unique in <i>parentPckID</i> .
<i>delete</i>	Delete a package	<i>packageID</i> : ID of the deleted package	The path defined with <i>packageID</i> is correct.

#### Operations with Instances

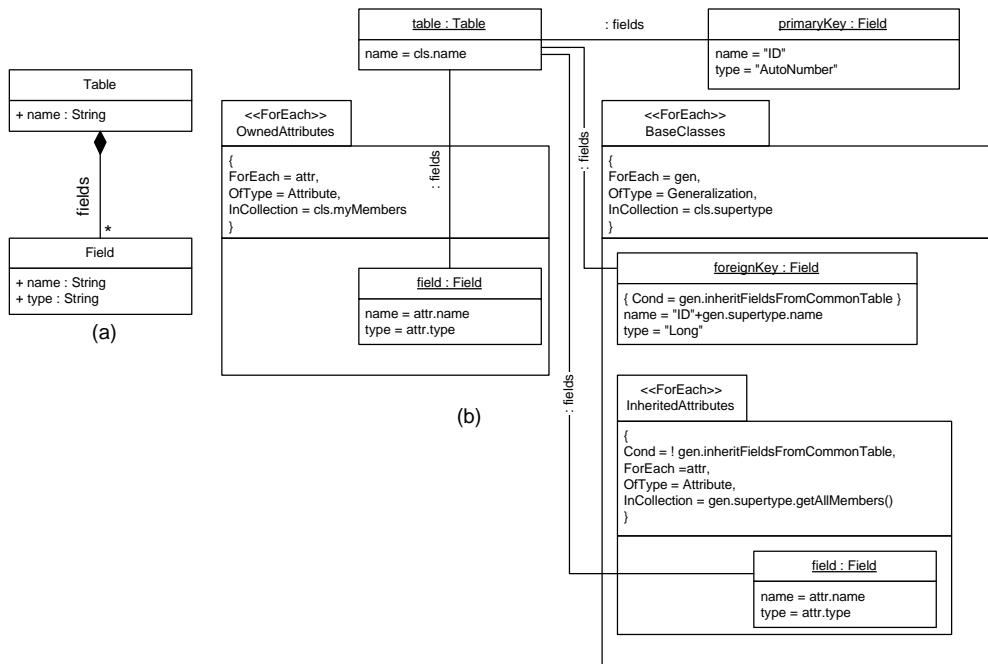
Operation	Meaning	Arguments	Preconditions
<i>create</i>	Create an instance	<i>parentPckID</i> : ID of the package to which the created instance will belong <i>instName</i> : name of the created instance <i>type</i> : reference to the type of the instance	The path defined with <i>parentPckID</i> is correct. <i>instName</i> is unique in <i>parentPckID</i> for the given type. The reference to the type is correct.
<i>modify</i>	Modify an attribute value of an instance	<i>instID</i> : ID of the modified instance <i>attribute</i> : reference to the modified attribute <i>newVal</i> : new value of the attribute	The path defined with <i>instID</i> is correct. The reference to the attribute is correct. The new value of the attribute is valid.
<i>delete</i>	Delete an instance	<i>instID</i> : ID of the deleted instance	The path defined with <i>instID</i> is correct.

#### Operations with Links

Operation	Meaning	Arguments	Preconditions
<i>create</i>	Create a link	<i>inst1ID</i> : ID of the first instance <i>inst2ID</i> : ID of the second instance <i>assoc</i> : reference to the association of which the link is an instance (the type of the instance)	The paths defined with <i>inst1ID</i> and <i>inst2ID</i> are correct. The reference to the association is correct.
<i>modify</i>	Modify an attribute value of a link (if a link is an instance of an association class)	<i>instID</i> : ID of the instance that represents the link (the instance of the association class) <i>attribute</i> : reference to the modified attribute <i>newVal</i> : new value of the attribute	The path defined with <i>instID</i> is correct. The reference to the attribute is correct. The new value of the attribute is valid.
<i>delete</i>	Delete a link	<i>inst1ID</i> : ID of the first instance <i>inst2ID</i> : ID of the second instance <i>assoc</i> : reference to the association of which the link is an instance (the type of the instance)	The paths defined with <i>inst1ID</i> and <i>inst2ID</i> are correct. The reference to the association is correct.

## 7 EXAMPLES AND CASE STUDY

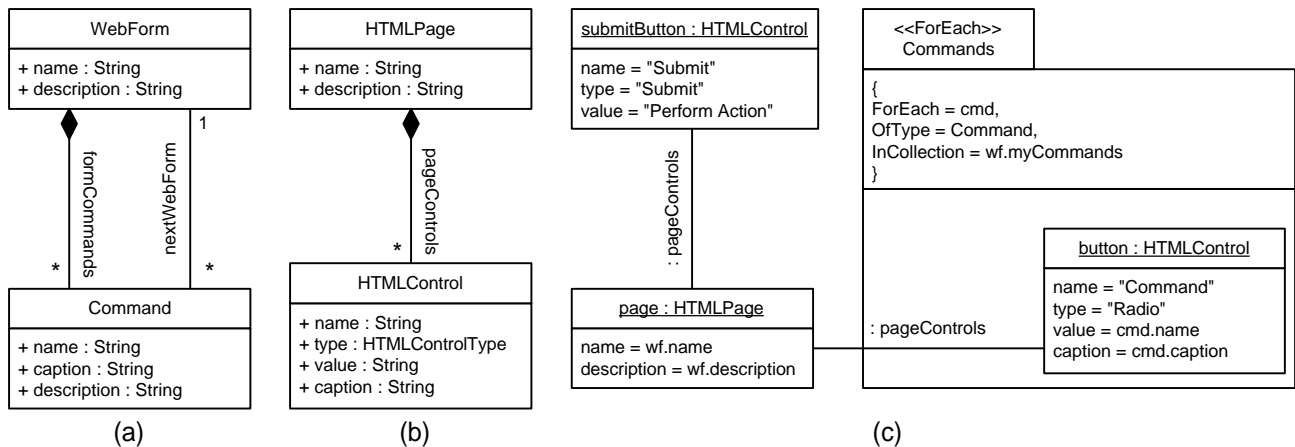
This section presents three simple examples that illustrate the applicability of the approach, and short descriptions of several more complex systems that have been developed using the proposed approach. The experiences from these projects are briefly reported and discussed.



**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 classifier.

### 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.) 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 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 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 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 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 mapping specification is shown in Figure 12b.



**Figure 13:** 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.

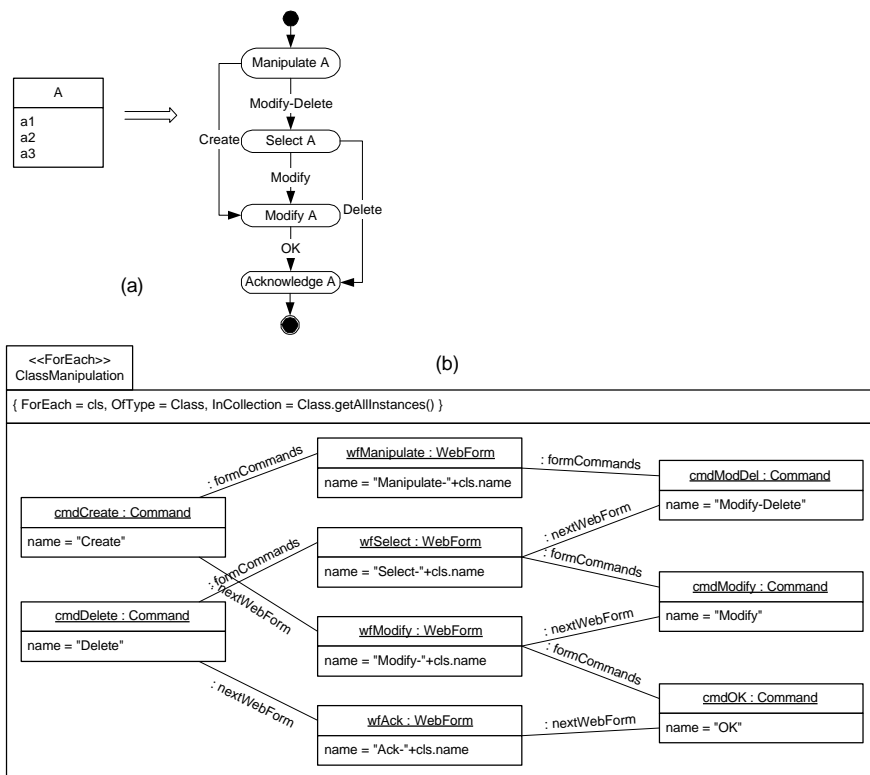
## Example 2: Web Application Development

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 with database-centric Web application development. The author has designed a method and infrastructure for rapid Web application development. A very small part of the idea is presented here, 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.

## Example 3: Automatic Implementation of Structural Use-Cases

The third example illustrates 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 class model with abstractions (classes) from that domain and relationships between them (most notably, associations and generalizations). On the other side, the behavioral aspect of the application 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 that simply create, modify (attribute values of), or delete instances of the abstractions and their links (as instances of associations). Often very few use-cases may be qualified as "behavioral," meaning that they navigate through the application's structure (instances and links) and provide the functionality that is specific for the application. Besides, structural use-cases may be designed using various patterns that are based on the semantics of the relationships between abstractions. Consequently, these use-cases may be implemented automatically.





**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.

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, a domain mapping specification may be created for the mapping from the UML domain, i.e. its structural part (class model), into the domain of web forms and navigation defined in the previous example (Figure 14b). On the other side, the database definition can be obtained from the class model, too, as in the first example. Thus, by using model pipelining, the implementation of the structural use-cases can be obtained from the very abstract structural definition of the 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 to a complete desired implementation. However, the strategy is still open enough so that the user may 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. This method seems to be powerful, 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. All of them have been developed by 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). This domain is referred to as the OOPL (OO programming language) domain here. 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. The implementation of the code generator took about three days, including testing, although the transformer from the source domain into OOPL was generated manually from the mapping specifications (this component of the 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 structure and behavior of circuits, 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.

This case study has addressed the issues and proved the properties of the approach as follows:

- (a) **Applicability.** The examples have been 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. This has proved the broad applicability of the approach.
- (b) **Acceptability and learning effort.** The projects have been accomplished by three different students without professional experience. They have all perceived the technique easily, without any help of the author, just by reading the technical reports and studying the available examples. They have all reported that they needed a short time to get ready to use the method, just as long as it took them to read the available materials.
- (c) **Production time.** To avoid the possibility of the author's subjectivity, the 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.

(d) Maintainability. In all three projects, the students had tasks to make two or more different versions of the output generation, by changing the starting mapping. They have reported no difficulties in accomplishing these tasks, because they have found the 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 supported by the proposed approach. The diagrams may show only parts of the complete mapping, i.e. only some elements that are relevant to the particular context of a 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 the same as the advantage of the visual software modeling approaches (such as UML) over the textual programming languages, when the maintainability is concerned.

(e) Reusability. The OOPL domain, its metamodel, and its C++ code generator have been reused in all three projects. This has shortened the development time at least twice. 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 Java.

(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. The experience is 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. Second, an example of a model from this domain and the desired output for that model should be constructed. The example 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 is the easiest task in the whole process, if the example and the metamodel are correctly constructed.

The case study has shown some weaknesses of the approach. 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 is not supported 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 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.

## 8 CONCLUSIONS

This work has studied the problem of 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 domains are at distant levels of abstraction, the mapping is difficult to specify, maintain, and reuse. 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 the instances of the target model that should be created by the transformer. The diagram is extended by the concepts of conditional, repetitive, sequential, parameterized, and polymorphic element creation. These concepts are implemented using the standard UML extensibility mechanisms.

Several examples from different engineering domains have been provided. All the examples have proved the expected benefits of the approach. The specifications are clear, concise, and easy to maintain. The domain mapping strategy leads to a better reuse of domain metamodels and to shorter production time.

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, and to D. Bojic from the University of Belgrade, J.-P. Tolvanen from MetaCase Consulting for their invaluable comments on the paper.

### **REFERENCES**

#### **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

#### **Metamodeling**

- [7] Artsy, S., "Meta-modeling the OO Methods, Tools, and Interoperability Facilities," *OOPSLA'95 Workshop in Metamodeling in OO*, October 1995
- [8] 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

- [9] MetaModel.com, *Metamodeling Glossary*, <http://www.metamodel.com>
- [10] 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**

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

### **Model Transformations**

- [20] Aitken, W., Dickens, B., Kwiatkowski, P., De Moor, O., Richter, D., Simonyi, C., "Transformation in Intentional Programming," <http://research.microsoft.com/ip>
- [21] 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
- [22] 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
- [23] Habermann, A. N., Notkin, D. S., "Gandalf: Software Development Environments," *IEEE Trans. Software Engineering*, Vol. 12, No. 12, December 1986, pp. 1117-1127
- [24] Karsai, G., Sztipanovits, J., Franke, H. "Towards Specification of Program Synthesis in Model-Integrated Computing," *Proc. IEEE ECBS'98 Conf.*, pp. 226-233, 1998
- [25] Karsai, G. "Structured Specification of Model Interpreters," *Proc. IEEE ECBS'99 Conf.*, pp. 84-91, March 1999
- [26] Milicev, D., *Automatic Model Transformations in Modeling Environments* (in Serbian), Ph.D. thesis, University of Belgrade, Faculty of Electrical Engineering, 2000
- [27] 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
- [28] Papazoglou, M. P., Russell, N., "A Semantic Meta-Modelling Approach to Schema Transformation," *Proc. ACM Conf. Information and Knowledge Management (CIKM'95)*, 1995

- [29] 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/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)
}

```