# Code Generation and Execution Framework for UML 2.0 Classes and State Machines

Romuald Pilitowski, Anna Derezińska
Institute of Computer Science, Warsaw University of Technology
15/19 Nowowiejska Street
Warsaw, 00-665 Poland

*Abstract* – **The paper presents Framework for eXecutable UML (FXU). FXU transforms UML models into programming code and supports execution of the resulting application according to the behavioral model. The code generation and execution is based on UML classes and their state machines. The FXU framework takes into account all concepts of state machines defined by the UML 2.0 specification. Ambiguities of UML state machine interpretation had to been resolved within the framework in order to obtain an executable application. During the runtime execution separate state machines and orthogonal regions are run as parallel threads. All kinds of events, states, pseudostates and activities are processed, as well. The framework was implemented and tested for C# code. The tool supports model-driven development of high quality applications.**

## I. Introduction

Unified Modeling Language (UML) is a widely accepted, graphical language for specification, analysis and design of object-oriented computer systems [1]. Emergence of UML has triggered development of many tools and methodologies that improve quality of computer systems [2-9]. One of such methodologies is Executable UML (xUML) [10] that is intended to enable UML models to be translated into executable code. As such, an Executable UML model can be treated as Platform Independent Model (PIM) – the concept defined within the Model Driven Architecture (MDA). MDA is a recent initiative of Object Management Group consortium (OMG) which contributed to the development of the UML standard [2]. MDA is a model-centric approach to the software engineering based on models as major artifacts of the development process. However, none of the MDA tools supports complete and precise transformation of UML 2.0 state machines into executable C# code.

Our solution, called Framework for eXecutable UML (FXU) developed at the Warsaw University of Technology is an example of xUML tool. It is a code generator which enables transformation of UML 2.0 class diagrams and state machines into code in C# programming language, which can be further compiled and executed. A distinguishing feature of FXU is its ability to handle every single element of state machines as defined by the OMG standard – the UML 2.0 specification [1].

FXU consists of two components – FXU Generator and FXU Runtime Library. The FXU Generator transforms an UML 2.0 model into the corresponding code in the programming language. It takes into account information from class diagrams and state machines. The FXU Runtime Library is a library that implements entire logic of state machines as described by the UML 2.0 specification, including states, events, transitions etc. Both generated code and the FXU Runtime Library are required in order to enable execution of state machines.

The paper presents core algorithms that were implemented by the FXU Runtime Library. It supports execution of concurrent state machines which can specify behavior of many different objects. Regions of orthogonal states are executed concurrently as well. Transitions across vertices are triggered by events. The selection of transitions to be fired is based on their priorities. Queues of events are handled for each active state machine.

During development of FXU we faced problems of inconsistency of UML 2.0 specification concerning state machines. Some of the issues are semantic variation points. Semantic variation points are aspects that were intentionally not determined in order to leave its interpretation to a user. The order in which events are removed from the event pool is an example of the semantic variation point. However, there are also aspects such as the order of execution of multiple transitions, that were clearly overlooked in the specification. There are about ten inconsistency issues that cannot be resolved on the basis of the specification. All of them had to be identified and interpreted before the implementation of FXU. A detailed description exceeds however the scope of this paper [11].

Section II describes principles of code generation and the architecture of the FXU Generator. The FXU Runtime Library and its essential algorithms are presented in Section III. Related work is discussed in Section IV and final remarks end the paper.



*T. Sobh (ed.), Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, 421–427.

## II. Code Generation From UML Models

Code generators from UML models transform an abstract model of a software system into its implementation. Output of the most simple code generators from UML models [4,5,7] is only a stub of a system, e.g. only class declarations that have to be implemented. Approach of MDA is different. It assumes generation of the most components of the target system without much human intervention afterwards. FXU enables generation of fully functional applications as long as their structure and behavior can be represented using UML 2.0 class diagrams and state machines.

The distinguishing feature of FXU is the ability to generate and execute every single aspect of state machines described in the UML 2.0 specification. They are all listed in Tab. 1.

TABLE 1
Elements of State Machines Supported by FXU

| |
|---|
| **States**: simple states, composite (including orthogonal) states, entry-, do-, exit activities, submachine states. |
| **Pseudostates**: initial pseudostate, deep and shallow history, join, fork, junction, choice, entry and exit point, terminate. |
| **Transitions**: external, local, internal transitions, guards, actions, priorities of transitions. |
| **Events**: call events, time events, completion events, change events, signals, deferred evens, priorities of dispatching. |

### A. Code generation from class diagrams.

Generation from class diagrams to an object-oriented language is straight-forward. The most of the concepts from class diagrams have their counterparts in object-oriented languages. The FXU Generator and most other similar generators transform classes, their operations and attributes as well as interfaces into corresponding instructions in programming languages. Some generators including the FXU Generator support also generation of associations. The FXU Generator provides also with a template of generated code that can be edited in order to adjust outcome of generation to individual requirements.

### B. Code generation from state machines.

Major goal of our solution is generation and execution of state machines. However, classes play an important role in both generation and execution of state machines because they create context for them. Every class can possess one or more state machines that describe behavior of its instances. Elements of the UML 2.0 model which are accessible within context of the class are also accessible within its state machines. Therefore, generation of state machines involves generation of corresponding classes.

Code generation from state machines is by far more difficult to accomplish than generation from class diagrams. The main reason for this is the lack of direct and precise mapping between concepts borrowed from state diagrams and general-purpose languages. For example, there is no counterpart of a transition or a state in the most general-purpose languages. Therefore every single concept defined in the UML 2.0 specification as a state, event, transition etc. was implemented and wrapped by a class of the FXU Runtime Library. The generated code consists mainly of instances of the FXU Library classes.

### C. Code template

Figure 1 illustrates how FXU merges an UML 2.0 class and its state machines into one file with source code.

```
1  $visibility class $class_name {
2
3  $visibility [static] [readonly] $type $attribute_name;
4
5  StateMachine sm =  new
6        StateMachine($state_machine_name);
7
8  $visibility [static] $return_type $method_name
9  ($type $parameter_name1, $type $parameter_name2) {
10             $method_implementation
11 }
12 public void InitFxu(){
13       Create regions
14       Create states
15             Add the states to the nesting regions
16             Add activities to the states
17             Add events deferred by the states
18       Create pseudostate
19             Add the pseudostates to the regions
20       Create transitions
21             Add triggers of the transitions
22             Add guards to the transitions
23             Add actions to the transitions
24 }
25 public void StartFxu(){
26       State::Enter ( sm );
27 }
28 }
```

Fig. 1. Template of generated source files.

Every class from the input model is generated as a separate file. Each class begins with declaration of its attributes (line 3). Keywords *static* and *readonly* are parameters of attributes and refer to UML properties of the same name. State machines of the class are treated as attributes of type *StateMachine* (rows 5-6). *StateMachine* and other types as *State*, *Transition*, *InitialPseudostate* are classes defined by the FXU Library.

Next, operations are declared (lines 8-9). If an operation is not abstract, its body is generated as well. Provided the input model contains an implementation, it is also included in the generated code. In case an operation triggers a call event, instructions broadcasting the appropriate event are added to