Realization of UML Class and State Machine Models in the C# Code Generation and Execution Framework

Anna Derezińska and Romuald Pilitowski
Institute of Computer Science, Warsaw University of Technology
Nowowiejska 15/19, 00-665 Warsaw, Poland
E-mail: A.Derezinska@ii.pw.eu.pl, http://ii.pw.edu.pl/~adr

Keywords: state machine, UML, code generation, MDE, C#

Received: April 3, 2009

Many benefits are expected due to usage of code generation tools. A reliable application should be created effectively based on complex structural and behavioral models. Model driven approach for program development is realized in Framework for eXecutable UML (FXU). The tool transforms UML models into C# source code and supports execution of the application reflecting the behavioral model. The framework consists of two components: code generator and run time library. The generated and executed code corresponds to structural model specified in class diagrams and behavioral model described by state machines of these classes. All single concepts of behavioral state machines included in the UML 2.x specification are taken into account, including all kinds of events, states, pseudostates, submachines etc. The paper discusses the transformation of UML state machines into C# language. It presents checking the correctness of classes and state machines decided in the framework in order to run a model-related and high quality C# application. The solution was tested on set of UML models.

Povzetek: Predstavljeno je orodje za avtomatsko generacijo kode iz UML v C#.

1 Introduction

Model Driven Engineering (MDE) represents software development approaches in which creation and manipulation of models should result in building of an executable system [1]. There are two general directions towards model execution. The first one is aimed at the direct model execution in a virtual machine of a modelling notation. This idea resulted in the development of the Foundation Subset for Executable UML Models specification (FUML [2]). It defines a basic virtual machine for UML.

The second trend assumes transformation of a model into possibly more refined models, and finally into a target code. The output code is usually expressed in a general purpose language. It can be further modified, completed and used for building a final application, with commonly used development environments. In this paper we discuss problems concerning building an executable C# application from UML classes and state machines.

Industrial product development puts a lot of attention on fast implementation of needed functionalities. Model-driven approach to program development offers a promising solution to these problems. Complex behavioral models can be designed and verified at early stages of the whole product creation cycle and automatically transformed into the code preserving the desired behavior.

State machines, also in the form of statecharts incorporated in the UML notation [3], are a widely used concept for specification of concurrent reactive systems. Proposal for execution of behavioral UML models suffers from the problem that no generally accepted formal semantics of UML models is available. Therefore, validation of UML transformation and model behavior depicted in the resulting code is difficult. Rather than completely formalizing UML models, we try to deal with selected aspects of the models.

Inconsistency and incompleteness allowed by UML can be a source of problems in software development. A basic type of design faults is concerned with the well-formedness of diagrams [3]. Typically, completeness of a design requires that model elements are specified with their features and usage of one element can imply a usage of another, directly related model element. In the current modeling CASE tools some completeness conditions can be assured automatically (e.g., default names of roles in associations, attributes, operations etc.). Incompleteness of models can be strongly related to their inconsistency, because it is often impossible to conclude whether diagrams are inconsistent or incomplete [4]. Therefore, within this paper we will refer to model defects as to correctness issues.

The Framework for eXecutable UML (FXU) offers a foundation for applying MDA ideas in automation of software design and verification [5]. The FXU framework was the first solution that supported generation and execution of all elements of behavioral state machine UML 2.0 using C# language. In order to build an application reflecting the modeled classes and their behaviors specified by state machines, we resolved necessary semantic variation points [6]. Semantic variation points are aspects that were intentionally not determined in the specification [3] and its interpretation is left for a user.

It was also necessary to provide some correctness checking of a model. This paper is devoted to these issues. To present potential problems we selected one target application environment, i.e., creation of application in C# language. The verification of an input UML model is based on a set of hard coded rules. Some of the rules are general and can be applied for any object-oriented language, as they originate directly from the UML specification [3]. Other rules depend on the programming environment because they take also into account the features of the target language - C#. The verification is performed during transformation of class and state machine models into the corresponding code; it is so-called static verification. Other set of rules is used during execution of the code corresponding to given state machines; so-called dynamic verification. For all correctness rules the appropriate reaction on the detected flaws were specified.

One of the contributions of the paper is exploitation of C# constructs to create concise representation of state machines, including also all complex concepts of UML behavioral state machines. Additionally, the correctness rules for UML models are presented, aimed at executing class and state machine models as C# applications.

In the next section we discuss the related works. Next, the FXU framework, especially solutions used for state machines realization, will be presented. In Sec. 4 we introduce correctness issues identified in the transformation process and during execution of state machines. Remarks about experiments performed and the conclusions finish the paper.

2 Related work

2.1 Code generation and execution support

There are different policies dealing with UML models to be transformed. Transformation of a model into the corresponding target code can be realized for any general UML model. The main restrictions concern model correctness but not the direct correspondence to any target notation. Many code generators incorporated in modeling tools, and also the FXU framework, support this approach. It helps dealing with not complete and not specialized models, which often encounter in software development and evolution praxis.

An opposite strategy is the refinement of a model towards the concepts of the target notation, which can be a programming language. This refinement can be completed, for example, using a set of stereotypes included in a UML profile dedicated for the considered notation. This approach is represented by IBM Rational modeling extension for Microsoft .NET [7]. However, it should be noted that the tool supports only selected C# concepts and the relations between refined model elements are not validated. Moreover, state machines are not taken into account in code generation.

Many modeling tools have a facility of transforming models into code in different programming languages. However, the most of them consider only class models. We compared functionality of twelve tools that could also generate code from behavioral state machines. Only few of them took into account more complex features of state machines, like choice pseudostates, deep and shallow history pseudostates, deferred events or internal transitions. The most complete support for state machines UML 2.0 is implemented in the Rhapsody tool [8] of IBM Telelogic (formerly I-Logix). However it does not consider C# language.

There exist different approaches to building an executable application basing on behavioral UML models. In the first one, the code created as the target of model transformation includes the mapping of the state machine structure as well as the logic supporting model execution [9, 10]. Therefore, large number of code must be generated even for simple state diagrams. All semantic issues have to be resolved directly in the generated code.

Another solution is usage of a kind of a run-time environment. It assumes an existence of a library or virtual machine that provides an engine for state machine execution [8, 11, 12]. The generated code depicts only the structure of the input state machine. The code is more compact and easier to understand and to modify. The FXU framework is based on the second solution, applying a run-time library.

2.2 State machine semantics

A huge amount of research efforts is devoted to formalization of UML models, specification of their semantics and verification methods [13]-[17]. However they are usually not resolving the practical problems which are faced while building an executable code, because of many variation semantic points of the UML specification.

An attempt for incorporation of different variation points into one solution is presented in [18]. The authors intend to build models that specify different variants and combine them with the statechart metamodel. Different policies should be implemented for these variants.

The semantics defined in the FUML specification [2] are generally a precise definition of a subset of the UML semantics given in the UML 2.2 Superstructure Specification. The FUML specification is limited to the selected UML elements considered as mostly used. Therefore it does not deal, for example, with all features of state machines.

2.3 Model correctness

Our work relates also to the field of correctness of UML models. The consistency problems in UML designs were extensively studied in many papers. It could be mentioned workshops co-located to the Models (former UML) series of conferences, and other works [4, 19-22].

Checking of models is important in Model Driven Architecture (MDA) approaches [23, 24] where new diagrams and code are automatically synthesized from the initial UML model: all the constructed artifacts would inherit the initial inconsistency [19].

Current UML case tools allow constructing incorrect models. They provide partial checking of selected model features, but it is not sufficient if we would like to create automatically a reliable application. More comprehensive checking can be found in the tools aimed at model analysis. For example, the OO design measurement tool SDMetrics [25] gives the rules according to which the models are checked. We used the experiences of the tool (Sec. 4), but it deals neither with state machine execution nor with C# language.

The consistency problems remain also using tools for building executable UML models [26-28]. Different subsets of UML being used and we cannot assure that two interchanged models will behave in the same way.

Solutions to consistency problems in class diagrams were presented in [29]. The problem refers to constrains specifying generalization sets in class diagram, which is still not commonly used in most of UML designs.

An interesting investigation about defects in industrial projects can be found in [30]. However the study takes into account only class diagrams, sequence diagrams and use case diagrams. It discusses mostly relations among elements from different diagram types. The state machines were not considered.

3 Code generation and execution in FXU

Transformation of UML models into executable application can be realized in the following steps.

1. A model, created using a CASE modeling tool, is exported and saved as an XML Metadata Interchange (XMI) file.
2. The model (or its parts) is transformed by a generator that creates a corresponding code in the target programming language.
3. The generated code is modified (if necessary), compiled and linked against a Runtime Library. The Runtime Library contains realization of different UML meta-model elements, especially referring to behavioral UML models.
4. The final application, reflecting the model behavior, can be executed.

It should be noted, that steps 1) and 2) can be merged, if the considered code generator is associated with the modelling tool.

The process presented above is realized in the FXU framework [5]. The target implementation language is C#. The part of UML model taken into account comprises classes and behavioral state machines. Protocol state machines are not considered.

The FXU framework consists of two components - FXU Generator and FXU Runtime Library. The Generator is responsible for realization of step 2. The FXU Runtime Library includes over forty classes that correspond to different elements of UML state machines. It implements the general rules of state machine behavior, independent of a considered model, e.g., processing of events, execution of transitions, entering and exiting states, realization of different pseudostates. It is also responsible for the runtime verification of certain features of an executed model.

3.1 Model transformation

Transforming class models into C# code, all model elements are implemented by appropriate C# elements. Principles of code generation from class models are similar to other object-oriented languages and analogues to solutions used in other tools. It is not so straightforward for state machine models.

State machines can be used at different levels of abstraction as behavioral state machines or protocol state machines. Protocol state machines are intended to model protocols. Behavioral state machines specify behavior of various model elements, like a class, a component, an operation. These elements constitute a context of a machine.

The primary application of behavioral state machine in an object-oriented model is description of a class. A class can have attributes keeping information about a current state of an object. Classes have operations that can trigger transitions, send and receive events. The FXU framework is limited to the most typical case, when a behavioral state machine models behavior of class instances. Model elements available in the context of the class are also available in the state machine.

A distinctive feature of FXU is dealing with all UML elements of behavioral state machines and their realization in C# application. Therefore we present selected concepts of state machines with their implementation in C#. We point out different C# specific mechanisms used in the generated application. Using selected solutions we would like to obtain an efficient and reliable application.

For any state machine of a class, a new attribute of *StateMachine* type is created. The structure of the state machine is build in a method of the class - *InitFXU()*. States, pseudostates, regions, transitions and events are created as local variables of the method.

Any state can have up to three types of internal activities *do, entry, exit.* The activities of a state are realized using a delegate mechanism of C#. Three methods *DoBody, EntryBody* and *ExitBody* with empty bodies are created for any state by default. If an activity exists a corresponding method with its body is created, using information taken from the model. Applying delegate mechanism allows defining the methods for states without using of inheritance or overloaded methods. Therefore the generated code can be simple, and generation of a class for any single state can be avoided. A state machine is not generated as a design pattern - “state” [31]. In the state design pattern, a single class is created for any state and any substate; and we would like to prevent an explosion of number of classes.

Signals, in opposite to other elements like states or events, are not created as local variables of the initialisation method. They are created as classes, because they can be generalized and specialized building a signals hierarchy. If a certain signal can trigger an event also all signals that are its descendants in the signal hierarchy can trigger the same event. This feature of signals was implemented using the reflection mechanism of C# [32].

Three transition kinds can be specified for a transition, *external, internal* and *local* transitions. Triggering an internal transition implies no change of a state, exit and entry activities are not invoked. If an external transition is triggered it will exit its source state (a composite one), i.e. its exit activity will be executed. A local transition is a transition within a composite state. No exit for the composite (source) state will be invoked, but the appropriate exits and entries of the substates included in the state will be executed.

A kind of a transition can be specified in a model, but in praxis this information is rarely updated and often inaccurate. Therefore we assumed that in case of composite states a kind of generated transition is determined using a following heuristics:

- If the target state is different than the source state of a transition and the source state is a composite state, the transition is external.
- Else, if the transition is defined in a model as internal it is treated as an internal transition.
- Otherwise, the transition is local.

A transition can have its guard condition and actions. They are created similarly to activities in states, using delegate mechanism of C#. If a body of an appropriate guard condition or action is nonempty in a model, it is put in the generated code. It should be noted that verification of logical conditions written in C# is postponed to the compilation time.

Events should have some identifiers in order to be managed. Change events and call events are identified by unique natural numbers assigned to the events. A time event is identified by a transition which can be triggered by this event. A completion event is identified by a state in which the event was generated. Finally, for a signal event the class of the signal, i.e., its type, is used as its identifier.

There are some elements of a UML model that include a description in a form not precisely specified in the standard, but dependent on a selected notation, usually a programming language. There are, for example, guard conditions, implementation of actions in transitions or in states, body of operations in classes. They can be written directly in a target implementation language (e.g., C#). During code generation these fragments are inserted into the final code. Verification of the syntax and semantics of such code extracts is performed during the code compilation and execution according to a selected programming language.

3.2 Model example

Fragments of an exemplary UML model are shown in Fig. 1. *Runway* class belongs to an airport control system. Selected attributes, operations and a state machine of the class are presented. The state machine describes different states of the runway. A runway can be opened, closed or deleted. State *deleted* is simple; two remaining states are composite ones. An opened runway can be either free or occupied. A runway can be closed due to temporary maintenance or emergency. Complex state *closed* consists of simple state *temporaryMainenance,* simple state *preparation* and complex state *restoration* including two orthogonal regions.

In guard conditions and triggers, the operations and attributes of the class are used. Several *entry* and *do* activities are omitted due to legibility reasons.

Using an FXU template the resulting programming class can be created for the *Runway* class and its behavioral model. Extracts of the C# code corresponding to the example and created by the FXU generator are given in the Appendix.

The *StateMachine* attribute of the *Runway* class defines the structure and features of its state machine. Except of methods implementing operations modelled in the class, the class has also two additional methods *InitFXU* and *StartFXU*. The *InitFXU* method is responsible for creation and initialization of all objects corresponding to all elements of state machine(s) associated with the class, such as regions, states, pseudostates, transitions, activities, events, triggers, guards, actions, etc. Bodies of *entry, do, exit* activities, guard conditions and actions are implemented with delegates. The *StartFXU* method is used for launching the behavior of the state machine.

3.3 Model execution

The structure of basic elements of the FXU Runtime Library corresponds to the simplified state machine meta-model (Fig. 2). A vertex of a state machine graph is handled as a state or a pseudostate. A specialized state can be a state machine. Any transition is defined by its source and destination vertices. A transition can be triggered by an event. Classes of all events are a direct specialization of the base class *Event*. Meta-model class *MessageEvent* was omitted, because it is an intermediate level abstract class and was not necessary to perform any tasks. An additional class *CompletitionEvent* is responsible for dealing with completion events. They are triggered once the entry actions, *do* activities and activities of the internal elements have been completed. The completion event can be triggered just after entering the state if there are no activities and no internal elements.

Event processing during state machine execution is performed according to the rules given in the UML specification [3]. The processing of a single event occurrence by a state machine is interpreted as a *run-to-completion step.* Before initiation and after completion of a step, a state machine is in a stable configuration. All *entry, exit* or internal activities of all states (complex and nested states) are completed, but *do* activities can last.

Basic algorithms of FXU realization, like execution of a state machine, entry to a state, exit from a state, were presented in [5]. For every state a queue was implemented that pools incoming events (Fig. 2). Event pool is served by a producers-consumer algorithm.

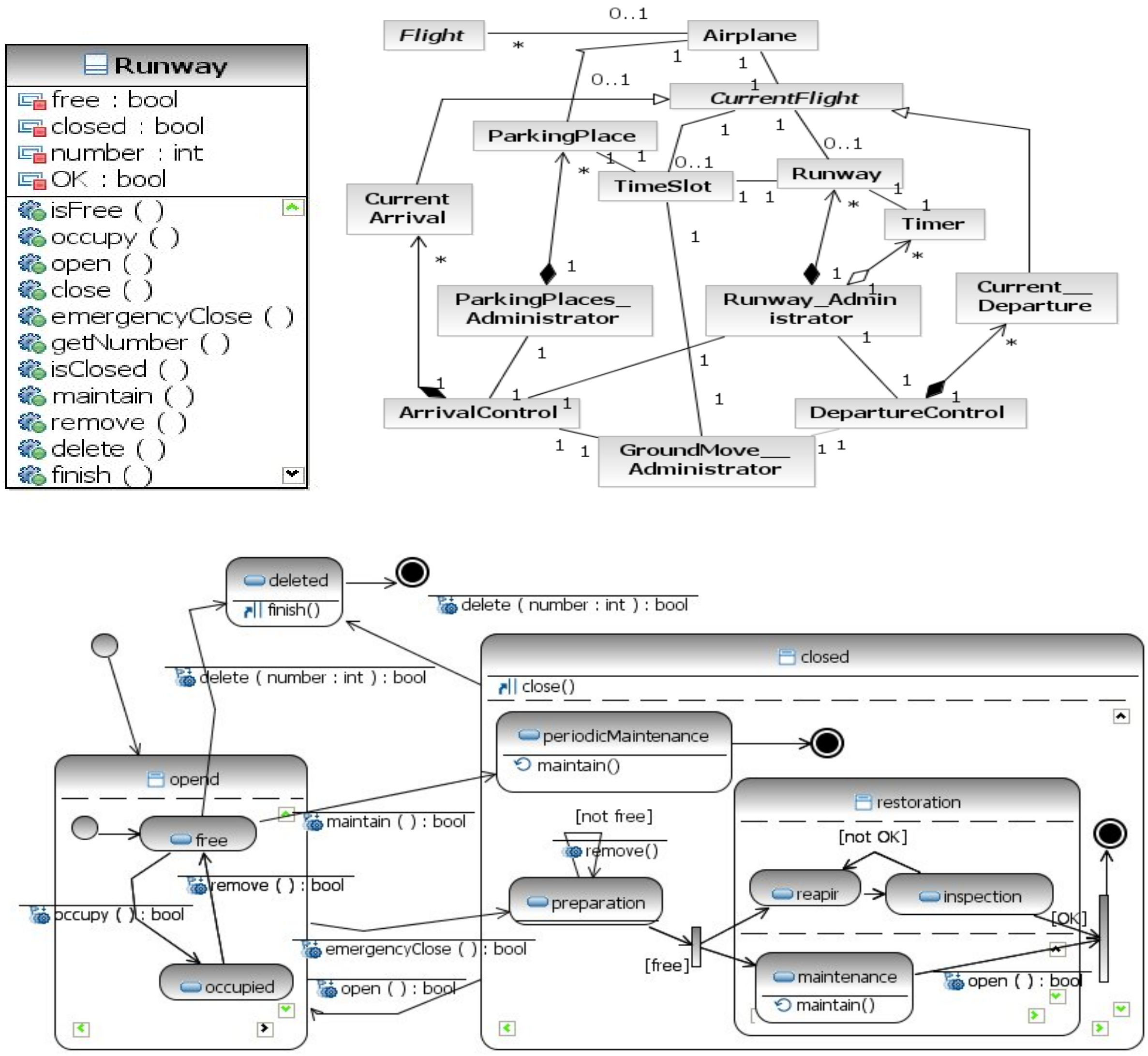

Figure 1: Example – *Runway* class, its state machine and class model

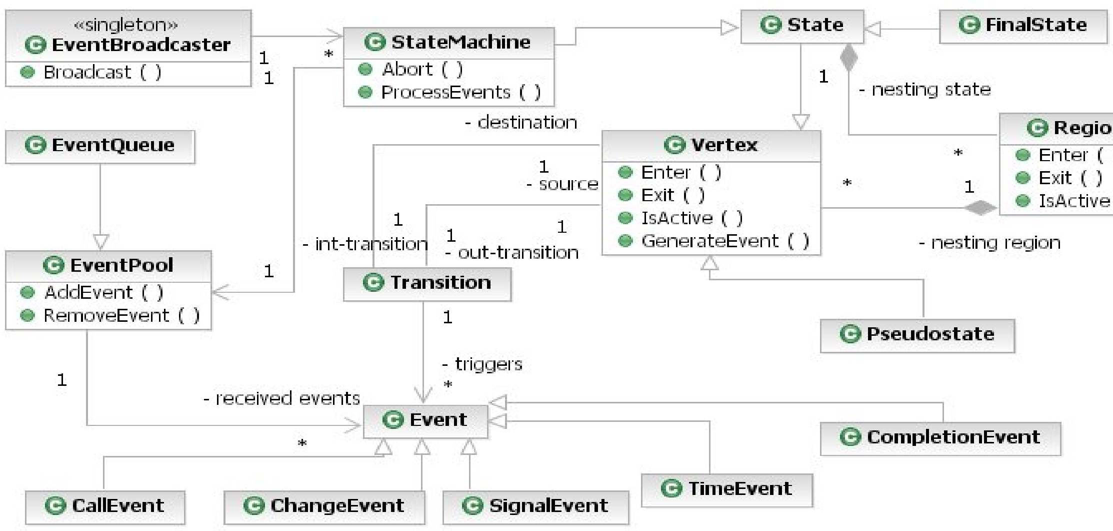

Figure 2: FXU Runtime Library - statemachines and event processing.

Events can be broadcasted or sent directly to the selected state machines. Events trigger transitions that have an active source state and their guard conditions evaluate to true. Transitions to be fired are determined as *the maximal set of non-conflicting transitions* [3]. If many transitions can be fired, transition priorities are used for their selection and resolve some transition conflicts. According to the specification, the priorities of conflicting transitions are based on their relative position in the state hierarchy. A transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.

Using this priority definition not all transition conflicts are resolved in case many transitions can be fired. Therefore, we had proposed and implemented an extended definition of transitions priority. We obtained one unique set of non-conflicting transitions in any situation. The detailed algorithm of selecting non-conflicting transitions and the extended firing priorities can be found in [6]. Also resolving of other variation points, especially dealing with entering and exiting orthogonal states, is shown in [6].

Interpreting different concepts of state machines we can use parallel execution. In the FXU RunTime Library it is implemented by multithreading. Multithreading is used for processing of many state machines which are active in the same time, e.g., state machines of different classes. It is used also for handling submachine states and orthogonal regions working within states, and for other processing of events. In the Appendix, examples of an output trace generated during execution of an application created from the model (Fig. 1) are shown. Different threads, which were created to deal with encountering events, are identified by number in brackets. For example, realization of a transition from the pseudostate fork to substate *maintenance* launched thread ”[14]”. Thread “[15]” was created to implement a transition from the fork pseudostate to substate *repair*. In other execution runs of the application, the numbers and ordering of the threads can be different.

4 Checking of model correctness

While generating valid C# code from UML class and state machine diagrams the certain conditions should be satisfied. There are many possible shortcomings of models that are not excluded by modeling tools, or should be not prohibited due to possible model incompleteness at different evolution stages. They were analyzed taking into account the practical weaknesses of model developers.

The prepared correctness rules were based on three main sources: the specification of UML [3], the rules discussed in related works and other comparable tools, in particular in [25], and finally our own study, especially taking into account the features of C# language - the target of the model transformation [32].

Various shortcomings can be detected during different steps of application realization (Sec. 3). Many of them can be identified directly in the model, and therefore detected during model to code transformation step (step 2). Verification of such problems will be called static, as it corresponds to an automated inspection of a model. Other flaws are detected only during execution of the resulting application (step 4). Such dynamic verification will be completed by the appropriate classes of the FXU Runtime Library.

In tables I-III defects identified in classes and state machines are presented. The last column shows severity associated to the shortcomings. Three classes of severity are distinguished. If a defect detected in a model is called as *critical* the model is treated as invalid and the code

Table I
Defects detected in UML class diagrams (static)

No	Detected defects	Reaction	Severity
1	A generalization of an interface from a class was detected	Stop code generation	critical
2	A name of an element to be generated (e.g. a class, an operation, an attribute) is a keyword of C# language	Stop code generation	critical
3	A class relates via generalization to more than one general class	Stop code generation	critical
4	A cycle in class generalization was detected	Stop code generation	critical
5	A name of an element to be generated is missing	Generate the element pattern without its name. The element name has to be supplemented in the generated code	medium
6	A name of an element to be generated is not a valid C# name. It is assumed that white characters are so common shortcoming that they should be automatically substituted by an underline character	As above	medium
7	An interface visibility is *private* or *protected*	Use *package* visibility	low
8	A class visibility is *private* or *protected*.	Use *package* visibility	low
9	An interface is *abstract*	Treat the interface as no abstract	low
10	An interface has some attributes	Ignore attributes of the interface	low
11	An interface has nested classes	Ignore classes nested in the interface	low
12	A class that is no *abstract* has abstract operations	Treat the class as *abstract*	low

generation is interrupted without producing the output. Later cases are classified as *medium* and *low*. In both cases the code generation is proceeded, although for *medium* severity it can require corrections before compilation. In all cases information about all detected shortcomings is delivered to a user. A detailed reaction to the found defect is described in the third column. While assigning severity levels and reactions to given defects we took into account general model correctness features but also requirements specific for C# applications.

4.1 Verification of class models

Class diagrams describe a static structure of a system, therefore many their features can be verified statically before code generation. Table I summaries defects that are checked during static analysis of UML class models. It was assumed that some improvements can be added more conveniently in the generated code than in a model. The class models can be incomplete to some extent and we can still generate the code. Admission of certain model incompleteness can be practically justifiable because of model evolution.

It should be noted that not all requirements of generated code are checked by the generator. Some elements are verified later by the compiler. It concerns especially elements that are not directly defined by the UML specification, like bodies of operations.

4.2 Verification of state machines

Similarly to class diagrams, different defects of state machines can be detected statically in the models. They are listed in Tab. II. Static detection of shortcomings in state machines is realized twice. First, it is made before model to source transformation (step 2). Second correctness checking is fulfilled before state machine execution. It is a part of step 4, during the initialization of the structure of a state machine.

For example, a static verification can be illustrated using a state machine from Fig. 1. Transition outgoing state *maintenace* has an event trigger - calling of an

Table II.
Defects detected in UML state machines (static)

No	Detected defects	Reaction	Severity
1	A cycle in signal generalization was detected	Stop code generation	critical
2	A signal inherits after an element that is not another signal	Stop code generation	critical
3	A signal relates via generalization to more than one general signal	Stop code generation	critical
4	A region has more than one initial pseudostate	Stop code generation	critical
5	A state has more than one deep history pseudostate or shallow history pseudostate	Stop code generation	critical
6	There are transitions from pseudostates to the same pseudostates (different than a choice pseudostate)	Stop code generation	critical
7	There are improper transitions between orthogonal regions	Stop code generation	critical
8	A transition trigger refers to an nonexistent signal	Stop code generation	critical
9	An entry point, join or initial pseudostate has no incoming transition or more than one incoming transition	Stop code generation	critical
10	A deep or shallow history pseudostate has more than one outgoing transition	Stop code generation	critical
11	A transition from an entry/exit point to an entry/exit point	Stop code generation	critical
12	An exit point has no any incoming transition	Stop code generation	critical
13	Transitions outgoing a fork pseudostate do not target states in different regions of an orthogonal states	Stop code generation	critical
14	Transitions incoming to a join pseudostate do not originate in different regions of an orthogonal state	Stop code generation	critical
15	There is a transition originating in an initial pseudostate or a deep/shallow history pseudostate and outgoing a nested orthogonal state	Stop code generation	critical
16	The region at the topmost level (region of a state machine) has no initial pseudostate	Warn a user	medium
17	A transition outgoing a pseudostate has a trigger	Ignore the trigger	medium
18	A transition outgoing a pseudostate (different from a choice or junction vertex) has a nonempty guard condition	Ignore the guard condition	medium
19	A transition targeting a join pseudostate has a trigger or nonempty guard condition	Ignore the trigger and/or condition	medium
20	A trigger refers to a non-existing operation	The transition will be generated but it cannot be triggered by this event	medium
21	A trigger refers to an abstract operation or to an operation of an interface	as above	medium
22	A time event is deferred	Treat the event as not being deferred	medium
23	A final state has an outgoing transition	Warn a user	medium
24	A terminate pseudostate has an outgoing transition	Warn a user	low

Table III.
Defects detected in UML state machines (dynamic)

No	Detected defects	Reaction	Severity
1	There is no enabled and no “else” transition outgoing a choice or junction pseudostate	Suspend execution - terminate	critical
2	A deep or shallow history pseudostate was entered that has no outgoing transitions and is “empty”, i.e. either a final state was a last active substate or the state was not visited before	Suspend execution - terminate	critical
3	More than one transition outgoing a choice or junction pseudostate is enabled	Select one enabled transition and ignore the others	medium
4	There is no enabled transition outgoing a choice or junction pseudostate and there is one or more “else” transition outgoing this pseudostate	Select one “else” transition and ignore other transitions	medium
5	More than one transition outgoing the same state is enabled	Select one transition and ignore the others	medium

operation *open()*. However, this transition targets the join pseudostate. Therefore neither a trigger nor a guard condition can be associated with the transition. It violates the correctness rule 19 (Tab. II). This model flaw is quite often and is not critical. The trigger will be omitted in the generated code and the designer will be warned about this exclusion.

The same rule is violated in a transition outgoing state *inspection*. The guard [OK] can not be used in this context. The generated code will be incomplete and the warning reminds of the correcting the state machine model.

State machines model system behavior; therefore not all their elements can be verified statically. A part of defects is detected dynamically, i.e., during execution of state machines. For example, a situation that two enabled transitions are outgoing the same choice pseudostate can be detected after evaluation of appropriate guard conditions, namely during program execution. Defects detected dynamically in state machines are listed in Tab. III.

5 Experiments

The FXU framework is not directly associated with any modelling tool but UML models are passed between tools using files. Input models in some XMI variants, UML2 and UML formats, supported by Eclipse, are accepted. Therefore the solution is not tool-dependent. However, all experiments mentioned in this Section were performed with UML models created using IBM Rational Software Architect [33].

The presented approach for building the C# code and executing the automatically created applications was tested on over fifty models. The first group of ten models was aimed at classes. In experiments the correct and incorrect constructions encountering in class diagrams were checked, concerning especially association and generalization. Moreover, two bigger projects were tested. The first one was a design of a web page, which was a part of MDA project called Acceleo [34]. The model described a design of a web page. The second one presented a metamodel of an object-oriented modeling language [35].

Models from the next group (above forty models) comprised different diagrams, including both classes and their state machines. All possible constructs of UML 2.x behavioral state machines were used in different situations in the models. The biggest design included five state machines with about 80 states and 110 transitions, using complex and orthogonal states, different kinds of pseudostates and submachine states.

The programs realizing state machines were run taking into account different sequences of triggering events. The behavior modeled by state machines was observed and verified using detailed traces generated during program runs. They helped to test whether the obtained program behavior conforms to desired state machine semantics. For complex models, filtered traces that included selected information were also used.

In the performed experiments, applications realizing behavior specified in state machine models were developed in an automated way. For example, different airport subsystems were modeled in order to simulate a desired behavior. The essential part of the class model of *Airport_FlightControl* subsystem is shown in Fig. 1. It models occupation of runways and airplane parking places. Behavior of classes can be defined by state machines realising different policies. One exemplary state machine is shown for *Runway* class. Comparison of the policies is easily performed combining code generated for different versions of state machines in final applications.

6 Conclusion and future work

In this paper we discussed the problems of creation of valid C# applications realizing ideas modeled by classes and their state machines. Different C# mechanisms were effectively used for implementation of the full state machine model defined in the UML 2.x specification. We showed which correctness issues of models have to be checked during model transformation (static verification) and during application execution (dynamic verification). The detailed correctness rules help a developer to cope with possible flaws present in UML models. In the difference to other tools, using FXU the state machines including any complex features can be effectively transformed into corresponding C# application. The tool support assists building of reliable applications including complex behavioral specifications. It can be especially useful for developing programs in which non-trivial state machines are intensely used, e.g., dependable systems, embedded reactive systems.

In the future work, we prepare other complex models implementing telecommunication problems. Capability of using advance state machine features and building reliable applications is very important in these cases.

As a complementary approach, another solution for C# code generation based on C# profiles is under development. Transform OCL Fragments Into C# (T.O.F.I.C.) tool supports labelling of UML model elements with stereotypes reflecting C# concepts. Target code is generated from a refined UML model and OCL constraints. In this approach, a model can be verified both during placing stereotypes and/or code generation process. Using dedicated profiles enforce more precise mapping to a given target language and therefore also checking of model correctness. However it requires more effort of a developer while creating a refined model.

Appendix

The appendix includes selected extracts of C# code generated for an exemplary class and its state machine shown in Fig. 1. Code of class operations is omitted. Method *InitFxu()* creates appropriate structure of the state machine and method *StartFxu()* initializes its behavior.

```
public class Runway    {
  private bool free;
// other attributes and operations (omitted)
//
  StateMachine sm1 =
       new StateMachine("RunwayStateMachine");
  public void InitFxu() {
     Region r1 = new Region("Region1");
     sm1.AddRegion(r1);
     InitialPseudostate v2 = new
               InitialPseudostate("");
     r1.AddVertex(v2);
     State v4 =  new State ("opend");
     r1.AddVertex(v4);
//...
     State v8 =  new State ("closed");
     r1.AddVertex(v8);
     v8.EntryBody = delegate(){ close(); };
     Region r3 = new Region("Region1");
     v8.AddRegion(r3);
//...
     State v11 =  new State ("restoration");
     r3.AddVertex(v11);
     Region r4 = new Region("Region1");
     v11.AddRegion(r4);
     Region r5 = new Region("Region2");
     v11.AddRegion(r4);
//...
     State v14 =  new State ("maintenance");
     r5.AddVertex(v14);
     v14.DoBody = delegate(){ maintain(); };
//
     Fork v15 = new Fork("");
     r3.AddVertex(v15);
//
     Transition t1 = new Transition(v2, v4);
     Transition t8 = new Transition(v8, v4);
     t8.AddTrigger(new CallEvent("open", 1))
//...
     Transition t11 = new Transition(v10, v10);
     t11.GuardBody = delegate()
              {return not free;};
     t11.ActionBody = delegate(){remove(); };
//...
} //End of InitFXU
  public void StartFxu(){
     sm1.Enter();     }
}
```

Log items selected from a detailed execution trace of the exemplary state machine (Fig. 1) are shown below. All labels of the items, including time stamps and item types (*Warning, Information, Debugging*), are omitted for the brevity reasons. A number in brackets denotes a number of a thread that realizes a considered part of machine execution.

[1] State diagram < RunwayStateMachine >: Entered.

[1] State diagram < RunwayStateMachine >: Execution of **entry-activity started**. State is now active.

[1] State diagram < RunwayStateMachine >: Execution of **entry-activity finished.**

[7] Initial **pseudostate** < RunwayStateMachine:: Region1{::UnNamedVertex}>: **Entered.**

[7] Transition from Initial pseudostate < RunwayState Machine::Region1{::UnNamedVertex}> to State < RunwayStateMachine::Region1::opend>: **Traversing** started.

[7] State < RunwayStateMachine::Region1:: opend>: Execution of **entry-activity started**. State is now active.

After *emergencyClose* trigger

[3] State diagram < RunwayStateMachine >: **Call-event** <emergencyClose [ID=1]> has been dispatched.

[9] State < RunwayStateMachine::Region1:: opend:: Region1::occupied>: Execution of **exit-activity started.**

Transition to fork from *preparation* state

[3] State diagram < RunwayStateMachine >: **Completion event** <> generated by State < RunwayStateMachine:: Region1::closed::Region1::preparation> has been dispatched.

[12] State < RunwayStateMachine::Region1::closed:: Region1::preparation >: Execution of **exit-activity started**.

[13] Transition from State < RunwayStateMachine:: Region1::closed::Region1::preparation > to Fork < RunwayStateMachine::Region1::closed::Region1{::UnNamedVertex}>: **Traversing** started.

[14] Transition **from Fork** < RunwayStateMachine:: Region1::closed::Region1{::UnNamedVertex}> to State < RunwayStateMachine::Region1::closed:::restoration::Region 2::maintenance>: **Traversing** started.

[16] State < RunwayStateMachine::Region1::closed::: restoration >: Execution of **entry-activity started**. State is now active.

[15] Transition **from Fork** < RunwayStateMachine:: Region1::closed::Region1{::UnNamedVertex}> to State < RunwayStateMachine::Region1::closed:::restoration::Region 1::repair>: **Traversing** started.

Acknowledgement

The authors would like to thank Kamil Raś for his help in preparing UML models.

References

[1] R. France, B. Rumpe (2007). Model-driven Development of Complex Software: A Research Roadmap. *Future of Software Engineering at ICSE'07*, IEEE Soc., pp. 37-54.

[2] Semantics of a foundation subset for executable UML models (FUML) (2008). http://www.omg.org/spec//FUML/

[3] Unified Modeling Language Superstructure v. 2.1.2 (2007). OMG Document formal/2007-11-02, http://www.uml.org

[4] C. Lange at al. (2003). An empirical investigation in quantifying inconsistency and incompleteness of UML designs. in *Proc. of 2nd Workshop on Consistency Problems in UML-based Software Development co-located at UML'03 Conf.*, San Francisko, USA, Oct 2003, pp. 26-34.

[5] R. Pilitowski, A. Derezinska (2007). Code Generation and Execution Framework for UML 2.0 Classes and State Machines. T. Sobh (eds.) *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, Springer, pp. 421-427.

[6] A. Derezinska, R. Pilitowski (2007). Event Processing in Code Generation and Execution Framework of UML State Machines. in L. Madeyski at al. (eds.) *Software Engineering in progress*. Nakom, Poznań, pp.80-92.

[7] L. K. Kishore, D. Saini, IBM Rational Modeling Extension for Microsoft .NET, http://www.ibm.com/developerworks/rational/library/07/0306_kishore_saini/

[8] Rhapsody, http://www.telelogic.com/

[9] A. Niaz, J. Tanaka (2004). Mapping UML Statecharts into Java code. in *Proc. of the IASTED Int. Conf. Software Engineering*, Acta Press, Anheim, Calgary, Zurich, pp. 111-116.

[10] SmartState, http://www.smartstatestudio.com

[11] Hugo/RT, http://www.pst.ifi.lmu.de/projekte/hugo/

[12] A. Knapp, S. Merz (2002). Model Checking and Code Generation for UML State Machines and Collaborations. In D. Haneberg, G. Schellhorn, and W. Reif, Eds, Proc. 5th Workshop Tools for System Design and Verif., pp. 59-64. Tech. Rep. 2002-11, Inst. für Informatik, Univ. Augsburg,

[13] D. Harel, H. Kugler (2004). The Rhapsody Semantics of Statecharts (or On the Executable Core of the UML) (preliminary version), *SoftSpez Final Report*, LNCS, vol. 3147, Springer, Heidelberg, pp. 325-354.

[14] STL: UML 2 Semantics Project, References, Queen's University http://www.cs.queensu.ca/home/stl/internal/uml2/refs.htm

[15] M. Crane, J. Dingel (2005). UML vs. Classical vs. Rhapsody Statecharts: Not All Models are Created Equal. in: *MoDELS/UML* 2005, LNCS, vol. 3713, Springer, Heidelberg, pp. 97-112.

[16] Y. Jin, R. Esser and J. W. Janneck (2004). A Method for Describing the Syntax and Semantics of UML Statecharts. *Software and System Modeling*, vol. 3 no 2, Springer, 2004, pp. 150-163.

[17] H. Fecher, J. Schönborn (2007). UML 2.0 state machines: Complete formal semantics via core state machines. in *FMICS and PDMC* 2006, LNCS vol. 4346, Springer, Hildelberg, pp. 244-260.

[18] F. Chauvel, J-M. Jezequel (2005). Code Generation from UML Models with Semantic Variation Points. *MoDELS/UML* 2005, LNCS, vol. 3713, Springer, Heidelberg, pp. 97-112.

[19] A. Baruzzo, M. Comini (2006). Static verification of UML model consistency. *Proc. of the 3rd Workshop on Model Development, Validation and Verific., at MoDELS'06,* Genoa, Italy, pp. 111-126.

[20] A. Egyed (2007). Fixing inconsistencies in UML designs, in *Proc. of 29th Intern. Conf. on Software Engineering,* ICSE'07, IEEE Comp. Soc.

[21] S. Prochanow, R. von Hanxleden (2007). Statecharts development beyond WYSIWIG, in G. Engels et al. (Eds.) *MODELS* 2007, LNCS 4735, Springer, Berlin Heidelberg, pp. 635-649.

[22] L-K. Ha, B-W Kang. (2003). Meta-Validation of UML Structural Diagrams and Behavioral Diagrams with Consistency Rules. *Proc. of IEEE PACRIM*, Vol. 2., 28-30 Aug. pp. 679-683.

[23] MDA Guide Ver. 1.0.1 (2003). OMG Document omg/2003-06-01.

[24] S. Frankel (2003). *Model Driven Architecture: Appling MDA to enterprise computing*. Wiley Press, Hoboken, NJ.

[25] J. Wuest, SDMetrics - the UML design measurement tool, http://www.sdmetrics.com/manual?LORules.html

[26] S. J. Mellor, M. J. Balcer (2002). *Executable UML a Foundation for Model-Driven Architecture.* Addison-Wesley.

[27] C. Raistrick, at al. (2004). *Model Driven Architecture with Executable UML* Cambridge University Press.

[28] K. Carter, iUMLite - xUML modeling tool, http://www.kc.com (visited 2009)

[29] A. Maraee, M. Balaban (2006). Efficient decision of consistency in UML diagrams with constrained generalization sets. *Proc. of the 1st Workshop on Quality in Modeling, co-located. at MoDELS'06*, Genoa, Italy, pp. 1-14.

[30] F. J. Lange, M. R. V. Chaudron (2007). Defects in industrial UML models - a multiple case study. *Proc. of the 2nd Workshop on Quality in Modeling, at MoDELS'07*, Nashville, TN, USA, pp. 50-64.

[31] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1995). *Design patterns: elements of reusable object-oriented software.* Boston Addison-Wesley.

[32] J. Liberty (2005). *Programming C#,* O'Reilly Media.

[33] IBM Rational Software Architect, http://www-306.ibm.com/software/rational

[34] Acceleo project http://www.acceleo.org

[35] G. Booch, Metamodel of object-oriented modeling language, http://www.booch.com/architecture/architecture/artifacts/architecture.emx.