Towards C# Application Development using UML State Machines – a Case Study

Anna Derezińska, Marian Szczykulski
Institute of Computer Science, Warsaw University of Technology
A.Derezinska@ii.pw.edu.pl

Abstract – **Using a state machine for modeling a class behavior can assist effective development of an application. We discuss a model-driven approach to building a C# application based on UML class models and behavioral state machines. A case study addressed in the paper is devoted to a social network of mobile users. The core of the system is a presence server for the status services in the network. There are three main tasks performed by the server: subscription of a status of another user, publication of a new status with given rules and notification another user about a status. The system architecture and exemplary state machine models are presented. Model to code transformation and development of an executable application was realized by a Framework for eXecutable UML (FXU). Verification of the application was supported by tracing of program execution in terms of model elements using FXU Tracer. On the basis of the gathered experience, we discuss design guidelines for carrying out the approach.**

Keywords: MDA, UML, state machine, C#, model transformation, code generation

I. Introduction

Human comprehension of graphical models can be more effective and less error-prone than a direct code analysis. Therefore Model Driven Engineering (MDE) technology promotes application development based on a model to code transformation [1]. A leading proposal for MDE is the initiative of Model Driven Architecture (MDA) [2,3] that encourages different ideas and tools around the UML notation.

There are various strategies to cope with models and their execution. A model can be directly executed according to its virtual machine as, for example, one given in the Foundation Subset for Executable UML Models (FUML) [4]. Another strategy is transformation of a source model into a target code. It can be completed in several steps. In the first step, a target level can be an intermediate model level. For instance, a level between UML models and a programming language is used in order to simplify transformation to a desired programming language [5]. Transformation can also be defined as a direct translation of a model to a code given in a general purpose programming language, e.g. Java. C++, C# [6-10]. This paper addresses the later approach.

Complex problems can be modeled with behavioral state machines. It is beneficial to support the development process with a tool capable to transform advance state machine features and to build a reliable application. We discuss this problem in a case study.

A social network deals with a set of mobile users with time and space dynamic relationships. Social networking services support different context-aware features. *Internet Protocol Multimedia Subsystem* (IMS) is an environment that provides a developer with multimedia services based on the Internet protocol. The environment was used for the implementation of presence services presented in [11]. The specification was tested using an emulator of IMS. In this work, we propose another solution using UML class and state machine diagrams [12]. Further they were transformed to a C# code and executed using a Framework for eXecutable UML (FXU) [6].

In the next section, we present the background of transformation of UML state machines to a C# code. The system considered in the case study and its modeling is introduced in Section III. Section IV discusses technology of application development and design guidelines. Finally, Section V concludes the paper.

II. State machine transformation to C# code

Providing with modeling facilities, CASE tools also support a model to code transformation, which can assist building a final application reflecting principles behind the model. However, as far as UML is concerned, a transformation is mostly restricted to a structural model, namely a class diagram as, for example, in Rational Modeling Extension for .NET [7].

Some tools also support the code transformation from behavioral diagrams, especially state machines. They usually consider only a subset of state machine elements, whereas such advanced concepts as orthogonal regions, deep and shallow history pseudostates, deferred events or internal transitions are omitted.

An exception is the IBM Rational Rhapsody tool [8] (previously Telelogic and I-Logic) that takes into account the entire state machine during transformations aimed at different programming languages, although it does not support C#.

The Framework for Executable UML (FXU), introduced in [6], performs transformation from UML class and state machine models into a corresponding C# code. It supports creation and execution of the resulting application that is intended to reflect the modeled behavior.

The framework, in the opposite to the most of similar tools, implements a complete specification of state machines, including all kinds of states, such as simple states, composite states, orthogonal states, *entry*-, *do*- and *exit*- activities, and submachine states. It also takes into account all possible pseudostates, i.e. initial psudostate, deep and shallow history, join, fork, junction, choice, entry and exit points, terminate, as well as different kinds of events (also deferred events).

In FXU several variation points of the UML state machine specification [13] had to be resolved in order to obtain an unambiguous interpretation of a model behavior [14].

In order to improve the usability of the framework a new FXU version was released. It was equipped with GUI, Application Wizard and FXU Tracer assisting comprehension of an application behavior in terms of state machine elements [15].

In some solutions, the code created as the target of a model transformation includes a mapping of the state machine structure as well as of the logic supporting a model execution [16]. Recently, Sparx Enterprise Architect [9] was extended with code generator from a state diagram to various programming languages, including C#. Also in this case, a subset of state machine concepts is transformed into predefined code extracts that are placed directly into a considered class. In case of many state machines the amount of generated code can be significant, aggravating evolution and maintenance of a program.

The approach implemented in FXU is different. The entire logic of state machines is hidden in the library code (runtime environment) and for each class is only generated a simple code defining a structure of its state machine and its initialization.

III. Presence Server Modelling

This section describes the basic idea of the system used in the case study (*A*) and its modeling with UML structural and behavioral diagrams (*B*).

A. System overview

The system under consideration was devoted to a status service for a social network. A user status is understood as information about a current user's presence and its context. A context is defined by a location, communication possibilities, occupation of a user, its activities, mood, etc. There are different relationships between users. The relations can be divided into various categories, like family, friends, coworkers, acquaintances of a common hobby.

A user description is specified according to a format FOAF (*Friend of a Friend*). This specification is based on the XML syntax and the RDF structure (*Resource Description Framework*) – a standard used for defining data in the net.

A presence service, which is offered in a social network provides users with a required data about a status of a given, selected user. The main job of the service is filtering of information in accordance to different inter-user relationships.

The services are supported by *Session Initiation Protocol* (SIP) - an open standard for creating a session between one or many clients proposed by *Internet Engineering Task Force*. Application of the SIP protocol for subscription and distributing presence information is described in [17].

A user, let called X, registers directly in a SIP server where information about its status is stored. Another user, so called observer, can request a subscription of the status of user X. Therefore the observer communicates with the *Presence Server* sending an appropriate message. The presence server communicates with the SIP server fetching a current status of user X. The presence server also connects to the XCAP server (*XML Configuration Access Protocol*) where rules defining a resulting status drawn on a current state and a relation between a user and his/her observer are gathered. Therefore the observer can be notified about the real status of user X.

In the case study, we focused on *Presence Server* that constitutes the main part of the system and is responsible for completing of the business logic. A presence server should realize the following main tasks:

- Status creation – creation of a predefined presence status that controls an access to presence information.
- Status publication – making known selected information to users subscribed in a contact list, according to rules of a formerly predefined status.
- Status subscription – requesting information about a presence state of an observed user.
- Delete subscription – deletion of a presence status. A user is no more registered in the service because do not want to be informed about changes of statuses of other users from the contact list.

B. System model

The system was divided into three main layers devoted to:
- communication with a client,
- controlling of presence statuses,
- communication with other systems.

Each layer is modeled by a package comprising its classes and other subpackages, if necessary. A rough class diagram of *Presence Server* is shown in Fig. 1.

Package *clientConnector* consists of four classes and is responsible for receiving client requests. It performs a preliminary preprocessing of requests and forwards them to further handling by package *presenceAgentController*.

This package is the core of the system. Its classes fulfill the basic functional requirements of the presence server. The package uses adapters, stored in package *systemAdapters*, for communication with other systems. Each adapter is created in a separate subpackage. There are three following adapters: an adapter for the server SIP, the server XCAP and the server FOAF. Finally, package *utils* comprises auxiliary classes used by the system.

For each class of the model a state machine was created that specifies a behavior of the class. A state machine complexity varies from a simple two-state model to a more complex state machine that uses different modeling techniques (history pseudostates, parallel execution with forks and joins, composite states, orthogonal regions, etc.).

The whole presence server model consists of about twenty classes and interfaces and about seventeen state machines. Two exemplary classes and their state machines are presented below. In the state machine diagrams some less important details are omitted in order to keep the figures legible enough.

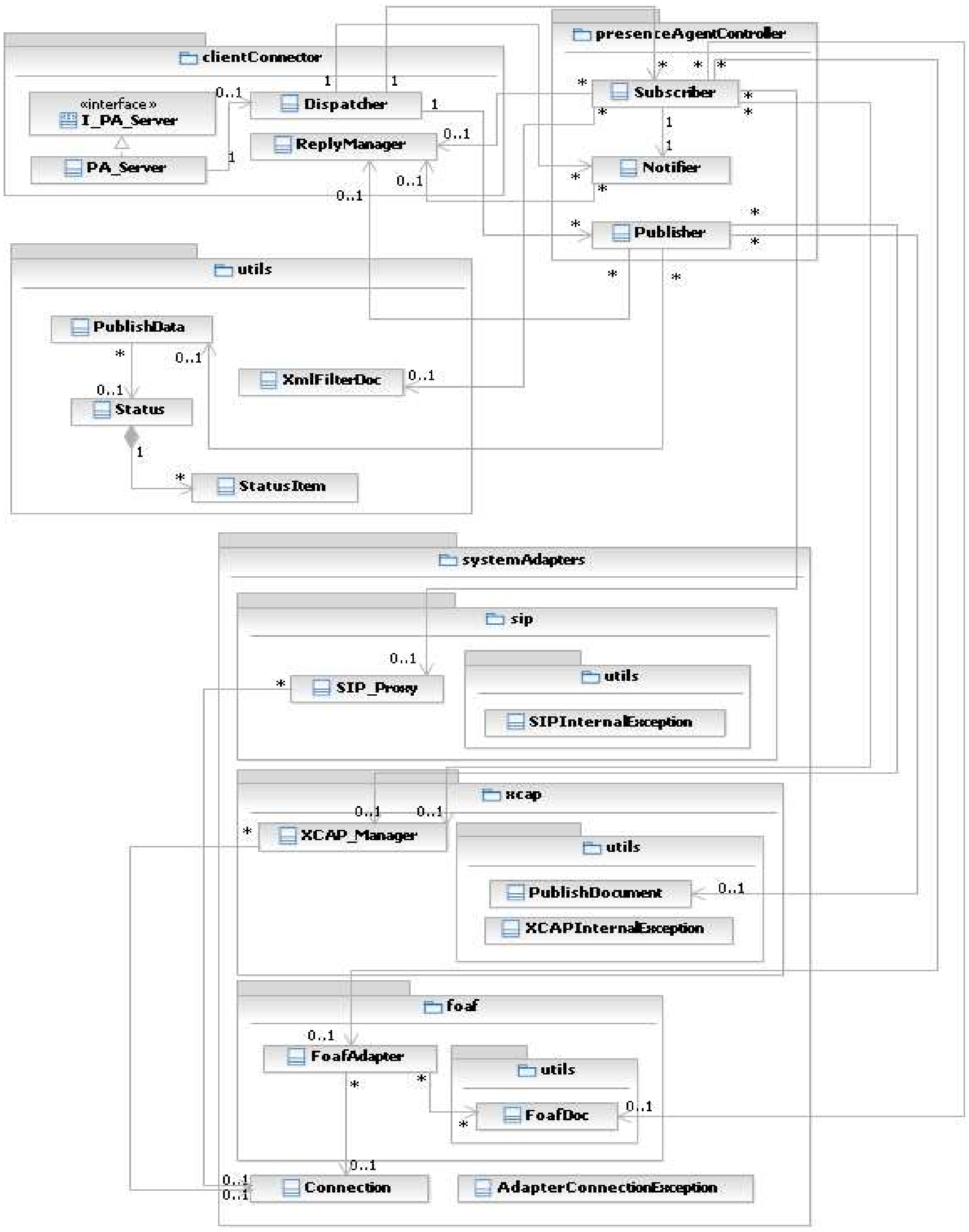

Fig. 1. Class diagram of the *Presence Server* system

The central part of the system constitutes package *presenceAgentControler* with its classes *Subscriber*, *Notifier* and *Publisher*. Class *Notifier* is responsible for distributing a notification about a subscribed user status. It handles NOTIFY request. For this purpose, it uses an object of class *ReplayManager*. The structural and behavioral model of class *Notifier* is given in Fig. 2. Class *Notifier* has an operation for notifying a state of a user that subscription was requested. Other operations of the class acknowledge a correct or erroneous notification accordingly; or handle a notification.

The given state machine specifies a notification process. The process is divided into several phases. Initially, an object of class *Notifier* is idle, waiting for a request. Once an appropriate request is received, a transition is triggered and the object enters a corresponding substate in the complex state (*Working*). It should be noted that events that are not handled in a state are omitted in the model. It is generally assumed that an occurrence of such an unexpected event will be skipped

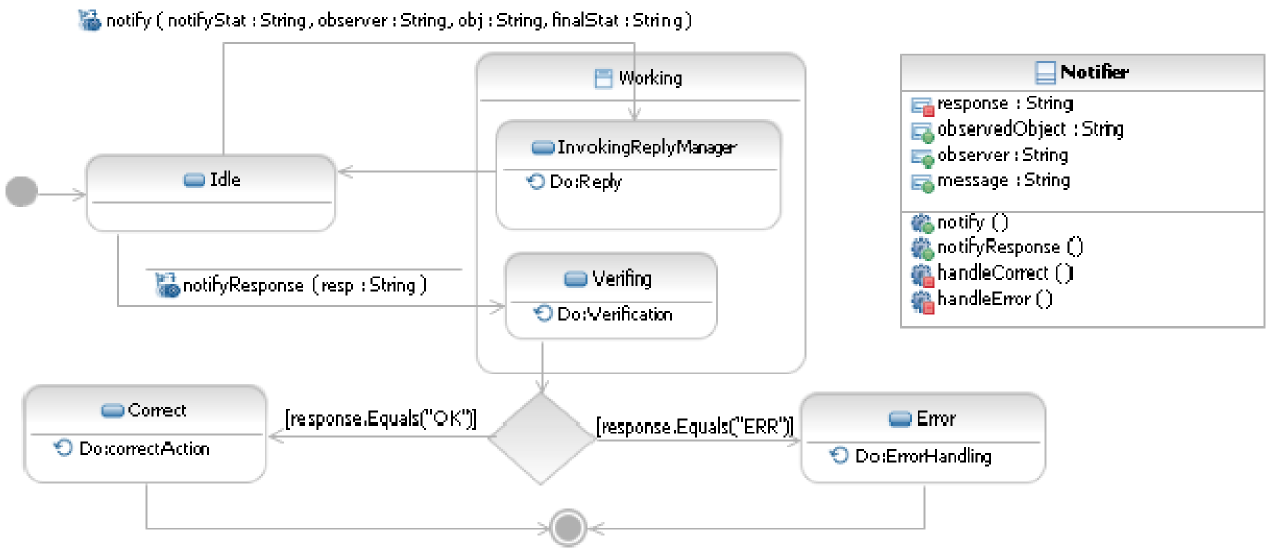

Fig. 2. Class *Notifier* and its state machine (a member of package *presenceAgentControler*)

during execution of the model and the model remains in the unchanged state.

Calling of operation *notify()* launches the major part of the notification process. The configuration of *Notifier* switches to <*Working*, *InvokingReplyManager*>. A notification comprising a subscribed status of a user is sent to an observer. An object of class *ReplyManager* is invoked. It replays whether the notification was successful. An object of *Notifier* returns to the idle state.

The next activity phase is triggered by a request of sending a response about a notification status. The automaton enters another substate of the complex working state. Verification of correctness of a response is performed. In dependence on the verification result, the object activity is modeled by two states that correspond to a correct response and an erroneous one. One of the states is chosen and an action that is appropriate to a correct situation or an error handling is performed. It finishes activity of the object and therefore the notification process.

Two remaining classes of package *presenceAgentControler* are dealing with publishing information about a client's presence and subscribing of a status of another user. They handle requests PUBLISH and SUBSCRIBE, accordingly. Classes *Publisher* and *Subscriber* are more complex than *Notifier* and have about 12-16 operations each. Their state machines include several complex states with substates, choice selections, and forks for parallel actions.

Package *systemAdapters* comprises subpackages responsible for particular adapters. The main classes placed in those subpackages, namely *SIP_Proxy*, *XCAP_Manager* and *FAOF_Adapter* are related to appropriate remote servers. The remaining classes of the package are devoted to supplementary functions. There are also classes handling exceptions that can occur during communication with the servers.

An example of adapter classes with its state machine is shown in Fig. 3. An object of class *XCAP_Manager* is responsible for the proper communication between an agent controller and the XCAP server, which keeps the rules regarding statuses of users. *XCAP_Manager* stores the current status of the connection to the server. The class owns operations for managing connection with the server, publishing rules of a status and evaluating the final user's status.

The behavior of any *XCAP_Manager* is specified by its state machine. Communication with the XCAP server is run in several steps. First, a manager object is initialized and connection parameters are collected. If the parameters are set, the manager tries to connect with the XCAP server.

In the next step, the connection status is checked. If the connection was established the manager moves on to state *ConnectionActive*. In this state, the final state of a user is evaluated according to the status document and the relation between users. The document is taken from the SIP server whereas the relation from the FOAF server. Then, the rules about the client status are published in the XCAP server.

Otherwise, when the connection is disabled, state *ConnectionInactive* is entered. In this state the object waits for the appropriate time delay before it tries again to establish a connection. This situation is modeled by means of a time event. It labels a transition outgoing from state *ConnectioInactive* to *XCAPServerConnectionChecking*.

Finally, when a disconnection of the manager is requested, regardless in an active or inactive state, the object ends its activity. A new manager object is created if re-connection with the XCAP server is required.

IV. Presence Server Application Development

In this section we discuss briefly how the application was constructed from the *Presence Server* model and comment on good design practices.

A. *Technology*

The system of *Presence Server* was modeled using a CASE

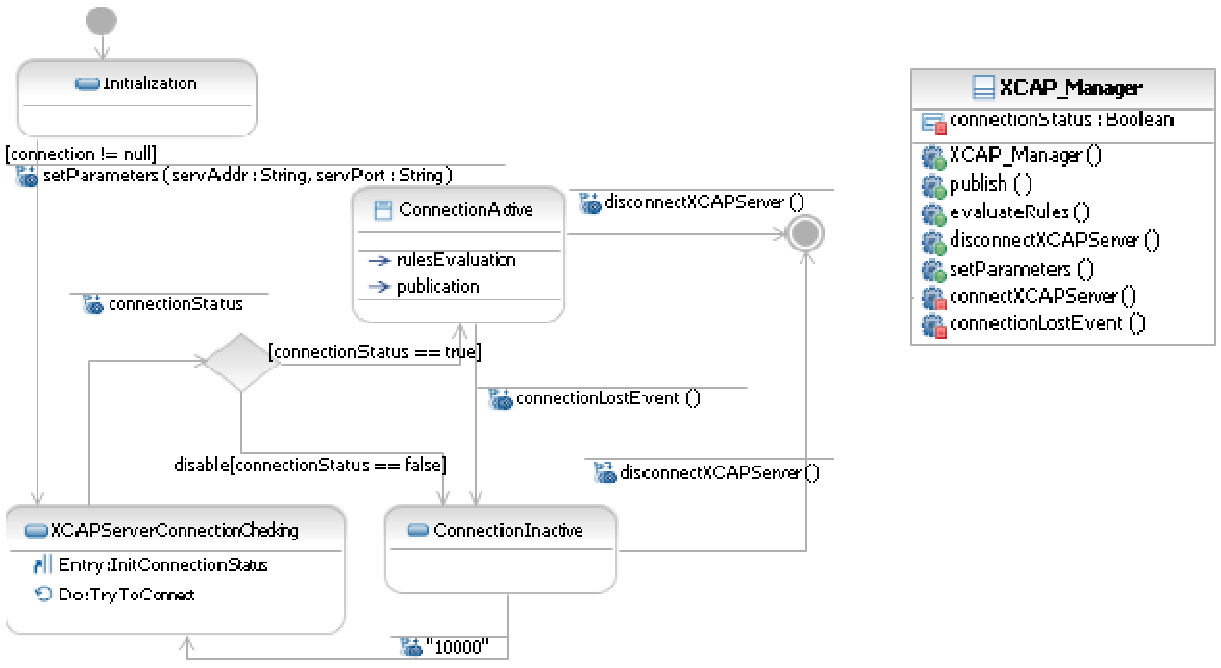

Fig. 3. Class *XCAP_Manager* and its state machine (an example of a system adapter)

tool. For this purpose we used the IBM Rational Software Architect [10] – a tool that is run within the Eclipse platform [18]. The model, exported in the UML 2.1 format of Eclipse, was forwarded to FXU Generator.

The generator extracted all information of classes and their behavioral state machines and transformed into a corresponding code in the C# language. Using Application Wizard, which is a part of the FXU generator, a project of Microsoft VS was built. At this stage, we declare objects that cooperate within the model, initialize their state machines and define required event occurrences.

Apart from the created code the project was coupled with an FXU Runtime Library. The library provided the application with a code implementing the behavioral logic of all concepts of the UML state machine.

The application was supplemented with the code, written directly in the C# development environment that implemented body of particular operations.

The correctness of class and state machine models was, to some extent, verified during the model to code transformation [19]. Moreover, some checking of selected dynamic features of state machines is incorporated in the library and was completed during application runs.

The developed application was further tested to verify correctness of business logic built-in the designed automata. The testing concentrated on handling of three main requests: PUBLISH, SUBSRIBE and NOTIFY. Each publishing request pointed at a user who produced it and the user's status. A subscription request was associated with data identifying two users: an observer that demanded subscription of a status, and a subscribed object. A notification request comprised identification about an observer, a subscribed object, and a status of the object.

Traces of application runs were stored in log files. They were further analyzed with assistance of FXU Tracer [15]. Application behavior was examined in terms of visited states and other elements of state machines constituting the *Presence Server* model. The observation was advanced step by step or using appropriate breakpoints if necessary. The application behavior was consent to the primary assumptions behind the presence server's activity.

B. Design guidelines

Basing on the experience gathered during system development the following design guidelines can be addressed.

- Each class should be specified by its behavioral state diagram. In some cases the diagram will be very simple, e.g. comprising only two simple states, but the whole generated code will be more completely and cover the application logic as much as possible.
- Nondeterministic transitions in state machines should be avoided or used very carefully. Otherwise the application can behave in an unexpected manner. If there is no synchronization provided state machines of various classes, but also different regions in orthogonal states can be scheduled in any possible order.
- It is helpful to document all elements of a state machine with associated names. It refers not only to states but also transitions, events, *entry*, *do* or *exit* actions. They can by further easily localized in the source code, e.g. in order to correct them.
- It is a good practice to specify *entry*, *do* or *exit* actions

with a short code extracts. This code should be brief and not complicated, as it is not verified before the compilation. It can be advised to call a private operation as an action. The operation can be further implemented in the code development environment. It can also be changed or substituted without interfering into the model.

- Each operation in a class model should be specified with its arguments and their types, unless a default type is consciously accepted. If a type is not determined it will be generated with a default type. The default type might be not consistent with intentions of a designer of a particular model.
- It is worthwhile to comment important diagram elements, filling in an appropriate “description” section. However, in case of an operation, the comments should obey the rules of a programming comment, i.e. be either proceeded by “//” sign or encompassed by “/*” and “*/”. The description will be a part of the operation body and can be compiled properly as a comment.
- Writing too much code implementing an operation directly as its textual description in a model should be avoided or at least made very carefully. The code will be transformed and appropriately placed in the resulting source code, but its correctness will be not checked until the program compilation time.
- It is not recommended to use modeling constructions that are variation semantic points of the UML specification [13]. Variation points are especially characteristic to state diagrams, and can be implemented in different ways in various tools, being still consistent with the UML specification. If necessary, a designer should check a solution selected in the tool (FXU) for a desired variation point in order to prevent an unexpected application behavior.
- While verifying a developed application, it is beneficial to interpret its trace in terms of elements of state models. It can be completed using FXU Tracer [15].

V. Conclusions

We have presented an approach to model-driven development of an application. The approach is intended to support projects which are built around concepts of behavioral state machines. It was illustrated by a case study of predefined presence status services for social networks. The considered process is supported by the FXU environment aimed at C# applications.

Our future work is focused on extending the environment with various implementations of selected semantic variation points of UML. We also plan to deploy model transformation and state machine execution in accordance to detailed time specification provided in a model with appropriate stereotypes of a UML profile.

References

[1] R. France, B. Rumpe, “Model-driven Development of complex software: A research roadmap”, *Future of Software Engineering at ICSE'07*, IEEE Soc., 2007, pp. 37-54.

[2] MDA home page, http://www.omg.org/mda/

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

[4] Semantics of a Foundation subset for executable UML models (FUML) , ptc/2009-10-05, 2009, http://www.uml.org

[5] T. Haubold, G. Beier, and W. Golubski, “A pragmatic UML-based meta model for object-oriented code generation,” in *Proc. of 21st Int. Conf. on Software Eng. & Knowledge Engineering*, SEKE’09 , 2009, pp. 733-738.

[6] R. Pilitowski, A. Derezińska, “Code generation and execution framework for UML 2.0 classes and state machines,” in T. Sobh (eds.) *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, Springer, 2007 pp. 421-427.

[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] IBM Rational Rhapsody, http://www-01.ibm.com/software/awdtools/rhapsody/ (visited 2010).

[9] http: *Sparx Systems. Enterprise Architect*, http://www.sparxsystems.com.au/ products/ea/index.html, (visited 2010).

[10] IBM Rational Software Architect, http://www-01.ibm.com/software/awdtools/swarchitect/ (visited 2010).

[11] A. M. Dziekan, “Context-aware services in the IP Multimedia Subsystem: social networks modeling and implementation,” MSc Thesis, Warsaw Univ. of Technology, Institut of Telecommunications, 2008 (in polish).

[12] M. Szczykulski, “C# code generation from UML 2.1 class diagram and state machine diagrams using FXU”, Bach. Thesis, Warsaw Univ. of Technology, Inst. of Computer Science Poland, 2009 (in polish).

[13] Unified Modeling Language Superstructure v. 2.2, OMG Document formal/2009-02-02, 2009, http://www.uml.org

[14] A. Derezińska, R. Pilitowski, “Event processing in code generation and execution framework of UML state machines,” in L. Madeyski at al. (eds.) Software Engineering in Progress, Nakom, Poznań, 2007, pp. 80-92.

[15] A. Derezińska, M. Szczykulski, "Tracing of state machine execution in model-driven development framework", A. Konczakowska, M. Hasse (Eds), *Gdansk Univ. of Techn. Faculty of ETI Annals, Information Technologies*, Vol. 18, No 8, Gdańsk 2010, pp. 199-204, and *IEEE Proc. of the 2nd Int. Conf. on Inform. Techno. ICIT'2010*, 2010, pp. 109-112.

[16] A. Niaz, J. Tanaka, “Mapping UML statecharts into Java code”, In *Proc. of the IASTED International Conference on Software Engineering*, Acta Press, Anheim, Calgary, Zurich, 2004, pp. 111-116.

[17] Rosenberg J.: RFC 3856 – *A Presence Event Package for the Session Initiation Protocol (SIP*). 2004, http://tools.ietf.org/html/rfc3856

[18] Eclipse Open Source Community, http://www.eclipse.org

[19] A. Derezińska, R. Pilitowski, “Realization of UML class and state machine models in the C# code generation and execution framework,” *Informatica* vol. 33, no 4, Nov. 2009, pp.431-440.