Finite State Machine Based Object-Oriented Applications Development Using UML and Eclipse Platform

Vadim Gurov, Maxim Korotkov, Maxim Mazin

eVelopers Corp.

Abstract

This paper describes methodology and tool for developing object-oriented applications using Eclipse platform, *UML* and finite state machine paradigm.

Described methodology is based on [1, 2] and suggests to model finite state machines using Class and Statechart *UML* diagrams, then to convert diagrams content into *XML* state machine description and finally to execute resulting *XML* using special finite state machine interpreter. For visual modeling purposes *UML* editor was created as Eclipse plugin using Graphical Editing Framework [3]. *UML* editor supports such originally text-oriented features as "markers" and "quick fixes".

Described tool was implemented as UniMod Project that may be accessed via *URL* http://unimod.sf.net.

1 Introduction

CASE tools are very popular now. They allows to describe application model with a help of set of different diagrams. Diagrams may be converted into target programming language code late on.

In *UML* there are means to model both static application structure using Class diagrams and application behavior using Statechart, Collaboration and Sequence diagrams.

UML itself doesn't define object-oriented applications modeling methodology, but only defines set of diagrams. There are number of methodologies [4, 5, 6, 7, 8] for application modeling. They have good formal description of approach for modeling static application structure, but there is no acceptable formal description of application behavior modeling process. Semantic relation with program behavior code is absent. Also, if application has complicated logic, Statechart diagrams as proposed by UML authors could hardly be used because of identifiers. Moreover there are limits on state machines nesting.

In [1, 2] methodology called SWITCH-technology for modeling behavior of event-driven applications with explicit state emphasis was suggested. Key feature of this methodology is that entity behavior described with a help of finite state machine. Finite state machine defined using labeled transition graph with special very compact notation for labels.

This paper describes methodology for designing object-oriented application behavior based on SWITCH-technology and plugin for Eclipse platform that supports created methodology.

2 Methodology

First step during creation of methodology was to adapt SWITCH-technology for object-oriented approach and *UML*.

SWITCH-technology defines two types of diagrams for describing application behavior:

- Connectivity schema;
- Transition graph.

2.1 Connectivity Schema

Connectivity schema describes relations between finite state machines, event sources and controlled entities. Event sources must be placed on the left. Controlled entities must be placed on the right. State machines must be labeled with *AR*, where *R*

– state machine number, and placed in the middle of schema.

For every event that produced by event sources link between event source and state machine is created. Link directed into state machine and labeled with short event name *eN*, where *N* – is event number, and long event description.

For every ingoing effect link between controlled entity that owns ingoing effect and state machine is created. Link directed into state machine and labeled with short ingoing effect description *xM*, where *M* - is ingoing effect number, and long ingoing effect description.

For every outgoing effect link between state machine and controlled entity that owns outgoing effect is created. Link directed into controlled entity and labeled with short outgoing effect name *zK*, where *K* – is outgoing effect number, and long outgoing effect description.

Connectivity schema may contain arbitrary amount of event sources, state machines and controlled entities.

2.2 Transition Graph

Transition graph is created for every state machine defined on Connectivity schema. Transition graph describes combined machines (C-machines or Moore-Mealy machines) [8].

Transition graph contains states and transitions between states. States labeled with it's name, list of outgoing effects short names that executed on-enter and list of included state machines names. Transitions labeled with logical condition that trigs transition and list of outgoing effects that must be executed if transition trigs. Logical condition contains logic formula in basis *AND*, *OR*, *NOT*. Terms of formula are short names of events, short names of ingoing effects and *1 (TRUE)*, *0 (FALSE)* constants.

A lot of sample projects based on SWITCH-technology may be found at SWITCH-technology home site http://is.ifmo.ru/projects_en/.

Adaptation process of described diagrams is introduced below.

2.3 Connectivity Schema Adaptation

SWITCH-technology Connectivity schema is converted into *UML* Class diagram. To do so, event source class, state machine class and controlled object class are introduced. Instances of these classes (objects) are placed on *UML* Class diagram. It's very important to note, that just objects are placed on Class diagram, because, for example, if there are two different state machines that drive the same controlled object, they must share the same instance of this controlled object in runtime.

Event sources connected with state machines with the only directed association without labels. State machines connected with controlled objects with the only directed association labeled *oP*, where *P* – is local number of controlled object from state machine point of view. State machine use this label as controlled object identifier to refer to controlled object instance. Also such approach allows two different state machines to refer to the same controlled object using different identifiers.

Controlled object class has two types of public methods: *xM()* – ingoing effect method, where *M* – is ingoing effect number, and *zK()* – outgoing effect method, where *K* – is outgoing effect number. Method name is used as short effect name. Long effect description is shown as method tag – standard *UML* extension mechanism.

Event source class has the only type of method – *eN(),* where *N* is event number. Method name is used as short event name. Long event description is shown using method tag.

State machine object has name *AR,* where *R* – state machine number. State machine class has no methods. State machine name also called state machine identifier.

2.4 Transition Graph Adaptation

SWITCH-technology Transition graph is converted into *UML* Statechart diagram. Statechart diagram syntax changed in the following way: state has only two internal transitions *enter* and *include*, state can't has concurrent regions, initial and final pseudo -states are the only pseudo-states allowed on diagram.

Enter internal transition defines list of methods of controlled objects to be executed on-enter into state. Methods of controlled objects are referred as *oP.zK(),* where *oP* is controlled object identifier defined on Class diagram (see section 2.3) and *zK()* – is controlled object method name. Labels *oP.zK()* and *oP.xK()* called *fully qualified method identifier* (*FQMI*).

Include internal transition defines list of included state machines identifiers. State machines referred as *AR* (see section 2.3).

Transition label has the following syntax: *eN[guard_condition]/list_of_FQMI*, where *eN* – is reference to method of event source, *guard_condition* – is logic formula is basis *AND*, *OR*, *NOT, list_of_FQMI* – the same list as list defined for *on-enter* state internal transition. Terms of *guard_condition* are *FQMI*, first order predicates <, >, >=, <=, !=, ==, Boolean constants and natural numbers constants.

Here is example of transition label: *e1[o1.x1&&o2.x2||o1.x10>10]/o1.z1,o2.z1*.

Statechart has number of *OCL* constraints that must be satisfied for well-specified diagram. Some additional constraints are introduced in methodology that is being described: all states on diagram must be attainable; the set of state outgoing transitions must be consistent and complete.

2.5 Methodology Process

Methodology suggests the following step-by-step process for creating application model:

- analyze problem domain, create conceptual model using classical methodologies such as [4];
- extract event sources, controlled objects and state machines from conceptual model;
- create *UML* Class diagram, put event sources on the left, put controlled objects on the right, put state machines in the middle, create associations between event sources, state machines and controlled objects. This Class diagram called *Connectivity diagram*;
- assign names to links between state machines and controlled objects using identifiers like *oP;*
- define public methods of two main types for every controlled object: ingoing effects *xM()* and outgoing effects *zK()*;
- create one Statechart diagram per state machine;
- implement controlled objects and event source on Java language. Note that *eN()* methods of event sources are not needed to be implemented – they are used for visualization purposes only;
- automatically create *XML* description of application behavior model using designed Connectivity and Statecharts diagrams for further interpreting;

Next section describes Eclipse plugin that supports described methodology.

3 Eclipse plugin

To support described methodology Eclipse plugin was developed. Plugin consists of two main parts:

- Core - Eclipse independent part;
- Eclipse plugin.

Core, in it's turn, consists of:

- finite state machine meta-model;
- algorithms for parsing and translation of guard conditions using ANTLR library [9];
- algorithms for state machines validation (includes *UML OCL* contains validation and additional constraints described in section 2.4).
- transformers between in-memory state machine model representation and *XML* state machine description;
- framework for runtime *XML* description interpreting.

XML description runtime interpreter operates in the following way: on interpreter start-up, state machine *XML* description is converted into in-memory state machine model once and completely; resulting system consists of runtime environment and object representation of state machine; to handle events this system analyzes next event and ingoing effects to chooses the transition, executes outgoing effects, invokes included state machines.

Eclipse plugin implements visual editor for Connectivity (see fig. 1) and Statechart diagrams (see fig. 2) using Graphical Editing Framework [3]. Plugin shows structure of model that is being developed as tree using *Outline* view (see fig. 3).

Plugin also starts validation process in background and reschedules it on every model change. Every found validation error is put into Problems view as marker (see fig. 4), graphical elements associated with error are outlined (see fig. 5).

A number of quick fixes are available for every found validation error. For example, for unattainable state suggested quick fixes will include such quick fix as "add transition from some attainable state to this one" (see fig. 6), for state that has incomplete set of outgoing transitions suggested quick fixes will include "create new transition labeled with rest condition to make transitions set complete" (see fig. 7)

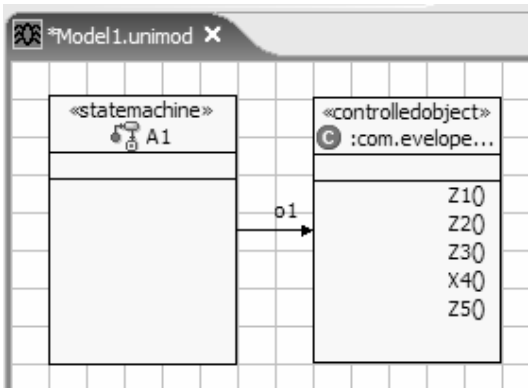

Figure 1: Connectivity diagram

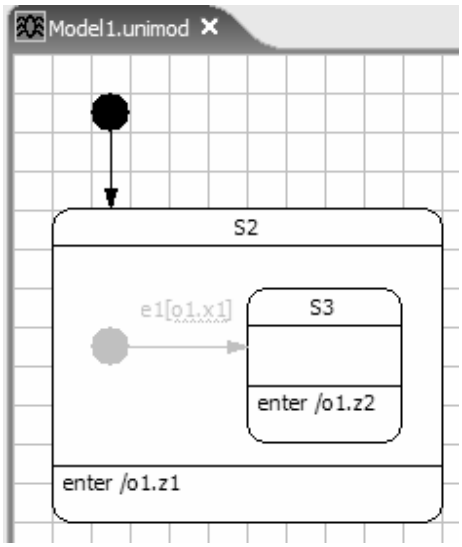

Figure 2. Statechart diagram.

In any moment of design process plugin allows to start interpreter to see how created model works. Behind the scene plugin converts diagrams content into state machine *XML* description and starts external process with interpreter passing path to generated *XML* to it.

Sometimes the diagram may be made easier to read by running the Layouter. Layouter tries to place diagram elements better (to minimize the number of intersections, to avoid overlaying states, to minimize area of the drawing and to match some other criteria). At present moment the simple modification of force-directed method [10] is used. In future we are going to adapt algorithm for orthogonal graph drawing described in [11] for our purposes.

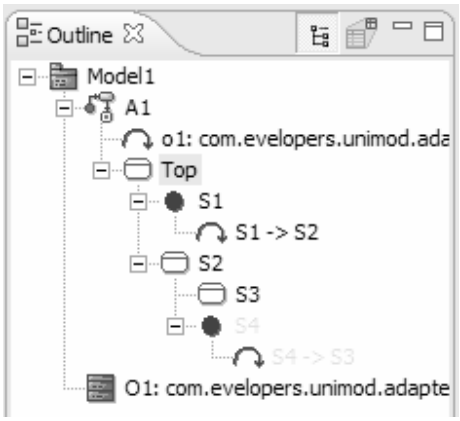

Figure 3. Outline view.

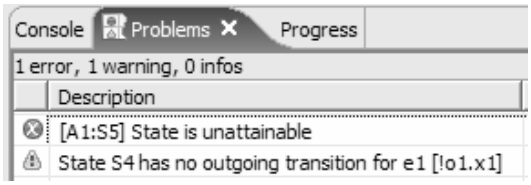

Figure 4. Problems view.

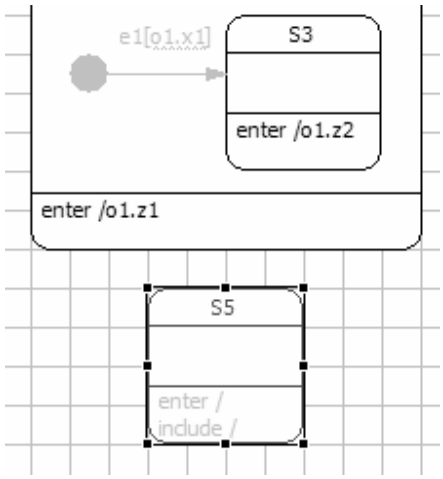

Figure 5. Elements with errors.

4 Conclusions

Developed methodology and Eclipse plugin implement complete design and development environment for modeling and executing object-oriented applications logic, allowing to reduce amount of manual programming and to increase quality of resulting code because of advanced on-the-fly model validation.

Created behavior diagrams used not only for design purposes, but as “graphical programs” - it

helps to remove the gap between design and development.

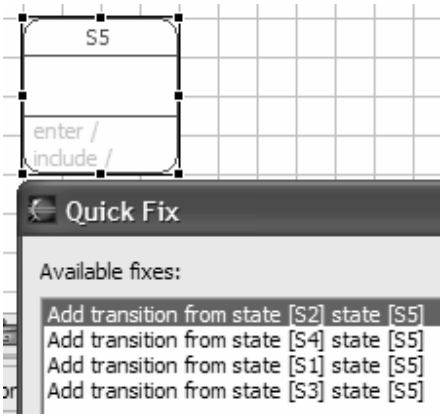

Figure 6. Unattainable state.

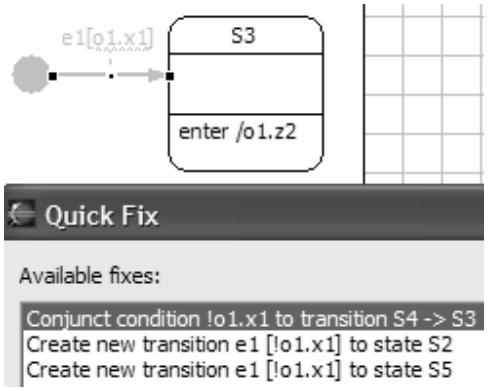

Figure 7. Incomplete set of transitions.

Acknowledgements

The author would like to thank Anatoly Abramovitch Shalyto for creating SWITCH-technology.

About the Author

Vadim Gurov is a Senior Software Developer in eVelopers Corp. and a post graduate student in Saint-Petersburg State University of Information Technologies, Fine Mechanics and Optics. His Internet address is vgurov@evelopers.com.

Maxim Korotkov is a Software Developer in eVelopers Corp. and a student in Saint-Petersburg State University of Information Technologies, Fine Mechanics and Optics. His Internet address is mkorotkov@evelopers.com.

Maxim Mazin is a Software Developer in eVelopers Corp. and a student in Saint-Petersburg State University of Information Technologies, Fine Mechanics and Optics. His Internet address is mmazin@evelopers.com.

References

[1] A.A. Shalyto. Logic Control and "Reactive" Systems: Algorithmization and Programming. Automation and Remote Control, Vol. 62, No. 1, 2001, pp. 1-29. Translated from Avtomatika i Telemekhanika, No. 1, 2001, pp. 3-39

[2] A.A. Shalyto. Switch-tekhnologiia: algoritmizatsiia i programmirovanie zadach logicheskogo upravleniia. Sankt-Peterburg: Nauka, 1998.

[3] Eclipse Graphical Editing Framework. http://eclipse.org/gef/.

[4] G. Butch. Object-Oriented Analysis and Design with Applications. Pearson Education, 1993.

[5] J. R. Rumbaugh, M. R. Blaha, W. Lorensen, et al. Object-Oriented Modeling and Design. Prentice Hall. 1990.

[6] C. Larman. Applying UML and Patterns. Prentice Hall PTR, 1997.

[7] P. Coad. Object Models: Strategies, Patterns, and Applications. Prentice Hall PTR, 1997.

[8] J.J. Odell. Advanced Object-Oriented Analysis and Design using UML, New York: SIGS Books, 1998.

[9] T.J Parr., R.W Quong. ANTRL: A Predicated-LL(k) Parser Generator. Software - Practice And Experience. 1995, №25(7). p. 789-810.

[10] T. M. J. Fruchterman, E. M. Reingold: Graph Drawing by Force Directed Placement. Software - Practice And Experience. 1991, №21(11). p. 1129-1164.

[11] G. D. Battista, P. Eades, R. Tamassia, I.G. Tollis: Graph Drawing. Prentice Hall PTR, 1999.