IEEE BT02/C16/CE08/COM19/CAS04/EM14 Societies Chapters' SECTION

Proceedings of St. Petersburg IEEE Chapters. Year 2005.
International Conference "110 Anniversary of Radio Invention".
SPb ETU "LETI". 2005. V. 2, pp. 106-110.

UniMod: Method and Tool for Development of Reactive Object-Oriented Programs with Explicit States Emphasis

Vadim S. Gurov, Maxim A. Mazin, Andrey S. Narvsky, Anatoly A. Shalyto

Abstract: **Modern software development processes usually outline phases of requirements gathering, design, coding and testing. Design phase outgoing artefact is system model, described with a help of diagrams and text. On coding phase, developers, using system model, implement system for target platform. There is a semantic gap between design and coding phases, because there's no formal relationship between system model and resulting program code. This paper describes method for development of reactive object-oriented programs, which resolves described problem. Method is based on *SWITCH-technology* and *UML*. For visual modelling purposes, *UML* editor was created as plug-in for Eclipse platform. Editor implements interactive model validation, errors highlighting, errors quick fixes, auto-completion of labels on diagrams, model execution in one click and model visual debugging,.**

***Index Terms*: Executable *UML*, *Statechart*, *SWITCH-technology*, *Eclipse*.**

I. INTRODUCTION

CASE tools are very popular now. They allow to describe program model using of set of different diagrams. Diagrams may be converted into target programming language code late on.

UML is the most popular modelling language now. *UML* allows to model both static program structure using Class diagrams and program behaviour using Use Case, Statechart, Activity, Collaboration and Sequence diagrams.

UML itself doesn't define method for object-oriented programs modelling, but only defines diagrams notation and semantics. There are number of methodologies [4, 5, 6, 7, 8] for modelling with *UML*. They have good formal description of approach for modelling static application structure, but have no acceptable formal description of application behavior modelling process. The most useful *UML* diagram for modelling behaviour is Statechart diagram. For big programs it could hardly be used with notation proposed in *UML*, because of: incompact notation for events, guard conditions and actions; there's no expressive way to show state machines collaboration; *UML* doesn't define clear operational semantics (or interpretation rules) for Statechart diagram.

Described disadvantages do not allow to create formal behavior models and isomorphously generate source code from them.

In [1, 2] methodology called *SWITCH-technology* for modelling behaviour of event-driven programs with explicit state emphasis was suggested. Key feature of this methodology is that entities behaviour is described using finite state machines. Finite state machine is defined using labelled transition graph with compact notation for labels.

This paper describes method for designing object-oriented programs behaviour. Method is based on *SWITCH-technology* and *UML*. To support created method, special tool was developed, that will be described too.

II. METHOD

To create method, *SWITCH-technology* was adapt for object-oriented programming and *UML* diagrams notation.

SWITCH-technology defines three types of diagrams for describing program model:

- Connectivity schema - describes connections between event providers, state machines and controlled objects;
- State machines collaboration schema - describes relationships between state machines;
- Transition graph - created for every state machine and describe its behaviour.

These types of diagrams will be briefly descried below.

A. CONNECTIVITY SCHEMA

Connectivity schema describes relations between finite state machines, event providers and controlled objects. Event providers must be placed on the left. Controlled objects 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. Note, that term “controlled object” here is being used not in the meaning of object-oriented programming.

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

For every input action link between controlled object that owns input action and state machine is created. Link directed into state machine and is labelled with name of input action - *xM*, where *M* - is input action number, and input action description.

For every output action link between state machine and controlled object that owns output action is created. Link directed into controlled object and labelled with name of output action - *zK*, where *K* – is output action number, and output action description.

Connectivity schema may contain arbitrary amount of event providers and controlled entities, but may contain only one state machine. If more then one state machine should be specified in the model, connectivity schema is created for every state machine. Also in this case additional diagram is created – state machines collaboration schema, which describes relationships between state machines.

B. TRANSITION GRAPH

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

Transition graph contains states and transitions between states. States are labelled with it’s name, list of output actions that will be executed on-enter and list of included state machines names. Transitions are labelled with Boolean condition that trigs transition and list of output actions that will be executed if transition trigs. Boolean condition contains logic formula in basis *AND*, *OR*, *NOT*. Terms of formula are short names of events, names of input actions and 1 (*TRUE*), 0 (*FALSE*) constants. Also, formula may contain first order predicates that analyses states of another state machines (for example, predicate ($y1 == 2$) means that current state of state machine A1 is being checked).

List of output actions may contain instruction to send event to another state machine.

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.

C. CONNECTIVITY SCHEMA ADAPTATION

SWITCH-technology Connectivity schema is converted into *UML* Class diagram. To do so, event provider stereotype, state machine stereotype and controlled object stereotype 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 controlled object in runtime.

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

Controlled object class should have two types of public methods: *xM()* –input action method, where *M* – is input action number, and *zK()* – output action method, where *K* – is output action number. Method name is used as action identifier. Action description may be defined as method tag – standard *UML* extension mechanism.

Event providers class should define set of public class fields *eN* – event identifier, where N – is event number. Event description a be defined used field *UML* tag as well.

State machine object has name *AR*, where *R* – state machine number. State machine class has no methods. State machine object name is used as state machine identifier.

If program model contains more then one state machine, it's suggested to put all state machines on one class diagram and show relationships between them on it. So, if in *SWITCH-technology* it's necessary to create one diagram per state machine plus state machines collaboration diagram, in suggested method only one class diagram should be created for all state machines.

D. TRANSITION GRAPH ADAPTATION

SWITCH-technology Transition graph is converted into *UML* Statechart diagram. Statechart diagram syntax changed in the following way: state may have two types of 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 output actions to be executed on-enter into state. Output actions are linked to methods of controlled objects using identifiers in form *oP.zK,* where *oP* is controlled object identifier defined on Class diagram (see previous section) and *zK* – is controlled object method name. Labels *oP.zK* and *oP.xK* are called *fully qualified method identifier* (*FQMI*).

Include internal transition defines list of included state machines identifiers. State machines referred as *AR*.

Transition label has the following syntax: *eN[guard_condition]/list_of_FQMI*, where *eN* – is reference to public field of event provider, *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 an example of transition label: *e1[o1.x1&&o2.x2||o1.x10>10]/o1.z1,o2.z1*.

In *UML*, *Statechart* has number of *OCL* constraints that must be satisfied for well-specified diagram. Some additional constraints are introduced: all states on diagram must be attainable; set of state outgoing transitions must be consistent and complete.

E. MODELLING PROCESS

Method suggests the following step-by-step process for creating program model:
- analyze problem domain and create conceptual model using classical methods such as [4];
- extract event providers, controlled objects and state machines from conceptual model;
- create *UML* Class diagram, put event providers on the left, put controlled objects on the right, put state machines in the middle, create associations between event providers, 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 two sets of public methods in every controlled object: set of methods that implements input actions *xM()* and set of methods that implements output actions *zK()*;
- define set of public class fields *eN* for every event provider;
- create one Statechart diagram per state machine;
- implement controlled objects and event providers manually using target programming language;
- automatically create model *XML*-description, that describes diagrams content, for further interpreting or generate source code for target programming language for further compiling and executing;

Next section describes tool that was created to support described method.

III. TOOL

To support described method, special tool named *UniMod* was developed. Tool consists of two main parts:
– Core;
– *Eclipse* plug-in.

Core consists of:
– finite state machine meta-model;
– algorithms for parsing and translation of guard conditions using *ANTLR* library [9];
– algorithms for state machines validation;
– transformers between in-memory state machine model representation and *XML*-description of state machine;
– converters of model into source code;
– framework for *XML*-description interpreting;
– model debugging engine;

XML description 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; after converting, interpreter initialises all event providers and they start to generate events to state machines; to handle events, system analyses event and incoming actions to chooses the transition; when transition is chosen, output actions written on transition are executed, state machines, that are included in target state of transition are invoked.

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

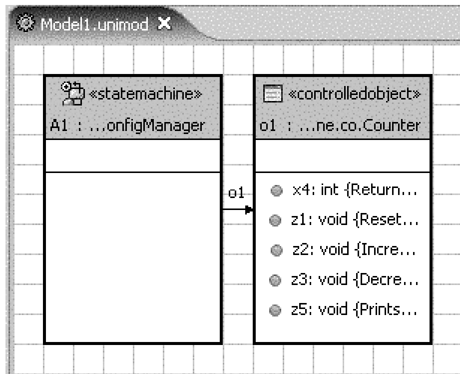


Fig. 1 - Connectivity diagram

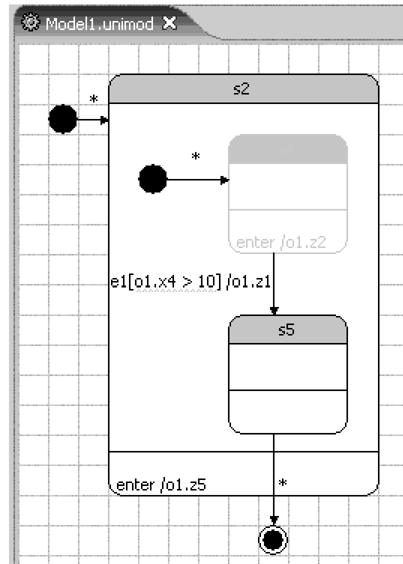


Fig. 2 - Statechart diagram

Plug-in 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.

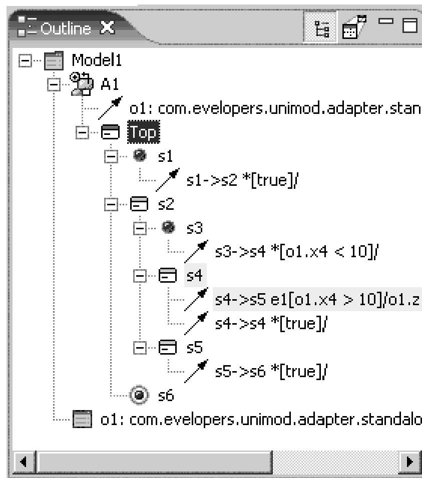


Fig. 3 - Outline view

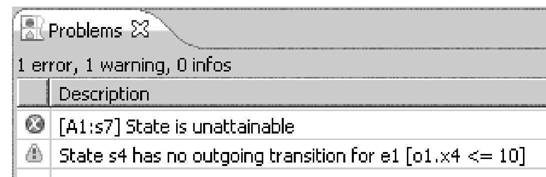
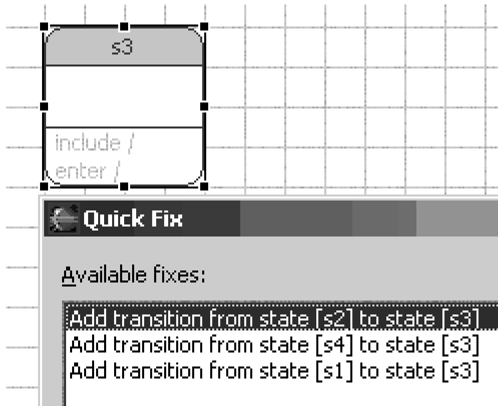


Fig. 4. - Problems view

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. 5), for state that has incomplete set of outgoing transitions suggested quick fixes will include “create new transition labelled with rest condition to make transitions set complete” (see fig. 6)



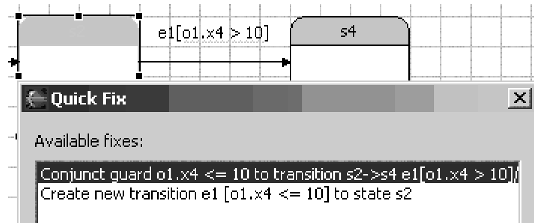


Fig. 5 - Unattainable state

Fig. 6 - Incomplete set of transitions

In any moment of design process plug-in allows to start interpreter to see how created model works. Behind the scene plug-in 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, algorithm for orthogonal graph drawing described in [11] is planned to be adapted for Statecharts.

Traditional debugging of programs is based on operator-by-operator code tracing with variables' values analysis. In *UniMod* application behavior is defined using *UML Statechart* diagram, so debugging is based on state-by-state diagram tracing with trigger events, guard conditions and output actions analysis. Debugger allows to set graphical breakpoints on states, transitions, output and input events. On fig. 7 breakpoints are set on state s2, on-enter output action o1.z1 in state s2, on input action o1.x4 in guard condition on transitions from state s2 to s3, on transition from state s3 to final state.

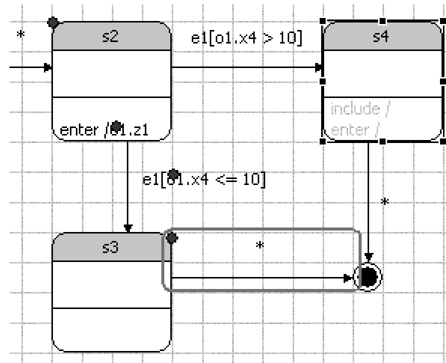


Fig. 7 – Graphical breakpoints

During debug session (see fig. 8) current state machine execution position is outlined on diagram and is shown in Debug window execution stack. As in usual textual programming languages, there is ability, to make execution step on diagram or to start program till next breakpoint hit.

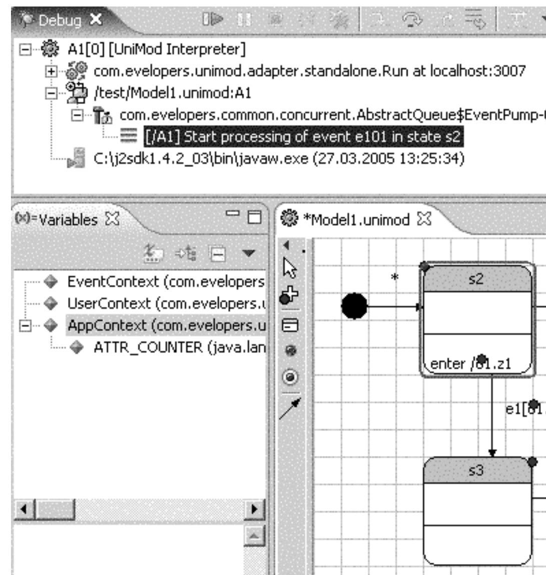


Fig. 8 – Debug session

IV. APPLICATION OF METHOD AND TOOL

Originally, described method and tool was used in *eVelopers* Corporation for *J2EE* Web application development. Client page flow modelled with a help of Statechart. A number of commercial projects was successfully developed.

Next, a set of *J2SE* applications was developed. As an example, client-server application was developed. This application implemented simple analogue of *ICQ* system. Application consists of client and server parts, both parts were designed using proposed method and implemented using described tool.

For *J2EE* and *J2SE* application interpretation approach was used. Next some amount of *J2ME MIDP* and *Symbian C++* application was developed using compiled approach – model was automatically converted into source code (*Java* and *C++*).

Also it's necessary to note, that last version of visual editor contains such features as visual model debugging and auto-completion of labels on diagrams. These features was developed using described method and previous version of tool.

V. CONCLUSION

Described method allows to close the semantic gap between phases of design and coding.

Developed *Eclipse* plug-in implements complete design and development environment for modelling, executing ad debugging object-oriented programs with explicit states emphasis, allowing to reduce amount of manual programming and to increase quality of resulting code because of interactive model validation. Model validation process marks error elements on diagrams and suggests possible resolutions to fix error.

Finally, an important point is that *UniMod* project is an attempt to design and implement programming language of the next generation — graphical programming language — based on *SWITCH-technology*, *UML* notation and *Eclipse* platform. *UniMod* is a successful attempt to port code assist technologies to diagrams.

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.

ABOUT THE AUTHORS

Vadim S. Gurov is a Senior Software Developer at the *eVelopers* Corporation. He has a master's degree in Computer Science from the St.Petersburg State University of Information Technologies, Mechanics and Optics and now is a post-graduate student in the same university. His research passion is model-driven development, applied *UML* and Web application modeling. He has 10 years of software development experience with focus on object-oriented programming. His email address is vgurov@evelopers.com.

Maxim A. Mazin is a lead developer in *UniMod* project at *eVelopers* Corporation. He has a bachelor's degree in applied mathematics from the St.Petersburg State University of Information Technologies, Mechanics and Optics and now is a graduate student (master degree) at the same university. The subject of his bachelor's degree research dedicated to state machine validation and verification algorithms. He is focused on automata-based approach in programming research. He is interested in technologies that increase development tools usability. His email address is mmazin@evelopers.com.

Andrey S. Narvsky is Chief Executive Officer of *eVelopers* Corporation. He is a candidate of technical sciences. He has more then 30 publications. His email address is anarvsky@evelopers.com.

Anatoly A. Shalyto is Head of Programming Technologies Department in Saint-Petersburg State University of Information Technologies, Mechanics and Optics. He is a doctor of technical sciences. He is creator of *SWITCH-technology*. He has more then 250 publications. His email address is shalyto@mail.ifmo.ru.