ISSN 0361-7688, Programming and Computer Software, 2007, Vol. 33, No. 6, pp. 343–355. © Pleiades Publishing, Ltd., 2007.
Original Russian Text © V.S. Gurov, M.A. Mazin, A.S. Narvsky, A.A. Shalyto, 2007, published in Programmirovanie, 2007, Vol. 33, No. 6.

Tools for Support of Automata-Based Programming

V. S. Gurov[a], M. A. Mazin[a], A. S. Narvsky[a], and A. A. Shalyto[b]

[a] *eVelopers Corporation, ul. B. Raznochinnaya 14-5, St. Petersburg, 197110 Russia*
[b] *St. Petersburg State University of Information Technologies, Mechanics, and Optics, Kronverkskii pr. 49, St. Petersburg, 197101 Russia*
e-mail: vadim.gurov@gmail.com

Received August 1, 2006

Abstract—A method for designing and implementing reactive object-oriented programs with explicit emphasis of states is suggested. The method relies on the automata-based programming (SWITCH-technology) and the UML notation. The UniMod tool based on this method, which is a plug-in module for the Eclipse platform, is described.

DOI: 10.1134/S0361768807060059

1. INTRODUCTION

To improve abstraction level of software development tools, a new direction in software engineering [1]—model-driven engineering (MDE) [2]—is being evolved.

This direction includes model-driven development (MDD), which is also referred to as model-driven design [3, 4]. A variant of the MDD is the model-driven architecture (MDA) [5], which is being developed by the Object Management Group (OMG).

When using the MDA, models of program systems are represented by means of the unified modeling language (UML) [6].

Over a period of years, this language was used only for representing models; however, the idea of *executable* UML is becoming very popular recently [7, 8]. This is explained by the fact that, in the majority of cases, practical use of UML is limited to modeling of only static part of programs by means of class diagrams and generating of program skeleton by them. This is not sufficient for effective software design.

The modeling of dynamical aspects of programs in UML is difficult because of the lack of formal unambiguous description of interpretation rules (operational semantics) for the behavior diagrams in the language standard.

Besides, none of the great number of methods for designing object-oriented systems described in [9] clearly explains how to bind static diagrams with dynamical ones.

Currently, in spite of the existence of many tools for automated transformation of behavior diagrams (state diagrams) into codes in various programming languages [10], the widely used design tools, such as, for example, Sun Studio Enterprise [11], are lacking such functionality.

Some tools have graphical editors for constructing these diagrams; however, the possibility of code generation by them is lacking.

Jacobson, one of the developers of UML, listed four most important trends in the process of software development in his talk [12].

He noted that the technological basis of the development of object-oriented software, which consists of UML and the standard development process RUP (rational unified process) [13], is well known. According to Jacobson, UML is taught in more than 1000 universities worldwide. In his opinion, the next step should be wide adaptation of UML and RUP.

As applied to the subject of this paper, the following two trends mentioned by Jacobson should be noted.

1. Executable UML. Currently, UML is used basically as a specification language for system models. The existing UML means make it possible to build various diagrams and to automatically create code "skeletons" for the target programming language (e.g., in Java or C#) by a class diagram. Some of these means make it possible also to automatically generate a program behavior code by the state diagrams. However, it should be noted that this functionality exists in only an inchoative stage, since the known tools cannot efficiently relate model behavior, which can be described by means of diagrams of four kinds (state, activity, cooperation, and sequence diagrams), and the generated code.

The lack of unambiguous operational semantics in the case of the traditional program development results in different descriptions of behavior in the model and in the program, as well as in arbitrary interpretation of the behavior diagrams by the programmers. Moreover, the behavior description in the model is often informal. The opposite situation, when the model is formal and its implementation is heuristic, is also possible. It often happens that the formal model is constructed by an architect and the programmer does not rely on it but writes the program on its own.

The appearance of operational semantics will make diagram understanding unambiguous and will allow one to construct an executable UML for which the code (in the ordinary sense of this word) may not be generated. This can be achieved owing to direct model interpretation.

2. The process of software development should be active. The existing development tools need much time to be studied. Jacobson believes that the development means should envision the developer's actions and suggest variants of solving encountered problems depending on the current context. Note that a similar approach is implemented in many modern development environments (e.g., Borland JBuilder, Eclipse, and IntelliJ IDEA) for text programming languages (but not for UML).

The admission of the fact that programs should not be written haphazardly but rather be carefully designed with the help of development tools of a high level of abstraction resulted in the appearance of new trends in software design, such as model-driven design and visual software design [14].

2. REACTIVE SYSTEMS

A wide class of software systems is that of reactive systems, i.e., systems that perform certain actions in response to external events. It is shown in [15] that such systems can nicely be modeled by means of extensions (for example, through the use of nested states) of finite automata transition diagrams called statecharts. For construction of such diagrams and code generation by them, special tools have been created. A list of such tools is presented in [10]; it includes, in particular, I-Logix Statemate (http://ilogix.com/sublevel.aspx?id=74), XJTek AnyState (http://www.xjtek.com/anystates/), StateSoft ViewControl (http://www.statesoft.ie/products.html), SCOPE (http://www.itu.dk/~wasowski /projects/scope/), IAR Systems visualSTATE (http:// www.iar.com/p1014/p1014_eng.php), and The State Machine Compiler (http://smc.sourceforge.net/) [16–21].

There also exist other tools for code generation by these diagrams (see, e.g., [22]).

The disadvantage of these tools is that they allow one to build and implement only the behavior part of the program model without regard to the program statics. Therefore, the **program on the whole cannot be constructed** by means of these tools.

It follows from the above said that the static part of the program can be constructed by means of one set of tool, whereas the dynamic part, by means of other tools. Therefore, the task of creation of tools for development of object-oriented programs on the whole remains open.

3. EXECUTABLE UML

As has already been noted, to solve the above-specified problem, executable UML, which will combine static and dynamic diagrams, is being developed.

One of the approaches to solving this problem is development of a *UML virtual machine* [23–25].

In the project [24], the model of software system is constructed as follows: the program structure is modeled by means of the UML class diagrams, and the behavior, by means of the description of each method of each class in the form of a UML sequence diagram. Such an approach is very inconvenient in the case of a complicated application logic, since it results in very cumbersome models.

In the project [25], UML is extended by a text platform-independent imperative language for action description, which results in overloading graphical diagrams by text information.

Speaking of industrial developments, the idea of an executable UML is implemented in the Telelogic TAU2 product [26]. However, since this is not an open-source project, it presents no interest from a scientific standpoint. IBM Rational Rose and Borland Together are also commercial closed-source tools.

Therefore, of interest is the project with the open-source code UniMod (http://unimod.sourceforge.net), which underlies this work.

Further in this paper, we describe an executable graphical language based on the use of the UML notation, an automata-based programming method, and the UniMod tool supporting this method.

4. EXECUTABLE GRAPHICAL LANGUAGE AND METHOD OF CONSTRUCTION OF AUTOMATON PROGRAMS BASED ON IT

A method of program design with explicit separation of states called the SWITCH-technology, or the automata-based programming, was proposed in [27]. Further, this method was extended to event systems [28] and to object-oriented systems [29].

A specific feature of this method is that programs are constructed in the same way as the automation of technological (and not only technological) processes is carried out. In the course of the latter, a connectivity scheme is first constructed, which contains sources of information, control system, controlled objects, and feedbacks from the controlled objects to the control system. In the proposed approach, the control system is implemented as a system of interacting finite automata each of which has several inputs and outputs.

The SWITCH-technology defines for each automaton two types of diagrams (a connectivity scheme and transition graph) and their operational semantics. If there are several automata, a scheme of their interaction is also constructed. For each diagram type, an appropriate notation is used (http://is.ifmo.ru/?i0=science\& i1=minvuz2).

In this work, having preserved the automaton approach, we propose to use the UML notation when constructing diagrams in the framework of the SWITCH-technology. In so doing, with the use of the UML class diagram notation, connectivity schemes are constructed, which determine interface of the automata, and the transition graphs are constructed with the help of the UML state diagram notation. If there are several automata, the scheme of their interaction is not constructed, and all of them are shown in the diagram of classes. The diagram of classes (as a connectivity scheme) and the state diagrams form the suggested graphical language.

To design programs on the basis of this language, the following method is proposed.

• Based on the analysis of the problem domain, a conceptual model of the system is developed, which determines entities and relationships between them.

• Unlike in traditional approaches of object-oriented programming [9], all entities are classified into event sources, controlled objects, and automata. The event sources are active: they affect the automata on their own initiative. Controlled objects are passive: they perform actions having received commands from the automata. Controlled objects can also form values of input variables for the automata. The automata is activated by the event sources; based on the values of input variables and current states, affect the controlled objects; and transit to new states.

• By using the class diagram notation, a connectivity scheme is constructed, which specifies interface of each automaton. The left part of this scheme shows the event sources; the central panel depicts the automata; and the right panel, the controlled objects. The event sources connect to the automata by means UML associations and supply them with events. The automata connect to the objects they control, as well as to other automata, which are either invoked by the former or nested in their states.

• In addition to the automaton interface, the connectivity scheme performs a function typical of the class diagram; namely, it specifies the object-oriented structure of the program.

• Each controlled object contains methods of two types, namely, methods implementing input variables (`xj`) and those implementing output actions (`zk`).

• For each automaton, by means of the state diagram notation, the transition graph of the *Moore–Mealy* type is constructed. The arcs of this graph are labeled by events (`ei`), by a Boolean formula relating the input variables, and by the output actions formed on the transitions.

• In the vertices, the output actions performed upon entering the state and the names of nested automata that are active while the state is active may be specified.

• In addition to the interaction between the nested automata, the automata can interact through calls. In this case, the calling automaton passes an event to the automaton being called, which is indicated in the transition or in the vertex as an output action. In the latter case, the event is sent to the automaton being called upon entering the state.

• Each automaton has one initial state and an arbitrary number of terminal states.

• The states in the transition graph may be simple or complex. If a state is nested into another state, the latter is called complex. Otherwise, the state is said to be simple. A specific feature of the complex states is that similar arcs from the nested states are replaced by one arc originating from the complex state.

• All complex states are unstable, whereas all simple states, except for the initial one, are stable. If there are complex states in an automaton, an incoming event may result in more than one transition. This is because any complex state is unstable, and the automaton performs transitions until the first simple (stable) state is attained. Note that, if there are no complex states in the transition graph, then, like in the SWITCH-technology, not more than transition is performed in each automaton run.

• Each input variable and each output action are methods of the corresponding controlled object, which are implemented manually in the target programming language. The sources of the events are also implemented manually.

• The use of symbolic notation in the transition graphs makes it possible to compactly describe complex behavior of the designed systems. The meaning of such symbols is specified by the connectivity scheme. The placement of the cursor on a symbol results in a pop-up prompt with its text description.

The proposed method makes it possible to **design the program on the whole.**

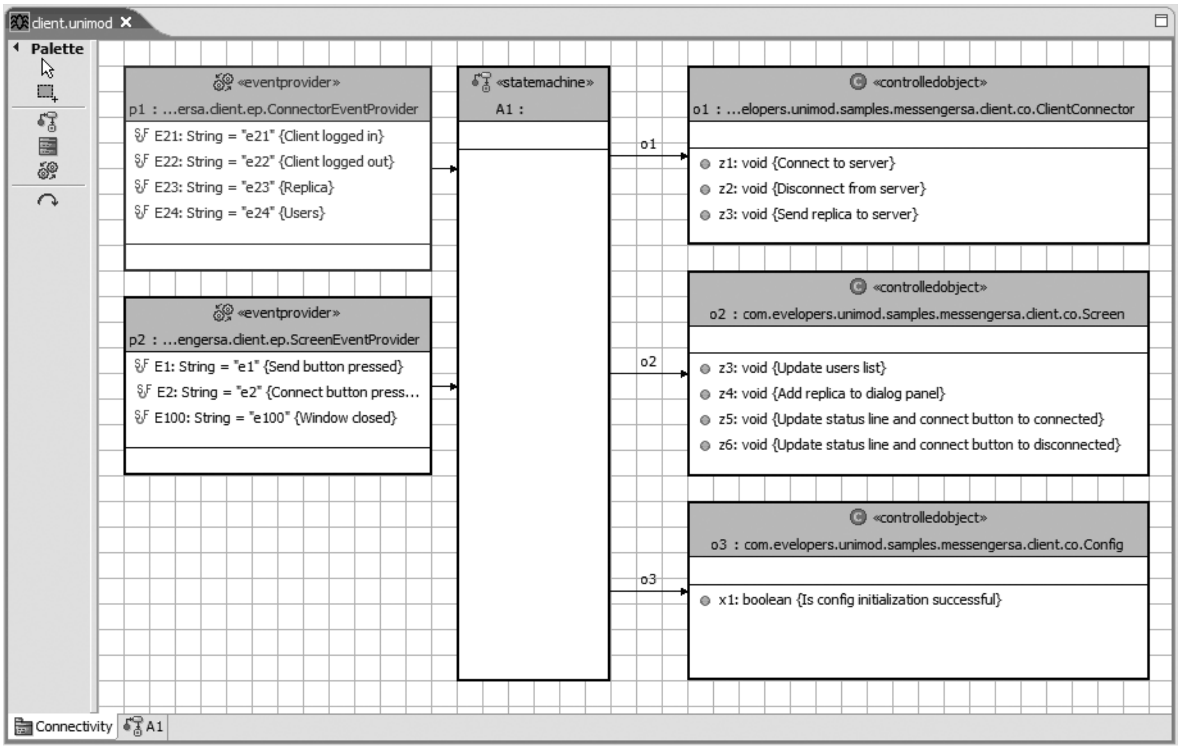
Figure 1 shows an example of a connectivity scheme, and Fig. 2, its transition graph.

Now, we describe the syntax and operational semantics of the proposed graphical language.

4.1. Syntax

For the text programming languages, syntax is usually described by means of formal grammars. UML is a graphical language and uses another approach: first, a metamodel defining a set of correct models is described; then, graphical primitives corresponding to the elements of the metamodel are defined. The diagrams are constructed from these primitives. The UML metamodel itself is described by means of MetaObject Facility (MOF) [30], a high-level means for specifying metamodels.

As noted earlier, the suggested graphical language uses UML diagrams of only two types and, hence, not



client.unimod

Palette

«eventprovider»
p1 : ...ersa.client.ep.ConnectorEventProvider
E21: String = "e21" {Client logged in}
E22: String = "e22" {Client logged out}
E23: String = "e23" {Replica}
E24: String = "e24" {Users}

«eventprovider»
p2 : ...engersa.client.ep.ScreenEventProvider
E1: String = "e1" {Send button pressed}
E2: String = "e2" {Connect button press...
E100: String = "e100" {Window closed}

«statemachine»
A1 :

o1
o2
o3

«controlledobject»
o1 : ...elopers.unimod.samples.messengersa.client.co.ClientConnector
z1: void {Connect to server}
z2: void {Disconnect from server}
z3: void {Send replica to server}

«controlledobject»
o2 : com.evelopers.unimod.samples.messengersa.client.co.Screen
z3: void {Update users list}
z4: void {Add replica to dialog panel}
z5: void {Update status line and connect button to connected}
z6: void {Update status line and connect button to disconnected}

«controlledobject»
o3 : com.evelopers.unimod.samples.messengersa.client.co.Config
x1: boolean {Is config initialization successful}

Connectivity A1

Fig. 1. Example of an automaton connectivity scheme.

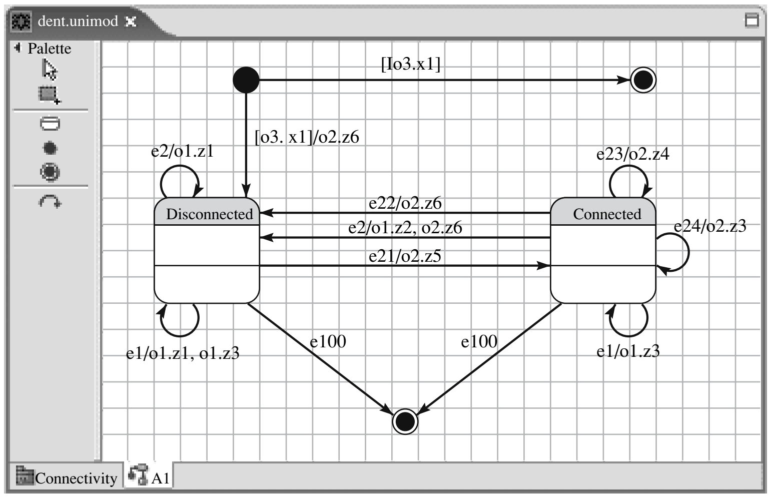


Fig. 2. Example of an automaton transition graph.

all elements of the metamodel. A formal description of the UML metamodel subset used is a list of elements of the metamodel. Such a description occupies too much space and is difficult to read. Therefore, below is a meaningful description of this subset.

In the framework of the UniMod project, a system model consists of one class diagram, which depicts classes with the following stereotypes: *EventProvider*, an event source; *StateMachine*, an automaton; and *ControlledObject*, an object of control. Between these stereotypes, there may exist directed associations (arcs with arrows of certain kinds) of the following three types: from an event source to an automaton, from an automaton to a controlled object, and from an automaton to another automaton. The associations are labeled by identifiers.

For each automaton, shown in the class diagram, it is required to create a state diagram. This diagram may depict initial, terminal, simple, and composite states. The composite states may include states of any other types.

Transitions between the states may have labels of the form

```
e1[o1.x1 && o2.x3 > 10]/o1.z1,
o2.z2, A2.e2
```

Here, `e1` is the event name, `o1` is the identifier labeling the association that leads to the first controlled object, `x1` is a method of the controlled object returning a value of the type `boolean` or `int`, `z1` is a method of the controlled object, `A2` is an identifier labeling the association that leads to the automaton being called, and `e2` is the event sent to the automaton `A2`. The square brackets contain a Boolean formula, which is the condition of the transition firing (guard condition).

Simple states may contain a string in which actions executed upon entering the state are specified. For example,

```
o1.z1, o2.z2
```

The language described does not support actions executed upon exiting the simple states.

As noted earlier, simple states may also contain a list of nested automata.

The UML states with parallel regions reflecting parallelism are not supported. This is because the design of objects with one control flow is relatively simple, whereas parallelism requires several objects (in our case, automata) executed in parallel [31].

4.2. Operational Semantics

Let us define operational semantics of the system model constructed as described above, which satisfies certain syntax.

• When launching the model, all event sources and controlled objects are initialized. Then, the event sources start to affect the automata related to them.

• Each automaton starts its operation from the initial state and stops at one of the terminal states.

• Upon receiving an event, the automaton selects all transitions from the current state that are labeled by the symbol of this event.

• The automaton searches the selected transitions and calculates the Boolean formulas written on them until it finds a formula with the `true` value.

• If a transition with such a formula is found, the automaton performs output actions written on the arc and turns to a new state. In this state, the automaton performs the output actions and launches the nested automata. If the new state turns out composite, the transition is performed from the initial state contained inside the given composite state.

• If, among the output actions, there is an automaton being called, it is called with an appropriate event.

• If a transition is not found, the automaton continues to search for a transition turning to the parent state, the state into which the current state is nested.

• When transiting to a terminal state, the automaton stops all event sources, and the system operation terminates.

A more detailed formal description of the operational semantics is given in [32].

Now, having described the graphical language based on the UML notation, its operational semantics, and the method of its use, we turn to the description of the tool for implementing the method.

5. THE UNIMOD TOOL FOR SUPPORTING AUTOMATA-BASED PROGRAMMING

The UniMod tool is designed for the development and execution of automaton programs. It allows one to create and edit UML diagrams of classes and states, which correspond to the connectivity schemes and transition graphs.

As noted earlier, the design of programs with the use of this tool consists in the following. The behavior of the application is described by a system of interacting automata given as a set of the above-mentioned diagrams constructed with the help of the UML notation. The event sources and controlled objects are implemented manually in the target programming language.

The tool under consideration supports two basic types of implementation of the diagrams constructed, namely, interpretation and compilation.

5.1. Interpretation

The interpretational approach implements the *UML virtual machine*.

The structural scheme of the interpretational approach is shown in Fig. 3.

As can be seen from Fig. 3, in the interpretational approach, **the source code is a UML model** (connectivity schemes and state diagrams by the SWITCH-technology) **and a Java code of the event sources and controlled objects.**

When launching the program, the interpreter contained in the UniMod tool loads the XML description of the model into the memory and creates instances of the event sources and controlled objects. The sources form events and direct them to the interpreter, which processes them in accordance with the logic described by the automata. The automata call methods of the controlled objects that implement input variables and output actions.

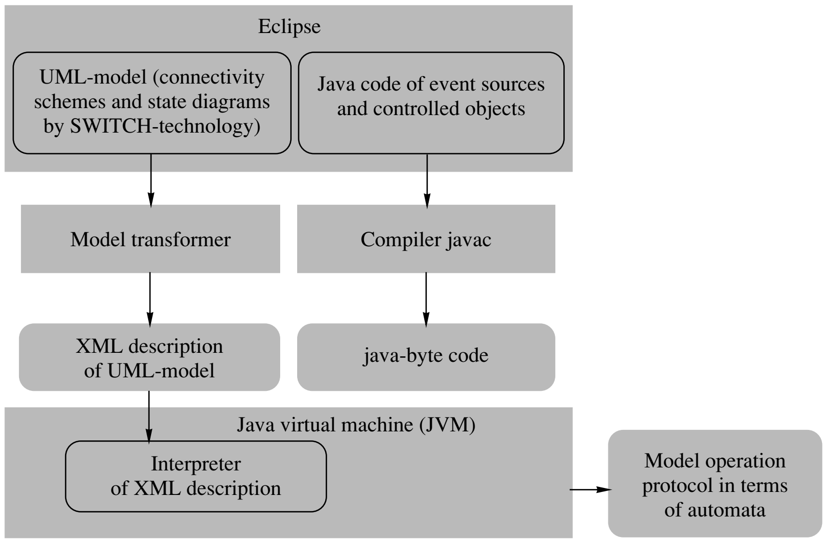

Eclipse
UML-model (connectivity schemes and state diagrams by SWITCH-technology)
Java code of event sources and controlled objects
Model transformer
Compiler javac
XML description of UML-model
java-byte code
Java virtual machine (JVM)
Interpreter of XML description
Model operation protocol in terms of automata

Fig. 3. Structural scheme of the interpretational approach.

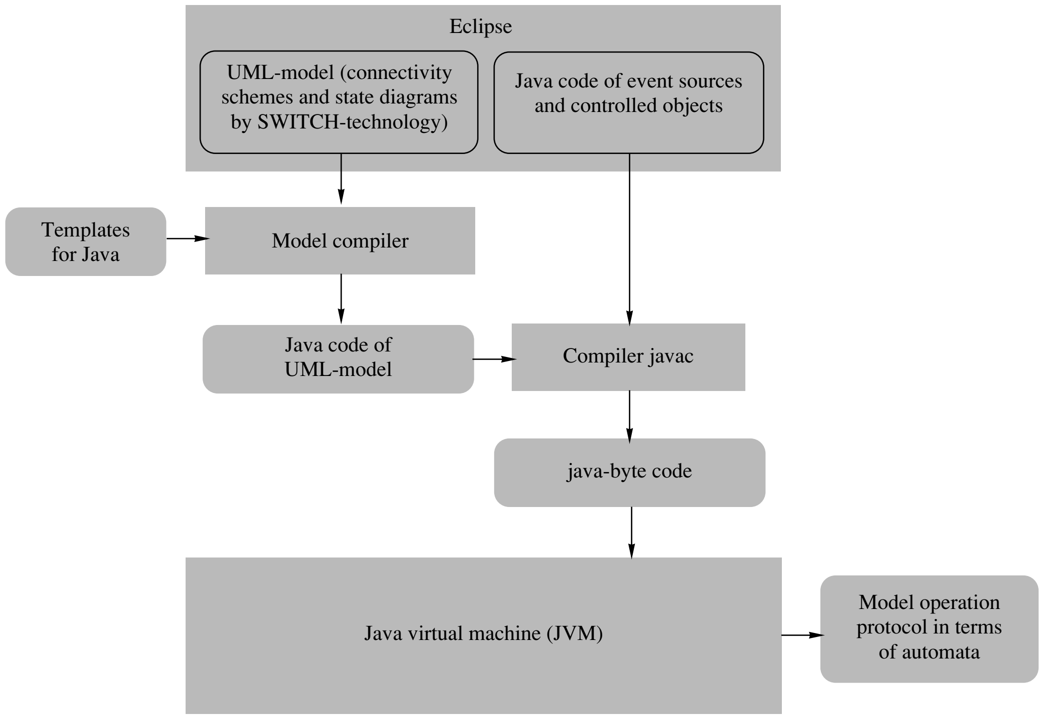
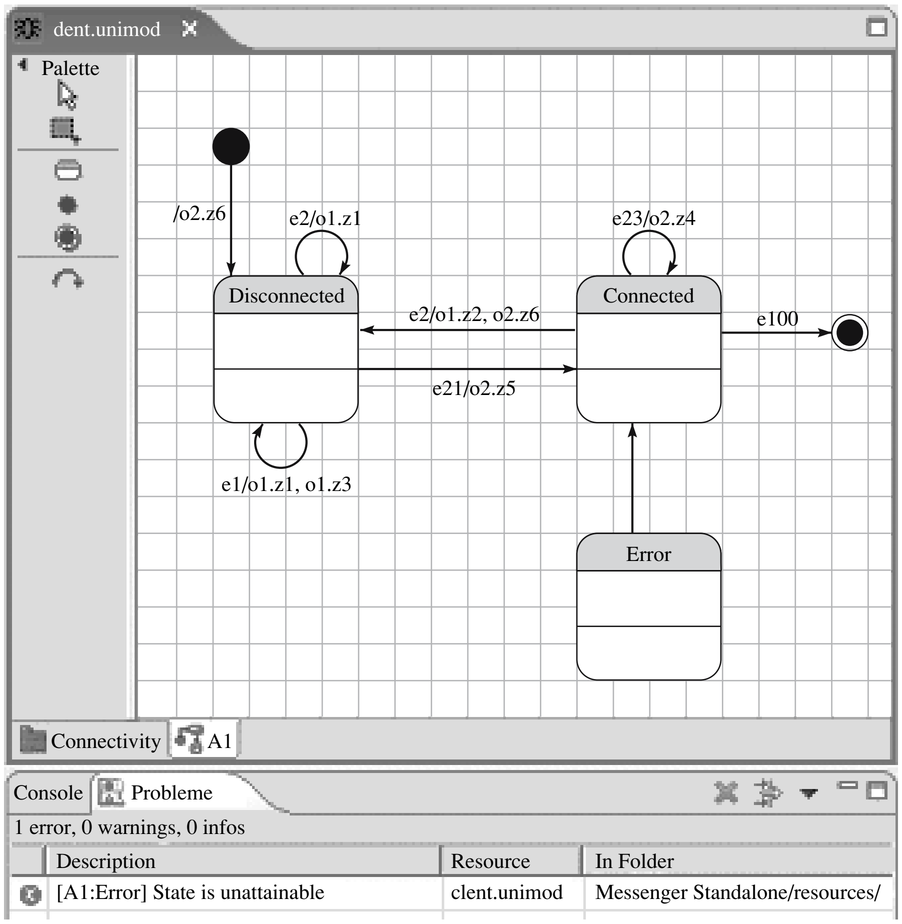

Eclipse
UML-model (connectivity schemes and state diagrams by SWITCH-technology)
Java code of event sources and controlled objects
Templates for Java
Model compiler
Java code of UML-model
Compiler javac
java-byte code
Java virtual machine (JVM)
Model operation protocol in terms of automata

Fig. 4. Structural scheme of the compilation approach.

5.2. Compilation

The structural scheme of the compilation approach is shown in Fig. 4.

In the case of the compilation approach, the UML model is transformed directly to a code in the target programming language, which is then compiled and run. When transforming to the code, *Velocity* templates are used [33]. This makes it possible to adapt the compilation approach for programming languages different from Java (for example, the authors of this paper made this for C++ used in development of applications for mobile devices).



1 error, 0 warnings, 0 infos

	Description	Resource	In Folder
✖	[A1:Error] State is unattainable	clent.unimod	Messenger Standalone/resources/

Fig. 5. Unattainable state in a transition graph.

This approach is reasonable to use in the cases of limited resources. Note that this approach is typical of the “classical SWITCH-technology.”

6. IMPLEMENTATION OF THE DIAGRAM EDITOR ON THE ECLIPSE PLATFORM

The diagram editor is a plug-in module for the Eclipse platform (http://www.eclipse.org). The advantages of this platform over IntelliJ IDEA or Borland JBuilder are as follows:

• it is a freeware with an open-source code,

• it contains the Graphical Editing Framework library for development of graphical editors,

• it is actively supported by IBM and, even now, its functionality is not less than that of the above-mentioned analogues.

To ensure the process of active development of programs in text languages, the following features are implemented in the above-listed development tools:

• highlighting of semantic and syntax errors,

• input auto-complete and correction of input errors,

• code formatting and refactoring [34],

• program debugging and execution in the development environment.

These features are called code assist. They have been implemented for editing diagrams in the developed module for the Eclipse platform.

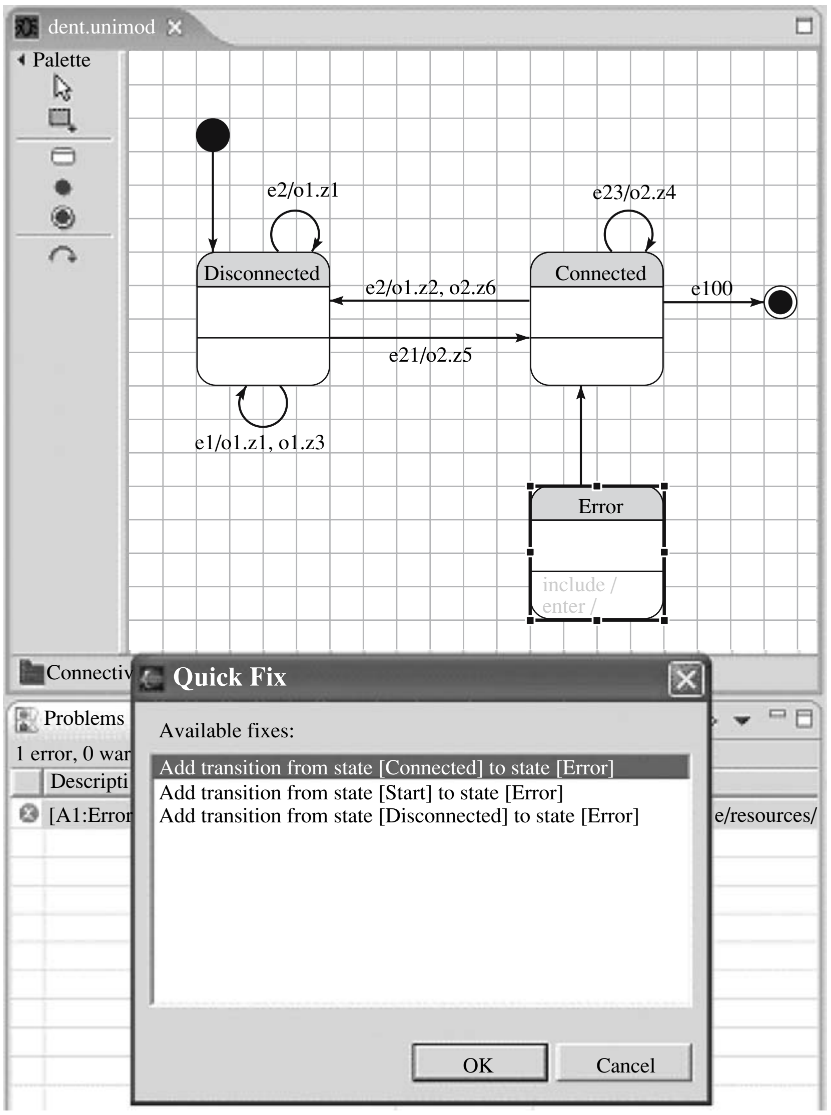
6.1. Model Validation

The editors for text programming languages check whether the program is written in a given language and highlight places containing syntax errors. Semantic errors in the case of text programming languages include, for example, use of undeclared variables, calls of non-existing methods, and incorrect type casting.

In the UML standard, the diagram syntax and semantics are defined by a set of constraints written in the object constraint language. This set of constraints must be satisfied for any correctly constructed diagram. The validation of the diagram syntax and semantics is based on just these constraints.

We suggest extending of the set of constraints as follows:

• All states in the state diagram must be attainable.



Available fixes:

Add transition from state [Connected] to state [Error]
Add transition from state [Start] to state [Error]
Add transition from state [Disconnected] to state [Error]

Fig. 6. Suggested variants of error correction in a diagram.

• The set of transitions originating from any state must be complete and consistent. The completeness means that, for any event, the disjunction of the protecting conditions on all arcs originating from the considered state is identically equal to unity. The consistency means that, when processing any event, there do no exist transitions that can be executed simultaneously (only deterministic finite automata are used).

The above-described constraints specify correctness conditions. Validation of the diagram correctness proceeds as follows. In a background mode, a process is launched that checks the above-specified conditions when the diagram is modified. If an error is found, the incorrect element is highlighted in the diagram by a color. Figure 5 shows an example of a diagram with an unattainable state.

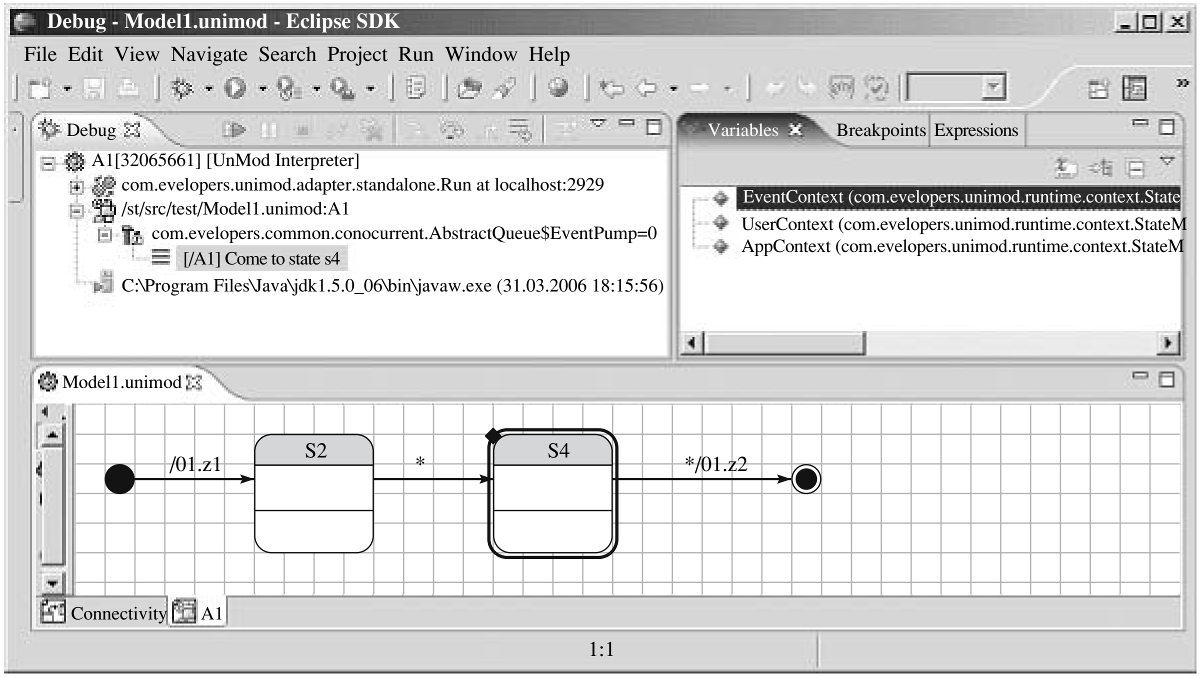
6.2. Input Auto-Complete and Correction of Input Errors

Traditionally, input auto-complete for text programming languages consists in the following: by a given beginning of a lexeme, a set of admissible constructs whose prefix is the given beginning is determined, and the user is suggested to select one of the lexemes.

In text languages, correction of input errors consists in displaying variants of correction for any error found.

In the proposed graphical language, both these approaches are used for editing labels of the transitions.

Since the proposed language contains not only text but also graphical information, correction of graphical input errors is also performed. For example, for the unattainable state in Fig. 5, the user will be advised to add a transition to this state from any attainable state (Fig. 6).


Debug - Model1.unimod - Eclipse SDK
File Edit View Navigate Search Project Run Window Help
Debug
Variables
Breakpoints
Expressions
Model1.unimod
S2
S4
/01.z1
*
*/01.z2
Connectivity
A1
1:1

```
A1[32065661] [UnMod Interpreter]
  com.evelopers.unimod.adapter.standalone.Run at localhost:2929
  /st/src/test/Model1.unimod:A1
    com.evelopers.common.conocurrent.AbstractQueue$EventPump=0
      [/A1] Come to state s4
  C:\Program Files\Java\jdk1.5.0_06\bin\javaw.exe (31.03.2006 18:15:56)
```

```
EventContext (com.evelopers.unimod.runtime.context.State
UserContext (com.evelopers.unimod.runtime.context.StateM
AppContext (com.evelopers.unimod.runtime.context.StateM
```

Fig. 7. Debugging session.

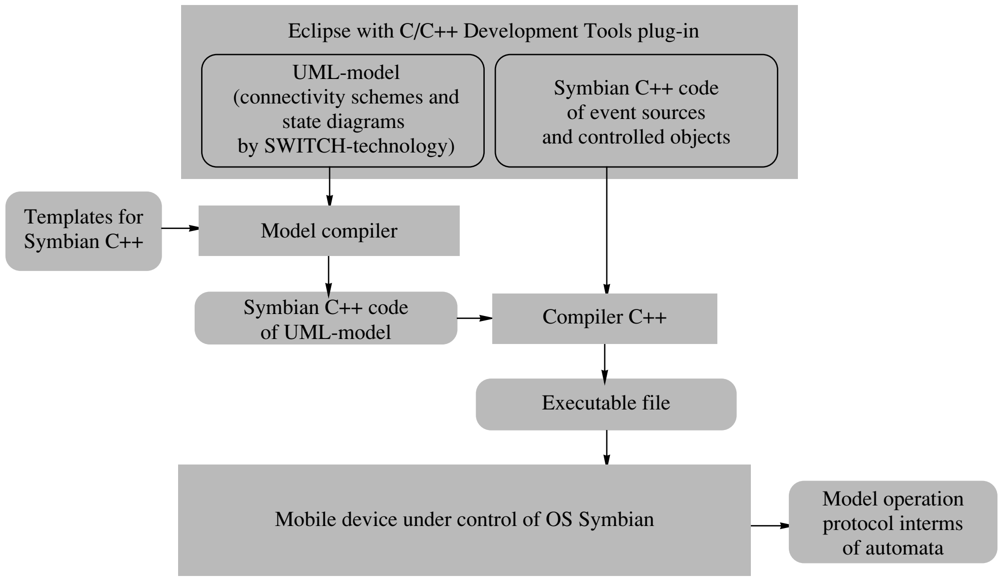

Eclipse with C/C++ Development Tools plug-in
UML-model (connectivity schemes and state diagrams by SWITCH-technology)
Symbian C++ code of event sources and controlled objects
Templates for Symbian C++
Model compiler
Symbian C++ code of UML-model
Compiler C++
Executable file
Mobile device under control of OS Symbian
Model operation protocol interms of automata

Fig. 8. Structural scheme of the compilation approach when using Symbian C++.

6.3. Formatting

Formatting of a code facilitates reading it. Many text editors format codes automatically.

As applied to the diagrams, an analogue of the code formatting is, in the authors' opinion, the diagram layout. The problem of the diagram layout is considerably more complicated than that of the code formatting, since there are no commonly accepted aesthetic criteria of the layout quality. In the UniMod project, the diagram layout is performed by the annealing method [35]. It yields satisfactory results, which can further be improved manually if needed.

6.4. Model Execution

Traditionally, the following variants of execution of programs written in text programming languages are used:

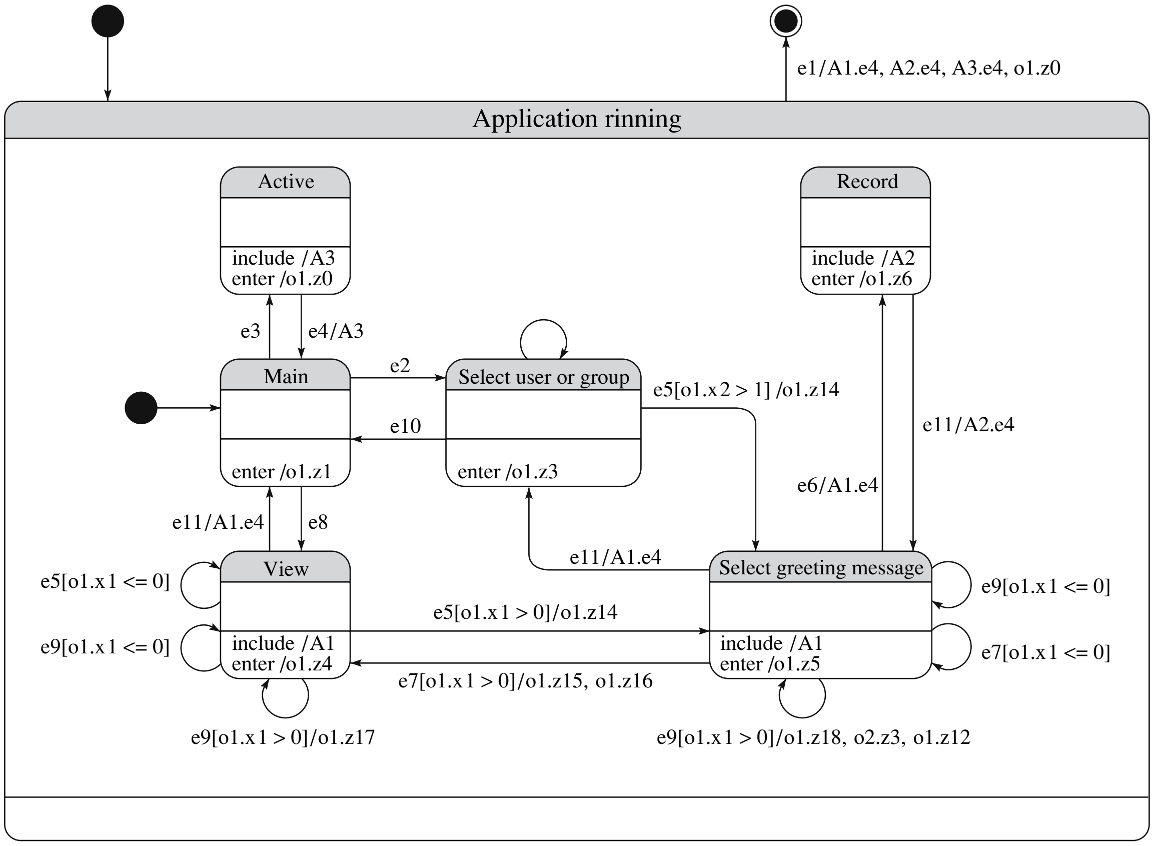


Fig. 9. State diagrams of a mobile phone answering machine.

• the text is compiled into a code executed by the operating system (Pascal, C++);

• the text is compiled into a code executed by a virtual machine (Java, C#);

• the text is executed directly by the interpreter (JavaScript, Basic).

Similar variants are used for the proposed graphical language. The second and third variants are basic ones. They are described in detail in Section 5. In some cases, the first variant is also used (see Section 7).

6.5. Model Debugging

Traditionally, program debugging is a tracing of the program code, one statement after another, with simultaneous analysis of variable values.

For the graphical automaton model, "graphical debugging" is suggested. It consists in the tracing of the transition graph accompanied by the analysis of the current state, events, and values of input variables marking the transition being analyzed. If necessary, text debugging of input variables and output actions is possible.

An example of a debugging session in the UniMod tool is shown in Fig. 7.

In the lower part of the figure, the debugged model is shown. The small circles depict stopping points. The state enclosed in the frame is the current debugger state.

7. APPLICATION OF THE PROPOSED APPROACH TO MOBILE DEVICES

It has been noted earlier that the considered tool is based on the Java language. However, in some cases (for example for mobile devices), to ensure the desired performance, C++ is used. In this case, only the compilation approach can be used.

Figure 8 shows how the structural scheme for the compilation approach (Fig. 4) is changed when C++ is used for the mobile platform Symbian (http://www.symbian.com).

It should be noted that the **development** of programs in this case is carried out in the same way as in Java. The only difference is that the C++ templates rather than those for Java are used. An example of program development in C++ with the help of the UniMod tool can be found at the address http://is/ifmo/ru/science/MD-Mobile.pdf.

For the illustration, the state diagram of the basic automaton for a mobile phone answering machine is presented in Fig. 9. This diagram contains all types of syntax constructs described above.

Thus, we can state that, although Eclipse and UniMod are designed for the Java language, the use of templates in the model compilation makes the UniMod tool a multilanguage platform. However, the graphical debugging of models for the languages different from Java is impossible.

8. CONCLUSIONS

In the paper, an approach to the creation of a tool for the automata-based programming is discussed.

This approach allows us to

• reduce the length of the code in a text programming language through the use of the graphical programming language;

• construct connectivity schemes and transition graphs suggested in the SWITCH-technology using the UML notation for the class diagrams and state diagrams, respectively, and include them into the project documentation [36];

• formally and clearly describe behavior of programs and change them, in the majority of cases, by modifying only the transition graphs;

• simplify project maintenance owing to increased centralization of program logic.

The proposed approach has been shown to possess the following advantages compared to its analogues:

• It is allowed to use a system of interrelated automata in the model, which makes it possible to decompose the behavior of a complex system into subproblems. It should be noted that each state also implements decomposition of a subproblem separating only those input and output actions that are related to this subproblem.

• The programs created on the basis of the suggested method are suitable for verification by construction. This is explained by the fact that, in the verification of behavior with the use of the Model Checking method [37], for programs written in a traditional way, it is required to construct models (for example, as a system of transitions), whereas, in the automata-based programming, models in the form of transition graphs are given upon program specification.

• The structure of the automaton programs, in which functions of the input and output actions are almost completely separated from the program logic, makes the verification of these functions on the basis of formal proofs with the use of pre- and postconditions [38, 39] practical.

• Along with the nested states, nested automata are also used; the number of which, as well as the nesting level, is not restricted.

• The automata can also interact through calls by receiving appropriate events on transitions or upon entering a state.

• In addition to the compilation approach, model interpretation is possible. In this case, the “source code” is diagrams themselves.

• The use of the Eclipse platform and the Java and XML languages makes the models and the program easily portable from one operating system to another (for example, from Windows to Linux).

• The interactive model validation makes it possible to localize many syntax, semantic, and logical errors as early as at the modeling stage.

• The program framework is specified by the tool, so that the user does not need to develop it for each application anew.

• The proposed method makes it possible to create the program on the whole.

• The project is open-source.

• Our experience shows that the use of the compilation approach in the case of complicated logic results in that more than half of the application code is constructed automatically.

The proposed operational semantics is deterministic owing to checking of the existence of conflicting transitions, which is not performed, for example, in Rational Rose and Borland Together [40]. This disadvantage presents also in the VisualSTATE tool [20]; however, here, it is eliminated by means of the SCOPE tool [19].

The source texts, documentation, and examples of using the UniMod package can be found at the website http://unimod.sourceforge.net.

In conclusion, we note that this study relies on the work [28] and the approach described in [41, 42].

Note also that, as stated in [43], the UML language obeys the “20–80 law.” Our study substantiates this assertion: out of the entire variety of UML diagrams, diagrams of only two types were used for constructing the programs. This is in accordance with Okkama’s razor principle, which says that entities should not be multiplied without need. The tool developed is used for teaching at the department of computer technologies of St. Petersburg State University of Information Technologies, Mechanics, and Optics. Student projects containing project documentation are published at the site http://is.ifmo.ru in the UniMod-projects section (e.g., [44]).

Results of this study were presented at a number of scientific conferences, such as **Methods and Tools for Information Processing,** MSU, 2005 (http://lvk.cs.msu.su/mco/part9.html), **Open Source Forum 2005** (http://www.opensource-forum.ru/rbio_view.php?num=41), **IEEE** International Conference “110 Anniversary of Radio Invention” (http://is.ifmo.ru/articles_en/_unimod.pdf), and **Software Engineering in Russia 2005** (http://secr.ru/rus/program/schedule.html). At the last conference, Jacobson in his talk noted the originality of the described approach [45], which stimulated the authors to write the paper [46].

The UniMod tool started its advance over the world. For example, a book on UML 2.0 (http://helion.pl/

ksiazki/juml2.htm) where the UniMod tool is mentioned among the tools for the Eclipse platform was published in Poland and its source texts are provided on an attached CD. It has been noted at the polytechnic institute of Turin (http://is.ifmo.ru/unimod_en/_torino.pdf) and the Borland corporation (http://www.pcweek.ru/?ID=504874) as a promising project. There is information that it is used in other organizations (http://is.ifmo.ru/unimod_en/_unimoduser.pdf).

The approach discussed in this work is close to that described in [47], which is used for designing software for important long-term systems.

ACKNOWLEDGMENTS

This paper relies on results of the work on the state contract "Technology of Automata-based Programming: Application and Tools" (http://www.fasi.gov.ru/fcp/technika/konkurs/it/izv-it-6.doc) carried out in the framework of the federal scientific–engineering program "Studies and Developments in Priority Fields of Science and Engineering" for 2002–2006.

REFERENCES

1. Sommerville, I., *Software Engineering*, Pearson Education, 2001, 6th ed.
2. Kuznetsov, S., UML 2.0: Promises and Disappointments, *Otkrytye systemy*, 2006, no. 2, pp. 75–79.
3. *1st European Conf. on Model-Driven Software Engineering*, Germany, 2003, http://www.agedis.de/conference/.
4. *Int. Workshop "e-Business and Model Based in System Design",* IBM EE/A. SPb.: SPb ETU, 2004.
5. *OMG Model Driven Architecture,* http://www.omg.org/mda/.
6. Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.
7. Mellor, S. and Balcer, M., *Executable UML: A Foundation for Model Driven Architecture*, Addison-Wesley, 2002.
8. Raistrick, C., Francis, P., and Wright, J., *Model Driven Architecture with Executable UML*, Cambridge University Press, 2004.
9. Graham, I., *Object-Oriented Methods: Principles and Practice*, Addison-Wesley, 2000, 3d ed.
10. *Wikipedia*, Finite state machine. Tools, http:// en.wikipedia.org/wiki/Finite_automaton#Tools.
11. *Sun Studio Enterprise*, http://developers.sun.com/prodtech/javatools/jsenterprise/reference/techart/whatis.html.
12. Jacobson, I., Four Macro Trends in Software Development Y2004, http://www.ivarjacobson.com/postnuke/html/modules.php?op=modload&name=UpDownload&file =index&req=getit&lid=9.
13. Jacobson, I., Booch, G., and Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.
14. Novikov, F., Visual Program Design, *Informatsionno-upravlyayushchie systemy*, 2005, no. 6, pp. 9–22, http://is.ifmo.ru/works/visualcons/.
15. Harel, D., Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Program*, 1987, vol. 8, pp. 231–274.
16. I-Logix Statemate, http://ilogix.com/sublevel.aspx?id=74.
17. XJTek AnyState, http://www.xjtek.com/anystates/.
18. StateSoft ViewControl, http://www.statesoft.ie/products.html.
19. SCOPE, http://www.itu.dk/~wasowski/projects/ scope/.
20. IAR Systems visualSTATE, http://www.iar.com/p1014/p1014_eng.php.
21. The State Machine Compiler, http://smc.sourceforge.net/.
22. Jia X. et al., Using ZOOM Approach to Support MDD, http://se.cs.depaul.edu/ise/zoom/papers/zoom/SERP_ZOOM.pdf.
23. Riehle, D., Fraleigh, S., Bucka-Lassen, D., and Omorogbe, N., The Architecture of a UML Virtual Machine, *Proc. of the 2001 Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, ACM 2001.
24. Matilda UML Virtual Machine, http://dssg.cs.umb.edu /projects/umlvm/.
25. Kennedy Carter iUML, http://www.kc.com/products/iuml/index.html.
26. Telelogic TAU G2, http://telelogic.com/corp/products/tau/g2/index.cfm.
27. Shalyto, A.A., *SWITCH-tekhnologiya. Algoritmizatsiya i programmirovanie zadach logicheskogo upravleniya* (SWITCH-Technology: Algorithmization and Programming of Logic Control Problems), St. Petersburg: Nauka, 1998, http://is.ifmo.ru/books/switch/1.
28. Shalyto. A.A. and Tukkel', N.I., SWITCH-Technology: An Automated Approach to Developing Software for Reactive Systems, *Programmirovanie*, 2001, no. 5, pp. 45–62. [*Programming Comput. Software* (Engl. Transl.), 2001, vol. 27, no. 5, pp. 260–276].
29. Shalyto, A.A. and Tukkel', N.I., Tanks and Automata, *BYTE*, Russia, 2003, no. 2, pp. 69–73, http://s.ifmo.ru/works/tanks_new/.
30. MetaObject Facility Core Speification Version 2.0. http://www.omg.org/technology/documents/formal/MOF_Core.htm.
31. Gomaa, H., Designing Concurrent, Distributed, and Real-Time Applications with UML, Addison-Wesley, 2000.
32. Gurov, V.S., Mazin, M.A., and Shalyto, A.A., Operational Semantics of UML State Diagrams in the UniMod Software Package, *Trudy XII Vserossiiskoi nauchno-metodicheskoi konferentsii "Telematika-2005"* (Proc. of the XII All-Russian Scientific Conference), St. Petersburg: SpbGU ITMO, vol. 1, pp. 74–76, http://tm.ifmo.ru/tm2005/scr/224as.pdf.
33. Velocity—Java-based template engine, http://jakarta.apache.org/velocity/index.html.
34. Fowler, M., Brant, J., Opdyke, W., and Roberts, D., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
35. Fruchterman, T.M.J. and Reingold, E.M., Graph Drawing by Force Directed Placemen, *Software—Practice and Experience*,1991, vol. 21, no. 11, pp. 1129–1164.

36. Shalyto, A.A., A New Initiative in Programming: The Demand for Open Project Documentation, *PC Week/RE,* 2003, no. 40, pp. 38–42, http://is.ifmo.ru/works/open_ doc/.

37. Clarke, E., Grumberg, O., and Peled, D., *Model Checking*, MIT, 2000.

38. Dijkstra, E.W., Notes on Structured Programming, *Structured Programming,* Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R, Eds., London: Academic, 1972.

39. Meyer, B., *Object Oriented Software Construction*, Prentice-Hall, 1997.

40. Borland Together, http://www.borland.com/us/ products/together/index.html.

41. Gorshkova, E.A. and Novikov, B.A., Use of Statechart Diagrams for Modeling Hypertext, *Programmirovanie*, 2004, no. 1, pp. 64–80 [*Programming Comput. Software* (Engl. Transl.), 2004, vol. 30, no. 1, pp. 47–51].

42. Gorshkova, E.A., Novikov, B.A., Belov, D.D., Gurov, V.S., and Spiridonov, S.V., A UML-Based Modeling of Web Application Controller, *Programmirovanie*, 2005, no. 1, pp. 44–51 [*Programming Comput. Software* (Engl. Transl.), 2005, vol. 31, no. 1, pp. 29–33].

43. Eckel, B., *Thinking in Java*, Prentice-Hall, 2002.

44. Parashchenko, D.A., Tsarev, F.N., and Shalyto, A.A., Modeling Technology Based on Automata-based programming for One Class of Multiagent Systems on the Example of Game "Competition of Flying Saucers," http://is.ifmo.ru (UniMod Projects Section).

45. Shalyto, A.A., Two Meetings with I. Jacobson, http://is.ifmo.ru/aboutus/uml_ph/, http://is.ifmo.ru/belletristic/jacobson/.

46. Gurov, V., Narvsky, A., and Shalyto, A., Executable UML from Russia, *PC Week/RE*, 2005, no. 26, pp. 18–19, http://is.ifmo.ru/works/_umlrus.pdf.

47. Regan, P. and Hamilton, S., NASA's Mission Reliable, 24–32, http://www.osp.ru/os/2004/03/045_print.htm.