2005 International Conference on "Integration of Knowledge Intensive Multiagent Systems. KIMAS '05: Modeling, Exploration, and Engineering". USA, MA: IEEE, 2005, pp. 460–465.

Methods of Object-Oriented Reactive Agents Implementation on the Basis of Finite Automata

Anatoly Shalyto, shalyto@mail.ifmo.ru
Lev Naumov, naumov@rain.ifmo.ru
George Korneev, kgeorgiy@rain.ifmo.ru
St. Petersburg State University of Information Technologies,
Mechanics and Optics
Computer Technologies Department
Sablinskaya street 14, St. Petersburg, Russia

Abstract—*This paper gives an overview of different automata implementation methods in the framework of object-oriented agent-based systems development. The general idea of this paper is the application of the finite automata to the reactive agents' development [1]. Support of several mechanisms of automata interactions allows using discussed methods for multi-agents systems implementation. So, this paper describes an approach to solution of one of the paramount problems of object-oriented programming – definition of connections between static and dynamic properties of object-oriented systems.*
Described methods are used in projects, which were developed in the framework of Foundation for Open Project Documentation [2].

1. INTRODUCTION

In paper [3] authors offered an approach to agents' implementation as automata for logical control systems.

Moreover authors offered to consider multi-agent systems as systems of interacting automata.

Before this, in 1991, Anatoly Shalyto offered an approach to logical control systems development, which was called "automata-based programming" or "Switch-technology" [4]. In English it was introduced and published for the first time in the paper [5].

Switch-technology is based on the concepts of "state", "input variables" and "output actions". Other basic concepts are compiled combining these terms. So, after uniting "states" and "input variables" we will get the concept of "automaton without output". In the same manner, after uniting the concept of "automaton without output" with "output actions" we will get the "automaton". Such kind of "automata" in the theory of automata are called "structural automata".

For development of systems with complicated behavior logic should be separated or shared by set of automata. So, Switch-technology supports several approaches to automata interactions and intercommunications. List of these approaches follows:

- interaction between automata by exchange of state numbers;
- intercommunication between automata, when one is invoked from (by) another one;
- intercommunication between automata, when one is nested in another one.

These three approaches are enough for separating logic between automata when solving tasks of arbitrary complexity.

When using Switch-technology, automata form the language of logic specification. So, logic can be described in the terms of "automata", "states", "transitions" and so on.

Switch-technology allows to build automata' source code formally and isomorphically to connection diagrams, which describe automata' interfaces, and transition diagrams, which define automata' functionality. So this process can easily be automated.

To distinguish states they are encoded with the help of single variable. This variable can take some integer value from the definite set. As a result, states are observable, so such kind of automata can be used not just for lexical analysis, but for purposes of control in any tasks.

Written in the framework of Switch-technology, programs are "living" in the terms of automata.

Important feature of "automata-based programming" is record-keeping - building logs while program is working. This mechanism provides "airborne recorder"-like functionality.

2. FOUNDATION FOR OPEN PROJECT DOCUMENTATION

In the industry of software development it is always needed to give a detailed description of the existing code for the person with average level of qualification. This description has to cover program development and its static and dynamic properties. The documentation should be understandable by everyone, who can be involved in project.

It is always good to have original source code, but the problem is that in most cases it is not enough. Understanding of any non-trivial program requires additional documentation. Code analysis for restoration of the initial project solutions and program understanding are two important branches of the technology of programming. For example, try to understand structure of the compiler if you have no definition of formal language, which it compiles.

Everybody, who has participated in large-scale software reengineering projects, remembers the sense of helplessness, which occurs when you see the heap of badly documented (but, may be, very good written) source code.

Availability of the source codes does not help when there is no access to key solutions' developers. If program is written, for example, in C programming language (relatively low-level one) and its documentation leaves much to be desired then all project solutions dissolve in the coding details. In such situations the value of high-level documentation like specifications, interfaces definitions and architecture description may raise the value of source code!

Lack of source codes for program understanding results in creation of methods, which unite code developing and documenting.

One of the most famous attempts of such solutions was taken by D. Knuth in his book "Literate Programming".

Probably, the most well-known prohibited book in the history of computer science was "Commentary on Unix. With Source Code", which contains high-level explanation of source codes of Unix operating system even with description of used algorithms. This book has been copied and distributed illegally in xerocopies for more then twenty years form the moment of publication in 1977!

Switch-technology solves this problem also. Verbal descriptions, automata definitions (connection diagrams and transition diagrams), source code, verification logs and, if necessary, class diagrams and structural diagrams of classes form "project documentation", which is good enough for understanding the functionality of software with complicated logic, for using and reusing it.

One of the authors (Anatoly Shalyto) declared "Foundation for Open Project Documentation" [2] on the opening of North-Eastern European semifinal competitions of ACM International Collegiate Programming Contest (Saint-Petersburg, November 2002). For support and propagation of the foundation site http://is.ifmo.ru was created.

At the Computer Technologies Department of Saint-Petersburg State University of Information Technologies, Mechanics and Optics the special pedagogical experiment began [2]. Students were divided into nearly 60 groups (one or two persons in each group). Each group was to develop some project, using automata-oriented programming technology. Usage of automata allows not only to specify the problem formally, but also verify software in terms of automata.

3. AGENTS, OBJECTS, AUTOMATA

Purpose of this paper is in reviewing different methods of object-oriented automata implementation, which were developed and used during this pedagogical experiment, mentioned above. Application of these methods to reactive agents' development is a key point. Support of several mechanisms of automata interactions allows to use the discussed methods for multi-agents systems implementation.

Reactive agents are widely used [1] for multi-agent systems construction. One of the most popular mathematical models for building agents of this class is a finite automaton. In paper [3] the special technology for such agents' implementation was suggested, but the procedural approach to programming was used.

In paper [6] this approach was further developed to allow designing and developing software for reactive systems. Described method is the procedural one that is why it was called "state-based procedural programming".

In paper [7] previous approach was extended to cover object-oriented programming that is why it was called "state-based object-oriented programming". In this approach automata should be implemented as member-functions of classes.

During a pedagogical experiment, described in paper [2], more than sixty projects (for the moment of this paper preparation) were developed using state based object-oriented programming.

Design and implementation of these projects caused the creation of many methods of automata implementation that significantly differed from the approach suggested in paper [7].

4. CLASSIFICATION OF METHODS

In the current paper these methods will be classified and described briefly. Their enumeration and comments follow.

1. Automata, as classes' member-functions [7]. This approach is very similar to procedural programming style, so it can be called as "the envelopment of automata into classes".

2. Automata, as classes without usage of base class that implements fundamental automata functionality [8].

3. Automata, as classes using base class that implements fundamental automata functionality. This approach is based on the combined usage of object-oriented and automata-oriented programming styles benefits. Using this method, automata should to be developed as descendants of a special class that provides necessary base functionality. This class and, optionally, necessary additional classes are to be compiled into a special library that can be used and, possibly, extended by the developer.

   3.1. Paper [9] describes one of the simplest possible libraries of such kind that provides all necessary functionality for implementing multi-agent systems with arbitrary complexity in the framework of state based object-oriented programming.

   Using this library, designing of each automaton consists of creation, which is based on verbal

description (declaration of intents), the connections diagram (description of the interface) and the transitions diagram (description of the behavior, dynamic properties). Source code can be automatically generated through these documents.

In terms of object-oriented paradigm, automata are to be implemented as descendants of basic class `Automaton`. This class implements basic and additional functions of automata.

Basic automata functions, implemented in class `Automaton`, are:

- providing actions execution at transition diagram's vertices (for Moore automata), at it's transition (for Mealy automata), and both for vertices and transitions (for Moore-Mealy automata, so named, mixed automata);
- providing automata interactions:
  - invokes automaton with specified event;
  - implementation of nested automata (calls from the inner one to the outer one and vice versa);
  - states' numbers interchange.

Important note is that if the first mechanism of automata interactions acts only "top-down", second and third mechanisms can be carried out in both directions: "top-down" and "bottom-up".

Class `Automaton` implements following additional automata functions:

- automatic building of logs:
  - when automaton starts in some state with some event;
  - when automaton changes its state from one to another;
  - when automaton stops in some state;
- description of input and output actions in logs manually, with verbal information, specific for given action or activity.

Descendant classes redefine some of parent's member-functions and add functions that correspond to input actions (events and variables), internal variables, output actions, objects to be controlled, nested and invokable automata.

Paper [9] suggests approach that was illustrated with the example of an elevator controlling system that is to be functionally equivalent with to Knuth's solution of the same task, described in "The Art of Computer Programming". Created program *Lift* is published on web-site http://is.ifmo.ru in "Projects" section.

This program was written in object-oriented style. It is rather handy to develop such kind of software on personal computers and they can be easily ported to the platform of PC-like controllers. But, as a matter of fact, microcontrollers are used in controlling systems extremely wide. Unfortunately compilers from object-oriented languages do not exist for microcontrollers (or, at least, for the overwhelming majority of them). So, procedure-oriented style of software developing is used for this kind of computing devices.

In paper [9] a special method of porting state based object-oriented programs written in *C++* to programs, written in *C*, using the framework of state-based procedural programming is suggested.

Obviously, only porting of programs' skeleton, fragment of code, excluding visual interface part and implementation of input and internal variables and output actions, is meant here.

This method was illustrated with the example of porting elevator's controlling systems skeleton to the platform of microcontroller *Siemens SAB 80C515*. It was made, using *Keil μVision 2* as the development environment. Resulting program is also published on the web-site http://is.ifmo.ru in "Projects" section.

In the paper [10] rather similar approach was suggested.

3.2. In paper [11] library *STOOL* (*Switch–Technology Object Oriented Library*) was introduced. In this library, not only automaton, but also all of its logical constituent parts (states, input actions, output actions, transitions and others) have their own base classes.

Another important feature of *STOOL* is that it allows using automata for multithreading software developing.

Automata should be implemented not as member-functions, but as descendants of a special base class `Auto`. This is a more common method to automata-based programs development. Mentioned approach (automata as member functions) can be brought to the second one (automata as classes), but not vice versa.

Class `State` represents concept of automaton's state. Class `Info` is used for providing automaton's description. It is used for building logs automatically.

Each automaton can execute no more than a single transition at startup.

Two variants of algorithms implementation were examined in that paper:

- automata are implemented inside `while`-like loop operators;
- automata are implemented directly, without the loop operator.

Automata of the first type are extremely useful for the implementation of various algorithms. Automata of second type are suitable for reactive agents' implementation.

Class's `State` overloaded operators, `operator int()` and `operator=(int)` allow dealing with object, that represents automatons state, as if it was an integer variable.

Using object instead of integer variable allows bringing all functions that do not provide main functionality (transition function, input variables and output actions) out of `switch` operator. Therefore two advantages are achieved:

- it is possible to determine and single out the "global state" of the whole system;
- it is possible to implement "actions" and "activities".

The suggested approach allows keeping the `switch` operator even in the object-oriented implementation. This operator allows implementing automata' transition diagrams integrally, formally and isomorphically.

3.3. In paper [12] one more library for object-oriented automata implementation was suggested. This library was called *Auto–Lib*. Paper also provides examples of the library's usage.

3.4. Authors of paper [13] suggest library that allows "assembling" of simple automata, using descendants of base classes as bricks. These classes implement concepts of "automaton's state" and "transition between states".

This library can supply system with isomorphism between its source code and transition diagram even if group states (so named, "metastates") exists in it.

3.5. In order to dispose reentrance (recurrent calls to main automaton's functions before leaving it after previous call), in paper [14] method of "delayed automaton's call" was offered. The idea is that one member-function of basic class stores all the events, which were sent to some automaton, in a special queue and also handles it in an independent thread (separate for each automaton). So, when using this method, the amount of threads equals to the amount of automata, in contrast to approach, offered in paper [3], where there is single thread always.

4. Usage of design patterns [14]. Side by side with usage of libraries for object-oriented automata implementation, design patterns can be also developed, used and reused.

4.1. Pattern *Automat*, described in paper [15], allows designing and implementing software, using classes, which implement following concepts: "state", "transition", "condition on transition", "action", and "automaton". Class, that implements the last concept, is the base class for developer's automata classes. This class contains the fundamental logic.

4.2. Usage of pattern *State*. This pattern was described in book [14]. It implements abstraction "state". For concrete state's implementation developers have to redefine transition function in it.

Similar approach was examined in paper [16]. In this paper, for each automaton it was necessary to develop base class for the state and then inherit particular states from that class. Transitions between states are provided by base classes, but their execution performs in the descendants.

4.3. As a culmination of design patterns and automata joint usage, pattern *State Machine* was developed [17]. Main advantages and features of this one are following:

- it allows to develop separate independent classes (for example, one class that represents some concrete state could be used in different automata);
- when using *State* pattern, transitions' conditions will be distributed between classes, which represent "states". So logic is not centralized. When using *State Machine* these logic will be assembled in "context";
- pattern *State Machine* keeps from duplicating of interfaces.

5. Dynamical automata definition.

5.1. A lot of automata building methods are static – automata should be described with some source code before execution. Such description is constant. Source code is to be interpreted or compiled and executed somehow. In papers [18, 19] the method of dynamical automata definition is presented. These methods allows to implement automata with unknown beforehand amount of states. So automata' connection and transitions diagrams can be changed dynamically at runtime.

All means (classes and basic functionality engines) for such kind of automata modification are to be gathered into developer libraries.

5.2. Successful alliance of object-oriented and automata-based approaches allows to use one very significant ability. When all automaton's functionality is encapsulated in single class, opportunity to create arbitrary amount of instances of this automaton, which can control some device (object, agent), communicating with self-similar automata, allows to build complicated multi-agent systems, where agents (or part of these agents) are identical.

6. Implementation of automata using interpretation.

6.1. In paper [20] method of automated conversion of transition diagrams into textual description in *XML* format was suggested. Special environment for execution of such descriptions was developed using *Java* language (so it is platform-independent).

6.2. First of all mentioned description should be fully converted into internal object program

representation at startup. System that consists of two parts: execution environment and object representation of program is formed. Each input and output action needs to be implemented manually, to provide automaton with necessary functionality.

When some event occurs, the system analyzes input variables and executes output actions. After that it applies to nested automata.

6.3. Papers [21, 22] describe software *UniMod* (official web-site http://unimod.sourceforge.net), that represents plug-in for the *Eclipse* environment and implements approach, described in the previous item. This software allows to create event-driven object-oriented programs automatically using state-based programming. But for designing of finite automaton Switch-technology is used in tandem with *UML (Unified Modeling Language)*. So connections diagrams are represented with the help of class diagrams and state diagram – with the help of *Statecharts*. Discussed software consists of following parts:

- kernel, that contains object's metamodel of finite automaton, implementation of description parsing, boolean functions interpretation mechanism, finite automaton correctness checkup tools and environment for *XML*-description execution (this part is common, not Eclipse-specific);
- built-in module for *UML*-diagrams development in the *Eclipse* environment. This module helps to create connection diagrams and transition diagrams as *UML*-diagrams. It also performs generation of *XML*-descriptions for systems, being developed by user.

It is possible to conclude this item with formula *UniMod* = *Switch-technology* + *UML* + *Eclipse*.

6.4. Authors of paper [23] suggest to use *XML* for automata description of virtual device appearance's dynamic properties. Virtual device here is video player *Crystal Player* (official web-site http://www.crystalplayer.com).

7. Automata and messages interchange mechanisms.

7.1. While studying classical problem of parallel programming, the synchronization of the shooters' chain [24, 25], it became clear that automata, built using template, described in paper [6] (template consists of two operators `switch`), does not allow to implement interacting parallel (or even pseudo parallel) processes. For overcoming this problem it was decided to use messages interchange mechanisms.

For this purpose special library *SWMEM* (*SWitch Message Exchange Mechanism*) was developed. In automaton's implementation template following changes were made:

- automaton's step was divided into three parts:
  - selection of the transition;
  - execution of actions on the transition;
  - state variable value's reassignment;
- special variables for taking conditions' priorities on diagram's edges into account were added;
- special variable for storage of selected action and its further execution was added.

7.2. In paper [26] mechanism of messages interchange between automata "located" in parallel is implemented due to addition such essence as "common bus", that allows implementing decentralized reactive multi-agent systems.

This approach allows implementing algorithms of the different kind (of the hierarchical, nested, parallel or any other) in the same manner.

For implementing automata that are working simultaneously, it was suggested to change templates, introduced in papers [6, 24]. The idea was to build automata with the help of two basic functions:

- transition/action function, that first of all executes input actions (of both types, in the state and on the transition), then defines the number of a new state and executes output actions in it;
- update function, that provides execution of the same operations (refreshing of automaton state's number and of array of received messages).

For synchronization automata should call all transition/action functions first and then call all update functions.

8. State-based programming language *State*. Usage of automata is limited by the lack of support of corresponding concepts in programming languages. For overcoming this problem in paper [27] the specialized programming language *State* was suggested. It is based on language *C#*, but it is extended with the support of basic abstraction "state".

That idea was not a successful one. So subsequently it was completely refactored (from the ideological point of view as well). As a result, language *State Machine* [28] appears. It extends the *Java* language by constructions, which represent concepts of "automaton", "state" and "event". It is based on design pattern *State Machine* [17] and realizes its main idea: this language should be suitable for describing essences, which vary their behavior in terms of automata.

5. CONCLUSIONS

In conclusion it is important to note, that current paper describes a lot of solutions of paramount problem of object-oriented programming [29] – definition of connections between static and dynamic properties of

object-oriented systems. This circumstance gives an ability to use different methods to implement reactive multi-agent systems.

All mentioned approaches are illustrated with the examples, projects, that were developed in the framework of "Foundation for Open Project Documentation" described in [2]. These examples are published on the web-site http://is.ifmo.ru.

REFERENCES

[1] Luger G., *Artificial Intelligence. Structures and Strategies for Complex Problems Solving*, Addison Wesley, 2002.

[2] Shalyto A., Naumov L., "Foundation for Open Project Documentation", *Linux Summit*, 2004. http://linuxsummit.org.

[3] Naumov L., Shalyto A., "Automata Theory for Multi-Agent Systems Implementation", *Proceedings of Integration of Knowledge Intensive Multi-Agent Systems*, MA, Boston, 2003. http://is.ifmo.ru in "Science" section.

[4] Shalyto A., *Switch-Technology. Algorithmization and Programming of Logic Control*, SPb.: Science (Nauka), 1998.

[5] Shalyto A., "Cognitive Properties of Hierarchical Representation of Complex Logical Structure", Proceedings of the 1995 ISIC Workshop. Architectures for Semiotic Modeling and Situation Analysis in Large Complex Systems, CA, Monterey, 1995.

[6] Shalyto A., Tukkel N., "Switch-Technology – Automata Approach to "Reactive" Systems Software Development", *Programming*, 2001, Vol. 5. http://is.ifmo.ru in "Articles" section.

[7] Shalyto A., Tukkel N. "Tanks and Automata", *BYTE/Russia*, 2003, Vol. 2. http://is.ifmo.ru in "Articles" section.

[8] Naumov A., Shalyto A., "Elevator Controlling System", SPb.: SPbSU ITMO, 2003. http://is.ifmo.ru in "Projects" section.

[9] Naumov L., Shalyto A., "The Art of Lift Programming. Object-Oriented Programming with Explicit States Separation", *Informational and Controlling Systems*, 2004, Vol. 7. http://is.ifmo.ru in "Projects" section.

[10] Korneev G., Shalyto A., "Finite Automata Implementation, Using Object-Oriented Programming", *Proceedings of X All-Russian Scientific-Methodical Conference "Telematics–2003"*, 2003, Vol. 2. http://tm.ifmo.ru.

[11] Shopyrin D., Shalyto A., "Object-Oriented Approach to Automata Programming", SPb.: SPbSU ITMO, 2003. http://is.ifmo.ru in "Projects" section.

[12] Feldman P., Shalyto A., "Joint Usage of Object-Oriented and Automata-Oriented Approaches to Programming", SPb.: SPbSU ITMO, 2004. http://is.ifmo.ru in "Projects" section.

[13] Zayakin E., Shalyto A., "Repeated Fragments of the State Based Source Code Elimination Method", SPb.: SPbSU ITMO, 2004. http://is.ifmo.ru in "Projects" section.

[14] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[15] Astafurov A., Shalyto A. "Automaton Design Pattern", SPb.: SPbSU ITMO, 2003. http://is.ifmo.ru in "Projects" section.

[16] Kuznetsov D., Shalyto A., "Tank Controlling System for "Robocode" Game. Variant 2", SPb.: SPbSU ITMO, 2004. http://is.ifmo.ru in "Projects" section.

[17] Shamgunov N., Korneev G., Shalyto A., "State Machine – New Object-Oriented Design Pattern", *Informational and Controlling Systems*, 2004, Vol. 5. http://is.ifmo.ru in "Articles" section.

[18] Naumov A., "State Based Object-Oriented Programming", SPb.: SPbSU ITMO, 2004. http://is.ifmo.ru in "Works" section.

[19] Feldman P. "Development of Toolkit for State Based Programs Debug", SPb.: SPbSU ITMO, 2004. http://is.ifmo.ru in "Works" section.

[20] Gurov V., Narvsky A., Shalyto A., "Automation of Event-Driven Object-Oriented Programming, using State Based Programming", *Proceedings of X All-Russian Scientific-Methodical Conference "Telematics–2003"*, 2003, Vol. 1. http://tm.ifmo.ru.

[21] Gurov V., Mazin M., Shalyto A., "UniMod – Software Package for Implementation of Object-Oriented Applications, Basing on Automata Approach", *Proceedings of XI All-Russian Scientific-Methodical Conference "Telematics–2004"*, 2004, Vol. 1. http://tm.ifmo.ru.

[22] Gurov V., Mazin M., Narvsky A., Shalyto A., "UML. Switch–technology. Eclipse", *Informational and Controlling Systems*, 2004, Vol. 6. http://is.ifmo.ru in "Articles" section.

[23] Bondarenko K., Shalyto A., "Development of XML-Format for Video Player Appearance Definition, Using on Finite Automata", SPb.: SPbSU ITMO, 2003. http://is.ifmo.ru in "Projects" section.

[24] Guisov M., Kuznetsov A., Shalyto A., "Integration of Messages Interchange Mechanism into Switch-Technology", SPb.: SPbSU ITMO, 2003. http://is.ifmo.ru in "Projects" section.

[25] Guisov M., Kuznetsov A., Shalyto A., "D. Mayhill's Problem of Shooters Chain Synchronization. Variant 2", SPb.: SPbSU ITMO, 2003. http://is.ifmo.ru in "Projects" section.

[26] Alshevsky U., Raer M., Shalyto A., "Turnstile Controlling System", SPb.: SPbSU ITMO, 2002. http://is.ifmo.ru in "Projects" section.

[27] Shamgunov N., Shalyto A., "State Based Programming Language with Compilation into Microsoft CLR", *Microsoft Research Academic Days in Saint-Petersburg*, April 21–23, 2004.

[28] Shamgunov N., Korneev G., Shalyto A., "State Machine – Java Language's Extension for Efficient Implementation of Automata", *Informational and Controlling Systems*, 2004, Vol. 7. http://is.ifmo.ru in "Articles" section.

[29] Graham I., *Object-Oriented Methods. Principles and Practice*, Addison-Wesley, 2001.