ISSN 0361-7688, Programming and Computer Software, 2007, Vol. 33, No. 5, pp. 283–292. © Pleiades Publishing, Ltd., 2007.
Original Russian Text © D.G. Shopyrin, A.A. Shalyto, 2007, published in Programmirovanie, 2007, Vol. 33, No. 5.

Graphical Inheritance Notation for State-Based Classes

D. G. Shopyrin and A. A. Shalyto

St. Petersburg State University of Information Technologies, Mechanics, and Optics,
Kronverkskii pr. 49, St. Petersburg, 197101 Russia
e-mail: danil.shopyrin@gmail.com

Received May 29, 2006

Abstract—State-based object-oriented programming combines basic advantages of object-oriented and automata-based programming technologies. Its basic features are flexibility, extensibility, and powerful mechanism of description of complex behavior, which is based on finite automata. The disadvantage of the state-based object-oriented programming is the lack of standard methods for designing and implementing state-based classes. In this work, graphical notation for designing state-based classes, which combines capabilities of the class diagrams of the object-oriented programming and behavior diagrams of the automata-based programming, is presented. The proposed graphical notation makes it possible to generalize, decompose, structure, and incrementally extend logic of the state-based classes by means of the inheritance.

DOI: 10.1134/S0361768807050040

INTRODUCTION

Program systems can be divided into the following three classes: *transforming, interactive*, and *reactive* systems [1]. *Transforming* systems are systems that stop after they transformed input data (e.g., archivers and compilers). *Interactive* systems interact with the environment in a dialog mode (e.g., text editors). *Reactive* systems interact with the environment through message exchange at the rate determined by the environment. Specific features of the reactive systems are as follows:

• response time is determined by the system environment,

• system behavior is deterministic,

• system behavior is parallel by its nature,

• system failures are extremely undesirable.

Means of the traditional object-oriented programming are often insufficient for designing and implementing complex behavior of reactive systems. On the other hand, automata-based programming suggests powerful mechanism of description and implementation of complex behavior based on finite automata. *State-based object-oriented programming* is a synthesis of object-oriented and automata-based technologies. It combines their basic advantages, such as flexibility, extensibility, and powerful mechanism of complex behavior description, which is based on finite automata. The basic concept of the state-based object-oriented programming is a state-based class. The *state-based class* is a class of objects the behavior of which depends on the current, explicitly specified, control state. The implementation of the state-based class relies on finite automata [2, 3].

The disadvantage of the state-based object-oriented programming is the lack of standard methods for designing and implementing state-based classes. The most popular approach to designing and implementing state-based classes is the *State* design pattern [4]. Some derivative variants of the *State* design pattern are described in [5–8].

The Statecharts graphical language [9] is used as a tool for the graphical design of behavior. Many modern approaches to designing reactive systems are based, to some extent, on this language. The Statecharts language is an extension of the traditional automaton model [10] supplemented by *hierarchy* and *parallelism* description facilities [11, 12]. The *hierarchy* is introduced by means of *embedded states*, which semantically corresponds to the `XOR` operation (exclusive `OR`). The *parallelism* is introduced by means of *orthogonal states*, which semantically corresponds to the logical `AND` operation. The automata-based means for system design include also the SDL language [13] and the SyncCharts synchronous programming language [14, 15].

One of the disadvantages of the above-listed graphical languages for behavior design that are based on the state-based classes is the lack of means for describing object-oriented nature of state-based classes. This disadvantage is (partially) removed in the object-oriented programming with explicit state emphasis [16], also known as SWITCH-technology [17]. Transition graphs used in the SWITCH-technology are combined with the communication diagrams that describe in detail their interface.

Another disadvantage of the above-listed languages is the absence of means for describing the *inheritance* relation for the state-based classes. Inheritance allows derived class to receive properties or characteristics of the base class, normally, as a result of some special relationship between the base and derived classes [18]. There exist approaches that allow one to use the inheritance when implementing state-based classes [19, 20]. However, the question of whether it is possible to use the inheritance in visual design of state-based classes is still open.

The goal of this work is to develop graphical notation for designing state-based classes, which combines capabilities of the object-oriented class diagrams and automata-based behavior diagrams. The proposed graphical notation makes it possible to generalize, decompose, structure, and incrementally extend logic of state-based classes by means of the inheritance.

1. TERMS AND DEFINITIONS

Before we proceed to description of the suggested graphical notation, we introduce some terms and definitions.

A state-based class A is defined by a triple $\langle I, S, J\rangle$, where I is a set of methods of the state-based class interface, S is a set of control states of the state-based class, and J is a set of transitions between the states.

On the set of states S of the state-based class, function $beg(S) \in S$ returning the initial state is defined. For each state $s \in S$, the following functions are defined:

- $den(s)$, action upon entering the state;
- $dex(s)$, action when exiting the state;
- $dact(s)$, activity in the state.

A transition $j \in J$ is defined by the quinary $\langle from, to, ev, cond, do\rangle$, where $from(j) \in S$ is an initial state of the transition, $to(j) \in S$ is a terminal state of the transition, $ev(j) \in I$ is a cause of the transition: a call of the interface method in the case of which the transition can be performed, $cond(j) \in \{true, false\}$ is the transition condition, and $do(j)$ is an action executed upon transition.

The transition j_0 can be done if the following conditions are satisfied:

- the current state of the state-based class is the state $from(j_0)$,
- the interface method $ev(j_0)$ of the state-based class is invoked, and
- the condition $cond(j_0)$ is fulfilled.

In this case, the actions $dex(from(j_0))$ and $do(j_0)$ are performed; the state $to(j_0)$ is set as the current state; and the action $den(to(j_0))$ is performed. After this, the transition j_0 is considered performed.

1.1. Inheritance of State-Based Classes

The inheritance allows one to define new classes based on the existing ones. When defining a new class, only the properties that are different from those of the base class are specified. The other properties are added to the new class *implicitly* and *automatically* [21]. In formal terms, the inheritance can be written as follows [22, 23]:

$$R = P_1 \oplus P_2 \oplus \ldots \oplus P_n \oplus \Delta R,$$

where R is the new class; $P_1, P_2, \ldots, P_n$ is a set of properties inherited from the base classes, ΔR are incrementally added new properties differing the new class R from the base classes; and $\oplus$ is the operation combing the class properties.

Consider the mechanism of the state-based class inheritance. All states and transitions of the base classes are implicitly present in the derived class. The derived class can extend and modify behavior of the base classes. Behavior modification of the base classes relies on state overriding. Some states of the base classes can be *overridden*, and transitions from these overridden states can be modified. The derived class can also be supplemented by new states and transitions between them.

A state-based class D is a descendant of a state-based class B ($B \le D$) if

- $I_b \subseteq I_d$; i.e., the set of interface methods I_b implemented by the state-based class B belongs to the set of interface methods I_d implemented by the state-based class D;
- $S_b \subseteq S_d$; i.e., the set of states S_b of the state-based class B is a subset of the set of states S_d of the state-based class D;
- $beg(S_b) = beg(S_d)$, i.e., the initial states of the classes B and D coincide;
- for any transition j_b from the set of transitions J_b of the automaton B, there exists a transition j_d from the set of transitions J_d of the automaton D such that
  - $from(j_d) = from(j_b)$ (initial states of transitions j_d and j_b coincide),
  - $ev(j_d) = ev(j_b)$ (methods that are causes of transitions j_d and $j_b$coincide),
  - $cond(j_d) = cond(j_b)$ (conditions of transitions (j_d and j_b coincide).

Transition j_d of a state-based class D overrides transition j_b of a state-based class B if

- $from(j_d) = from(j_b)$; i.e., the initial states of transitions j_d and j_b coincide;
- $ev(j_d) = ev(j_b)$; i.e., the methods that are causes of transitions j_d and j_b coincide;

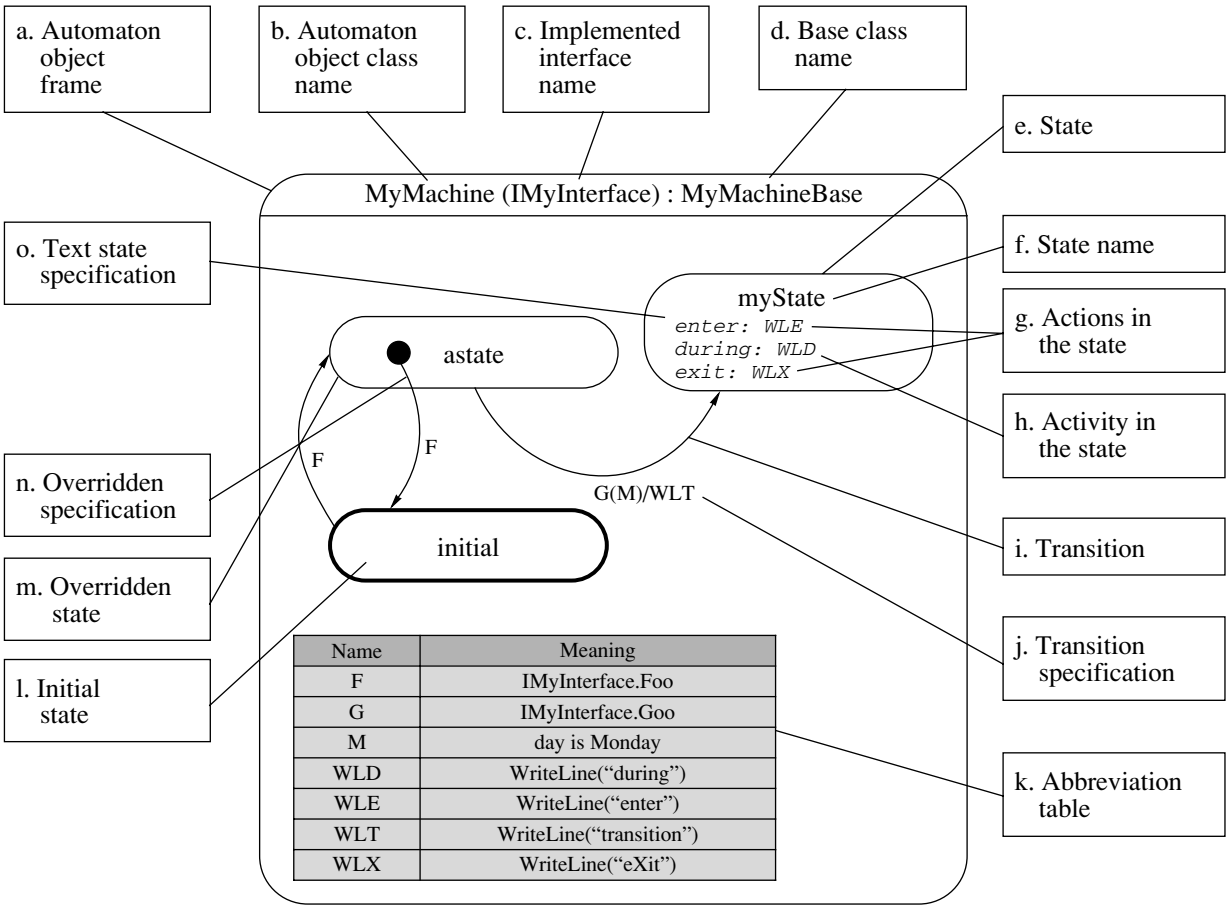


Fig. 1. Basic elements of the proposed graphical notation.

• $cond(j_d) = cond(j_b)$; i.e., the conditions of transitions j_d and j_b coincide;

• $to(j_d) \neq to(j_b)$ or $do(j_d) \neq do(j_b)$; i.e., the terminal states or actions upon the transitions do not coincide.

1.2. Decomposition and Structuring of the State-Based Class Logic

Decomposition and structuring of the state-based class logic is implemented by means of *state groups*, which are used in the Statecharts language [9] and SWITCH-technology [24]. The state groups can be embedded in each other forming a hierarchy of the state groups.

The state groups can have *group transitions*, which are also referred to as *beams*. A beam is similar to a *transition*, except for the fact that the initial state is not specified for the beam. A beam $b \in B$ is defined by a quadruple $\langle to, ev, cond, do\rangle$. For each state $s \in S$, a function $beams(s)$ is defined, which returns the set of beams corresponding to the given state. The set $beams(s)$ is equivalent to the subset of transitions that have state s as the initial state:

$$beams(s) \equiv \mathrm{j} \in J, \quad from(j) = s.$$

A group $g \in G$ is defined by the triple $\langle gbeams, msub, gsub\rangle$, where $gbeams(g) \subseteq B$ is the set of beams corresponding to the group g, $msub(g) \subseteq S$ is the set of states belonging to the group g, and $gsub(g) \subset G$ is the set of groups embedded into the group g.

For each group $g \in G$, the following assertions are valid:

• $\forall s \in msub(g)$, $gbeams(g) \subseteq beams(s)$; i.e., the set of beams of the state s belonging to the group g is a superset of the set of beams corresponding to the group g;

• $\forall g_0 \in gsub(g)$, $gbeams(g) \subseteq gbeams(g_0)$; i.e., the set of beams of the state group g_0 belonging to the group g is a superset of the set of beams corresponding to the group g;

• $\forall g_0 \in gsub(g)$, $\forall s \in msub(g_0)$, $s \in msub(g)$; i.e., if the state s is embedded into the group g_0 that is, in turn, embedded into the group g, then it is embedded into the group g as well;

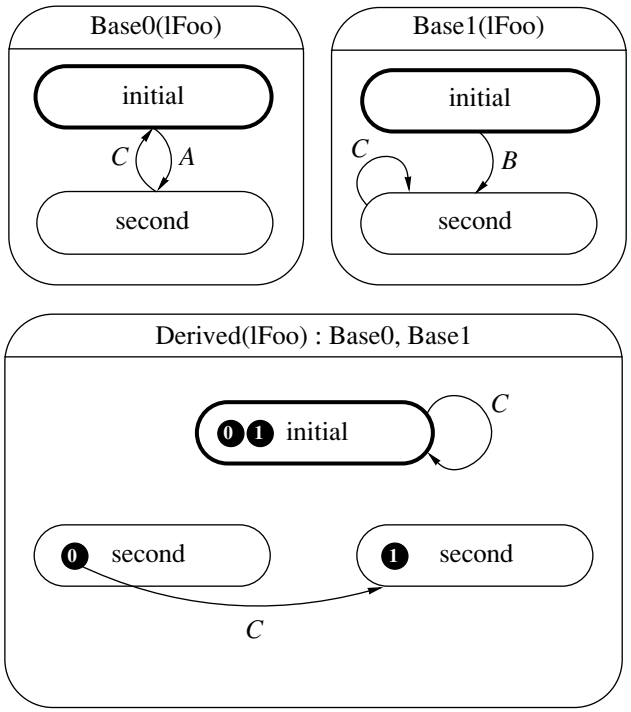


Fig. 2. Multiple inheritance of state-based classes.

• $\forall g_0, g_1 \in G$, if $g_0 \in gsub(g_1)$ and $g \in gsub(g_0)$, then $g \in gsub(g_1)$; i.e., if the group g is embedded into the group g_0 that is embedded into the group g_1, then the group g is embedded into the group g_1 as well.

2. GRAPHICAL NOTATION

To design state-based classes, we propose to use behavior diagrams, which are an extended version of the transition graphs used in the SWITCH-technology [3, 16]. A specific feature of the behavior diagrams suggested in this work is a capability to describe decomposition and structuring of the state-based class logic by means of the inheritance. Basic elements of the graphical notation used for constructing the behavior diagrams are shown in Fig. 1.

Text specification of transitions is written in the form

```
E[(C)][/D],
```

where `E` is the transition cause $ev(j_0)$, `C` is the transition condition $cond(j_0)$, and `D` is the transition action $do(j_0)$.

The transition condition and action are optional parts of the specification and can be omitted. The transition cause and condition (if available) constitute the *permissive* part of the transition specification.

For example, transition j in Fig. 1 has specification `G(M)/WLT`, where, in accordance with the abbreviation table,

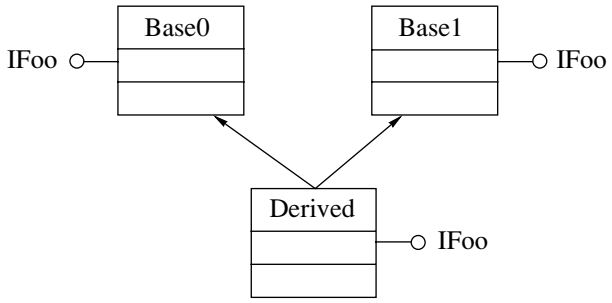


Fig. 3. Static diagram of state-based classes.

• `G` is equivalent to the call of the `IMyInterface.Goo` method,

• `M` is equivalent to the condition `"day is Monday"`,

• `WLT` is equivalent to the call of the `WriteLine("transition")` method.

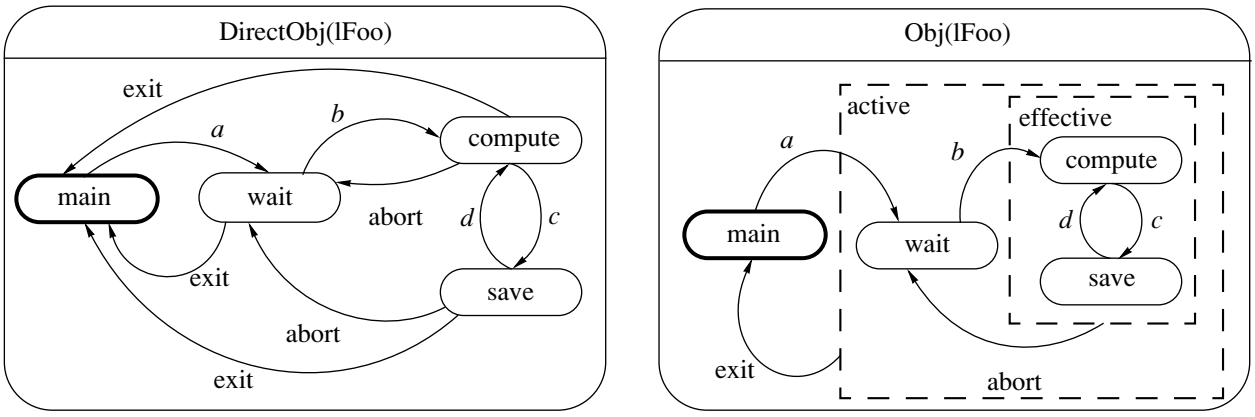
Hence, the transition to the state `myState` is possible only by Mondays (`day is Monday`) if method `Goo()` is invoked and the automaton is in the state `astate`. In this case, before changing the state, the string `"transition"` is printed to the standard stream.

2.1. Graphical Representation of the State-Based Inheritance

Let us consider representation of the inheritance relation by means of the suggested graphical notation. The base state-based class (if exists) is indicated in the heading of the state-based class frame after the colon. All states and transitions of the base class are implicitly inherited by the derived state-based class.

Overriding of the states and transitions of the base class are permitted. The overridden state is marked by the bold dot. In the case of multiple n-ary inheritance, one numbered *dot* for each base class is depicted in the order they appeared in the heading. The dot with number `i` corresponds to the base class with number `i`. The state marked by more than one dot is a union and extension of the states with the same name that are present in several base classes (Fig. 2). Note that, if multiple inheritance is used, various contradictions inherent in multiple inheritance may appear. The detailed formal analysis of possible contradictions is beyond the scope of this work and is a subject of future studies.

The transition overriding is implemented in accordance with the *permissive* part of the transition specification. As a result of the overriding, the transition action can be modified. The overridden transition begins at the bold dot depicting the corresponding base class. For example, in Fig. 2, the class `Derived` overrides the transition by the cause `C` from the state `sec-`



(a) Without logic structuring (b) With logic structuring

Fig. 4. Example of using state-based class logic structuring.

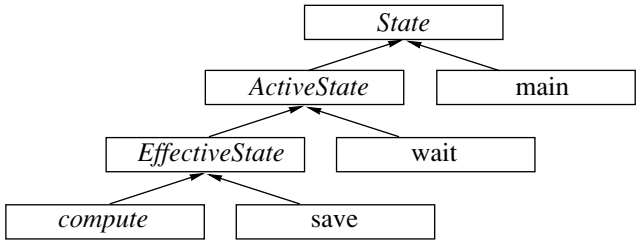


Fig. 5. Hierarchy of states.

`ond` of the base class `Base0` and indicates state `second` of the base class `Base1` as a terminal state.

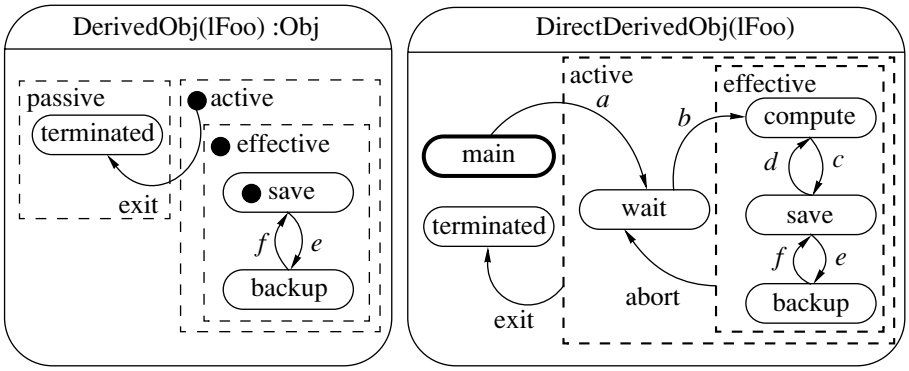
Note that the inheritance relations for the state-based classes can be represented by means of the class diagrams of the UML language [25]. The class diagrams make it possible to represent static aspect of the state-based class inheritance; however, they do not provide graphical syntax for detailed representation of the dynamic aspect of inheritance. An example of the class diagram is shown in Fig. 3.

2.2. Graphical Representation of Structuring of the State-Based Class Logic

Consider the behavior diagram of the state-based class `DirectObj` depicted in Fig. 4a. In class `DirectObj`, structuring is not used; therefore, its diagram contains redundant transitions.

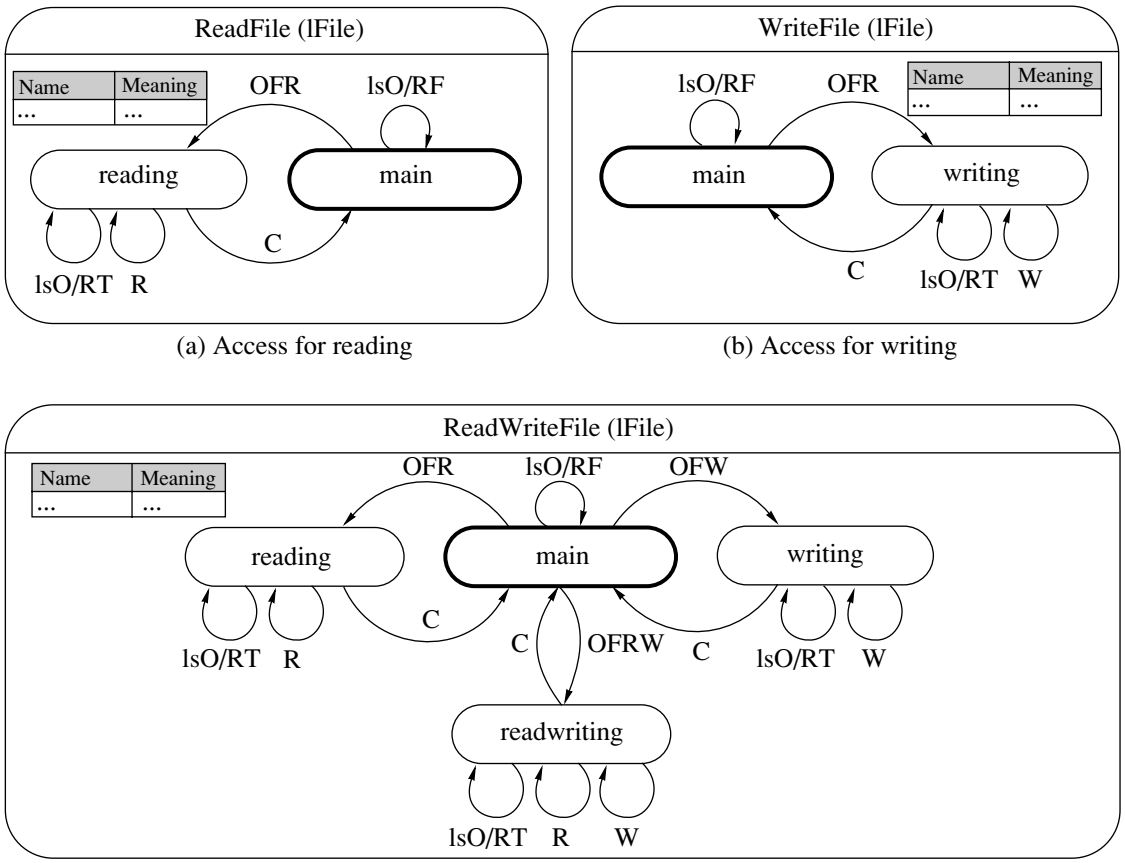
Structuring of the state-based class logic is performed by means of the state groups, which are depicted as dashed rectangles (e.g., state groups `active` and `effective` in Fig. 4b). State groups make it possible to generalize behavior common for several states. As a result, duplication of transitions can be reduced.

State groups may contain group transitions. The group transition can be performed when the state-based class is in one of the states belonging to this group.



(a) Combined use of inheritance and structuring (b) Without combined use of inheritance and structuring

Fig. 6. Example of combined use of state-based class inheritance and logic structuring.



(a) Access for reading

(b) Access for writing

(c) Access for reading/writing

Fig. 7. Behavior diagrams for a family of file access classes without using inheritance.

States belonging to one group are descendants of one base class of states. State groups can be embedded one into another forming hierarchy. The diagram of state classes corresponding to the state-based class shown in Fig. 4 is depicted in Fig. 5 [26].

Logic structuring can be used together with the inheritance of state-based classes. All state groups defined in the base class are implicitly inherited by the derived class. The state groups of the base class mentioned in the derived class are marked by the bold dot, like the states of the base class. The derived state-based class can override behavior in the groups of its base class.

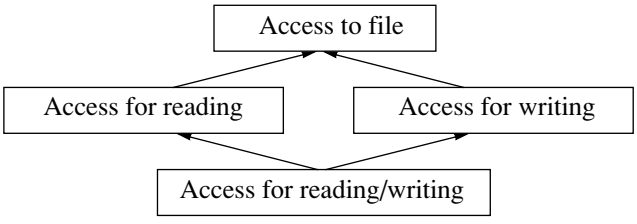


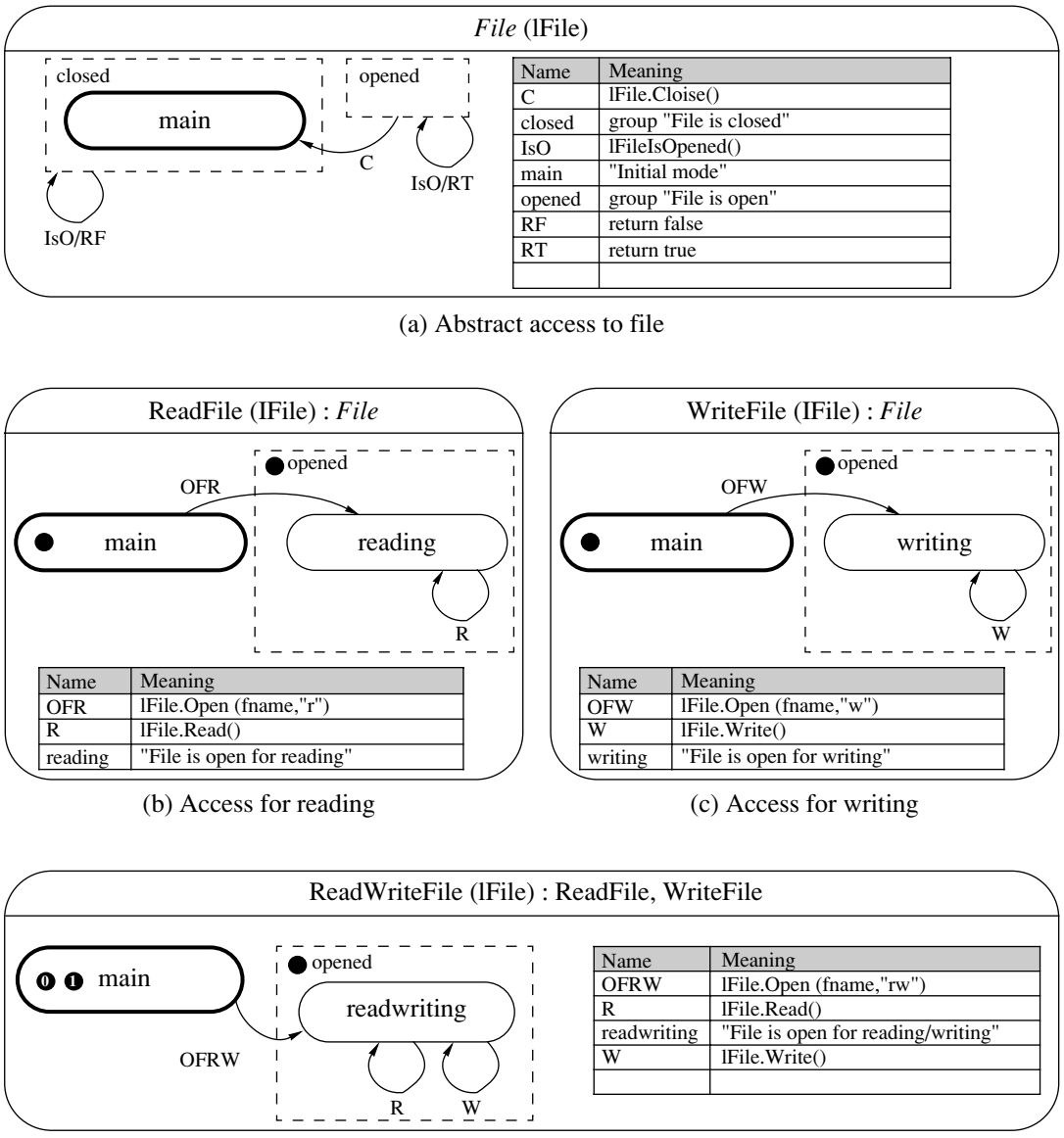
Fig. 8. Hierarchy of file access classes.

As an example, we consider class `DerivedObj`, which is a descendant of class `Obj` (Fig. 4b). The behavior diagram for class `DerivedObj` is depicted in Fig. 6a.

In the state-based class `DerivedObj`, state groups `active` and `effective` of the base class `Obj` are overridden. Class `DerivedObj` adds the new state group `passive`. In addition, the state-based class `DerivedObj` (i) overrides transition from group `active` in state `main` setting state `terminated` defined in class `DerivedObj` as the terminal state and (ii) adds state `backup` to the state group `effective` and connects it by transitions with state `save`.

The behavior diagram for the state-based class `DirectDerivedObj` that is similar to class `DerivedObj` (Fig. 6a) but is constructed without combined use of structuring and inheritance of the state-based class logic is presented in Fig. 6b.

Structuring of the state-based class logic by grouping states allows us to *considerably* reduce duplication and improve readability both at the stage of designing state-based classes and at the stage of their implementation.



(a) Abstract access to file

(b) Access for reading

(c) Access for writing

(d) Access for reading/writing

Fig. 9. Behavior diagrams for file access classes with the use of inheritance.

3. APPLICATION EXAMPLE

Let us consider an example illustrating the proposed graphical notation. Suppose that there exists a family of classes providing access to a file:

- access for reading (state-based class `ReadFile`),
- access for writing (state-based class `WriteFile`),
- access for reading, writing, and reading/writing (state-based class `ReadWriteFile`).

These classes are of automaton nature (with states "Closed," "Open for reading," etc.). The behavior diagrams for these state-based classes are presented in Fig. 7.

The behavior of these classes can be generalized (similar components are distinguished) and structured by means of the inheritance. These classes form the hierarchy shown in Fig. 8.

The root of the suggested hierarchy is the abstract class generalizing some aspects of access to a file. The behavior diagram for these classes constructed by means of the inheritance is depicted in Fig. 9.

The behavior of any state-based class can be extended by means of the inheritance. Figure 10 shows the behavior diagram for the state-based class `AppendFile`, which extends logic of the state-based class `ReadWriteFile` by adding one more state (`appending`) to it. The extension proceeds incrementally, without modifications of the already exiting classes.

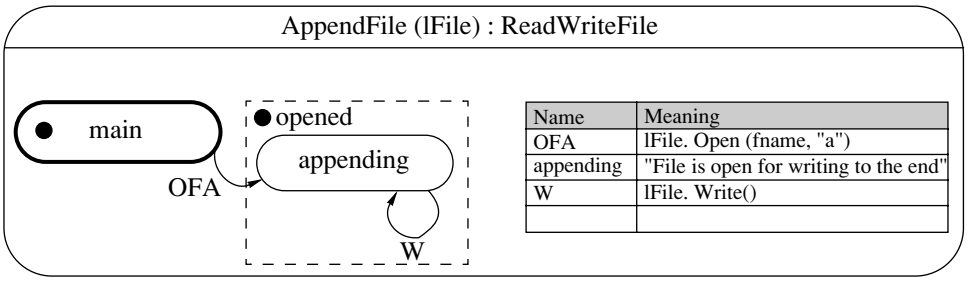


Fig. 10. Behavior diagram for the `AppendFile` class with the use of inheritance.

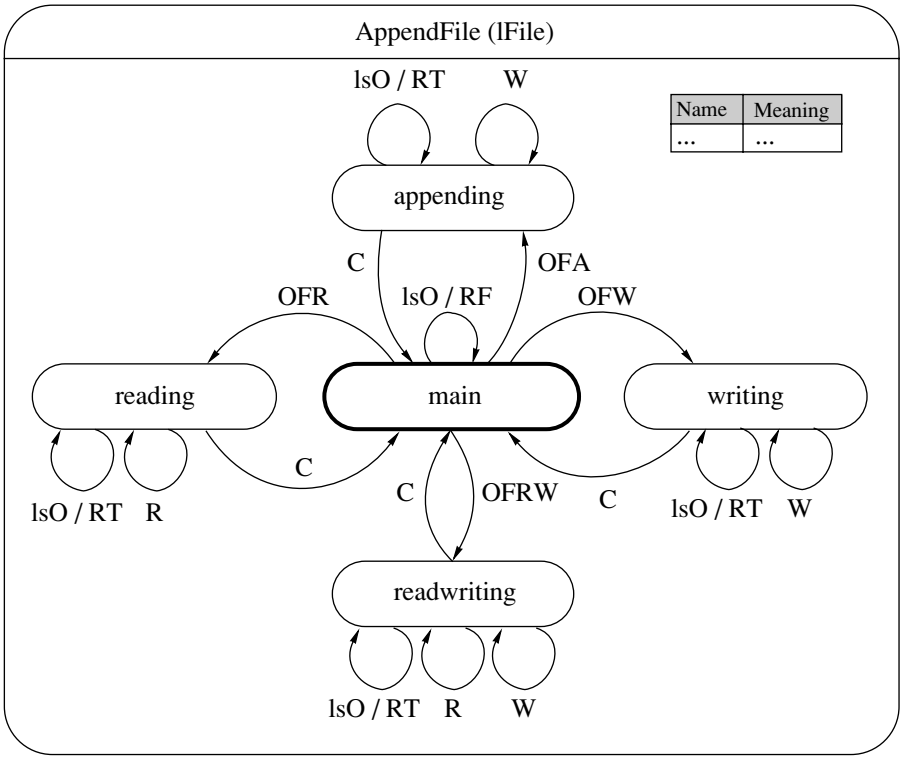


Fig. 11. Behavior diagram for the `AppendFile` class without using inheritance.

The behavior diagram for class `AppendFile` shown in Fig. 10 is equivalent to the behavior diagram for class `AppendFile` (Fig. 11) but constructed without use of the inheritance. Note that the use of the inheritance *considerably* reduced duplication of the states and transitions.

Comparison of the design methods

	Without using inheritance and logic structuring	With the use of inheritance and logic structuring
States	13	5
Overridden states	–	4
State groups	–	2
Overridden state groups	–	4
Transitions	42	12

The table shows results of calculation of the number of the states, state groups, and transitions used in designing behavior of state-based classes with the use of decomposition and structuring their logic by means of the inheritance (Figs. 9 and 10) and without it (Figs. 7 and 11).

As can be seen from the table, the proposed method of decomposition and structuring of the state-based class logic allows us to considerably reduce the number of the transitions used by removing duplication. In this case, the diagram is supplemented by new entities, such as overridden states and state groups.

4. CONCLUSIONS

In conclusion, we note that, for the graphical notation described, two methods of implementation of the state-based classes have been suggested:

- one based on virtual methods [27],
- the other based on virtual embedded classes [28].

These methods completely agree with the basic principles of object-oriented programming and allow us, as required in the automata-based programming, to isomorphically map the proposed graphical notation when implementing state-based classes.

The practical importance of the suggested graphical notation and of the related methods of implementation of state-based classes is substantiated by the results of their application to programming practice at Transas Technologies Co. In particular, it was used in the design and implementation of the editing manipulators in the Iris reusable framework. The latter is designed for constructing spatial data visual editing subsystems and is used for developing navigation, coastal, and training systems.

The graphical notation discussed makes it possible to generalize, decompose, structure, and extend logic of the state-based classes by means of the inheritance. Decomposition and structuring of the state-based class logic considerably reduces duplication in designing and implementing systems whose behavior is described by finite automata.

REFERENCES

1. Harel, D. and Pnueli, A., On the Development of Reactive Systems, *Logic and Models of Concurrent Systems. Nato Advanced Study Institute on Logic and Models for Verification and Specification of Concurrent Systems*, Springer, 1985, pp. 477–498.
2. Shalyto, A.A. and Tukkel', N.I., From Turing Programming to Automaton Programming, *Mir PK*, 2002, no. 2, pp. 144–149, http://is.ifmo.ru/works/turing/.
3. 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].
4. Gamma, E., Khelm, R., Johnson, R., and Vlissides, J., *Methods of Objective-Oriented Projection. Design Patterns*, St. Petersburg: Piter, 2001, p. 368.
5. Adamczyk, P., The Anthology of the Finite State Machine Design Patterns, *The 10th Conf. on Pattern Languages of Programs*, 2003, http://hillside.net/plop/plop2003/Papers/Adamczyk-State-Machine.pdf.
6. Shalyto, A.A. and Naumov, L.A., Methods of Objective-Oriented Implementation of Reactive Agents on the Basis of the Finte Automata, *Iskusstvennyi intellect,* 2004, no. 4, pp. 756–762, http://is.ifmo.ru/works/_aut_oop.pdf.
7. Adamczyk, P., Selected Patterns for Implementing Finite State Machines, *The 11th Conf. on Pattern Languages of Programs*, 2004, http://pinky.cs.uiuc.edu/~padamczy/docs/fsm_updated.pdf.
8. Odrowski, J. and Sogaard, P., Pattern Integration—Variations of State, *Proc. of PLoP96*, http://www.cs.wustl.edu/~schmidt/PLoP-96/od-rowski.ps.gz.
9. Harel, D., Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Program*, 1987, vol. 8, pp. 231–274.
10. *Automata Studies*, Princeton: Princeton Univ. Press, 1956.
11. Harel, D. and Naamad, A., The Statemate Semantics of Statecharts, *ACM Trans. Softw. Eng. Methodology,* 1996, vol. 5, pp. 293–333.
12. Mikk, E., Lakhnech, Y., Petersohn, C., and Siegel, M., On Formal Semantics of Statecharts as Supported by STATEMATE, *Proc. of the 2nd BCS-FACS Northern Formal Methods Workshop*, Ilkley, 1997.
13. Specification and Description Language (SDL), *Int. Engineering Consortium,* http://www.iec.org/acrobat.asp?filecode=125.
14. Benveniste, A., The Synchronous Languages 12 Years Later, *Proc. of the IEEE,* 2003, vol. 91, no. 1, pp. 64–83. http://www-sop.inria.fr/aoste/benveniste2003synchro nous.pdf.
15. André, C., SyncCharts: A Visual Representation of Reactive Behaviors, *Tech. Report RR 95-52. I3S*, Sophia-Antipolis, 1995.
16. Shalyto, A.A. and Tukkel', N.I., Tanks and Automata, *BYTE,* Russia, 2003, no. 2, pp. 69–73, http://is.ifmo.ru/works/tanks_new/.
17. Shalyto, A.A., SWITCH-technology, *Algoritmizatsiya i programmirovanie zadach logicheskogo upravleniya* (Algorithmization and Programming of Logic Control Problems), St. Petersburg: Nauka, 1998, no. 1, p. 628.
18. Danforth, S. and Tomlinson, C., Type Theories and Object-Oriented Programming, *ACM Comput. Surv.,* 1988, no. 1, pp. 29–72.
19. Sane, A. and Campbell, R., Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity, *OOPSLA'95*, http://choices.cs.uiuc.edu/sane/home.html.
20. Lee, J., Xue, N., and Kuei, T., A Note on State Modeling Through Inheritance, *SIGSOFT Softw. Eng. Notes*, 1998, no. 1, pp. 104–110.
21. Taivalsaari, A., On the Notion of Inheritance, *ACM Comput. Surv.*, 1996, no. 3, pp. 438–479.
22. Bracha, G. and Cook, W., Mixin-based Inheritance, *OOPSLA/ECOOP'90, Conf. Proc., ACM SIGPLAN Not.*, 1990, no. 10, pp. 303–311.
23. Wegner, P. and Zdonik, S., Inheritance As an Incremental Modification Mechanism or What Like Is and Isn't Like, *ECOOP'88 Conf. Proc.*, Springer, 1988, pp. 55–77.
24. Shalyto, A.A. and Tukkel', N.I., Implementation of Automata in Programming of Event Systems, *Program-*

mist, 2002, no. 4, pp. 74–80, http://is.ifmo.ru/works/evsys/.

25. Buch, G., Rambo, J., and Jacobson, A., *UML. Rukovodstvo pol'zovatelya* (UML. Guidance for a User), Moscow: DMK, 2000, p. 432.
26. Zayakin, E.A. and Shalyto, A.A., Method of Elimination of Repeated Code Fragments in Implementation of Finite Automata, *Mir PK (CD)*, 2005, no. 8, http://is.ifmo.ru/projects/life_app/.
27. Shopyrin, D.G., Objective-Oriented Implementation of Finite Automata on the Basis of Virtual Methods, *Informatsionno-upravlyayushchie sistemy,* 2005, no. 3, pp. 36–40, http://is.ifmo.ru/works/runewstate/.
28. Shopyrin, D.G., A Method of Design and Implementation of Finite Automata on the Basis of Virtual Embedded Classes, *Informatsionnye tekhnologii modelirovaniya i upravleniya*, 2005, no. 1, pp. 87–97, http://is.ifmo.ru/works/ruvstate/.