НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ноябрь–декабрь 2024 Том 24 № 6 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
November–December 2024 Vol. 24 No 6 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

doi: 10.17586/2226-1494-2024-24-6-1035-1043

Specification language for automata-based objects cooperation

Fedor A. Novikov[1], Irina V. Afanasieva[2], Ludmila N. Fedorchenko[3]✉, Taisia A. Kharisova[4]

[1] Peter the Great St. Petersburg Polytechnic University (SPbPU), Saint Petersburg, 195251, Russian Federation
[2] Special Astrophysical Observatory of the Russian Academy of Sciences (SAO RAS), Nizhny Arkhyz, 369167, Russian Federation
[3] St. Petersburg Federal Research Center of the Russian Academy of Sciences, Saint Petersburg, 199178, Russian Federation
[4] Ioffe Institute, Saint Petersburg, 194021, Russian Federation

[1] fedornovikov51@gmail.com, https://orcid.org/0000-0003-4450-0173
[2] riv615@gmail.com, https://orcid.org/0000-0003-4225-4124
[3] lnf@iias.spb.su✉, https://orcid.org/0000-0002-4008-9316
[4] tais.harisova@mail.ru, https://orcid.org/0009-0002-3456-0471

Abstract
Automata-based programming is a programming paradigm that has been successfully used in the development of reactive systems, distributed control systems, and various mission-critical applications where the ability to verify the compliance of a real system with its model given in the form of specifications is critical. The traditional testing of such systems can be difficult; thus, more advanced verification tools are required to increase confidence in the reliability of real systems. The previously proposed language for the specification of the Cooperative Interaction of Automata-based Objects (CIAO) was successfully used to develop several different reactive systems as a result of which a number of shortcomings were identified and eliminated in the new version of CIAO v.3. This new version of the language was developed for the automatic verification of automata-based programs according to the formal specifications of a certain class of real-time systems. Three innovations distinguish CIAO v.3 from previous versions. First, an explicit distinction between automata classes and automaton objects as instances of these classes. Second, we specify the binding of automaton objects through interfaces using a connection scheme. Third, we describe the semantics of the behavior of a system of interacting automaton objects using a semantic graph. This paper presents the main concepts of the new language version including the abstract syntax, operational semantics, and metamodel. The third version of the CIAO language naturally includes almost all the advantages of object-oriented programming into the paradigm of automata programming. The connection of automaton objects through the corresponding interfaces is arbitrarily reflected by the connection scheme. A semantic graph describing the semantics of the behavior of the automata-based program is used to implement automatic verification with respect to formal specifications.

Keywords
behavior model, automata-based programming, state transition graph, UML, state machine diagram, class diagram, concurrent behavior, software architecture, reactive system

For citation: Novikov F.A., Afanasieva I.V., Fedorchenko L.N., Kharisova T.A. Specification language for automata-based objects cooperation. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2024, vol. 24, no. 6, pp. 1035–1043. doi: 10.17586/2226-1494-2024-24-6-1035-1043

УДК 004.415.52, 004.434

Язык спецификации взаимодействия автоматных объектов

Федор Александрович Новиков[1], Ирина Викторовна Афанасьева[2], Людмила Николаевна Федорченко[3]✉, Таисия Анваровна Харисова[4]

[1] Санкт-Петербургский политехнический университет Петра Великого, Санкт-Петербург, 195251, Российская Федерация

[2] Специальная астрофизическая обсерватория РАН, Карачаево-Черкесская Республика, Зеленчукский р-н, пос. Нижний Архыз, 369167, Российская Федерация

[3] Санкт-Петербургский Федеральный исследовательский центр РАН, Санкт-Петербург, 199178, Российская Федерация

[4] Физико-технический институт им. А.Ф. Иоффе РАН, Санкт-Петербург, 194021, Российская Федерация

[1] fedornovikov51@gmail.com, https://orcid.org/0000-0003-4450-0173

[2] riv615@gmail.com, https://orcid.org/0000-0003-4225-4124

[3] lnf@iias.spb.su✉, https://orcid.org/0000-0002-4008-9316

[4] tais.harisova@mail.ru, https://orcid.org/0009-0002-3456-0471

Аннотация

Введение. Автоматное программирование — парадигма программирования, успешно применяемая при разработке реагирующих систем, распределенных систем управления и различных ответственных приложений, где критически важна возможность верификации соответствия реальной системы ее модели, заданной в виде спецификаций. Традиционное тестирование таких систем может быть затруднено, поэтому требуются более совершенные средства верификации для повышения степени доверия к надежности реальной системы. Предложенный ранее язык спецификации кооперативного взаимодействия автоматных объектов (Cooperative Interaction of Automata Objects, CIAO) был успешно применен для разработки нескольких реагирующих систем. Однако он также выявил ряд недостатков, которые устранены в CIAO v.3. **Метод.** Новая версия языка разработана с целью автоматической верификации автоматных программ по формальным спецификациям определенного класса систем реального времени. CIAO v.3 содержит три нововведения в отличие от предшествующих версий. Во-первых, явное разграничение автоматных классов и автоматных объектов как экземпляров этих классов. Во-вторых, спецификация связывания автоматных объектов через интерфейсы с помощью схемы связей. В-третьих, описание семантики поведения системы взаимодействующих автоматных объектов с помощью семантического графа. **Основные результаты.** В работе представлены основные концепции новой версии языка, приведены абстрактный синтаксис, операционная семантика и метамодель. **Обсуждение.** CIAO v.3 позволяет естественным образом включить в парадигму автоматного программирования почти все преимущества объектно-ориентированного программирования. Подключение автоматных объектов через соответствующие интерфейсы произвольным образом отражает схема связей. Семантический граф, описывающий семантику поведения автоматной программы, используется для реализации автоматической верификации относительно некоторых формальных спецификаций.

Ключевые слова

модель поведения, автоматное программирование, граф переходов состояний, унифицированный язык моделирования, UML, диаграмма конечного автомата, диаграмма классов, параллельное поведение, архитектура программного обеспечения, реагирующая система

Ссылка для цитирования: Новиков Ф.А., Афанасьева И.В., Федорченко Л.Н., Харисова Т.А. Язык спецификации взаимодействия автоматных объектов // Научно-технический вестник информационных технологий, механики и оптики. 2024. Т. 24, № 6. С. 1035–1043 (на англ. яз.). doi: 10.17586/2226-1494-2024-24-6-1035-1043

Introduction and Historical Background

The automata-based programming paradigm [1] describes the behavior of technical systems based on an explicit definition of discrete states and their transitions. This paradigm dates back to the pioneering work of D. Harel [2–4] which was significantly advanced in [5, 6] and is currently being developed by many researchers [7–11]. Automata-based programming has many undeniable advantages:

— a reliable mathematical basis for the theory of finite automata;
— a stable tradition of using the concept of states to describe the behavior of various devices in many engineering fields;
— the possibility of simple and efficient implementation on any software platform developed in the automata-based programming paradigm.

Many authors have successfully used automata-based programming in various fields and for different purposes which has led to the emergence of various forms of automata-based programming [6]. For example, we used automata-based programming to implement domain-specific languages, developed the Cooperative Interaction of Automata Objects (CIAO) language [8], and used CIAO v.1 to implement a mission-critical astronomical data collection and processing system [9]. Based on the results of its use, some improvements were made to CIAO v.2, and the language was applied to verify low-level communication protocols [8] and other tasks [10–13]. Moreover, CIAO v.2 can be used to automatically verify device control programs [13]. Subsequent research in this area led us to the need to refine the CIAO behavior model, improve the graphical notation, and conduct practical testing of the algorithms.

The goal of this work is to implement the third version of the CIAO language which is suitable for automatic verification of conformance to formal specifications.

Basic Concepts of the CIAO Specification Language

The central concept of all versions of the CIAO language is the automaton object which is discussed in [8, 13]. An automaton object has much in common with a state machine packaged in a component. Here, the terms "state machine" and "component" are understood in the sense of the Unified Modeling Language (UML) [14, 15], but at the same time, they have significant differences. A state machine is defined by a state transition graph where for each transition an event is specified that initiates this transition. A transition can have a guard condition, and an effect (action) can be achieved due to the transition. We use the following extensions of the model each of which derives the proposed model of an automaton object from the class of finite automata [16]:

— events can have parameters and provide an arbitrary amount of information to the automaton object;
— guard conditions can depend on the current values of the local variables of a given automaton object and not only on the current state;
— effects can have parameters and write an arbitrary amount of information to the external memory.

A component in the context of the UML language is a set of provided and required one. A component in the context of the UML language is a set of provided and required interfaces. According to B. Meyer's principle [17], the operations of the interface of any object should be divided into queries (designated by the stereotype "query") which deliver values and do not change the state of the object, and commands (designated by the stereotype "command") which change the state of the object but do not deliver values. In this case, we obtain exactly four kinds of possible interfaces for interaction between objects:

— provided commands (events);
— required commands (actions);
— provided queries (assertions);
— required queries (conditions).

The list of combinations is exhaustive, and in this sense the CIAO model is final and considers all possible interactions between objects satisfying B. Meyer's principle.

The basic concept of the CIAO behavior model is that the behavior of an entire system can be described as the cooperative behavior of several automaton objects interacting through strictly predefined interfaces. Complex technical systems usually consist of many standard parts that are necessarily connected to each other. Similarly, software systems consist of instances of classes that interact with each other. Based on these observations, three new significant concepts were added to the CIAO v.3 language.

First, the CIAO v.3 language version explicitly distinguishes between *classes* and *instances* of automaton objects. A class describes the general behavior of a set of objects. Instances are created from a class by executing a constructor (instantiation), with the initial values of all local variables, including the initial state specified. Note that this method eliminates the erroneous use of uninitialized variables in the CIAO language.

Second, the *connection scheme* of automaton objects which are instances of automaton classes is explicitly described. Some pairs consisting of the provided interface of one automaton object and the required interface of another (or the same automaton object) can be related in the connection scheme if these interfaces correspond to each other in terms of type (command or query) and number and types of parameters. This means that this pair of interfaces forms an internal connection, in other words, forms a *connection* in the system. However, not all interfaces are related. Unrelated interfaces are considered *free*. Free interfaces are necessary to interact with the external environment. This means that such interfaces are *public*. Sometimes there is no need to show all interacting objects in a single diagram, and some free interfaces are actually connected to hidden objects. This means that such interfaces are *private*.

Third, according to the connection scheme, a *semantic graph* is explicitly (fully or partially) defined as a finite description of the semantics of the behavior of interacting automaton objects. The task of defining behavioral semantics is not simple. We mainly followed the ideas presented in the book [18]. Each specific execution of the algorithm determines a sequence of performed actions which is typically referred to as the *execution occurrence protocol*. The set of all possible execution occurrence protocols forms the *operational semantics of the system*. A compact representation of sets of sequences (i.e., languages), including infinite sets using oriented graphs, is well known. If we select the initial and final vertices in the oriented graph, we can easily determine a set of paths leading from the initial vertices to the final ones. Each path corresponds to an execution protocol. If the graph contains contours (loops), the length of paths passing along the contours is not limited; therefore, the set of paths is infinite. The method to compactly represent sets of sequences is not universal. Sets of sequences (languages) that can be represented in this way are called regular languages; they are well studied [16, 19]. A regular language can be described in different ways: grammatically using generative automaton grammar, algebraically using a regular expression, and graphically using an oriented graph. We call a *semantic graph* an oriented graph that describes a set of execution protocols, i.e., the semantics of the behavior of an automaton program.

Fig. 1 shows the main components of the CIAO v.3 language. On the Fig. 1 (*a*, left) is a diagram of an automata class, including a state machine, local variables, and a set of provided and required interfaces. In this example, the state machine has two states, **s** and **t**, and two transitions: from **s** to **t** with event **a**, guard condition **c**, and effect **d** (where **a**, **c**, **d** are interfaces of the state machine class), and the second transition from **s** to **s** with event **a** and negation of guard condition **!c**. Here, local variable **s** is the name of the current state. The provided interface **b** is a Boolean function that returns true if the current state has the name "**s**". The connection scheme in Fig. 1, *b*, shows the interfaces through which state machine objects interact. Both objects **p** and **q** are instances of the state

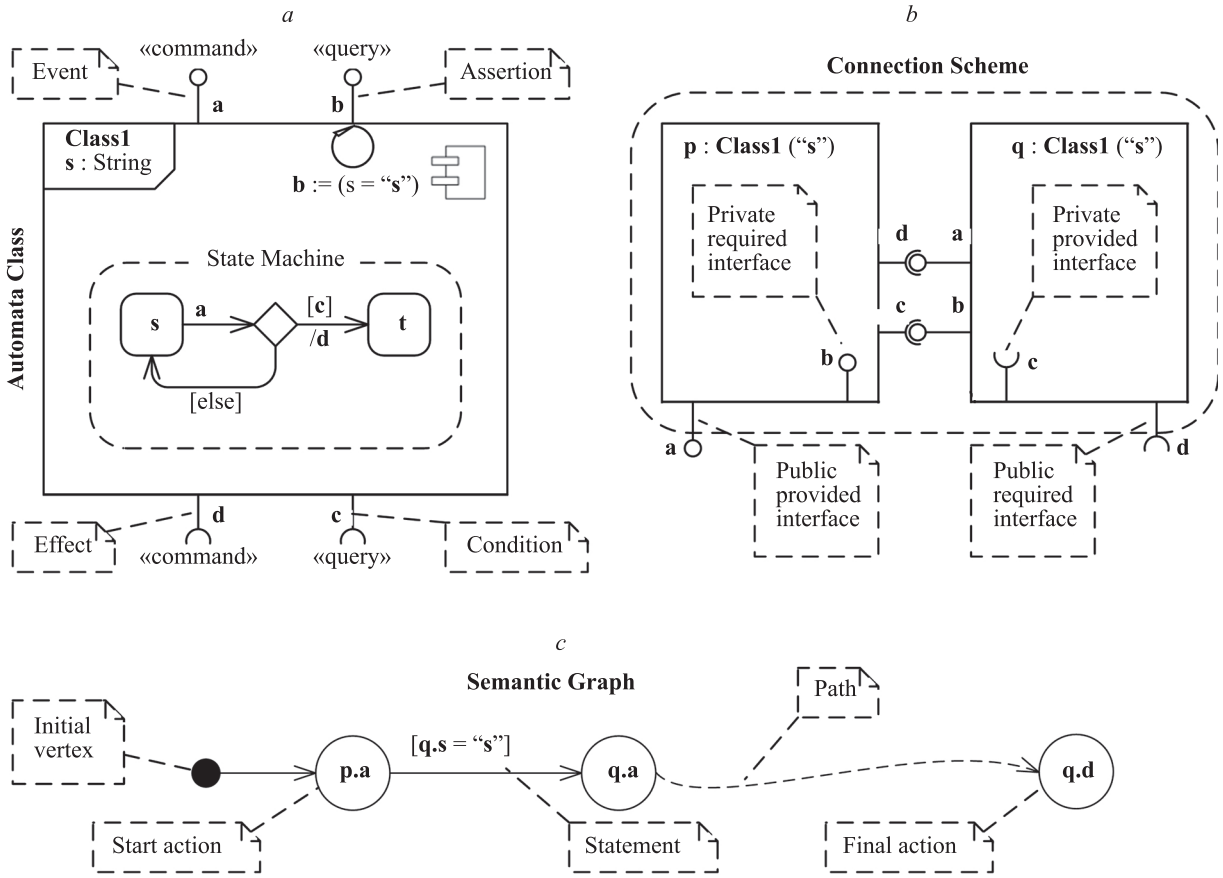

Fig. 1. Automata Class (*a*), Connection Scheme (*b*), and Semantic Graph (*c*)

machine class **Class1**, and the initial state in both cases has the name "**s**". A fragment of the semantic graph is shown in Fig. 1, *c*. In this example, the path begins with the execution of command **p.a** by the automaton object **p** (input action) and possibly ends with the issuance of command **q.d** by the automaton object **q** to the outside world (output action). Part of the path is shown as a dotted line because the evaluation of guard condition **q.c** is not specified by the provided connection scheme; therefore, the semantic graph is only partially defined.

Metamodel and Abstract Syntax of the CIAO Specification Language

Let us consider in more detail all the constructs of the CIAO v.3 language, focusing on their purpose and indicating the limitations introduced into the language to achieve the main goal of this version of the language — the ability to automatically verify programs. The metamodel in the form of a UML class diagram is used as the main means of describing the formal language, allowing us to describe in detail not only the abstract syntax but also the contextual conditions of the language [20].

Fig. 2 presents the metamodel of the CIAO v.3 language which defines metaclasses and relationships between them for all specific constructs of the CIAO v.3 language. Well-known concepts such as "named element" (*NamedElement* in the metamodel), "typed element" (*TypedElement*), "expression" (*Expr*), "Boolean expression" (*BoolExpr*) are mentioned here, but they are not disclosed. These constructs are naturally defined on the basis of a specific programming language used in the implementation of the CIAO language, and there is no need to clutter the diagram with technical details.

In addition, abstract syntax is duplicated in the form of familiar productions of context-free grammar in regular form [21]. Because only abstract syntax is described, the terminal symbols in the productions are omitted. The following metasymbols were used:

— colon **:** to separate the left from the right;
— sign # to denote iteration with a separator, or the iteration by G.S. Tseytin [21];
— braces { } to denote an unordered set of objects;
— sign + to denote concatenation;
— sign | to denote alternation;
— dot symbol **.** to denote the end of a rule.

This way of describing the language is preferable in this case. The CIAO v.3 language has two presentation formats: a graphical representation in the form of diagrams in UML notation and a textual representation in the form of a sequence of symbols. We use the graphical form as a human-readable representation of an automaton program, and we use the textual form as a machine-readable form in tools. Of course, both forms are reducible to each other in a one-to-one manner.

We provide text explanations for all constructions in Fig. 2. For ease of comprehension, when comparing the explanatory text and metamodel diagram, the names of

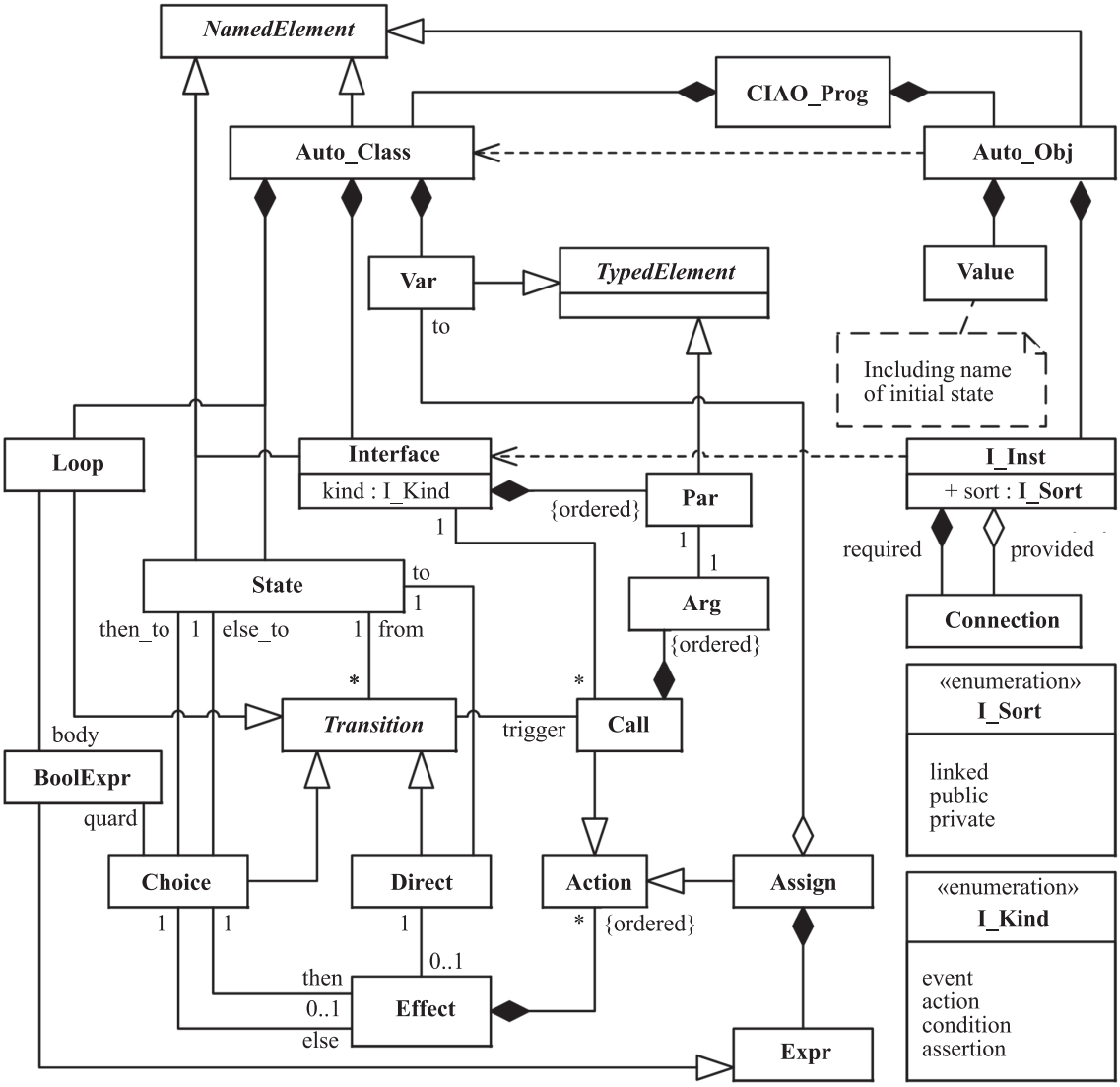

Fig. 2. Metamodel of the Language CIAO v.3

the diagram elements are repeated in the explanatory text in brackets.

A program in the CIAO v.3 language (class **CIAO_Prog**) comprises a set of descriptions of named automata classes (class **Auto_Class**) and a connection scheme of named automaton objects (class **Auto_Obj**), which are instances of these automata classes.

Abstract syntax of a program in the CIAO v.3 language:

CIAO_Prog : {Auto_class} + {Auto_Obj} .

For now, we allow only the use of automata classes in the object relationship diagram described in the given program. This somewhat old-fashioned restriction will be lifted when we develop algorithms for the incremental compilation and construction of semantic graphs. A program in the CIAO v.3 language is conceived as a single system of automaton objects interacting with each other in a single environment. Due to its uniqueness, the program does not require a personal name; however, it can, of course, be specified in the comment.

An automaton class (class **Auto_Class**) has the following components: a set of interfaces (class **Interface**), a set of local variables (class **Var**), and a set of states (class **State**). All of these sets are not empty; the elements of these sets have names that must be unique to the given program.

Abstract syntax of an automaton class:

Auto_class : {Interface} + {Var} + {State} .

Each interface has a specific kind (**I_Kind** enumeration) and can have several formal parameters (**Par** class) of a specific type. The positional method is used to identify parameters; thus, the parameters do not require names. The use of an interface in an automaton class has its own specific features for each kind of interface. The features of all four kinds of interfaces are described below.

The provided command, i.e., the *event* kind interface (the first element of the **I_Kind** enumeration), is a trigger (*trigger* role), a "trigger" of the transition between states (abstract class ***Transition***). An appeal to an interface of the *event* kind is a call (**Call** class), and the call arguments must be the names of local variables of the automata class. Call

arguments must correspond in number, order, and type to the formal parameters of the interface. The semantics of an event is that the transition is initiated, and the values of actual arguments passed from outside are assigned to local variables specified as call arguments. Thus, the event changes the state of an object in a given automaton class.

Abstract syntax for calling an event:

Call : Interface.name + **Var**.name # **.**

Here, as in the following, the construction of the form **Class_name.**attribute_name denotes, as usual, the value of a class attribute. In this case, the elements of the **Interface** and **Var** classes are named, and the constructions of **Interface**.name and **Var**.name denote the names of the interface and variable, respectively.

The provided query, i.e., the *assertion* kind interface (the fourth element of the **I_Kind**), is a Boolean function that delivers the truth value of some assertion relative to the values of local variables. This function is defined by a transition loop (class **Loop**) inherent in the automata class as a whole, i.e., a transition defined for all states of the class at once. An assertion kind interface invocation comprises a call (role *trigger*) and a logical assertion (class **BoolExpr**) which plays the role of the body of the Boolean function (role *body*). The arguments of the call are the names of temporary variables that differ from the names of all other elements of the automaton class, and the body of the function is a logical expression in which the atomic formulas are the comparison operators of local and temporary variables, as well as calls to conditions, i.e., calls to assertions of other objects. The logical expression is constructed in the usual manner from atomic formulas using logical connectives. We have omitted the well-known details of construction here in order not to clutter the metamodel. The semantics of an assertion is that the logical expression is evaluated, and the resulting truth value is returned to the state machine object from which the assertion is called. Thus, the assertion does not change the state of the object in the given state machine class.

Abstract syntax of statement definition:

Loop : Interface.name + **BoolExpr .**

The required command, i.e., the *action* kind interface (the second element of the **I_Kind** enumeration), is used to specify the effect (class **Effect**) during the transition between states. The effect during the transition is a finite sequence of elementary steps each of which is either a call of the interface (class **Call**) of the action kind, or an assignment (class **Assign**). Note that a sequence may be empty; in this case, the real effect of the transition is exhausted by the state change. Arguments in the action call may be arbitrary expressions over local variables of the automata class. Call arguments must correspond to the formal parameters of the interface in terms of number, order, and type. In the semantics of action, the values of the arguments are calculated and passed out as actual arguments of the associated interface of the event kind. Thus, the action call does not change the state of an object in the given automata class but may change the computational state of another object. The semantics of the assignment is that the value of the expression (class **Expr**) is calculated, and this value is assigned to a specified local variable (role *to*). The expression is constructed in the usual manner from the names of variables and constants using the signs of operations on the values of built-in types. We have omitted the well-known details of constructing expressions here in order not to clutter the metamodel. Thus, the assignment changes the computational state of an object in a given automaton class.

Abstract syntax of the effect:

Effect : # Step .
Step : Call | **Assign .**
Assign : Var.name + **Expr .**

The required query, i.e., the *condition* kind interface (the third element of the **I_Kind** enumeration), is used to determine the value of the guard condition (role *guard*) on the branching transition (class **Choice**) between states. An invocation of the condition kind interface is a call, and the arguments of the call can be expressed using any local variables of the automata class. The arguments of the call must correspond to the formal parameters of the interface in quantity, order, and type. The semantics of the condition are summarized as follows. The truth value of the corresponding assertion of another object is calculated, and the obtained truth value is used to calculate the value of the guard condition. If the guard condition is met, then a transition to a new state (role *then_to*) occurs, and the effect (class **Effect**) is performed if provided (role *then*). If the condition is not satisfied, then a transition to a new state (role *else_to*) occurs, and the effect is performed if it is provided in this case (role *else*). Thus, the condition does not change the state of the object in the given automata class. Moreover, the branching transition in CIAO v.3 is always executed, and events are never lost (unlike the UML language where if the guard condition is not met, the transition simply does not occur and the event is lost [22]).

Abstract syntax of condition:

$$\textbf{Choice : Guard} + \textbf{Effect}_{\textbf{then}} + \textbf{Effect}_{\textbf{else}} + \\ + \textbf{State}.\text{name}_{\text{then}} + \textbf{State}.\text{name}_{\text{else}} .$$

Guard : Interface.name + **Expr # .**

From here on, the subscript on the name of the construct in the production rule allows us to distinguish between two occurrences of the same construct in the right part of the rule.

Thus, all interface-related constructions are described. Next, variables and states with transitions are introduced.

The construction of variables in the CIAO v.3 language is extremely simple — only scalar variables of built-in types are used, local in the automaton class. A variable has a name and type. Among the local variables, there is a predefined variable *s* of the built-in String type, which is intended to store the name of the current state.

States (class **State**) in the CIAO v.3 language are considered only stable; that is, transition to another state

1. **CIAO_Prog : {Auto_class} + {Auto_Obj} .**
2. **Auto_class : {Interface} + {Var} + {State} .**
3. **Auto_Obj :** name + **Auto_class**.name + **Vars** + **Sorts** + **Links .**
4. **Call : Interface**.name + **Var**.name **# .**
5. **Loop : Interface**.name + **Var**.name **# .**
6. **State : name + {Transition} .**
7. **Transition : Direct | Choice .**
8. **Choice : Guard + Effect**$_{then}$ **+ Effect**$_{else}$ **+ State**.name$_{then}$ **+ State**.name$_{else}$ **.**
9. **Guard : Interface**.name **+ Expr # .**
10. **Direct : Call + Effect + State**.name **.**
11. **Effect : # Step .**
12. **Step : Call | Assign .**
13. **Assign : Var**.name **+ Expr .**
14. **Vars : Value # .**
15. **Sorts : {Interface**.name **+ I_Sort} .**
16. **Links : {Interface**.name$_{required}$ **+ Interface**.name$_{provided}$**} .**

Fig. 3. Grammar of CIAO v.3

occurs upon an event (role *trigger*). State transitions (abstract class ***Transition***) are linked in the CIAO v.3 language to the state from which they are carried out (role *from*). A state that does not have incoming transitions can be used as the initial transition and is specified in the constructor of an automaton object. A state that does not involve outgoing transitions can be used as the final state. In the final state, the automaton object does not respond to events but stores variables and provides assertion values. A direct transition (class **Direct**) has one source and one target state, and a branching transition (class **Choice**) has one source and two target states. For a loop (class **Loop**), the source and target states do not matter because the transition along the loop (that is, the evaluation of the assertion) does not change the state and is executed in any state in the same manner.

Abstract syntax for describing a state and its transitions:

State : name + {***Transition***} **.**
***Transition* : Direct | Choice .**
Direct : Call + Effect + State.name **.**

Let us now consider the connection scheme, i.e., the set of automaton objects (class **Auto_Obj**). Each automaton object in the relationship diagram includes three sorts of interconnected constructions.

First, a set of variable values (class **Value**). The initial values of all variables must be specified when executing the constructor of each object [23, 24].

Second, for each interface of each object, it is necessary to specify its sort (attribute + sort). This indicates the sort of interface as follows:

1) public, i.e., intended to interact with the external environment (the second element of the **I_Sort** enumeration);
2) private, i.e., hidden in this relationship diagram (the third element of the **I_Sort** enumeration);
3) linked, i.e., intended to interact with automaton objects (the first element of the **I_Sort** enumeration).

Third, it is necessary to specify the connections (class **Connection**), that is, to indicate which required interface of one object (role *required*) is linked to which provided interface of another object (role *provided*). The linked interfaces must match in type (action — event, condition — statement), and number and types of parameters. The relationships are described with reference to an automaton object on the side of the required interface.

Abstract syntax of an automaton object in a connection scheme:

Auto_obj : name + **Auto_class**.name +
+ Vars + Sorts + Links .
Vars : Value # . **Sorts : {Interface**.name **+ I_Sort} .**
Links : {Interface.name$_{required}$ **+ Interface**.name$_{provided}$**} .**

A complete list of CIAO v.3 abstract syntax rules is shown in Fig. 3.

Discussion and Conclusions

The CIAO language has been tested in practice many times and has shown good applicability. The new version of the CIAO language v.3 has retained the main competitive advantages of the previous versions. To simplify the language, some constructions ("syntactic sugar") were removed and three fundamental innovations were added: automaton classes, automaton objects, a scheme link, and a semantic graph.

Automaton classes and objects allow including all the advantages of object-oriented programming in the automaton-based programming paradigm.

Due to the introduction of a scheme link, it became possible to link automaton objects through arbitrary corresponding interfaces. The provided interface of one automaton object can be linked to the required interfaces of several other automaton objects, while the first automaton object will "obey several commanders". And vice versa — the required interface of one automaton object can be linked to the provided interfaces of several other automaton objects, while the first automaton object will "command several subordinates". Finally, it is possible to link the interfaces recursively with the interfaces of the same object, and send events to itself. The introduction of the concept of a semantic graph made it possible to develop and implement automatic verification of automaton programs with respect to formal specifications (example in [22]).

References

1. Shalyto A.A. Software implementation of the control automata. *Sudostroitel'naja promyshlennost'. Serija «Avtomatika i telemehanika»*, 1991, vol. 13, pp. 41–42. (in Russian)
2. Harel D. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 1987, vol. 8, no. 3, pp. 231–274. https://doi.org/10.1016/0167-6423(87)90035-9
3. Harel D., Pnueli A. On the development of reactive systems. *Logics and Models of Concurrent Systems*. Berlin, Heidelberg, Springer, 1985, pp. 477–498. https://doi.org/10.1007/978-3-642-82453-1_17
4. Harel D., Feldman Y.A. *Algorithmics: The Spirit of Computing*. London, Pearson Education, 2004, 513 p. https://doi.org/10.1007/978-3-642-27266-0
5. Shalyto A.A. *Switch-Technology. Algorithmization and Programming of the Logical Control Problems*. St. Petersburg, Nauka Publ., 1998, 617 p. (in Russian)
6. Polikarpova N.I., Shalyto A.A. *Automata-Based Programming*. St. Petersburg, Piter Publ., 2011, 176 p. (in Russian)
7. Shalyto A. Automata-based programming paradigm. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2008, vol. 53, pp. 3–23. (in Russian)
8. Novikov F.A., Afanasieva I.V. Cooperative interaction of automata objects. *Information and Control Systems*, 2016, no. 6, pp. 50–64. (in Russian). https://doi.org/10.15217/issn1684-8853.2016.6.50
9. Afanasieva I.V. Data acquisition and control system for high-performance large-area CCD systems. *Astrophysical Bulletin*, 2015, vol. 70, no. 2, pp. 232–237. https://doi.org/10.1134/S1990341315020108
10. Levonevskiy D., Novikov F., Fedorchenko L., Afanasieva I. Verification of internet protocol properties using cooperating automaton objects. *Proc. of the 12th International Conference on Security of Information and Networks (SIN'19)*, 2019, pp. 1–4. https://doi.org/10.1145/3357613.3357639
11. Afanasieva I., Novikov F., Fedorchenko L. Methodology for development of event-driven software systems using CIAO specification language. *SPIIRAS Proceedings*, 2020, no. 19, no. 3, pp. 481–514. (in Russian). https://doi.org/10.15622/sp.2020.19.3.1
12. Novikov F., Fedorchenko L., Vorobiev V., Fatkieva R., Levonevskiy D. Attribute-based approach of defining the secure behavior of automata objects. *Proc. of the 10th International Conference on Security of Information and Networks (SIN'17)*, 2017, pp. 67–72. https://doi.org/10.1145/3136825.3136887
13. Novikov F.A., Ivanov D.Iu. *UML Modeling. Theory, Practice, and Video Course*. St. Petersburg, Professional'naja Literature Publ., 2010, 649 p. (in Russian).
14. Afanasieva I.V., Novikov F.A., Fedorchenko L.N. Verification of event-driven software systems using the specification language of cooperating automata objects. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2023, vol. 23, no. 4, pp. 750–756. https://doi.org/10.17586/2226-1494-2023-23-4-750-756
15. Rumbaugh J., Jacobson I., Booch G. *The Unified Modeling Language Reference Manual*. 2nd ed. Addison-Wesley Professional, 2010.
16. Hopcroft J.E., Motwani R., Ullman J.D. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Boston, Addison-Wesley, 2006, 535 p.
17. Meyer B. *Object-Oriented Software Construction*. 2nd ed. Prentice-Hall, 1997, 1254 p.
18. Lavrov S.S. *Programming. Mathematical Foundations, Instrumentation, Theory*. St. Petersburg, BHV-Petersburg Publ., 2001, 320 p. (in Russian)
19. Friedl J.E.F. *Mastering Regular Expressions*. 3rd ed. O'Reilly, 2006.
20. Novikov F.A., Tikhonova U.N. An automata based method for domain specific languages definition. Part 3. *Information and Control Systems*, 2010, no. 3, pp. 29–37. (in Russian).
21. Fedorchenko L., Baranov S. Equivalent Transformations and Regularization in Context-Free Grammars. *Cybernetics and Information Technologies*, 2015, vol. 14, no. 4, pp. 29–44. https://doi.org/10.1515/cait-2014-0003
22. Novikov F.A., Afanasieva I.V., Fedorchenko L.N., Kharisova T.A. Application of conditional regular expressions in the problems of verification of control automata programs. *Proc. of the XIV All-Russian Conference on Management Problems (VSPU-2024)*, Moscow, V.A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences, 2024, pp. 2960–2964. (in Russian)

Литература

1. Шалыто А.А. Программная реализация управляющих автоматов // Судостроительная промышленность. Сер. Автоматика и телемеханика. 1991. № 13. С. 41–42.
2. Harel D. Statecharts: a visual formalism for complex systems // Science of Computer Programming. 1987. V. 8. N 3. P. 231–274. https://doi.org/10.1016/0167-6423(87)90035-9
3. Harel D., Pnueli A. On the development of reactive systems // Logics and Models of Concurrent Systems. Berlin, Heidelberg: Springer, 1985. P. 477–498. https://doi.org/10.1007/978-3-642-82453-1_17
4. Harel D., Feldman Y.A. Algorithmics: The Spirit of Computing. London: Pearson Education, 2004. 513 p. https://doi.org/10.1007/978-3-642-27266-0
5. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. 617 с.
6. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. СПб.: Питер, 2011. 176 с.
7. Шалыто А.А. Парадигма автоматного программирования // Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. 2008. № 53. С. 3–23.
8. Новиков Ф.А., Афанасьева И.В. Кооперативное взаимодействие автоматных объектов // Информационно-управляющие системы. 2016. № 6. С. 50–63. https://doi.org/10.15217/issn1684-8853.2016.6.50
9. Afanasieva I.V. Data acquisition and control system for high-performance large-area CCD systems // Astrophysical Bulletin. 2015. V. 70. N 2. P. 232–237. https://doi.org/10.1134/S1990341315020108
10. Levonevskiy D., Novikov F., Fedorchenko L., Afanasieva I. Verification of Internet protocol properties using cooperating automaton objects // Proc. of the 12th International Conference on Security of Information and Networks (SIN'19). 2019. P. 1–4. https://doi.org/10.1145/3357613.3357639
11. Афанасьева И.В., Новиков Ф.А., Федорченко Л.Н. Методика построения событийно-управляемых программных систем с использованием языка спецификации CIAO // Труды СПИИРАН. 2020. Т. 19. № 3. С. 481–514. https://doi.org/10.15622/sp.2020.19.3.1
12. Novikov F., Fedorchenko L., Vorobiev V., Fatkieva R., Levonevskiy D. Attribute-based approach of defining the secure behavior of automata objects // Proc. of the 10th International Conference on Security of Information and Networks (SIN'17). 2017. P. 67–72. https://doi.org/10.1145/3136825.3136887
13. Новиков Ф.А., Иванов Д.Ю. Моделирование на UML. Теория, практика, видеокурс. СПб.: Профессиональная литература, 2010. 640 с.
14. Afanasieva I.V., Novikov F.A., Fedorchenko L.N. Verification of event-driven software systems using the specification language of cooperating automata objects // Научно-технический вестник информационных технологий, механики и оптики. 2023. Т. 23. № 4. С. 750–756. https://doi.org/10.17586/2226-1494-2023-23-4-750-756
15. Rumbaugh J., Jacobson I., Booch G. The Unified Modeling Language Reference Manual / 2nd ed. Addison-Wesley Professional, 2010.
16. Hopcroft J.E., Motwani R., Ullman J.D. Introduction to Automata Theory, Languages, and Computation / 3rd ed. Boston: Addison-Wesley, 2006. 535 p.
17. Meyer B. Object-Oriented Software Construction / 2nd ed. Prentice-Hall, 1997. 1254 p.
18. Лавров С.С. Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001. 320 с.
19. Friedl J.E.F. Mastering Regular Expressions / 3rd ed. O'Reilly, 2006.
20. Новиков Ф.А., Тихонова У.Н. Автоматный метод определения проблемно-ориентированных языков. Ч. 3 // Информационно-управляющие системы. 2010. № 3. С. 29–37.
21. Fedorchenko L., Baranov S. Equivalent transformations and regularization in context-free grammars // Cybernetics and Information Technologies. 2015. V. 14. N 4. P. 29–44. https://doi.org/10.1515/cait-2014-0003
22. Новиков Ф.А., Афанасьева И.В., Федорченко Л.Н., Харисова Т.А. Применение условных регулярных выражений в задачах верификации управляющих автоматных программ // Сборник трудов XIV Всероссийского совещания по проблемам управления (ВСПУ-2024). М.: ИПУ РАН, 2024. С. 2960–2964.
23. Meyer B. Touch of Class: Learning to Program Well with Objects and Contracts. Berlin: Springer, 2009. 876 p. https://doi.org/10.1007/978-3-540-92145-5

23. Meyer B. *Touch of Class: Learning to Program Well with Objects and Contracts*. Berlin, Springer, 2009, 876 p. https://doi.org/10.1007/978-3-540-92145-5
24. Weisfeld M. *The Object-Oriented Thought Process*. 5th ed. Addison-Wesley Professional, 2019.

24. Weisfeld M. The Object-Oriented Thought Process / 5th ed. Addison-Wesley Professional, 2019.

Authors

Fedor A. Novikov — D.Sc., Senior Researcher, Professor, Peter the Great St. Petersburg Polytechnic University (SPbPU), Saint Petersburg, 195251, Russian Federation; SC 16441904500, https://orcid.org/0000-0003-4450-0173, fedornovikov51@gmail.com

Irina V. Afanasieva — PhD, Head of Laboratory, Special Astrophysical Observatory of the Russian Academy of Sciences (SAO RAS), Nizhny Arkhyz, 369167, Russian Federation, SC 57210431774, https://orcid.org/0000-0003-4225-4124, riv615@gmail.com

Ludmila N. Fedorchenko — PhD, Senior Researcher, St. Petersburg Federal Research Center of the Russian Academy of Sciences, Saint Petersburg, 199178, Russian Federation, SC 36561350100, https://orcid.org/0000-0002-4008-9316, lnf@iias.spb.su

Taisia A. Kharisova — Engineer, Ioffe Institute, Saint Petersburg, 194021, Russian Federation, https://orcid.org/0009-0002-3456-0471, tais.harisova@mail.ru

Авторы

Новиков Федор Александрович — доктор технических наук, старший научный сотрудник, профессор, Санкт-Петербургский политехнический университет Петра Великого, Санкт-Петербург, 195251, Российская Федерация, SC 16441904500, https://orcid.org/0000-0003-4450-0173, fedornovikov51@gmail.com

Афанасьева Ирина Викторовна — кандидат технических наук, заведующий лабораторией, Специальная астрофизическая обсерватория Российской академии наук, Карачаево-Черкесская Республика, Зеленчукский р-н, пос. Нижний Архыз, 369167, Российская Федерация, SC 57210431774, https://orcid.org/0000-0003-4225-4124, riv615@gmail.com

Федорченко Людмила Николаевна — кандидат технических наук, старший научный сотрудник, Санкт-Петербургский Федеральный исследовательский центр Российской академии наук, Санкт-Петербург, 199178, Российская Федерация, SC 36561350100, https://orcid.org/0000-0002-4008-9316, lnf@iias.spb.su

Таисия Анваровна Харисова — инженер, Физико-технический институт им. А.Ф. Иоффе Российской академии наук, Санкт-Петербург, 194021, Российская Федерация, https://orcid.org/0009-0002-3456-0471, tais.harisova@mail.ru

Received 02.05.2024
Approved after reviewing 30.10.2024
Accepted 25.11.2024

Статья поступила в редакцию 02.05.2024
Одобрена после рецензирования 30.10.2024
Принята к печати 25.11.2024