НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
январь–февраль 2026 Том 26 № 1 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
January–February 2026 Vol. 26 No 1 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

doi: 10.17586/2226-1494-2026-26-1-154-164

Implementation of cooperative interaction of automaton objects

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

[1] Peter the Great St. Petersburg Polytechnic University, 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
[3] St. Petersburg State University (SPbSU), Saint Petersburg, 199034, 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

This paper addresses the issues related to the implementation of the interaction of automaton objects formalized using specialized state transition graphs. This representation approach, similar to state machine diagrams in Unified Modeling Language, significantly simplifies the development and subsequent maintenance of software. Each automaton object manages specific behavioral aspects of the system while their interaction through the corresponding interfaces ensures the achievement of common goals. Visualization of these objects is implemented using the CIAO (Cooperative Interaction of Automaton Objects) v.3 automata-based programming language. The implementation of the interaction mechanism involves developing a software system that supports the joint execution and interaction of automaton objects. To implement the proposed automaton objects interaction, the bootstrapping technique, known since the mid-1960s, is used. This method involves creating a compiler or interpreter in the same language for which it is being developed. The stepwise refinement method is used to construct the initial interpreter. Subsequently, using transformation patterns from imperative to automata-based constructs, the interpreter is modified into a system of interacting automaton objects, thus achieving the result of the bootstrapping process. This research yielded data structures for representing CIAO v.3 programs. The interpreter's structure was described in pseudocode using the stepwise refinement method. A set of patterns is proposed to implement imperative constructs through automata-based programming techniques. The structure of the CIAO v.3 language interpreter is presented using CIAO v.3 itself. A Python-based interpreter prototype was realized. The conducted study demonstrates the successful software self-implementation of the CIAO v.3 using the bootstrapping method. The CIAO v.3 language provides efficient design and implementation of software solutions, and also guarantees fault-tolerant component interaction due to the ability to automatically check the properties of CIAO v.3 programs. The proposed approach can be utilized for implementing domain-specific languages in multi-agent systems and human-machine interaction interfaces.

Keywords

automata-based programming, state transition graph, UML, finite state machine diagram, bootstrapping method, stepwise refinement method, transformation of imperative programs into automata-based programs

Acknowledgments

The author, L.N. Fedorchenko, carried out the research within the framework of the State Research Assignment (topic No. FFZF-2025-0006).

For citation: Novikov F.A., Afanasieva I.V., Fedorchenko L.N., Kharisova T.A. Implementation of cooperative interaction of automaton objects. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2026, vol. 26, no. 1, pp. 154–164. doi: 10.17586/2226-1494-2026-26-1-154-164

УДК 004.415.52, 004.434

Реализация кооперативного 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, Нижний Архыз, 369167, Российская Федерация
[3] Санкт-Петербургский Федеральный исследовательский центр РАН, Санкт-Петербург, 199178, Российская Федерация
[3] Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, 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. Рассмотрены вопросы реализации взаимодействия автоматных объектов, формализованных посредством специализированных графов переходов состояний. Данный подход к представлению подобен диаграммам автоматов в UML (Unified Modeling Language), существенно облегчает разработку и последующее сопровождение программного обеспечения. Каждый автоматный объект управляет определенными аспектами поведения системы, а их взаимодействие через соответствующие интерфейсы обеспечивает достижение общих целей. Визуализация таких объектов реализуется с помощью языка автоматного программирования CIAO (Cooperative Interaction of Automaton Objects) v.3. Использование механизма взаимодействия предполагает создание программной системы, которая поддерживает совместное выполнение и взаимодействие автоматных объектов. **Метод.** Для выполнения предлагаемого автоматного взаимодействия применяется метод раскрутки, который известен с середины 1960-х годов и представляет собой технику создания компилятора или интерпретатора на том же языке, для которого он создается. Для построения начального интерпретатора используется метод пошагового уточнения. Затем, на основе шаблонов преобразования императивных конструкций в автоматные, интерпретатор модифицируется в систему взаимодействующих автоматных объектов, что является итогом процесса раскрутки. **Основные результаты.** Разработаны структуры данных для представления программ на языке CIAO v.3, описана структура интерпретатора на псевдокоде методом пошагового уточнения. Предложены шаблоны для реализации императивных конструкций средствами автоматного программирования. Представлена структура интерпретатора языка CIAO v.3 на данном языке. Реализован прототип интерпретатора на языке Python. **Обсуждение.** Полученные результаты продемонстрировали успешную программную самореализацию языка CIAO v.3 методом раскрутки. Показано, что язык CIAO v.3 обеспечивает эффективное проектирование и реализацию программных решений, а также гарантирует безотказное взаимодействие компонентов за счет возможности автоматической проверки свойств программ на языке CIAO v.3. Предложенный подход может быть использован для реализации предметно-ориентированных языков в мультиагентных системах и интерфейсах человеко-машинного взаимодействия.

Ключевые слова

автоматное программирование, граф переходов состояний, UML, диаграмма конечного автомата, метод раскрутки, метод пошагового уточнения, преобразование императивных программ в автоматные

Благодарности

Автор Л.Н. Федорченко выполняла исследование в рамках Государственного научно-исследовательского задания (тема № FFZF-2025-0006).

Ссылка для цитирования: Новиков Ф.А., Афанасьева И.В., Федорченко Л.Н., Харисова Т.А. Реализация кооперативного взаимодействия автоматных объектов // Научно-технический вестник информационных технологий, механики и оптики. 2026. Т. 26, № 1. С. 154–164 (на англ. яз.). doi: 10.17586/2226-1494-2026-26-1-154-164

Introduction

This article examines mechanisms for implementing interactions between automaton objects based on the concept of self-applicability, presented through the development of a CIAO (Cooperative Interaction of Automaton Objects) language interpreter using the same language. Interacting automaton objects are described in the CIAO v.3 automata-based programming language [1], which allows for the automatic verification of certain properties of programs written in this language [2]. The implementation is based on the bootstrapping method; that is, the system that enables interaction between automaton objects is described as a system of interacting automaton objects. This testing technique is generally accepted in the field of programming language implementation.

The novelty of the proposed approach lies in the combined use of domain-specific languages in text form and specialized diagrams in graphical form.

The CIAO v.3 language strictly adheres to the automata-based programming paradigm [3] in its core concepts of explicit state extraction and event-based control. Implementation issues of automata-based programming systems have been addressed in numerous sources, starting with the seminal book [4] and continuing in many other works, including [5]. These studies, along

with related research, have significantly influenced the presented approach, especially the UniMod project [6], as discussed in [7]. A previous publication [8] described the implementation of the CIAO v.2 language using the bootstrapping method and its subsequent compilation to the C++ programming language. This article emphasizes the differences between the new implementation and its predecessors, which arise from the unique features of the CIAO v.3 language.

The purpose of this article is to describe the representation of the language automaton objects and demonstrate the expressive power of the language through an example implementation of the CIAO v.3 language interpreter using the bootstrapping method. The article is structured as follows. The storage structure of a system of automaton objects in computer memory during interaction is examined. The history and main principles of the bootstrapping method are briefly outlined. The first step of the bootstrapping method is presented: the implementation of a starting interpreter using stepwise refinement. The second step involves defining patterns for transforming imperative constructs into automaton constructs, and using these templates to transform the starting interpreter into a system of interacting automaton objects in the CIAO v.3 language.

Efficient representation of automaton objects

The CIAO v.3 language is implemented using a hybrid method consisting of two stages: first, compiling a CIAO v.3 program into an internal representation (described in this section), followed by interpreting the internal representation using the algorithm detailed in the subsequent sections. In this context, the interpreter implementation language remains unspecified, with pseudocode being utilized. The internal representation during interpretation can vary significantly based on specific interpretation requirements (e.g., execution safety requirements or code obfuscation needs). However, this implementation imposes no special requirements beyond the simplest possible data structure for representing a CIAO v.3 program according to the language metamodel. The language metamodel [1, Fig. 2], is absolutely essential for constructing the internal representation. On the other hand, it proves to be redundant during interpreter creation, where behavior is completely determined by the fixed internal representation.

The complete definition of the internal representation of automaton objects used in interpretation is provided in Listing 1. This definition utilizes four types of constructors: the direct product of types (denoted by the **struct** keyword), mapping (denoted by **array**), discriminated union (denoted by **union** [9]), and enumerated types (denoted by **enum**). The internal representation is implemented as a set of definitions of the object types forming the representation. Notably, the components of the internal representation share identical names with both the metamodel classes and the nonterminals of the abstract syntax grammar [1]. This correspondence is intentional and results from the domain language implementation approach described in this article. Note that the primitive types **nat** (natural number), **bool** (truth value), and **name** (character string) are considered predefined. The type constructors Value (representing values of any supported type), Type (enumeration of types supported by a particular implementation), and Expr (expressions constructed from operations and values of types supported by a particular implementation) are excluded from the type structure in Listing 1 for three reasons. Firstly, these types are highly implementation-dependent. Secondly, the methods for constructing these types are well-known, and there is no need to repeat them here. Thirdly, a comprehensive description of all implementation details would take up considerable space without being specific to this article.

This representation provides direct, efficient access to all elements of automaton objects using combinations of indices and selectors. Let us illustrate this with an example. Suppose we have an automaton object named *a* and a provided interface named *p*. Then the expression `CIAO_Prog[a].s` retrieves the current state name *s* of the automaton object *a*, and the expression `CIAO_Prog[a].S[s][p].t.n` retrieves the number of parameters *n* on the transition from state *s* for the event received via interface *p*. Notably, that not all classes of the CIAO v.3 metamodel are represented in the components of the internal representation. In particular, the **Auto_class** is not represented, and a CIAO v.3 program consists of named automaton objects created at the stage of compiling the internal representation. This solution leads to copying the general class information in each instance, which slightly increases memory usage but greatly simplifies the implementation of dynamic mutation of automaton objects. Furthermore, the connection scheme is not a separate entity. This is because the information carried by the interface connection scheme can be processed immediately during compilation (or, more precisely, during parsing), allowing the processed data to be directly embedded into the internal representation.

For example, if a transition invokes an action *d*, which is implemented by an interface *p*, and the connection scheme specifies that the required interface *p* of a given automaton object is associated with the provided interface *q* of another (or even the same) automaton object *a*, with the provided interface *q* being implemented by an event *e*, then the pair (*a*, *e*) can be written directly into the transition internal representation instead of the entire long chain of connections required for implementing action *d*. This technique is called compile-time constant computation and is one of the most powerful efficiency enhancers.

Along with the internal representation of the automaton program, the interpreter requires dynamic memory during execution to store intermediate data during interpretation. These internal data, represented by the **Event** class, is referred to as events. In CIAO v.3, an event is considered to have a certain number of arguments (possibly zero) and is addressed to a specific automaton object. Since each automaton object is always in a certain state, and this information is stored in the internal representation, any event uniquely defines the transition it triggers. Therefore, an event provides sufficient information to access all elements of the internal representation. It is convenient to store all event-related information in a single data structure (Listing 2).

Listing 1. Internal representation of the program in the CIAO v.3 language

```
CIAO_Prog = array [ name ] of Auto_Obj ;                \\ automaton objects
Auto_Obj  = struct { s : name ,                         \\ current state
                     V : array [ name ] of Var ,        \\ variables
                     P : array [ name ] of Interface ,\\ interfaces
                     S : array [ name ] of State } ;    \\ states
Var       = struct { v : Value ,                        \\ current value
                     t : Type ,                         \\ variable type
                     r : bool } ;                       \\ is initialized
I_Kind    = enum (event, action, condition, assertion) ; \\ interface kind
I_Sort    = enum (linked, private, public) ;            \\ interface sort
Interface = struct { kind : I_Kind ,                    \\ interface kind
                     sort :  I_Sort ,                   \\ interface sort
                     link : struct { a : name ,         \\ linked Auto_Obj name
                     p : name } } ;                     \\ linked interface name
State     = array [ name ] of Transition ;              \\ outgoing transitions
Transition = struct { t : Trigger ,                     \\ transition event
                      m : union tag : enum (loop, direct, choice) of { \\ cases
                            l : Loop ,                  \\ loop transition
                            d : Direct ,                \\ direct transition
                            c : Choice } } ;            \\ choice transition
Loop      = struct { b : BoolExpr } ;                   \\ assertion body
Direct    = struct { f : Effect ,                       \\ transition effect
                     s : name } ;                       \\ target state
Choice    = struct { c : BoolExpr ,                     \\ guard condition
                     f1 : Effect ,                      \\ then effect
                     s1 : name ,                        \\ then target state
                     f0 : Effect ,                      \\ else effect
                     s0 : name } ;                      \\ else target state
Trigger   = struct { i : name ,                         \\ interface name
                     n : nat ,                          \\ arity
                     X : array [1..n] of name } ;       \\ variable names
Effect    = struct { n : nat ,                          \\ number of actions
                     A : array [1..n] of Action } ;     \\ sequence of actions
Action    = union tag : enum (call, assign) of {        \\ action
              d : Call ,                                \\ call provided effect
              m : Assign } ;                            \\ variable assignment
Call      = struct { i : name ,                         \\ interface name
                     n : nat ,                          \\ arity
                     X : array [1..n] of Expr } ;       \\ parameters
Assign    = struct { v : name ,                         \\ LHS variable name
                     x : Expr } ;                       \\ RHS expression
```

If simplicity of implementation is prioritized, organizing the interpreter work merely involves placing events in the event queue and retrieving them from it. In this scenario, it is necessary to implement queue methods (Listing 3), which is a basic programming exercise for junior students and is beyond the scope of the detailed discussion in this article.

Let us emphasize once again that the internal representation stores *all* the information contained in the source program written in the CIAO v.3 language.

However, not all of this information is utilized during program interpretation. Some of this information is required for performing other operations on the program, such as context condition validation, possible optimization,

Listing 2. Event storage structure

```
Event = struct { e : name ,                     \\ event name
                 a : name ,                     \\ automaton object name
                 n : nat ,                      \\ arity
                 X : array [1..n] of Value } ; \\ arguments
```

Listing 3. Event queue methods

```
initQ () ;            \\ queue setup
isReadyQ () : bool ;  \\ the queue has items
putQ (Event) ;        \\ put event in the queue
moveQ () ;            \\ advance queue
getQ () : Event ;     \\ retrieve next event
```

checking compliance with requirements, etc. In such cases, the most reliable solution is to apply the encapsulation principle. Access to the internal representation data structures should be organized using a set of methods. These methods are traditionally divided into two categories: value retrieval methods (method names typically start with *get*) and value modification methods (method names usually begin with *set*). Listing 4 lists the signatures of internal representation access methods, which are sufficient for implementing a basic "zero interpreter".

The list is completed by the *newEvent* method, which differs from the others as it is not an accessor for the static internal representation, but rather a constructor for dynamically created events.

Using the bootstrapping method for language implementation

The term "bootstrapping" in the development of translators (compilers or interpreters) refers to a technique that enables the creation of a translator for a new programming language using the language itself. This approach solves the fundamental problem of how to compile a compiler when no existing compiler is available. This method has been successfully applied in the development of numerous modern languages including C, Pascal, Java, Go, Rust, and others.

The fundamental concept operates as follows. Utilize an existing compiler (or interpreter) for language X to develop the initial, basic version of a translator for the new language L. Subsequently, employ this starting version to compile a more advanced version of the translator, now written in L itself. This process can be repeated iteratively, continuously "lifting" the translator by its own "bootstraps" — hence the origin of the term.

The term "bootstrapping" emerged in the 1950s within the IBM development community, but its author is uncertain. The concept was initially proposed by G. Hopper[1], who described the process of loading a compiler program using an existing minimal loader (A-0 System). Although the underlying concept existed, the term "bootstrapping" was not yet in use.

The first documented reference appears in the IBM manual for the 701 computer (1953) within the section titled "Symbolic Coding Programs". This manual contains the following phrase: "The process of loading itself into the computer by its own bootstraps is called bootstrapping". This phrase was used to describe the loading process of the Symbolic Assembly Program (SAP) assembler. The SAP source code, written in the assembler, was manually translated into machine code. This machine code was subsequently utilized to compile newer versions of the SAP program.

In 1954, Donald Gill, a member of the IBM team, formalized this method in a paper[2]. The paper contains a detailed account of the following: 1) creation of a minimal hand-coded compiler (T_0); 2) development of a compiler T_1

[1] Hopper G. IBM 701 Preliminary Manual of Operation, 1953, p. 25.

[2] Gill D. Bootstrapping and the Use of Subroutines. Proc. ACM Meeting, 1954.

Listing 4. Internal representation access methods

```
getArgsNum (e : Event) : nat ;      \\ number of event arguments
getNewS (e : Event) : name ;        \\ direct transition target state name
getEffect (e : Event) : Effect ;    \\ direct transition effect
getNewS1 (e : Event) : name ;       \\ then transition target state name
getEffect1 (e : Event) : Effect ;   \\ then transition effect
getNewS0 (e : Event) : name ;       \\ else transition target state name
getEffect0 (e : Event) : Effect ;   \\ else transition effect
getActsNum (f : Effect) : nat ;     \\ number of actions in the effect
getTrTag (e : Event) : enum (loop, direct, choice) ;  \\ transition tag
getActTag (a : Action) : enum (call, assign) ;  \\ action tag
getAct (f : Effect, i : nat) : Action ;  \\ i-th action of the effect
getGuard (e : Event) : bool ;       \\ guard condition on the transition
setArgVal2Var (e : Event, i : nat) ;  \\ assignment of the i-th event
                                    \\ argument value to the i-th variable
                                    \\ in the trigger parameter list
setExprVal2Var (a : Action) ;       \\ assignment of expression value
                                    \\ to a variable in the assignment action
setS (a : name, s : name) ;         \\ transition of object a to state s
newEvent (a : Action) : Event ;     \\ event creation by call action
```

written in the input language T_0; and 3) subsequent use of T_0 to generate machine code for T_1. N. Rochester, the architect of the IBM 701, confirmed in a 1984 interview[1]: "We called it 'bootstrapping' — like pulling yourself up by your shoelaces".

A classic example of this method is the development of the ALGOL 60 compiler (1960–1962). E. Dijkstra's team in the Netherlands successfully applied bootstrapping to the Algol 60 compiler, as explicitly documented in the report X1 Algol 60 Compiler (1962): "The compiler was bootstrapped from a minimal hand-coded version". This statement was later repeated in the Annual Review[2].

In more detail, the bootstrapping process steps in this case can be described as follows.

1. Creating the initial "zero" translator (C_0) in language *X*: The developer writes a minimal, possibly unoptimized and incomplete version of a translator for language *L*. This translator (C_0) is written in an existing language *X* (e.g., C, C++, Java, Python), for which there is already a readily available and reliable compiler or interpreter. The C_0 translator must be able to translate at least a subset of language *L* that is sufficient to enable the writing of a subsequent, more advanced translator for *L* in *L*.
2. Writing a translator in the target language (C_1): The developer writes a fully functional, well-structured compiler or interpreter for language *L* — this is C_1. The crucial aspect is that the C_1 translator is written in *L* itself.
3. Translation of C_1 using C_0: Using the C_0 zero translator (written in *X*), the C_1 source code (written in *L*) is translated into executable code (e.g., machine code or bytecode). The result is an executable file, the first version of the *L* language translator written in *L* (C_{1_exec}).
4. "Bootstrapping": using C_{1_exec} to compile itself: At this stage, we have a C_{1_exec} translator, written in *L*, which is capable of compiling code in *L*. Next, we take the C_1 source code (written in *L*) and translate it using the newly created C_{1_exec}. This process yields a new translator executable ($C_{1'\_exec}$). Theoretically, C_{1_exec} and $C_{1'\_exec}$ should be functionally identical (if the translator is correct).
5. In subsequent possible iterations, the developer improves the translator source code (C_1) in language *L*, adding new features, optimizations, and bug fixes. To produce a new executable version, they compile the improved source code using the most recent version of the translator written in *L* (C_{1_exec} or $C_{1'\_exec}$). This process continues iteratively, with each new translator version being used to compile an even more refined version of the source code.

As demonstrated above, bootstrapping represents a fundamental and elegant technique in compiler development. It allows the creation of a self-sufficient system where the programming language is capable of "pull itself up by its bootstraps", ensuring platform independence and stimulating the development of both the language itself and its compiler. Despite the initial difficulties (creating the initial compiler and debugging it), the long-term benefits make bootstrapping the preferred approach for implementing new programming languages.

Implementation of the "zero" interpreter

According to the bootstrapping method, the first step should be developing a hand-written interpreter. However, the interpreter uses an internal representation, so the zero step should involve constructing the internal representation of the CIAO program. The CIAO language syntax is quite simple, and the proposed internal representation is so straightforward that it can be implemented in many ways (as we have confirmed through experience):

— to utilize domain-specific language implementation tools, such as the syntax-driven translator SynGT [10], as described in our previous paper [8];
— to use general-purpose compiler construction tools, such as LEX/YACC [11], and/or parser generators, such as ANTLR [12];
— to develop a custom implementation using syntax-driven translation [13] or another translation technique.

The construction of an internal representation can be accomplished using well-established methods. It should also be noted that each of these techniques provides proven syntax error handling tools. These tools require significant effort to implement. However, they provide reliable protection against creating an incorrect internal representation of a CIAO v.3 program. Therefore, we can assume that the interpreted internal representation (storage) contains no errors.

The interpreter receives the internal representation of the automaton program as input, along with the name of the start event (including actual arguments if the event has formal parameters) and the name of the automaton object to which this event is addressed. To write the interpreter in pseudocode, we utilized the stepwise refinement method [14]. When considering the stepwise refinement method, it should be noted that from an implementation perspective, it is simply a macro processor [15] that performs text substitutions. Conventionally, stepwise refinement is generally applied to refining the structure of a control program. We expand the capabilities and applicability of this method by adding data connections in the form of parameters that each refinement receives [16]. In other words, we assume that macros can have typed parameters. Under these conventions, the interpreter implementation shown in Listing 5 becomes straightforward and clear.

In accordance with our omission of the details of the Expr class, which represents expressions, we also omit the implementation of the methods newEvent and setExprVal2Var, which handle expression value computation.

Based on the outlined scheme, a student team successfully developed a Python-based interpreter prototype in 2025.

[1] Rochester N. Oral History. Computer History Museum, 1984.

[2] Dijkstra E.W. Making a Translator for ALGOL 60. Annual Review in Automatic Programming, 1963. Eindhoven University of Technology.

Listing 5. Description of the CIAO v.3 language interpreter using the stepwise refinement method

```
<CIAO Language Interpreter> (e : Event) ::=
    initQ () ;                          \\ queue setup
    putQ (e) ;                          \\ put start event in the queue
while isReadyQ () do                    \\ the queue has items
    moveQ () ;                          \\ advance queue
    e = getQ () ;                       \\ retrieve next event
    <Process event> (e) ;               \\ pass event to automaton object
<Process event> (e : Event) ::=
    for i in 1..getArgsNum (e) do       \\ assign event arguments to variables
        setArgVal2Var (e, i) ;          \\ variable assignment
    switch getTrTag (e)                 \\ 3 tag transition cases
    case direct :                       \\ direct transition
        <Unconditional transition> (e) ; \\ event transition
    case choice :                       \\ guard condition transition
        <Conditional transition> (e) ;  \\ event transition
    case loop : skip ;                  \\ error ignored for now
<Unconditional transition> (e : Event) ::=
    <Execute Effect> (getEffect (e)) ;  \\ effect execution
    setS (e.a, getNewS (e)) ;           \\ state change
<Conditional transition> (e : Event) ::=
    if getGuard (e) then                \\ guard condition is satisfied
        <Execute Effect> (getEffect1 (e)) ; \\ effect execution
        setS (e.a, getNewS1 (e)) ;      \\ state change
    else                                \\ guard condition is not satisfied
        <Execute Effect> (getEffect0 (e)) ; \\ effect execution
        setS (e.a, getNewS0 (e))        \\ state change
<Execute Effect> (f : Effect) ::=
    for i in 1..getActsNum (f) do       \\ sequence of actions
        act = getAct (f, i) ;           \\ next action
        switch getActTag (act)          \\ 2 action cases
        case call :                     \\ create an event
            e = newEvent (act) ;        \\ event constructor
            putQ (e) ;                  \\ put event in the queue
        case assign :                   \\ execute the assignment
            setExprVal2Var (act) ;      \\ variable assignment
```

Methods for converting imperative programs into automata-based solutions

According to the bootstrapping method, after implementing the zero-version of the target language interpreter, the next step is to construct the first interpreter of the target language in that language itself. The target language in this case is the CIAO v.3 automata-based programming language. There are numerous methods for constructing automata-based programs [5]. The purpose of this section is not to propose any new extravagant method, but rather to demonstrate that traditional, well-tested methods work quite effectively in the case of automata-based programming using the CIAO v.3 language. In this case the following three key aspects are used.

The first aspect, which is based on [17], is that an imperative program composed according to the rules of structured programming [9] can be automatically transformed into a functionally equivalent automata-based program. This statement can be expressed most precisely and rigorously in the language of graph theory [18] as follows. A structured program, in the sense of the work [9], can be conceptualized as an algorithmic diagram consisting of diamonds with testable conditions and rectangles with executable operators connected by control flow arrows. The property of "structuredness" lies in the fact that the algorithmic diagram must allow end-to-end decomposition into three basic constructs — "sequence", "branching" and "cycle" — which have the "one input — one output" property.

The process of automated conversion to a state transition graph requires preliminary normalization of the structured algorithm scheme. The normalization procedure includes the following steps:

1) merging linear sequences of rectangles into single rectangle;
2) inserting intermediate rectangles with null operators between two consecutive branching diamonds. After normalization, according to graph theory terminology, the structured algorithm scheme can be conceptualized as a directed hypergraph where nodes correspond to rectangles, and diamonds along with their incoming/outgoing arrows serve as hyperarcs (as described in [17]).

The proposed interpretation redefines the normalized structured algorithm scheme as a directed graph composed of four fundamental node categories:

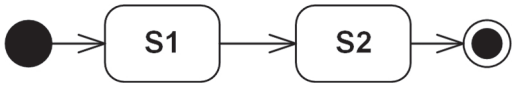
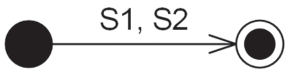
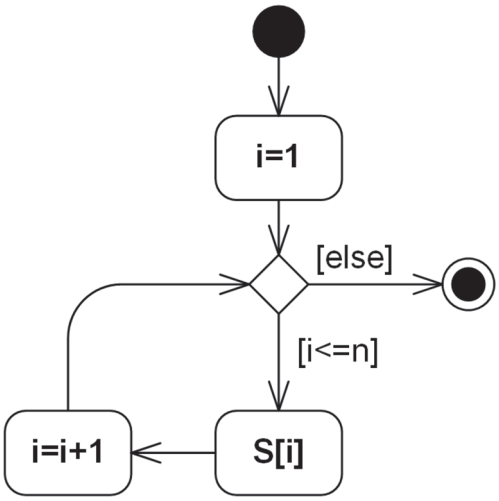
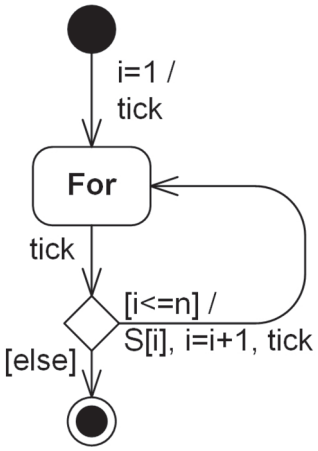
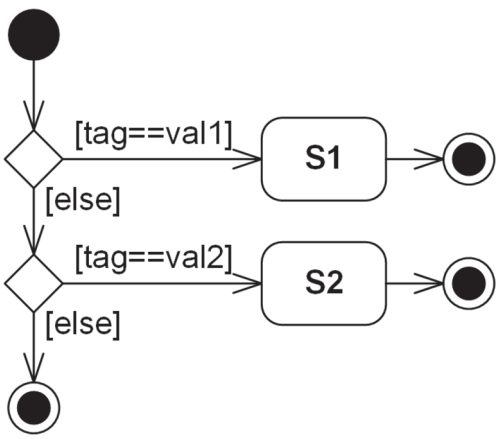
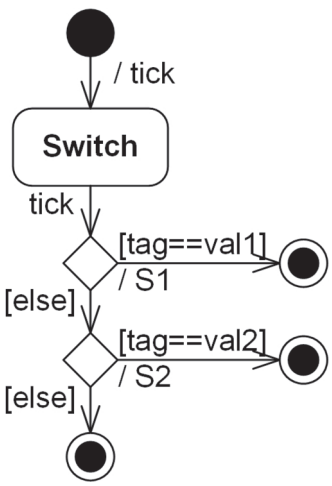
1) rectangles (operators) have exactly one input and one output arc;
2) diamonds (branching/merging points) have either one input arc and two output arcs, or two input and one output arc;
3) black circles (start points) have exactly one output arc;
4) bull's eyes (termination points) have exactly one input arc.

In the case of the interpretation presented in [17], the transformation into a state transition graph of a finite automaton involves constructing an edge graph for the algorithm scheme. In contrast, the proposed interpretation involves constructing a directed graph for which the normalized algorithm scheme serves as an incidence structure. In any case, the practical transformation occurs as follows: the diamonds become states, and the operators from the rectangles become effects on transition arcs. The transition conditions, as well as the initial and final states, remain unchanged.

The second aspect focuses on identifying "automata-based programming patterns". This involves establishing specific "building blocks" within automata-based programs that replicate the exact behavior of conventional imperative programming constructs. In this case, the following constructs were required: sequence, conditional branching, value-driven branching, precondition loop, and counter loop. Sequence and conditional branching are directly implemented in the conversion from imperative to automata form, as discussed previously. Loops can also be transformed into automata form, with a detailed example of counter loop conversion presented in Table 1.

It should be noted that the transformation from imperative to automata form is described under the assumption that automata can perform spontaneous transitions, also known as completion transitions. This principle of sequential execution — the assumption that the program executes autonomously: after one operator is

Table 1. Implementation patterns for imperative constructs in automata programs

Construction	Imperative form	CIAO language
Sequential composition **S1**; **S2**		
Counter loop **for** i **from** 1 **to** n **do** **S[i]**		
Value-driven branching **switch tag** **case** val1 : **S1** **case** val2 : **S2**		

executed, the next operator begins to execute automatically, without any external stimuli — is accepted "by default" in all forms of imperative programming. In CIAO v.2 [8], spontaneous transitions were included for convenience and to maintain compatibility with the imperative programming style.

However, in CIAO v.3 [1], completion transitions are excluded to ensure compatibility with automatic property verification algorithms [2]. Consequently, all transitions in CIAO v.3 are strictly event-driven. Therefore, the state transition graphs in Table 1 utilize a specific technique: an event called a `tick` is introduced. This event signifies that an action has been completed and the next action can begin. The automaton object self-generates this event thereby enabling full emulation of sequential execution in event-driven programs.

The third aspect in constructing automata-based programs is that the stepwise refinement method [14] works effectively in this context as well but takes on a graphical form. Specifically, any diagram fragment (a shape, a line, or a load on a shape or line) can be considered a parameter subject to refinement. In this case, by graphically substituting a syntactically appropriate refinement for the parameter (a shape instead of a shape, a line instead of a line, or text instead of text), it is possible to achieve the necessary level of detail through successive refinements of a simple initial scheme. Examples are provided in the following section.

Implementing a CIAO language interpreter in CIAO

Taking into account the discussion regarding the transformation of imperative programs into automata programs (Table 1), we apply the described methods and transform the initial version of the interpreter written in pseudocode (Listing 5) into a CIAO v.3 implementation (Figs. 1 and 2).

Initiating the process of translating the pseudocode interpreter realization into a CIAO v.3 diagram. The first refinement step (Listing 5) involves calls to `initQ; putQ(e)`, followed by a precondition loop. These method calls are combined into an effect on the initial **Idle → Running** transition. The precondition loop is implemented using the tick event, following the patterns demonstrated in Table 1. The outcome of these transformations is illustrated in Fig. 1. Three clarifying remarks are necessary here.

Firstly, the **Idle** state represents the initial state, denoted by a filled circle according to Unified Modeling Language (UML) notation. Since we combine the connection scheme diagram with the automata class diagram for brevity, the visual representation of the initial state significantly improves the diagram clarity. Secondly, the diagram displays two seemingly unconnected interfaces: `start` and `handle`. The provided `start` interface remains unconnected within the connection scheme and is intended for launching the interpreter externally. However, the required `handle` interface is actually connected to the provided interface, as illustrated in Fig. 2. This is a consequence of splitting one diagram into two separate figures for better readability. Thirdly, the `tick` event/action mechanism, which enables loop implementation in state machine form, must be unique for each control flow.

The `Queue` automaton object exists in a single instance. It should be noted that this presents a basic implementation using an unbounded dynamic array, and potential errors such as buffer overflow and underflow are not taken into account.

Continuing the transformation of the imperative interpreter implementation into an automaton-based one, we obtain two automaton objects — `Handler` and `Storage` (Fig. 2). The `Handler` object functions as the core of the interpreter and implements the algorithm for performing a single interpretation step: processing a single event. This automaton object sequentially passes through several states corresponding to the transition implementation stages: transition triggering, guard condition checking, transition effect execution, and state change.

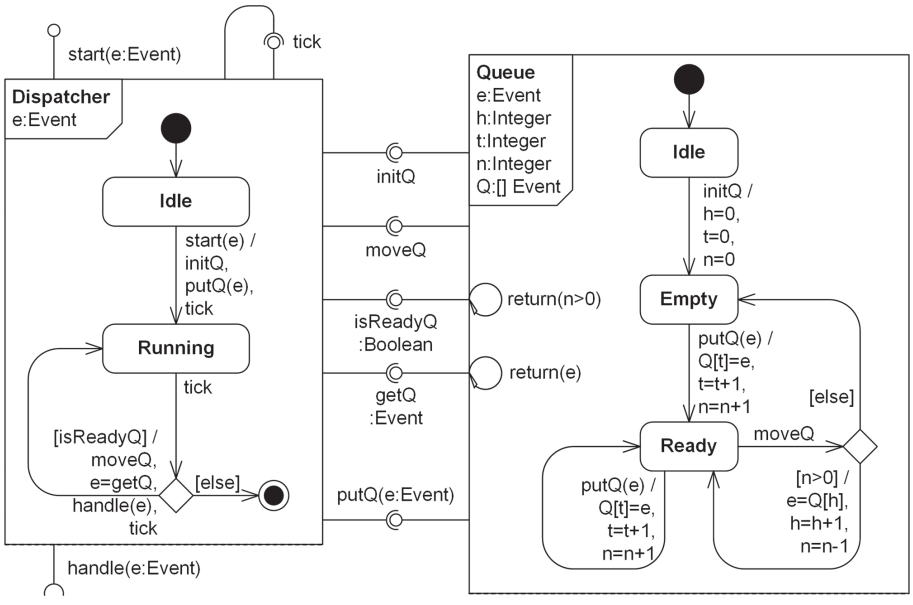


Fig. 1. `Dispatcher` and `Queue` automaton object diagrams with a connection scheme

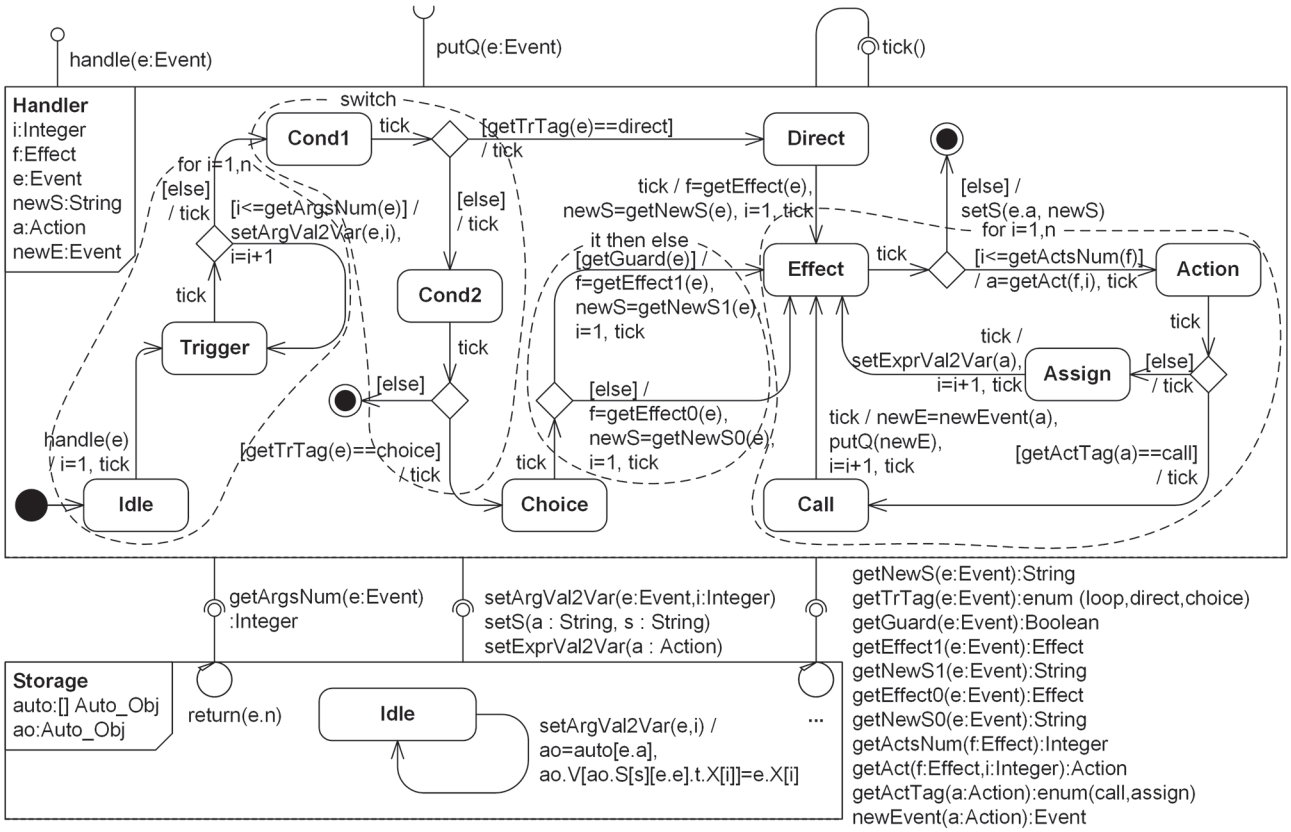


Fig. 2. `Handler` and `Storage` automaton object diagrams with a connection scheme

In the diagram in Fig. 2, the sequence is indicated by the direction of the transition arrows from left to right. The rather complex set of states of the `Handler` object was obtained almost automatically using the patterns from Table 1. In Fig. 2, the pattern applications are highlighted with dashed lines which are not part of the diagram notation and are provided for clarity purposes.

The `Storage` object is designed to store the internal representation of the automaton program. It serves as a typical example of a degenerate case of an automaton object, which maintains a constant state throughout execution due to its history-independent behavior. The object's interfaces comprise access methods to the internal representation, as listed in Listing 4. For clarity, the implementation of the two methods is presented here, while the remaining methods implementation is similar and is omitted to save space.

Conclusion and directions for further research

This article highlights the expressive capabilities of the CIAO v.3 language and its compatibility with both traditional programming methodologies (such as stepwise refinement) and modern visual programming approaches. This is demonstrated through self-implementation of CIAO v.3 using the bootstrapping method.

The future development of the CIAO language is planned to focus on designing and verifying the behavior of multi-agent asynchronous nondeterministic event-driven systems.

References

1. Novikov F.A., Afanasieva I.V., Fedorchenko L.N., Kharisova T.A. Specification language for automatabased objects cooperation. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2024, vol. 24, no. 6, pp. 1035–1043. https://doi.org/10.17586/2226-1494-2024-24-6-1035-1043
2. Novikov F.A., Afanasieva I.V., Fedorchenko L.N., Kharisova T.A. Verification of the formal requirements for the system behavior based on automaton objects. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2025, vol. 25, no. 2, pp. 328–338. https://doi.org/10.17586/2226-1494-2025-25-2-328-338
3. Shalyto A. Automata-based programming paradigm. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2008, vol. 53, pp. 3–23. (in Russian)

Литература

1. Novikov F.A., Afanasieva I.V., Fedorchenko L.N., Kharisova T.A. Specification language for automatabased objects cooperation // Scientific and Technical Journal of Information Technologies, Mechanics and Optics. 2024. V. 24. N 6. P. 1035–1043. https://doi.org/10.17586/2226-1494-2024-24-6-1035-1043
2. Novikov F.A., Afanasieva I.V., Fedorchenko L.N., Kharisova T.A. Verification of the formal requirements for the system behavior based on automaton objects // Scientific and Technical Journal of Information Technologies, Mechanics and Optics. 2025. V. 25. N 2. P. 328–338. https://doi.org/10.17586/2226-1494-2025-25-2-328-338
3. Шалыто А.А. Парадигма автоматного программирования // Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. 2008. № 53. С. 3–23.

4. Shalyto A.A. *Switch-Technology. Algorithmization and Programming of the Logical Control Problems*. St. Petersburg, Nauka Publ., 1998, 617 p. (in Russian)
5. Polikarpova N.I., Shalyto A.A. *Automata-Based Programming*. St. Petersburg, Piter Publ., 2011, 176 p. (in Russian)
6. Gurov V.S., Mazin M.A., Narvsky A.S., Shalyto A.A. Tools for Support of Automata-Based Programming. *Programming and Computer Software*, 2007, vol. 33, no. 6, pp. 343–355. https://doi.org/10.1134/s0361768807060059
7. Novikov F.A. Visual designing of programs. *Information and Control Systems*, 2005, no. 6 (19), pp. 9–22. (in Russian)
8. Afanasieva I., Novikov F., Fedorchenko L. Methodology for development of event-driven software systems using CIAO specification language. *SPIIRAS Proceedings*, 2020, vol. 19, no. 3, pp. 481–514. (in Russian) https://doi.org/10.15622/sp.2020.19.3.1
9. Dahl O.-J., Dijkstra E.W., Hoare C.A.R. *Structured Programming*. Academic Press, 1972, 220 p.
10. 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
11. Levine J.R., Mason T., Brown D. *Lex & Yacc*. O'Reilly Media, 1992, 459 p.
12. Parr T. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013. 328 p.
13. Aho A.V., Ullman J.D. *The Theory of Parsing, Translation and Compiling*, vol. 1: Parsing. Prentice-Hall, 1972. 560 p.
14. Wirth N. Program development by stepwise refinement. *Communications of the ACM*, 1971, vol. 14, no. 4, pp. 221–227. https://doi.org/10.1145/362575.362577
15. Brown P.J. *Macro Processors and Techniques for Portable Software*. Wiley, 1974, 244 p.
16. Novikov F.A. *Algorithms of Celestial Mechanics (Computer Mathematical Support Materials)*, Leningrad: Institute of Theoretical Astronomy, USSR Academy of Sciencesб 1979, 16 p. (in Russian)
17. Lukyanova A.P. *Behavior-preserving program transformations*. Master's thesis. St. Petersburg, ITMO University, 2007. URL: https://is.ifmo.ru/diploma-theses/28_07_2007_Lukianova (in Russian)
18. Novikov F.A. *Discrete Mathematics for Programmers*. St. Petersburg, Piter, 2009, 384 p. (in Russian)

4. Шалыто А.А. Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. 617 с.
5. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. СПб.: Питер, 2011. 176 с.
6. Gurov V.S., Mazin M.A., Narvsky A.S., Shalyto A.A. Tools for Support of Automata-Based Programming // Programming and Computer Software. 2007. V. 33. N 6. P. 343–355. https://doi.org/10.1134/s0361768807060059
7. Новиков Ф.А. Визуальное конструирование программ // Информационно-управляющие системы. 2005. № 6 (19). С. 9–22.
8. 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
9. Dahl O.-J., Dijkstra E.W., Hoare C.A.R. Structured Programming. Academic Press, 1972. 220 p.
10. 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
11. Levine J.R., Mason T., Brown D. Lex & Yacc. O'Reilly Media, 1992. 459 p.
12. Parr T. The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2013. 328 p.
13. Aho A.V., Ullman J.D. The Theory of Parsing, Translation and Compiling, V. 1: Parsing. Prentice-Hall, 1972. 560 p.
14. Wirth N. Program development by stepwise refinement // Communications of the ACM. 1971. V. 14. N 4. P. 221–227. https://doi.org/10.1145/362575.362577
15. Brown P.J. Macro Processors and Techniques for Portable Software. Wiley, 1974. 244 p.
16. Новиков Ф.А. Алгоритмы небесной механики: Материалы математического обеспечения ЭВМ. Ленинград: Институт теоретической астрономии АН СССР, 1979. 16 с.
17. Лукьянова А.П. Преобразования программ, сохраняющие поведение. Магистерская диссертация. СПб: СПбГУ ИТМО, 2007. URL: https://is.ifmo.ru/diploma-theses/28_07_2007_Lukianova
18. Новиков Ф.А. Дискретная математика для программистов. СПб.: ПИТЕР, 2009. 384 с.

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; PhD, St. Petersburg State University (SPbSU), Saint Petersburg, 199034, 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, SC 59492735600, https://orcid.org/0009-0002-3456-0471, tais.harisova@mail.ru

Received 24.09.2025
Approved after reviewing 22.11.2025
Accepted 25.01.2026

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, 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; доцент, Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, 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, 195251, Российская Федерация, SC 59492735600, https://orcid.org/0009-0002-3456-0471, tais.harisova@mail.ru

Статья поступила в редакцию 24.09.2025
Одобрена после рецензирования 22.11.2025
Принята к печати 25.01.2026