ISSN 0361-7688, Programming and Computer Software, 2009, Vol. 35, No. 1, pp. 43–55. © Pleiades Publishing, Ltd., 2009.
Original Russian Text © E.V. Kuzmin, V.A. Sokolov, D.Ju. Chalyy, 2009, published in Programmirovanie, 2009, Vol. 35, No. 1.

Application of the Trace Assertion Method to the Specification, Design, and Verification of Automaton Programs

E. V. Kuzmin, V. A. Sokolov, and D. Ju. Chalyy

Yaroslavl State University, ul. Sovetskaya 14, Yaroslavl, 150000 Russia

e-mail: kuzmin@uniyar.ac.ru, sokolov@uniyar.ac.ru; chaly@uniyar.ac.ru

Received April 20, 2008

Abstract—The paper considers the application of the trace assertion method [1] for specification and verification of automaton programs [2–4]. The trace assertion method allows the programmer to define an externally visible behavior of an automaton program in a rigorous way, without considering details of its implementation. The method is employed at the requirements specification stage of the system development. The paper introduces techniques for defining semantics of some elements of an automaton program, especially those involved in interactions with the control system. A formal approach to defining states of automaton programs is described. Results of studies related to the verification of specification requirements for automaton programs are also presented.

DOI: 10.1134/S036176880901006X

1. INTRODUCTION

As noted by many researchers (see, for example, [5, 6]), software is currently the most vulnerable and error-prone part of large software–hardware complexes. It is noted in [5] that the lack of confidence in the software is explained by the following reasons: history of software failures in the past (for example, [7, 8]), difficulties in software understanding and inspection, complexity of testing, and the lack of agreement between the customer and contractor about desired software functions. Therefore, the development of reliable methods for designing, inspecting, testing and verifying program systems is an important task.

The majority of approaches to the software development (comparative analysis is presented in [9], and discussion of methods used in logical control problems can be found in [2]) agree that the initial stage of a program system design should include requirements specification [6], i.e., specification of requirements that the constructed system must satisfy (in what follows, we call it simply specification). The specification is the main source of information about the functions that the system *must implement.* However, the specification should not impose any restrictions on the *way* the system is constructed. Based on this document, the program system is designed, a program code is created, and the system is tested and verified. The majority of researchers agree that an efficient analysis, design, and verification of program systems are possible only with the use of rigorous mathematical methods on all stages of its development.

It is in this direction that the technology of the automaton programming is being developed [2–4]. In the framework of this technology, new formal models of automaton programs are being proposed [10]. This technology suggests a method for designing programs as systems of interacting finite Moore–Mealy automata. The design of each automaton consists in the creation of a link scheme describing its interface and a transition graph determining its behavior by a verbal description of the desired automaton. However, the use of informal descriptions for constructing a system of interacting automata is a shortcoming of this technology, which may result in errors in the automaton program. Indeed (see, for example, [6, 11]), the most critical errors in program systems result from omissions made at the stage of specification of requirements to the system, and correction of them at later stages of program design is much more difficult than timely elimination of these errors [12]. An informal verbal problem statement greatly hampers correct determination of essential requirements raised by the customer. According to [13], when reading an informal verbal specification, the designer may face with an ambiguous interpretation, absence of a number of requirements (incomplete specification), inconsistent requirements, or unclear description. All this makes the analysis of the posed problem and the automaton program design difficult. Moreover, it may happen that that the customer will find and impose new requirements to the program, increasing, thus, the number of iterations "problem refinement–automaton program design." The situation becomes more difficult if the automaton program is designed for a system with "complex behavior" [14]. The latter is a system whose behavior depends not only on the current state but also on the previous history of system operation. Thus, there is a need for a method that would allow us to carry out a formal analysis of the posed problem, gradually refining its statement and

improving understanding of what the program is supposed to do. This method should be rather abstract in order to avoid focusing on internal, not externally visible, aspects of the implementation. For such a method, we suggest using the trace assertion method (TAM method) [1].

The paper is organized as follows. In Section 2, the approach to specification of automaton programs—the trace assertion method—is described. Basic concepts of the technology of the automaton programming relying on a hierarchical model of automaton programs [10] are described in Section 3. The use of specifications for designing automaton programs is discussed in Section 4. Verification of automaton programs on the basis of the constructed specification is discussed in Section 5. Conclusions of this work and discussion of directions of future studies are presented in the last section.

2. TRACE ASSERTION METHOD

The trace assertion method was first discussed in [1]. It is designed for specification of program modules in accordance with the information hiding principle [15]. Thus, a specification in terms of the TAM method is a "black box" description; i.e., only external observable typical features of the module are visible, whereas internal details of the implementation are hidden. Currently, there exist several modifications of this method [16–18] and program tools supporting them [19, 20]. The method can be used in practice for specification of actual control systems [13, 21–23] and, thus, presents not only theoretical, but also practical, interest.

For each module, a finite number of actions, which can change its behavior, are specified. These actions form an interface of the module with the environment and may represent different types of the interaction. In response to actions, the module can return reactions to the environment. The trace assertion method is a method for describing externally visible modes (or states) of a module by means of finite sequences of events the elements of which are pairs "action–output."

Let M be a program module and $P_M = \{p_1, \ldots, p_n\}$ be a set of actions for this module, which form an interface with the environment. Let, for each action p_i, the module can return an output, which is an element of some set T_i, and let $T_M = \bigcup_{1 \leqslant i \leqslant n} T_i$. Then, a trace is a sequence $p = p_{i_1} : o_{i_1} p_{i_2} : o_{i_2} p_{i_3} : o_{i_3} \ldots p_{i_k} : o_{i_k}$, where p_{i_m} $(1 \leqslant m \leqslant k)$ denotes an action on the module, $o_{i_m} \in T_{i_m}$ denotes the returned output, and "." denotes the concatenation operation. If no output is returned in response to some action, we assume that the output with the value *nil* is returned. Actually, a trace is a sequence belonging to the alphabet $\Delta \subseteq P_M \times T_M$. In what follows, ε will denote an empty sequence. The notation Δ^* and Δ^+ is used to denote sets of all sequences over the alphabet Δ with the empty sequence included and not included, respectively.

Let $\nu\colon \Delta^* \longrightarrow T_M \cup \{nil\}$ denote a projection that specifies the value of the last output for an arbitrary trace:

1. $\nu(\varepsilon) = nil$;
2. $\forall p : o \in \Delta\ \nu(p : o) = o$;
3. $\forall x, y \in \Delta^+\ \nu(x.y) = \nu(y)$.

Let $\pi\colon \Delta^* \longrightarrow P_M^* \cup \{\varepsilon\}$ denote a projection that, for an arbitrary trace, determines a sequence of actions applied to the module:

1. $\pi(\varepsilon) = \varepsilon$;
2. $\forall p : o \in \Delta\ \pi(p : o) = p$;
3. $\forall x, y \in \Delta^*\ \pi(x.y) = \pi(x)\pi(y)$.

A trace s is called *feasible* if, for any prefix s' of the trace s, the module admits satisfiability of the trace s'. Feasible traces determine possible scenarios of module usage.

If, for any feasible traces s_1 and s_2 of some module M, the condition

$$s_1 = s_2 \Leftrightarrow \pi(x) = \pi(y)$$

holds, then this module is said to be *output-independent.* Output-independent modules have no "internal" memory for storing the history of outputs. Such modules can store only the history of the applied actions. In this paper, we consider modules of only this kind.

Two feasible traces s_1 and s_2 are said to be *equivalent* (which is denoted as $s_1 \stackrel{E}{=} s_2$) if any subsequent, externally observable behavior of module M after the firing of either of the two traces is one and the same: $s_1 \stackrel{E}{=} s_2$ if and only if, for any trace $s \in \Delta^*$, the traces $s_1.s$ and $s_2.s$ are simultaneously feasible or not feasible. By means of the introduced equivalence relation, the set of all feasible traces is divided into equivalence classes. In this paper, we confine ourselves to systems with a finite set of equivalence classes.

A trace τ is said to be *canonical* if it represents some equivalence class. Now, let us turn to defining specifications in terms of the trace assertion method.

A *specification in terms of the trace assertion method* is given by

- *a set of actions* Σ;
- *a set of outputs* $\mathbb{O}$;
- *a set of pairs "action–output"*, $\Delta \subseteq \Sigma \times \mathbb{O}$;
- *a set of all canonical traces*, $Q \in \Delta^*$;
- *an initial canonical trace* $q_0 \in Q$;
- *a transition function* $\delta\colon Q \times \Sigma \longrightarrow Q$; and
- *a function* $v\colon Q \times \Sigma \longrightarrow Q$ *determining the returned output.*

Function δ determines what actions result in transitions between externally visible modes of system operation, which are specified by canonical traces. Function v determines outputs transmitted to the environment in

response to the actions. The output value depends on the current mode of system operation and on the action applied to the system. Upon creation of a specification, it is assumed that, at any moment, the environment can apply any action from the set Σ defined in the specification.

Consider an example of a specification prepared with the help of the TAM method for an "alarm clock" system from [14]. The alarm clock has three buttons *H*, *M*, and *A*. The buttons *H* and *M* increase the values of hours and minutes, respectively, by one (if the values of hours or minutes are equal to 23 and 59, respectively, they are set equal to zero), and the button *A* is used to turn the alarm clock on and off. After the first press of this button, the alarm clock can be set. The second press activates the clock mode (with the alarm clock being already set). The third press removes the alarm clock setting.

First, it is required to determine entities that are objects of control (they will be referred to as controlled entities) and those affecting behavior of the logical control system (observable entities). A system may have entities that are simultaneously controlled and monitored ones.

In the above example, there are two controlled entities: a clock and an alarm clock. They are simultaneously controlled and observable entities (for example, when values of hours and minutes must be set to zero upon pressing buttons *H* and *M*). A formal definition of the set of possible values of the clock and alarm clock is given in Tables 1 and 2.

From the problem description, we can also determine monitored entities, which specify actions applied to the module by means of buttons *H*, *M*, and *A*. When pressing buttons *H* and *M*, the controlled system is changed. Therefore, these actions have arguments corresponding to the current state, and the new state of the control system is an output.

Specification of the operation of the module responsible for the logical control of the clock is shown in Fig. 1. Sets of actions and outputs are defined in the section "Module interface." Canonical traces are defined in the corresponding section; they determine externally visible alarm clock system modes. The definition of the set of canonical traces is not an easy task. It can be solved by analyzing the problem under study with the help of heuristic methods, for example, by analyzing informal description of scenarios of operation of the created system provided by the customer. In our example, the definition of such a set of canonical traces followed from the fact that button *A* modified values of the output variables in different ways (difference in the behavior for sets of traces $\{A\}$ and $\{\varepsilon, A.A\}$) and from the necessity to distinguish the case where the alarm clock is set (trace $A.A$) from the case where it is not set (the set of traces $\{\varepsilon, A\}$). A set of assertions about traces that describe the transition function δ and function v specifying the returned outputs can be defined by means of an interview with the customer.

Table 1. Data types

Name	Type	Comments
tHour	{0 … 23}	The number of hours
tMinute	{0 … 59}	The number of minutes
tClock	tHour × tMin	Specifies clock

Table 2. Observable and controlled variables

Name	Type	Initial value	Comments
(h_w, m_w)	tClock	(0, 0)	Current time
(h_a, m_a)	tClock	(0, 0)	Alarm clock settin

In practical use of the trace assertion method, the so-called tabular expressions are often used [24, 25] for specifying values of the transition function δ and function v determining the outputs. Tabular expressions define complex relations and functions in a simple and clear tabular form.

Consider semantics of tabular expressions specifying functions δ and v. The tabular expressions are read row by row. Each table of a specification consists of three columns: *Condition, Trace pattern*, and *Result.* The first column contains a predicate that imposes a condition on the values of arguments of the input action. The second column contains a predicate that imposes a condition on the form of the canonical trace that is an argument of the specified function. The third column contains the value of function δ (or v) for the subset of function arguments satisfying conditions of the predicate in the first two columns joined by the logical operator "AND." In practice, if one cannot define functions in a compact and clear form, the designer of the specification may propose different form and semantics for the tabular expressions. Discussions on this subject can be found in [5, 13, 21, 24, 25].

Thus, the use of the trace assertion method suggests specifying desired behavior of the system being created. This is implemented by defining reactions to all possible input actions, which is based on the history of system operation represented in a compact form through specification of a set of canonical traces and transitions between them.

3. TECHNOLOGY OF THE AUTOMATON PROGRAMMING

The automaton programming technology [2–4, 14] is a modern Russian development, which is actively studied and supported by a number of Russian research groups. In the automaton approach to the program design, the program contains a system-independent part, which specifies logic of the automaton program

Specification of module Clock

Module interface

Action	Argument	Result
H	$(h_w, m_w) \times (h_a, m_a)$	$(h_w, m_w) \times (h_a, m_a)$
M	$(h_w, m_w) \times (h_a, m_a)$	$(h_w, m_w) \times (h_a, m_a)$
A	$\varnothing$	$\varnothing$

Canonical traces

A trace is canonical if and only if it is a prefix of the trace $t = A.A$
$t_0 = \varepsilon$

Assertions about traces

$\delta(t, H((h_w, m_w) \times (h_a, m_a))) = t$
$\delta(t, M((h_w, m_w) \times (h_a, m_a))) = t$

$\delta(t, A) =$

Condition	Trace patterns	Equivalent trace
	$\lvert t\rvert = 2$	ε
	$\lvert t\rvert < 2$	$t.A$

Outputs

$\nu(t, H((h_w, m_w) \times (h_a, m_a))) =$

	Condition	Trace patterns	Result
	$0 \le h_w < 23$	$t \ne A$	$(h_w + 1, m_w) \times (h_a, m_a)$
=	$h_w = 23$	$t \ne A$	$(0, m_w) \times (h_a, m_a)$
	$0 \le h_a < 23$	$t = A$	$(h_w, m_w) \times (h_a + 1, m_a)$
	$h_a = 23$	$t = A$	$(h_w, m_w) \times (0, m_a)$

$\nu(t, M((h_w, m_w) \times (h_a, m_a))) =$

	Condition	Trace patterns	Result
	$0 \le m_w < 59$	$t \ne A$	$(h_w, m_w + 1) \times (h_a, m_a)$
=	$m_w = 59$	$t \ne A$	$(h_w, 0) \times (h_a, m_a)$
	$0 \le m_a < 59$	$t = A$	$(h_w, m_w) \times (h_a, m_a + 1)$
	$m_a = 59$	$t = A$	$(h_w, m_w) \times (h_a, 0)$

$\nu(t, A(k)) = \varepsilon$

Notation

$|s|$ is length of trace s

Fig. 1. Specification of the alarm clock system.

and is given by a system of interacting finite Moore–Mealy automata. The automaton programming does not depend on the platform, operating system, or programming language and represents an approach that uses formal methods for constructing correct programs.

In this paper, we consider a formal hierarchical model of reactive systems and logical control systems proposed in [10]. In accordance with this model, an automaton system is considered as a system of interacting deterministic automata given by

$$\mathcal{A} = (A_0, A_{11}, \ldots, A_{1k_1}, \ldots, A_{n1}, \ldots, A_{nk_n}),$$

where n and k_i $(1 \leqslant i \leqslant n)$ are positive integers. The automaton A_0 is called main, and the others are called

nested automata. All automata are related through a hierarchy with respect to nesting. An automaton A_{ij} can transfer control to an automaton A_{i+1k} that occupies a lower level in the hierarchy. In this case, the automaton A_{ij} is said to be principal, and the automaton A_{i+1k}, nested. The automaton hierarchy forms a tree; i.e., for each nested automaton, there exists only one principal automaton in which it is nested.

The automaton system $\mathcal{A}$ is considered to be a reactive control system for a plant. The system $\mathcal{A}$ receives from the plant events that characterize, for example, change of its states and asks the plant about its current parameters, which is also considered to be an input action on $\mathcal{A}$. At the same time, the control system reacts to the arriving information and, thus, affects the plant. In addition to the above-described interaction with the "environment," the automata interact with one another inside the system by transferring control from the principal automaton to the nested ones when certain events occur and watching their current states.

For the entire system of the interacting automata $\mathcal{A}$, $Y = \{y_0, y_{11}, \ldots, y_{nk_n}\}$ will denote the set of variables that help us to track down states of the automaton system; i.e., the current state of an automaton A_{ij} is stored in variable y_{ij}.

Let us denote by $E_A = \{e_1, \ldots, e_k\}$ the set of the events to which the automaton A reacts. Let e be the variable where the current event for the automaton A is placed.

Let us introduce the set $X_A = \{x_1, \ldots, x_n\}$ of queries of the automaton A to the control system. Each query is considered as a certain predicate the truth of which depends on the state of the control system.

Let also $Z_A = \{z_1, \ldots, z_r\}$ denote the set of output actions of A. Actions z_i are classified into two groups. The first group includes direct actions on the control system. The actions from the second group model control transfers to the nested automata occurring after some events (generation of events for nested automata). In this case, the output action z_i has the form $A'(e'_j)$, where A' is a nested automaton and e'_j is an event generated by the automaton A for the nested automaton A' to which the control is transferred for processing this event.

Then, the automaton A of the control system $\mathcal{A}$ can be represented as a tuple $(\Sigma, Q, q_0, E, X, Z, \delta)$, where

1. $Q = \{q_0, q_{11}, \ldots, q_n\}$ is a finite set of states of the automaton,

2. q_0 is an initial state,

3. $\Sigma = \{a_1, a_2, \ldots, a_k\}$ is a finite alphabet of labels of the transition arcs,

4. δ: $Q \times \Sigma \longrightarrow Q$ is a function of transitions from one state to another.

Each transition fires by a certain rule. Before describing the transition rules, we introduce some notation.

For a transition label $a \in \Sigma$, $E(a)$ denotes the event to which A reacts upon firing the transition with the label a.

Let $X(a)$ denote the set of queries to the control object the truth of which is required for firing the transition with the label a.

Let Z^* be a set of finite sequences of output actions. Then, for $a \in \Sigma$, $Z^*(a) \in Z^*$ denotes the sequence of the output actions that occur when the transition with the label a fires.

For an arbitrary state $q \in Q$ of an automaton A, we introduce the notation $Z^*(q) \in Z^*$ for the sequence of output actions that are to be performed when the automaton A comes to the state q.

Finally, let $Y(a)$ be a predicate depending on the states of the nested automata. Then, the transition with the label a fires if and only if $Y(a)$ takes the true value.

The rule of the transition from a state q to a state q' by label a has the following form:

q, a : **if** $e = E(a)$ **and** $(\forall x \in X(a) : x = true)$

and $Y(a) = true$ **then** $Z^*(a)$; $Z^*(q)$; **goto** q'.

Having received an event, the automaton reacts (or does not react) to it (with reaction being determined by its current state), asks the control system about its parameters (input variables), takes into account states of the nested automata and, then, performs a sequence of output actions, including the actions that are required to perform when it occurs in a new state. Only after this, it switches to a new state.

An output action of the first kind, which is aimed at the control system, is considered to be performed immediately after the application. An output action of the second kind, which is a control transfer to a nested automaton in response, is considered to be performed only after the reaction of the nested automaton to this event. The latter reaction consists in the following: either the automaton transfers to the new state (one of the transitions fires) or the event is ignored by the nested automaton (none of the transitions can fire). Until the output action of the second type is performed, the operation of the principal automaton is postponed.

The transition firing rules for all automata of the hierarchical model are deterministic. If none of the transitions can fire in the current state when an event occurs, then the event is ignored.

4. DESIGN OF AUTOMATON PROGRAMS

In [2, 14], it is proposed to begin the design of an automaton program with the analysis of an informal text description of the system and to determine components of the control system and states of the system of the interacting automata. Such an approach has a num-

ber of disadvantages [13] typical of informal descriptions, which can be avoided if we apply mathematical specification methods, such as the trace assertion method.

Ambiguous interpretation of a specification. The specification structure expressed in terms of the TAM method eliminates ambiguous interpretation of requirements since it is formulated in a formal mathematical language.

Specification incompleteness. Unlike in the analysis of the text description of a problem, when using the trace assertion method, it is possible to prove that all admissible variants of system operation have been considered (see also [26]). The specification completeness is meant in the sense that functions δ and ν are defined in the entire domain, If we represent each row of a tabular specification of, say, function δ as a triple (c_i, m_i, r_i), then the proof of completeness of the specification of function δ reduces to proving the assertion that the condition

$$\bigcup_{i=1}^{n}\{(t, a)|t \in \mathcal{T}(m_i) \wedge a \in \mathcal{S}(c_i)\} = Q \times \Sigma$$

holds, where n is the number of rows in the table, $\mathcal{T}(m_i)$ is the number of canonical traces satisfying the predicate m_i, and $\mathcal{S}(c_i)$ is a set of input actions satisfying the predicate c_i.

The assertion about completeness of the specification of function ν is formulated similarly.

Specification inconsistency. When informal descriptions are used, inconsistent requirements can appear, especially if the description is lengthy or specifies complex logic of the system behavior. If the specification is expressed in terms of the trace assertion method, it can be proved that it does not contain inconsistent requirements (see also [26]). In other words, it can be proved that the value of function δ or ν for each element belonging to the domain is defined only once. This reduces to proving the assertion that, for any i and j ($i \neq j$), the following assertion holds:

$$\{(t, a)|t \in \mathcal{T}(m_i) \wedge a \in \mathcal{S}(c_i)\}$$
$$\cap\{(t', a')|t' \in \mathcal{T}(m_j) \wedge a' \in \mathcal{S}(c_j)\} = \varnothing,$$

where $\mathcal{T}(m_k)$ is the set of canonical traces satisfying the predicate m_k and $\mathcal{S}(c_k)$ is the set of input actions satisfying the predicate c_k.

In fact, this assertion states that two different rows of a specification cannot define function for one and the same subset of the domain. The assertion for function ν is formulated similarly.

Specification conciseness. Operation of even quite complicated systems can be specified in a clear, precise, and concise manner rather than by using a lengthy informal description.

Thus, a specification is a correct contract between the customer and contractor, which makes it possible to unambiguously and consistently formulate the task. Now, let us consider techniques used for designing an automaton program from a specification prepared by means of the trace assertion method.

When creating a system of interacting automata that presents an automaton program, it is required to specify values of sets of the tuple (Σ, Q, q_0, E, X, Z, δ). We strongly believe that the determination of the number of automata, their hierarchy, and the set of input and output actions is a design task to be solved by the programmer. Indeed, the purpose of the specification is to describe characteristics of the set of admissible implementations, i.e., instances of the automaton programs. Therefore, in designing an automaton program, the specification may help the programmer to determine some components of the automaton that is a part of the whole program, with these components being visible for an external observer. Let us consider construction of separate automaton components.

The finite alphabet of the transition arcs Σ is defined in terms of elements of the sets E, X, Y, and Z. Elements of set Y are used to model requests about current states of nested automata, which is related to the internal hierarchical structure of the automaton program and cannot be expressed by means of the specification. Therefore, we consider techniques that can be helpful for specifying sets E, X, and Z (excluding actions of the second kind, since they are also classified as internal interactions between the automata).

Transfer of elements directly from the specification. This technique suggests direct transfer of some elements of the specification to the definition of the automaton program. For example, actions H, M, and A described in the specification of the alarm clock system can be placed to the set of events E to which the automaton program responds as elements e_1, e_2, and e_3.

Defining semantics of elements by means of the specification. Actions on the control system returned by the automaton program can be rather complicated; however, semantics of these actions can uniquely be defined by means of a specification written in terms of the trace assertion method. Let us demonstrate this on the example of the alarm clock system. Analyzing the definition of the specification function ν, one can see all changes of the control system, which are registered in column *Result* (which can be viewed as that with respect to the arguments of function ν). Accordingly, we may conclude that the actions applied to the control system include increase of the number of hours or minutes by one or setting these values equal to zero (for the clock and alarm clock). Based on this conclusion, we may extract these actions from the specification and formally describe their semantics as, for example, shown in Table 3.

Elements placed to the considered sets and their semantics can be defined in different ways, and the efficiency of this procedure from the point of view of simplicity of the automaton program design depends on the

programmer. For the alarm clock system, semantics of output actions can be defined in a way different from that shown in Table 3. Indeed, one can see that the number of hours is increased by one modulo 24, and the number of minutes is increased by one modulo 60. For example, we can define action z_1' that increases the value of hours with the semantics $\langle h_a\rangle \longrightarrow \langle (h_a + 1) \bmod 24\rangle$. Such a definition of the action semantics can be justified from the specification standpoint. Indeed, whatever the history of actions τ on the system, the value of hours is either increased by one modulo 24 (the history of actions is equivalent to canonical trace A) or is not changed (the history of actions is equivalent to any other canonical trace). The proof is not difficult and follows directly from the specification.

The facts that the specification allows us to define semantics of output actions of an automaton program and that the specification completeness can be proved imply that it is possible to determine all necessary actions on the control system that may be required in the automaton program. The responsibility for the selection of a particular set of actions and their semantics rests completely with the programmer.

The selection of the set of output actions may affect the selection of the set of queries to the control system (set X) and semantics of these queries. For example, if we rely on the definition of the set of output actions shown in Table 3, then, when creating an automaton program, we will need queries to the control system that will allow us to determine which output action and when can be applied. As can be seen from the specification, each suggested action occurs only under certain conditions (column *Condition* in the definition of function v) of the control object. Hence, it is reasonable to specify the set of queries based on these conditions. The set of queries for the alarm clock system and their semantics are shown in Table 4.

As noted earlier, the set of conditions being selected depends on the selected set of output actions. Indeed, if we selected actions that increase the numbers of hours and minutes by one modulo 24 and 60, respectively, for the output actions, the set of queries to the control object would be empty since the semantics of these actions already assumes necessary checks.

Thus, the specification prepared by the trace assertion method allows us to formally define set Σ, in particular, elements of the set of input actions E; the set of queries to the control object X; and the set of output actions Z.

The next important step in the design of an automaton program is determination of states of the control automaton system. Let us show how the specification can help here. The specification is a more abstract description of the problem than its implementation in the form of an automaton program. Therefore, the set of states of an automaton program constructed by a specification cannot uniquely be defined. For example, if we have a system processing user's requests a and b, then,

Table 3. Semantics of the automaton program outputs (set Z of actions of the first kind)

Output	Semantics
z_1	$\langle h_w\rangle \longrightarrow \langle h_w + 1\rangle$
z_2	$\langle h_w\rangle \longrightarrow \langle 0\rangle$
z_3	$\langle m_w\rangle \longrightarrow \langle m_w + 1\rangle$
z_4	$\langle m_w\rangle \longrightarrow \langle 0\rangle$
z_5	$\langle h_a\rangle \longrightarrow \langle h_a + 1\rangle$
z_6	$\langle h_a\rangle \longrightarrow \langle 0\rangle$
z_7	$\langle m_a\rangle \longrightarrow \langle m_a + 1\rangle$
z_8	$\langle m_a\rangle \longrightarrow \langle 0\rangle$

Table 4. Semantics of queries to the control object for the automaton program (set X)

Query	Semantics
x_1	$h_w = 23$
x_2	$m_w = 23$
x_3	$h_a = 23$
x_4	$m_a = 23$

from the specification standpoint, in order to present an externally visible behavior of the system without focusing on its internal structure, it is sufficient to describe it as a transition system shown in Fig. 2a. At the same time, a clearer implementation is depicted in Fig. 2b. Here, the states where queries a and b are processed are explicitly indicated. The desire to develop a more understandable program can lead us to its specification as the transition system shown in Fig. 2c, where the state in which the system operation begins is explicitly indicated.

In the design of an automaton program by an informal specification, the identification of system states is a creative process, the correctness of implementation of which is ensured exclusively by the programmer. The originality of the trace assertion method consists in the fact that we can determine externally visible modes by applying sequences of actions to the module and observing outputs, i.e., traces, with a human factor being excluded.

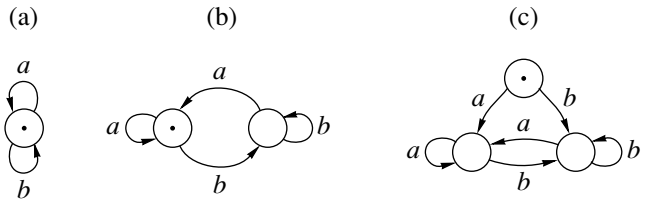

Fig. 2. Transition systems for the module processing queries a and b.

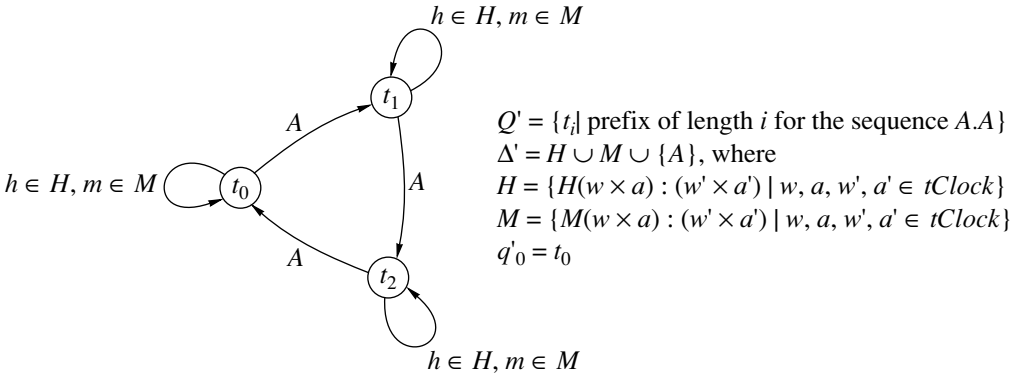

Fig. 3. The automaton presenting the clock specification.

To construct the set of states of an automaton program, for a given specification $(\Sigma, \mathbb{O}, \Delta, Q, q_0, \delta, v)$, we consider a deterministic finite automaton $\mathcal{A}_T = (Q', \Delta', \delta', q_0', F')$ [27] defined as follows:

• the set of states of automaton Q' coincides with the set of canonical traces of specification Q;

• the set of input symbols of automaton Δ' coincides with the set Δ of pairs "action–output" of the specification;

• the transition function for the automaton δ': $Q' \times \Delta' \longrightarrow Q'$ is defined as

$$\delta'(q, a) = q' \Leftrightarrow \delta(q, a) = q';$$

• the initial automaton state is $q_0' = q_0$;

• the set of accept states F' coincides with the set of all states Q'.

This automaton explicitly defines the set of externally visible modes that are states of the control system and transitions between these modes. Such an automaton for the specification of the alarm clock system is shown in Fig. 3. Based on this automaton, the programmer can make design decisions, such as definition of auxiliary automata and hierarchy of interactions between them.

For the alarm clock control system, it is possible to create the automaton program as a system of interacting automata $\mathcal{A} = \{A_0, A_{11}, A_{12}, A_{13}, A_{14}\}$. Here, A_0 is the principal automaton; automata A_{11} and A_{12} are used for modifying the values of hours and minutes, respectively, in the clock; and automata A_{13} and A_{14} control modification of values of hours and minutes in the alarm clock. The scheme of links between the principal automaton, auxiliary automata, and control object is shown in Fig. 4. The principal automaton is depicted in Fig. 6. As can be seen, automaton A_0 reminds very much the automaton in Fig. 3; i.e., it presents externally visible modes of the system operation. The other automata illustrate the clock and alarm clock setting process in a clearer way. The scheme of links for the auxiliary automaton A_{11} is depicted in Fig. 5, and the automaton itself is presented in Fig. 7. The schemes of links and transition systems for other auxiliary automata are constructed similarly to those for automaton A_{11}.

Thus, the proposed scheme of construction of automaton programs for logical control systems includes the following steps:

1. After discussions with the customer, based on informal descriptions, a specification of the developed system in terms of the trace assertion method is created. If necessary, the specification is checked, extended, and revised. The specification is a contract between the designer and customer.

2. The specification is used for designing and constructing the automaton program: elements of the sets of input and output actions and queries to the control system and their semantics are defined and states of the automaton program are determined. After this, on the basis of design decisions made by the programmer, a detailed automaton program is constructed.

3. On the basis of this specification, the constructed automaton program is tested and verified to show that it is correct from the specification standpoint.

The proposed scheme agrees with the ideas put forward by Mills in [28]. In the framework of his approach, the system is refined stepwise: first, the system is described as a "black box"; then, a more detailed description appears, system states are determined, and transitions between the states are defined; and, finally, the system is described explicitly. Mills notes that, in the case of application of heuristic methods to defining states of complex control systems (which is currently a common practice [2, 14]), it is difficult to understand whether the number of states found is sufficient or we missed something. The creation of a specification is considered to be a useful analytical step to understanding system functioning.

5. VERIFICATION OF AUTOMATON PROGRAMS

A "black box" specification formulates conditions that the constructed automaton program must satisfy;

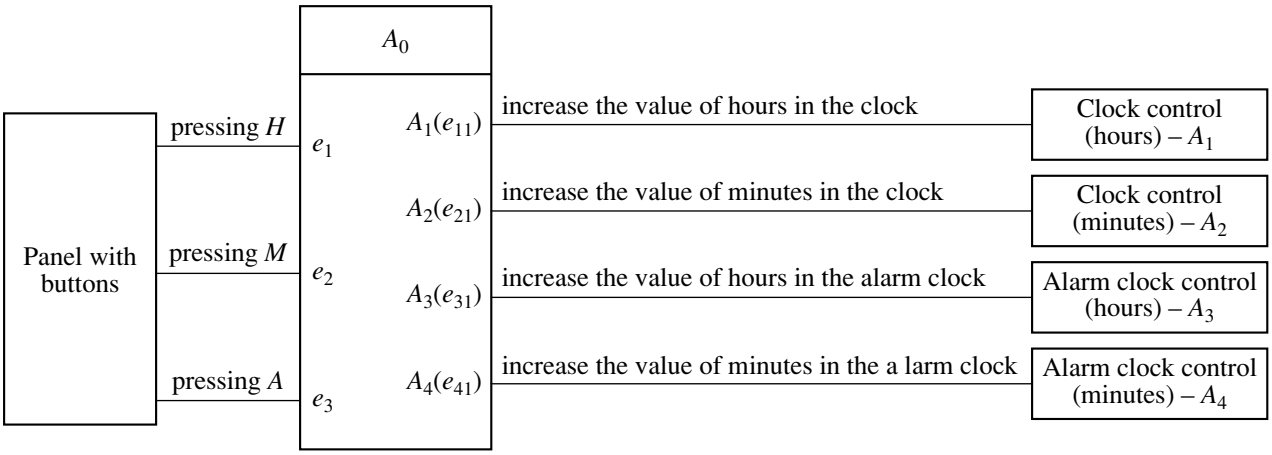

Fig. 4. The scheme of links for the clock control automaton.

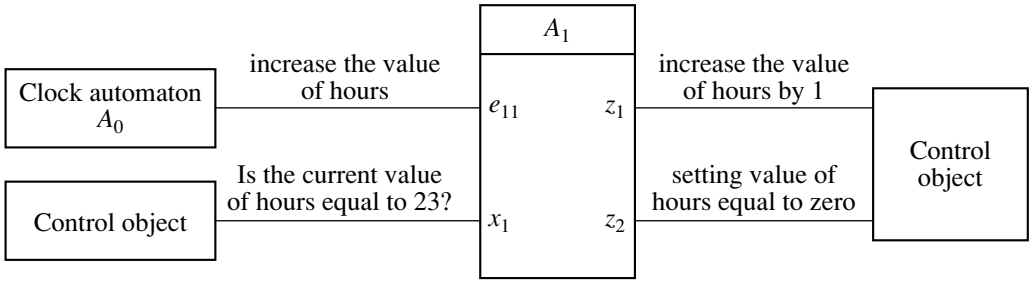

Fig. 5. The scheme of links for the automaton increasing the number of hours in the clock.

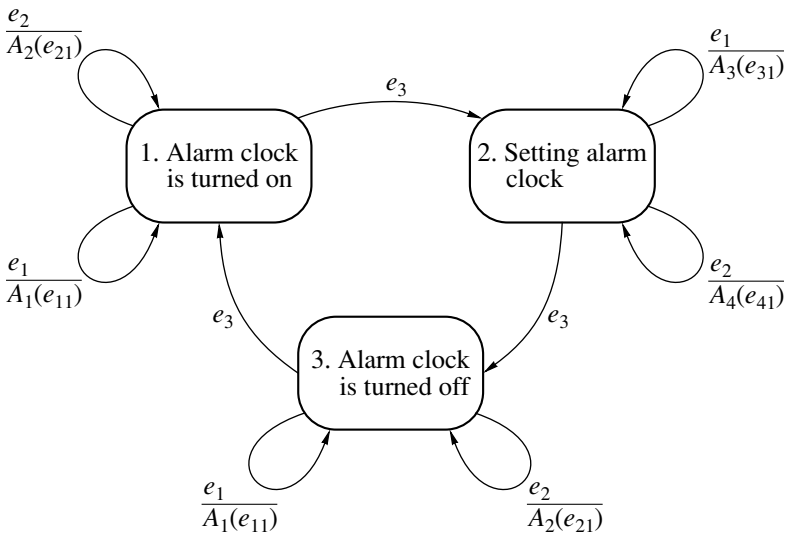

Fig. 6. Transition system for the clock control automaton.

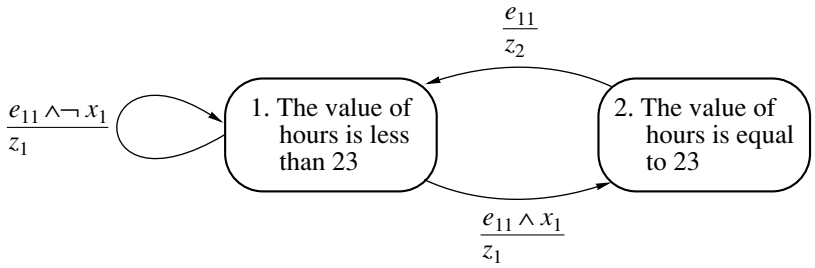

Fig. 7. The automaton controlling setting of hours in the clock.

therefore, the process of proving correctness of this program can be formulated as the following task. *Suppose that we have a specification written by means of the TAM method and an automaton program constructed by this specification. It is required to prove that the constructed automaton program satisfies the specification requirements.*

Currently known methods for analyzing correctness of complex systems include simulation, testing, proving, and model checking. Consider application of these methods to checking correctness of automaton programs from the point of view of a specification constructed by means of the trace assertion method.

The simulation can be applied as early as at the stage of the specification design. If the system is analyzed with the help of simulation, some scenario of the development of events is specified, and it is required to learn how the system behaves under this course of events. If this scenario can be specified as a trace τ, the latter can be executed by means of the specification. Indeed, as has already been shown, a specification in terms of the trace assertion method defines, in fact, a finite automaton, and the trace analysis suggests modeling the process of recognition of τ by this finite automaton. Another interesting way to use simulation is the trace rewriting method suggested in [29] and further developed in [30]; it transforms an arbitrary trace to its canonical equivalent. Thus, the applicability of the simulation is improved by inspecting the system being created by the customer. This allows us to evaluate correctness of externally visible behavior of the system as early as at the stage of constructing its specification. In design of an automaton program, such a "prototype" may help a lot to the programmer, allowing him to better understand the operation of the created logical control system.

Another popular method for checking program correctness is testing. Generally, testing suggests constructing a predicate on the set of all possible traces, which takes true value in the case of correct execution and false value otherwise. The testing methods can be classified into three categories: black box testing, when tests are generated without knowledge of the internal structure of the system under testing; clear box testing, when the internal structure of the tested system is completely known; and grey box testing, when some information about the internal structure is known. Since the trace assertion method imposes requirements on the system from the standpoint of the information hiding principle, it is reasonable to use the black box testing methods. The use of formal specifications given in the tabular notation for generation of test sequences is discussed in some works (see, for example, [31, 32]). The approach discussed in [31] is based on the verification of system operation under conditions close to the boundaries of intervals that constitute the domain of the function determining dependence between inputs and outputs. Such conditions are to be found for each externally visible mode of system operation and each state of the control system. If such a mode is given by a canonical trace τ, the boundaries of intervals for this mode must be specified based on those rows from the definition of function v of the specification in which the predicate in the column *Trace pattern* is true for τ. The boundaries of the intervals have to be determined by means of the predicates in the column *Condition* from these rows. For example, for the alarm clock system and the mode specified by trace $\tau = A$ (the alarm clock setting mode), for the test sequence specifying correct behavior, we can take the trace

$$\tau.H[(h_w, m_w) \times (22, m_a)] : [(h_w, m_w) \times (23, m_a)],$$

$$.H[(h_w, m_w) \times (23, m_a)] : [(h_w, m_w) \times (0, m_a)].$$

If the earlier defined semantics of input and output actions for automaton programs is used (Section 4 and Table 3), this trace can be transformed to the sequence $\langle e_3, e_1 : z_1, e_1 : z_2 \rangle$ with the initial state of the control system given by $(h_w, m_w) \times (22, m_a)$ and the alarm clock turned on.

Although testing is a widely used method for studying correctness of program systems, many researchers agree that the use of only one method is not sufficient to guarantee that the system is reliable. Moreover, there is an opinion that only with the help of mathematical proofs, i.e., the proving method [33], it is possible to formally prove correctness of system operation. However, even such a formal proof does not guarantee correctness of system functioning if the informal program specification was incorrect or some component of the program did not possess the property that was assumed

to be a premise of the proof. As for automaton programs, it is noted in [10] that the automaton structure of programs facilitates using the proving in spite of the fact that it is labor-consuming, possesses low level of automation, and is strongly tied to the semantics of the programming language. Rigorous mathematical definition of the semantics of elements of the automaton program performed with the use of a specification in terms of the trace assertion method significantly facilitates formal proof of the program correctness.

One more approach to the correctness analysis is the model checking method [34], the idea of which is as follows. The program system is specified as a finite transition system called Kripke structure. Further, in the framework of the Kripke structure with the use of a temporal logic language, properties of the program model are formulated and, then, checked. In formal terms, this can be defined as follows.

Given a finite transition system (Kripke structure), an initial state s_0 of this system, and a temporal logic formula φ, it is required to determine whether the state s_0 satisfies formula φ.

Methods of verification of automaton programs with the use of the model checking method are considered in detail in [10]. The disadvantage of the considered approach is that the formulas being checked are suggested based on the informal system specification. Hence, if the formula is composed with an error, it may happen that we check something different from what was originally planned.

Based on a specification performed with the help of the trace assertion method, the following categories of formulas can be constructed. (We do not write down particular temporal formulas, since this was considered in detail in [10].)

Conditions for externally visible modes. An automaton program should model transitions from one externally visible mode to another. On a set of states of an automaton program

$$\mathcal{A} = (A_0, A_{11}, \ldots, A_{1k_1}, \ldots, A_{n1}, \ldots, A_{nk_n}),$$

where n and k_i are positive integers ($1 \leqslant i \leqslant n$), we define a set of predicates $S_1, \ldots, S_m$; here, m is the number of canonical traces in the specification

$$S_j : Q_0 \times Q_{11} \times \ldots \times Q_{1k_1} \times \ldots$$

$$\times Q_{n1} \times \ldots \times Q_{nk_n} \longrightarrow \mathbb{B},$$

and each predicate S_i takes the true value if the system of automata $\mathcal{A}$ is in a state corresponding to the externally visible mode given by the i-th canonical trace of the specification. Then, based on the definition of function δ in the specification, one can define temporal logic formulas that check transition from one externally visible mode to another and the fact that the system does not occur in two states simultaneously.

Since the set of predicates for the alarm clock system can be defined as $S_i(q) = [q = q_i]$, where q is a state from the set of states of the automaton A_0. Accordingly, transition between two modes can be specified as the following condition: *if the automaton occurs in a state that corresponds to the externally visible mode S_1, then, after pressing button A (and, thus, receiving event e_3), the automaton will pass to a state corresponding to the externally visible mode S_2.*

The fact that an automaton does not present in two externally visible modes simultaneously is stated as follows: *an automaton will never present in the states where S_i and S_j, $i \neq j$, are simultaneously fulfilled.*

Reactivity condition. A constructed specification assumes that the module must process any action under any conditions; i.e., the control system constructed does not occur in a state where it does not respond to any external actions. As applied to the above example of the alarm clock system, this means that it is always possible to press any button, and the system will always respond to it.

Condition of producing certain outputs. This type of conditions helps us to define formulas that specify necessity of producing certain outputs in response to input actions if the system is in some externally visible mode.

An example of such a condition is as follows: *let the alarm clock system occur in a state corresponding to an externally visible mode S_1, and let an action e_1 be applied to its input; then, the outputs are either z_1 or z_2.*

It should be noted that the temporal formulas used for the verification of an automaton program are formed by means of the specification rather than are suggested for verification spontaneously with the only interest to check some properties.

Thus, based on the specification, we can check correctness of automaton programs using methods of all kinds.

6. CONCLUSIONS

We have shown that the creation of a specification is an important stage of the process of software development for logical control systems. Indeed, the specification design makes it possible to deeper analyze the posed problem without going into detail of the implementation and relying only on operation scenarios given as traces and to make sure that all important requirements are determined and do not contradict one another.

The specification obtained is not only a basis for defining semantics of such important components of an automaton program as input and output actions or queries to the control object. It also provides formal means for determining the set of states of the automaton program. Note that the programmer is not restricted in

making design decisions on specifying the hierarchy of the interacting automata.

We have also discussed verification of the automaton program obtained by means of various methods such as simulation, testing, proving, and the model checking method. In this case, the specification plays the role of a "prototype" used for determining system properties necessary for the verification.

REFERENCES

1. Bartussek, W. and Parnas, D.L., Using Assertions about Traces to Write Abstract Specifications for Software Modules, *Lecture Notes in Computer Science* (Proc. of the 2nd Conf. on European Cooperation in Informatics), Springer, 1978, no. 65, pp. 211–236.
2. Shalyto, A.A., *SWITCH-tekhnologiya. Algoritmizatsiya i programmirovanie zadach logicheskogo upravleniya* (SWITCH-Technology: Algorithmization and Programming of Logic Control Problems), St. Petersburg: Nauka, 1998.
3. Shalyto, A.A., Software Automaton Design: Algorithmization and Programming of Problems of Logical Control, *Izv. Ross. Akad. Nauk, Teor. Sist. Upr.*, 2000, no. 6, pp. 63–81. [*J. Comput. Systems Sci. Int.* (Engl. Transl.), 2000, vol. 39, no. 6, pp. 899–916].
4. Shalyto, A.A. and Tukkel, N.I., SWITCH-Technology: An Automated Approach to Developing Software for Reactive Systems, *Programmirovanie,* 2001, no. 5, pp. 45–62. [*Programming Comput. Software* (Engl. Transl.), 2001, vol. 27, no. 5, pp. 260–276].
5. Parnas, D.L. and Vilkomir, S.A., Precise Documentation of Critical Software, *The 10th IEEE High Assurance Systems Engineering Symposium*, IEEE, 2007, pp. 237–244.
6. Faulk, S.R., Software Requirements: A Tutorial, *Tech. Report NRL-7775*, Naval Research Lab., Washington.
7. Nuseibeh, B., Ariane 5: Who Dunnit?, *IEEE Software*, 1997, vol. 14, no. 3, pp. 15–16.
8. Leveson, N., Role of Software in Spacecraft Accidents, *J. Spacecraft Rockets*, Am. Inst. of Aeronautics and Astronautics, vol. 41, no. 4, pp. 564–575.
9. Davis, A., A Taxonomy for the Early Stages of the Software Development Life Cycle, *J. Systems Software*, 1988, vol. 8, no. 4, pp. 297–311.
10. Kuzmin, E.V. and Sokolov, V.A., Modeling, Specification, and Verification of Automaton Programs, *Programmirovanie,* 2008, no. 1, pp. 38–60. [*Programming Comput. Software* (Engl. Transl.), 2008, vol. 34, no. 1, pp. 27–43].
11. Lutz, R.R., Targeting Safety-related Errors during Software Analysis, *Proc. of the 1st ACM SIGSOFT Symp. on the Foundations of Software Engineering*, 1993.
12. Boehm, B.W., *Software Engineering Economics*, Prentice Hall, N.J., 1981.
13. Baber, R.L., Parnas, D.L., Vilkomir, S.A., Harrison, P., and O'Connor, T., Disciplined Methods of Software Specification: A Case Study, *Int. Conf. on Information Technology: Coding and Computing'2005*, 2005, vol. 2, pp. 428–437.
14. Polikarpova, N.I. and Shalyto, A.A., *Avtomatnoe programmirovanie* (Automaton Programming), St. Petersburg: SPbGU ITMO, 2007.
15. Parnas, D.L., On the Criteria to be Used in Decomposing Systems into Modules, *Commun. ACM*, 1972, vol. 15, no. 12, pp. 1053–1058.
16. Hoffman, D.M., The Trace Specification of Communication Protocols, *IEEE Trans. Comput.,* 1985, vol. C-34, no. 12, pp. 1102–1113.
17. Iglewski, M., Kubica, M., and Madey, J., Trace Specifications of Non-Deterministic Multi-Object Modules, *Lecture Notes in Computer Science* (Proc. of ASIAN'95), Springer, 1995, no. 1023, pp. 381–395.
18. Janicki, R. and Sekerinski, E., Foundations of the Trace Assertion Method of Module Interface Specification, *IEEE Trans. Software Eng.*, 2001, vol. 27, no. 7, pp. 577–598.
19. Iglewski, M., Kubica, M., and Madey, J., Editor for the Trace Assertion Method, *Proc. of the 10th Int. Conf. of CAD/CAM, Robotics and Factories of the Future: CARs&FOF'94*, Ottawa, Canada, 1994, pp. 876–881.
20. Peters, D.K., Lawford, M., and Widemann, B.T., An IDE for Software Development Using Tabular Expressions, *Proc. of CASCON 2007*, Ontario, Canada, 2007, pp. 248–251.
21. Van Schouwen, A.J., The A-7 Requirements Model: Re-examination of Real-time Systems and an Application to Monitoring Systems, *Tech. Report 90-276, Queen's C& IS, TRIO*, Kingston, Ontario, Canada, 1990.
22. Bojanowski, J., Iglewski, M., Madey, J., and Obaid, A., Functional Approach to Protocol Specification, in *Protocol Specification, Testing and Verification XIV*, Chapman & Hall, 1995, pp. 195–402.
23. Wassyng, A. and Lawford, M., Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project, *Lecture Notes in Computer Science* (Proc. of FME 2003: Int. Symp. of Formal Methods Europe), Springer, no. 2805, pp. 133–153.
24. Parnas, D.L., Tabular Representation of Relations, *CRL Report 260*, Telecom, Research Institute, McMaster University, Ontario, Canada, 1992.
25. Janicki, R. and Khédri, R., On a Formal Semantics of Tabular Expressions, *Sci. Comput. Programming*, 2001, vol. 39, nos. 2–3, pp. 189–213.
26. Parnas, D.L., Some Theorems We Should Prove, *Lecture Notes in Computer Science* (Int. Workshop on Higher Order Theorem Proving and Its Applications), Springer, 1993, no. 780, pp. 154–162.
27. Hopcroft, J.E., Motwani R., and Ullman J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 2001.
28. Mills, H.D., Stepwise Refinement and Verification in Box-Structured Systems, *IEEE Comput.,* 1988, vol. 21, no. 6, pp. 23–36.

29. Wang, Y. and Parnas, D.L., Simulating the Behavior of Software Modules by Trace Rewriting, *IEEE Trans. Software Engineering,* 1994, vol. 20, no. 10, pp. 750–759.

30. Brzozowski, J. and Jürgensen, H., Theory of Deterministic Trace-Assertion Specifications, *Tech. Report CS-2004-30*, School of Computer Science, Univ. of Waterloo, Ontario, Canada, 2004.

31. Clermont, M. and Parnas, D.L., Using Information about Functions in Selecting Test Cases, *ACM SIGSOFT Software Engineering Notes*, 2005, vol. 30, no. 4, pp. 1–7.

32. Liu, S., Generating Test Cases from Software Documentation, *MS Thesis,* School of Graduate Studies, McMaster University, 2001.

33. Gries, D., *The Science of Programming*, New York: Springer, 1981. Translated under the title *Nauka programmirovaniya*, Moscow: Mir, 1984.

34. Clarke, E.M., Grumberg, O., and Peled, D., *Model Checking*, MIT Press, 1999. Translated under the title *Verifikatsiya modelei program: Model Checking*, Moscow: MTsNMO, 2002.