ISSN 0361-7688, Programming and Computer Software, 2009, Vol. 35, No. 6, pp. 301–313. © Pleiades Publishing, Ltd., 2009.
Original Russian Text © I.B. Bourdonov, A. S. Kossatchev, 2009, published in Programmirovanie, 2009, Vol. 35, No. 6.

Complete Open-State Testing of Limitedly Nondeterministic Systems

I. B. Bourdonov and A. S. Kossatchev

Institute for System Programming, Russian Academy of Sciences,
ul. Solzhenitsyna 25, Moscow, 109004 Russia
e-mail: igor@ispras.ru, kos@ispras.ru

Received March 19, 2009

Abstract—An approach to the problem of complete testing is proposed. Testing is interpreted as the check of an implementation's conformance to the given requirements described by a specification. The completeness means that a test suite finds all the possible implementation errors. In practice, testing must end in a finite amount of time. In the general case, the requirements of completeness and finiteness contradict each other. However, finite complete test suites can be constructed for certain classes of implementations and specifications provided that there are specific test capabilities. Test algorithms are proposed for finite specifications and finite implementations with limited nondeterminism for the case of open-state testing. The complexity of those algorithms is estimated.

DOI: 10.1134/S0361768809060012

INTRODUCTION

Testing is the experimental check of an implementation's conformance to the requirements given in a specification. The testing is complete if it unambiguously decides whether the implementation contains errors or not. An error is a violation of requirements, that is, nonconformance. For practical purposes, testing must terminate in a finite amount of time. In the general case, testing turns out to be either incomplete or infinite. The solution of the problem can be sought by restricting the class of the implementations under examination or using advanced testing capabilities.

The main causes of the infiniteness of complete testing are the size of the implementation and (or) specification and the nondeterminism of the implementation.

If the specification is infinite, all its requirements cannot be checked in a finite amount of time. Therefore, a finite testing of an infinite specification is certainly incomplete. If a requirement must be checked in the implementation for an infinite number of situations, this also cannot be done in a finite amount of time. For the finite testing to be complete, the implementation must be also finite.

However, it is not sufficient for the implementation to be complete. Indeed, if the size of the implementation is unknown, we cannot be sure at any given time that all the possible situations have been checked. We need to estimate the size of the implementation in advance (assuming that it is not only finite but also bounded) or in the course of testing. In the latter case, advanced testing capabilities are needed to observe the part of the implementation that has already been tested and draw conclusions concerning the presence or absence of other parts.

If the implementation's behavior is arbitrarily nondeterministic, we cannot know at any particular time whether or not the implementation has demonstrated all the variants of its behavior. To make finite testing complete, one has to impose certain restrictions on the nondeterminism of the implementation.

In this paper, we consider testing of a *finite implementation* against a *finite specification* under two additional assumptions: (1) *open-state testing*, which means that we can observe the implementation's states at which it comes in the course of testing; (2) *the implementation is limitedly nondeterministic*, which means that, if a certain test input repeats at the same state of the implementation a sufficient (known in advance) number of times, then the implementation demonstrates all its behavior patterns. For this case, we propose algorithms for finite complete testing and give estimates of the number of test inputs and the amount of computations.

In the first section of the paper, we outline the conformance theory that was developed in [1–3]. In Section 2, we discuss problems of practical testing and different lines of attack on them. In Section 3, we propose traversal algorithms. In Section 4, testing algorithms are proposed and the finiteness and completeness of testing are proved; complexity estimates are also given.

Fig. 1. Testing machine.

1. CONFORMANCE THEORY

1.1. Semantics of the Interaction and Safe Testing

The verification of conformance is interpreted as the check of the system's conformance to the given requirements. In the model world, the system is mapped to its implementation model (implementation), the requirements are mapped to the specification model (specification), and their conformance is mapped to a binary conformance relation. If the requirements are formulated in terms of the system's interaction with its environment, testing can be performed as the verification of conformance in the course of testing experiments when a test replaces the environment. The conformance relation and its testing are based on a certain interaction model.

We consider the interaction semantics that are based only on the external observable behavior of the system but do not take into account its internal organization (which is mapped to the concept of state at the level of the model). In this case, the black box (or functional) testing can be performed. We can only observe the behavior of the system that is, first, induced by a test input and, second, can be observed in an external interaction. Such an interaction can be simulated using the so-called testing machine [1–6]. This machine is a black box containing the implementation (see Fig. 1). An operator controls the testing machine by pressing buttons on the machine's keyboard thus instructing (or allowing, or enabling) the implementation to perform certain actions that can be observed. Observations on the machine's display are classified into two types. Observation of an *action* that is enabled by the operator and is performed by the implementation, and observation of a *refusal*, which means that no actions allowed by the buttons pressed by the operator are observed. We denote actions by lowercase letters and refusals (considered as sets of actions) by uppercase letters.

We stress that the operator may allow the machine to perform a set of actions (but not necessarily only a single action). Each button is assigned an individual set of actions. After an observation (of an action or a refusal) has been made, the button is released, and all the external actions are disabled. Then, the operator may press another (or the same) button.

The testing capabilities are determined by the "button" sets available in the machine and by the set of buttons for which refusal can be observed. Thus, the semantics of the interaction is determined by the alphabet of external actions L, the set of actions that the test can enable (the set of buttons of the testing machine), and by two sets of buttons for which the corresponding refusals are observable (the family $\mathfrak{R} \subseteq \mathcal{P}(L)$) and not observable (the family $\mathfrak{Q} \subseteq \mathcal{P}(L)$). It is assumed that $\mathfrak{R} \cap \mathfrak{Q} = \emptyset$ and $\cup\mathfrak{R} \cup\cup \mathfrak{Q} = L$. Such kind of semantics is called $\mathfrak{R}/\mathfrak{Q}$ semantics.

It was shown in [1, 3] that its is sufficient to consider only the semantics in which all the refusal are observable ($\mathfrak{Q} = \emptyset$). Any $\mathfrak{R}/\mathfrak{Q}$ semantics is equivalent to an $\mathfrak{R} \cup \mathfrak{Q}/\emptyset$; namely, for any specification in an $\mathfrak{R}/\mathfrak{Q}$ semantics, there exists (and can be constructed under certain practically reasonable constraints) a specification in the $\mathfrak{R} \cup \mathfrak{Q}/\emptyset$ semantics that defines the same class of conformal implementations and a not smaller class of testable implementations. For that reason, we consider only the $\mathfrak{R}$ semantics; that is, we assume that $\mathfrak{Q} = \emptyset$.

For an action to be executable, it is necessary that it is defined in the implementation and is enabled by the operator. If this condition is also sufficient, the system does not have priorities. We consider only the systems without priorities.

In addition to the external actions, the implementation can execute internal (unobservable) actions, which are denoted by τ. These actions are always enabled. It is assumed that any finite sequence of arbitrary actions terminates in a finite amount of time and an infinite sequence of actions terminates in an infinite amount of time. The infinite sequence of τ-actions (an infinite loop) is called *divergence;* it is denoted by Δ. We also define a special action called *destruction* that is not controlled by the buttons; this action is denoted by γ. It models any prohibited behavior of the implementation. The divergence per se is not harmful, but, when the operator presses any button after its occurrence to escape the divergence, he does not know whether an observation should be expected or the implementation will endlessly continue its internal activity. Therefore, the operator cannot continue testing, nor can he finish it. The testing in which there are no attempts to escape the divergence and there is no destruction is said to be safe.

1.2. LTS-Model and Its Traces

We will use a *labeled transition system* (LTS) as a model of the implementation and the specification. LTS is a directed graph with a distinguished initial vertex in which the arcs are labeled by certain symbols. Formally, LTS is the collection $\mathbf{S} = LTS(V_S, L, E_S, s_0)$, where V_S is the nonempty set of states (graph vertices), L is the alphabet of external actions, $E_S \subseteq V_S \times (L \cup \{\tau, \gamma\}) \times V_S$ is the set of transitions (labeled arcs), and $s_0 \in V_S$ is the initial state (the initial vertex of the graph). The transition from the state s to the state s' by the action z is denoted by $s \xrightarrow{z} s'$. Define $s \xrightarrow{z} \stackrel{\Delta}{=} \exists s'$

$s \xrightarrow{z} s'$ and $s \xrightarrow{z}\!\!\!\!\!\!/\ \triangleq \nexists s'\, s \xrightarrow{z} s'$. A route in an LTS is a sequence of adjacent transitions such that the beginning of each transition except for the first one coincides with the end of the preceding transition.

The execution of an LTS in a testing machine is reduced to performing the transition defined in the current state and enabled by the pressed button (τ- and γ-transitions are always enabled).

A state is called *stable* if there are no τ- and γ-transitions outdoing from it. A state is said to be *divergent* if it begins an infinite chain of τ-transitions (in particular, a τ-cycle). The refusal $P \in \Re$ is induced by a stable state that contains no transitions initiated by the actions from P. The transition $s \xrightarrow{z} s'$ is called *destructive* if the beginning of a γ-transition can be reached from s' by a chain (possibly empty) of τ-transitions.

In each stable state of the LTS **S**, we add virtual loops labeled by the induced refusals and add Δ-transitions in all the divergent states. In the resulting LTS, consider the routes that begin in the initial state and do not continue beyond a Δ- or γ-transition. An $\Re$-trace in the LTS **S** is the sequence of labels of the transitions in such a route in which the symbols τ are omitted. The set of $\Re$-traces of the LTS **S** is denoted by $\Re(\mathbf{S})$.

1.3. Safety Hypothesis and Safe Conformance

Let us define the safety relation *a button is safe in the LTS after the $\Re$-trace* as follows. Pressing the button P after the $\Re$-trace σ does not entail an attempt of exiting from the divergence (there is no divergence after th trace) and does not entail destruction (after an action enabled by the button). In the case of safe testing, only safe buttons are pressed. Formally, the button P is *safe* in the state s (denoted by P *safe* s if the state s is not divergent, there are no γ-transitions in the states s' that are reachable from s by τ-transitions, and all the transitions $s' \xrightarrow{z} s''$ where $z \in P$ are nondestructive.

The safety relation in the set of states S is defined as follows: P *safe* $S \triangleq \forall s \in S\ P$ *safe* s.

The safety relation after the trace s is defined as follows: P *safe* **S** *after* σ $\triangleq$ P *safe* (**S** *after* σ). Here, **S** *after* σ is the set of states of the LTS **S** after the trace σ; that is, this is the set of states that are reachable from the initial state by the trace σ.

The safety of buttons implies the safety of the actions and refusals after the trace. The refusal R is safe after the trace σ (R *safe* (**S** *after* σ)) if the button R is safe after σ. The action z is safe after the trace σ: z *safe* (**S** *after* σ) if it is enabled ($z \in P$) by a button such that P *safe* (**S** *after* σ). An $\Re$-trace is safe if the following conditions hold. (1) The LTS is not destructed at the very beginning; that is, it does not contain the trace $\langle\gamma\rangle$. (2) Every symbol of the trace is safe after the trace prefix that immediately precedes this symbol. The set of safe traces of the LTS **S** is denoted by *Safe*(**S**).

The requirement of testing safety defines the class of *safe* implementations that can be safely tested to check their conformance to the given specification. This class is defined by the following *safety hypothesis*. The implementation **I** is *safe* for the specification **S** if the following holds. (1) The implementation is not destructed at the very beginning if this requirement is not included in the specification. (2) After any safe trace that is common for the specification and the implementation, any button that is safe in the specification is safe after this trace in the implementation:

$$\mathbf{I}\ \textit{safe for}\ \mathbf{S} \triangleq (\langle\gamma\rangle \notin F(\mathbf{S}) \Rightarrow \langle\gamma\rangle \notin F(\mathbf{I})) \,\&\, \forall\sigma \in \textit{Safe}(\mathbf{S}) \cap T(\mathbf{I})\ \forall P \in \Re\ (P\ \textit{safe}\ \mathbf{S}\ \textit{after}\ \sigma \Rightarrow P\ \textit{safe}\ \mathbf{I}\ \textit{after}\ \sigma).$$

Note that the safety hypothesis cannot be checked in the course of testing; this hypothesis is the testing precondition. Then, we can define the conformance relation: The implementation **I** *conforms* to the specification **S** if it is safe and the following *condition under test* is fulfilled: any observation that is possible in the implementation as a response to pressing a safe (in the specification) button is allowed by the specification:

$$\mathbf{I}\ \textit{saco}\ \mathbf{S} \triangleq \mathbf{I}\ \textit{safe for}\ \mathbf{S} \,\&\, \forall\sigma \in \textit{Safe}(\mathbf{S}) \cap T(\mathbf{I})\ \forall P\ \textit{safe}\ \mathbf{S}\ \textit{after}\ \sigma\ \textit{obs}(\mathbf{I}\ \textit{after}\ \sigma, P) \subseteq \textit{obs}(\mathbf{S}\ \textit{after}\ \sigma, P),$$

Here, $\textit{obs}(M, P) \triangleq \{u \mid \exists m \in M\, u \in P \,\&\, m \xrightarrow{u} \vee u = P \,\&\, \forall z \in P\, m \xrightarrow{z}\!\!\!\!\!\!/\ \}$ is the set of observations that can be obtained by pressing the button P in the states belonging to the set M.

1.4. Test Suite Generation

In terms of the testing machine, a test is an instruction for the machine's operator consisting of terminal and nonterminal items. In each item, a button is specified that must be pressed by the operator and, for each observation that is possible after pressing this button, one must indicate the instruction item that must be executed next or the verdict *pass* or *fail* (if the testing must be stopped). In [1–3], such an instruction corresponds to the formal definition of a controllable LTS test that uniquely determines the operator's behavior (without excessive nondeterminism).

An implementation *passes* a test if its testing always results in the verdict *pass*. An implementation passes a suite of tests if it passes every test in this suite. A suite is called *significant* if it is passed by every conformal implementation; a test suite is called *exhaustive* if no nonconformal implementations passes it. A test suite is *complete* if it is both significant and exhaustive. The task is to generate a complete test suite given a specification.

A complete test suite always exists; in particular, the set of all *primitive* tests is complete. A primitive test is constructed on the basis of a distinguished safe $\Re$-trace of the specification. To this end, before every $\Re$-refusal R, the button R is included, and before every action z, an arbitrary safe (after the trace prefix) button P is included that enables the action z. The safety of the trace guarantees that the button R is safe and that an appropriate safe button P exists. The button P is generally, not unique; therefore, given the same safe trace, many different primitive tests can be generated. If the observation after pressing a button continues the trace, the testing is continued. The last observation in the trace and any "distracting" observation terminate the testing. The verdict *pass* is assigned if the specification contains the corresponding trace; otherwise, the verdict *fail* is returned. Such verdicts correspond to *strict* tests; these are the tests that, first, are significant (do not detect false bugs) and, second, do not miss detected bugs. Any strict test can be reduced to a set of primitive tests that detect the same bugs.

2. PROBLEMS OF PRACTICAL TESTING

2.1. Nondeterminism and Global Testing

In the absence of priorities, the implementation may execute any defined external action that is enabled by the operator; also, it may execute defined internal actions, which are always enabled. If there are several such actions, one of them is chosen in a nondeterministic way. We assume that nondeterminism is a phenomenon pertaining to the abstraction level that is determined by the testing observation and control capabilities; that is, the nondeterminism depends on the interaction semantics. In other words, the behavior of the implementation depends on certain "weather conditions" (which we do not take into account) that uniquely determine the choice of an action.

For the testing to be complete, one has to assume that arbitrary weather conditions can be reproduced in a testing experiment for every test. If this is possible, the testing is said to be *global* [6]. In this paper, we disregard the number of possible variants of the weather conditions; it is only important that the system's behavior can be checked at any weather conditions and any behavior of the operator. In practice, only a finite number of each test can be run; therefore, without additional assumptions, we cannot be sure that all the necessary test runs were performed for every test at all possible weather conditions. Various approaches to overcoming this difficulty are possible.

One of these approaches is to use special test tools for "weather control." Here, we go beyond the scope of the model in which we neglected the influence of external factors (the weather). Testing becomes dependent not only on the specification but also on the implementation details; in other words, it depends on the operation environment in which the implementation operates. For every variant of the operation environment, a special test suite is designed. In some cases, practical advantages can be gained using this approach.

Another approach uses special hypotheses about the implementation. They assume that, if the implementation's behavior s correct under certain weather conditions, it will also behave correctly under all weather conditions [7].

The third approach is based on the knowledge of the probability distribution of different weather conditions. In this case, testing is complete with a certain probability [8].

The fourth approach assumes that, in any situation (after any trace), the number of nonequivalent weather conditions is bounded by a known number t; then, t test runs guarantee that the implementation's behavior is reproduced under any weather conditions [9, 10]. This can be called *limited nondeterminism* (t-nondeterminism). For $t = 1$, all the implementations under test are deterministic. This is used in practice (see [11–13]] when it is known that all the implementations of interest are deterministic.

2.2. Infiniteness of Complete Test Suites

Since the tests are finite, the complete test suite usually contains an infinite number of tests; in particular, the set of primitive tests is also infinite. (A complete test suite is finite only for models with a finite behavior, that is, with a finite number of traces; this corresponds to a finite LTS specification without cycles.) However, only finite test suites can be used in practice. This problem can be solved differently, but all the solutions are reduced to special testing capabilities and (or) implementation hypotheses. In essence, such hypotheses assume that, if the implementation behaves correctly on the tests of a certain finite test suite, it also behaves correctly on all the tests of the infinite suite. The knowledge of the structure of the implementation of a certain class can justify such a hypothesis. Such finite test suites are significant but not complete on the class of all implementations; however, they are complete on the subclass of the implementations satisfying the implementation hypothesis.

The conformance theory uses the generic coverage criterion that requires that all the specification requirements are checked in all possible implementation situations. Often, special coverage criteria are used that check not all the situations but only certain classes of situations (based on an error model) [14–16]. A fairly general approach uses a coarser specification (a so-called test model) instead of the original specification. The coarse specification is obtained by factorizing the original specification with respect to the transition equivalence relation, which usually reduces to the equivalence of states and (or) actions [17]. Sometimes, the factorization also eliminates

nondeterminism thus solving this problem as well. Such an approach is justified if it is true that the errors that can appear in the implementation are found by testing against the factored specification (that is, the errors are found when testing against a finite set satisfying the coverage criterion).

An example is testing a finite state machine (FSM) against a specification given as an FMS [7, 11, 12, 18–22]. The conformance under check is the equivalence of finite state machines. It is usually assumed that the specification and implementation are deterministic and that the implementation has not more states than the specification (up to the equivalence of states) or it is assumed that the number of states in the implementation is greater than the number of states in the specification by given number.

2.3. Open State Testing

If an operation is available that allows one to reliably examine (query) the current state of the implementation (*status message*), the testing is said to be open state. It is known that the complete testing of the equivalence of finite state machines is reduced to a traversal of the transition graph of the FMS with querying every state being passed [7, 11, 21, 23–26].

In the LTS model, state querying can be interpreted in terms of the testing machine as pressing a special button after every observation. This is equivalent to the situation when the poststate appears on the machine's display as a part of the observation. It is important that, if the implementation is in an unstable state i after performing an action, the poststate on the display is not necessarily i but can be any state that is reachable from i by an empty trace ϵ, that is, by a chain of τ-transitions. Such a state $i \in (\mathbf{I}\ \textit{after}\ \epsilon)$ is said to be initially reachable. Therefore, we do not require that the internal operation of the implementation be stopped between the observation and the state query. Similarly, we do not require that the implementation be stopped between the query of the state and the next test input or the completion of the testing. Several queries in succession make it possible to observe the progression of the implementation through τ-transitions; however, we believe that this option does not increase the capacity of the *saco* conformance testing.

The start (or restart) of the system is interpreted as one of the test inputs included in $\Re$. The difference from the other test inputs is that an initially reachable state is always obtained after the start (restart). In the general case, the restart is not always defined at every state of the implementation. We assume that, if the restart is defined, then the action *restart* is observed; otherwise, the restart refusal is observed. The implementation LTS is completed by the transitions by restart, which "reset" the trace; that is, the trace of a route is the trace of its postfix after the last restart.

2.4. Conditions for the Testing to Be Finite

Let us formulate the restrictions on the implementation and specification that make it possible to perform a finite complete testing. We consider the open state *saco* conformance testing and the generic coverage criterion. For each restriction, we specify its weakened variants, which we do not consider in detail for the sake of simplicity.

t-Nondeterminism of the implementation. We assume that, in any implementation state i, all possible pairs (observation, poststate) are obtained after pressing any button t times; that is, all the routes that begin at i and have a trace ⟨observation⟩ are passed. This restriction can be weakened by considering only the reachable implementation states. For every given specification, only the implementation states that are reachable by safe traces of the specification are important. The t-nondeterminism implies that, for any button (with any button set) not more than t transitions by the actions enabled by this button are defined.

To solve the problem of infiniteness of the complete test suite, we impose restrictions on the size of the $\Re$-semantics, specification, and implementation.

The number of buttons is finite. Without this restriction, we would have to apply an infinite number of test inputs after each passed trace. For every given specification, this restriction is weakened if only the buttons that are safe in the specification after the trace are taken into account.

An algorithm for enabling the button for all the actions is available. For every action z and every button P, such an algorithm determines in a finite amount of time whether $z \in P$ or $z \notin P$. This is required to check the safety of the button after the passed trace: the button is safe if there are no destructive transitions from the states after the trace by the actions initiated by the button. Such an algorithm can be easily constructed if all the button sets are finite (then, the action alphabet is also finite).

The number of specification states is finite. If the number of states is infinite, the complete testing of the specification is infinite as well. This restriction can be weakened if only the states belonging to the safe traces are taken into account. For every given implementation, not all such states of the specification are needed; however, we assume that the specification is used to test arbitrary implementations (with regard for the restrictions on the nondeterminism and the size of the implementation).

The number of transitions in the specification is finite. This restriction is required to ensure the safety of a button after the passes trace. All the transitions that outgo the states after the trace must be examined. If such a transition is destructive, one must check if the action that labels this transition belongs to the given button. This check can be performed in a finite amount of time in two cases: (1) The number of transitions after the trace is finite. (2) The button is finite,

the number of transitions is enumerable, and the specification is given in such a way that the transitions by the same action are listed in succession. Since the number of states is finite, the number of transitions by the given action is also finite. In the course of examining the transitions, we mark the actions the transitions by which are nondestructive and that belong to the button until a destructive transition by the action is found or until all the actions initiated by the button are marked. If the number of transitions is infinite and the specification is given in a different way, the check is not guaranteed to terminate in a finite amount of time. The same is also true if the set of transitions and the button are infinite. This restriction can be weakened if only the transitions that continue routes with safe traces are taken into account.

The number of implementation states is finite. Otherwise, it is not guaranteed that an error is found in a finite amount of time; furthermore, if there are no errors, the testing is infinite. This restriction can be weakened by taking into account only the implementation states that are reachable by safe specification traces. If this weakened restriction does not hold, the conformal implementations take infinitely long time to be tested.

Strong connectivity of the implementation. For the testing to be complete, all the transitions in the implementation belonging to the routes with the traces that are safe in the specification must be checked. When the LTS generated by such transitions is tested, it is traversed; namely, the route containing all the reachable transitions is passed. For such a traversal to exist, the LTS must be strongly connected; that is, every state must be reachable by transitions from any other state. More precisely, the LTS must be a chain of strongly connected components and exactly one transition must lead from every component (except for the last one) to the next component [11]. However, an algorithm for such a traversal can be constructed only if every component in the chain except for the last one is a single state without loops. An LTS is strongly connected if the restart is defined in every state. In what follows, we assume that the LTS is strongly connected.

3. LTS TRAVERSAL ALGORITHM

First, we consider the traversal of an LTS implementation that does not contain the reachable destruction and divergence and that satisfies the following conditions: the LTS is strongly connected, t-nondeterministic, and the number of buttons is finite.

3.1. LTS of a Traversal

Denote by **I** the LTS implementation. When performing the traversal, we are going to construct an LTS that is called the passed LTS (it is denoted by **G**). The transition $i \xrightarrow{z} i'$ by the external action z is added when the action $z \in P$ and the poststate i' are observed after pressing the button P at the state i. This indicates that **I** contains a route beginning at i and ending at i' that has the trace $\langle z \rangle$; in other words, this route contains one transition by z and τ-transitions. If a refusal with *the same poststate* $i' = i$ is observed, no transitions are added. If the refusal P is observed with *another* poststate $i' \neq i$, then the transition $i \xrightarrow{\tau} i'$ is added. This indicates that **I** contains a τ-route beginning at i and ending in i' such that the state i' is stable and contains a refusal P. Together with the transition, we store a button that was pressed to cause this transition (any such button).

The traversal is completed when no new transition can be added to **G**. It is easy to verify that, after such a traversal, the LTSs **I** and **G** have the same set of reachable states and the same set of $\mathfrak{R}$ traces. Any algorithm that constructs a traversal of the LTS **G** guarantees a traversal of the LTS **I**.

For every button P and every passed state i, we define the counter $c(P, i)$ of the number of clicks on P at the state i. The button P is said to be *completed at the state i* if one of the following conditions is satisfied. (1) $c(P, i) = 1$, **G** does not contain τ-transitions from i, and it does not contain the transition $i \xrightarrow{z} i'$, where $z \in P$ or (2) $c(P, i) = t$. This indicates that all the possible transitions from the state i by pressing the button P have been obtained. In case (1), after pressing P at the state i, a refusal P at the same state was observed; another click on this button gives the same refusal. In case (2), all the possible transitions were obtained after pressing the button t times. The state i is said to be *completed* if every button is completed in this state. This indicates that all the possible transitions from the state i in **G** have been obtained.

Initially, **G** contains a single state $i \in \{\mathbf{I}\ \textit{after}\ \epsilon\}$ and $\forall P \in \mathfrak{R}\ c(P, i) = 0$.

3.2. General Scheme of the Traversal Algorithm

The general scheme of the traversal algorithm is shown in Fig. 2. If the current state is uncompleted, a certain button i is uncompleted in i. Then, we press this button to obtain an observation P (an action or refusal) and a poststate i', which becomes the new current state. Increment the counter $c(P, i) := c(P, i) + 1$, and add to **G** the transition $i \xrightarrow{o} i'$ if o is an action and add the transition $i \xrightarrow{\tau} i'$ if o is a refusal and $i \neq i'$.

If the current state is completed, all the test inputs have already been executed the required number of times and all the possible observations and poststates have been obtained. To continue the traversal, one must go to an uncompleted state. If there are no such states, the algorithm terminates because all the passed states are completed and, therefore, all the reachable transitions have been passed.

Consider a transition to an uncompleted state. In the graph of the LTS **G**, there exists a forest of trees that cover all the states and are directed to their roots

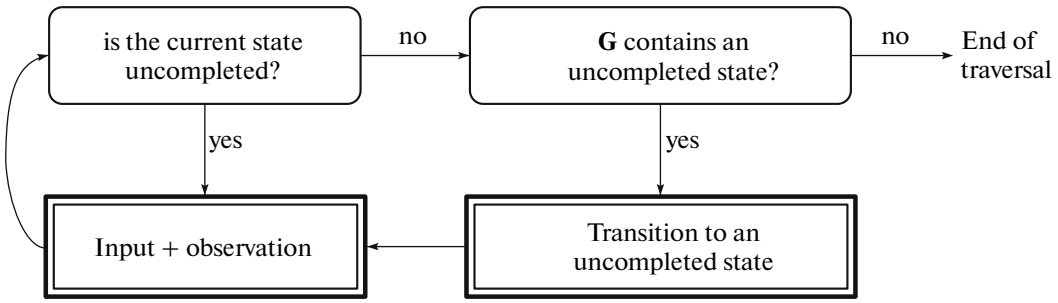

Fig. 2. General scheme of the traversal algorithm.

(the roots are all uncompleted states). Take an arbitrary forest and, for every transition $i \xrightarrow{o} i'$, denote by $P(i)$ the associated button. We perform the traversal by pressing the button $P(i)$ at every current state i. Because of the nondeterminism, we can find ourselves in a state i'' different from i', in which we press the button $P(i'')$.

We prove that any uncompleted state is reached in a finite number of steps. The proof is by contradiction. Assume that we performed an infinite number of steps while always passing through completed states. The length of the path in the forest from the state i to the uncompleted state (which is unique for the given i) is called the distance r_i. Since the number of states is finite, some completed states are visited an infinite number of times. Take among them the state i with the minimal distance r_i. We leave the state i by various transitions an infinite number of times by pressing the button $P(i)$; therefore, due to the restricted nondeterminism, we would have left i by a transition $i \xrightarrow{o} i'$ an infinite number of times. Consequently, we also would have visited the state i' an infinite number of times. But $r_{i'} = r_i - 1$, which contradicts the assumption that r_i is minimal. This contradiction proves the desired result.

3.3. Estimation of the Number of Test Inputs

In Fig. 2, test inputs are performed in the blocks surrounded by bold frames. Let n be the number of states, b be the number of buttons (taking into account the restart if it is defined at least in one state), and t be the nondeterminism restriction.

Consider the block *input + observation*. In each state, the test input is performed not more than bt times (even less if refusals with the same poststate are observed). In total, we have not more that btn test inputs.

Consider the block *"transition to an uncompleted state"*. Let us estimate the number of test inputs when this block is executed once and the number of completed states is c. Consider all the visited completed states. If the distance of a state is $r = 1$, then we leave it not more than t times due to the t-nondeterminism. Therefore, we leave the states with $r = 2$ not more than t^2 times. In the general case, we leave the states with the distance r not more than t^r times. Let a_r be the number of states with the distance r. Then, the number of test inputs is not greater than $f(c) = \sum_r (a_r t^r) \le t + t^2 + \ldots + t^c = (t^{c+1} - t)/(t-1)$ for $t > 1$ and not greater than c for $t = 1$.

Now, we estimate the total number of test inputs. Let us enumerate the states in the order they become completed. Assume that, at the time that immediately precedes obtaining the jth triple (button, observation, poststate) in the state i, there are c_{ij} uncompleted states. If we moved to the uncompleted state i before obtaining the jth triple, then denote by y_{ij} the number of test inputs in the course of this transition; otherwise, set $y_{ij} = 0$. As was proved above, we have $y_{ij} \le f(c_{ij})$. Since $c_{ij} \le i - 1$ and the function f is monotone increasing, we have $f(c_{ij}) \le f(i-1)$. The number of triples in each state does not exceed bt. Therefore, the upper bound on the number of test inputs in the block is $\sum_{ij}(y_{ij}) \le \sum_{ij}(f(c_{ij})) \le \sum_{ij}(f(i-1)) \le \sum_i (btf(i-1)) = bt\sum_i((t^i - t)/(t-1)) = O(bt^n)$ if $t > 1$, and $b\sum_i (i-1) = O(bn^2)$ if $t = 1$.

This bound is attained in terms of the order of magnitude for the LTS shown in Fig. 3. Here, there is a single action and a single button enabling this action. If, in the course of traversing the LTS, every transition shown by a dotted line is preceded by $t - 1$ transitions to the initial state (bold arrows) due to nondeterminism, then the lower bound is $O(t^n)$. Note that this bound is attained for any traversal algorithm and not only for the algorithm described above.

3.4. Estimation of the Amount of Computations

We assume that querying a state gives its unique identifier but we do not know its structure; we only can find out if two identifiers are identical. When querying a state, we must determine whether it is new or old; if it is old, we must identify it with the corresponding

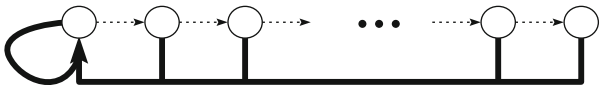

Fig. 3. Example of an LTS.

state. This requires $O(n)$ comparison operations. Therefore, the amount of computations does not exceed the length of the traversal more than by a factor of n: $O(nbt^n)$ for $t > 1$ and $O(bn^3)$ for $t = 1$. Let us prove that the other computations do not affect this order if the algorithm is implemented properly.

Note that the forest of trees required for a pass to an uncompleted state can be used multiply without modifications until an uncompleted state becomes completed.

Let us define the main data structures. We enumerate the buttons from 1 to b and enumerate all the passed states. Given the state index i, we can determine the following parameters:

the state's identifier $I(i)$;

the index of the current button $B(i)$; initially, $B(i) = 1$, and, if i is completed, $B(i) = b + 1$;

the counter of the current button; initially, $C(i) = 0$;

the list $T(i)$ of different indexes of the states where the transitions leading to i begin; initially, the list $T(i)$ is empty;

the index of the button in the forest of trees; initially, $P(i) = 0$.

The index of the current state i_c, the counter N of the passed states (initially, $N = 0$), the counter of completed states C (initially, $C = 0$), and the matrix of order $N \times N$ are stored separately; the entry $M(i, j)$ is the index of the button associated with a transition $I(i) \xrightarrow{o} I(j)$ or 0 if there is no such button.

The block *is the current state uncompleted?* checks if $B(i_c) \le b$. The number of comparisons is equal to the number of transitions, which is not greater than $O(btn)$.

The block *input + observation*. In the state i_c, we press the button $B(i)$ to obtain an observation o and a poststate. Determine the index of the poststate i' by looking through the array of states using $O(n)$ operations. If i' is a new state, it is assigned the index $N + 1$, its description is created, the matrix M is extended by adding $N + 1$ zero rows and a column, and we set $N := N + 1$. Furthermore, we set $C(i_c) := C(i_c) + 1$. If a refusal is observed and $i' = i_c$ or $C(i_c) = t$, we choose the next button: $B(i_c) := B(i_c) + 1$ and set $C(i_c) := 0$. If $M(i_c, i') = 0$, then the state i_c is added to the list $T(i')$ and the button $B(i)$ is added to the matrix: $M(i_c, i') := P$. The current state is changed: $i_c := i'$. The block is executed not more than btn times, and the total estimate of the amount of computations is $O(btn^2)$.

When the state i_c becomes completed ($B(i_c) = b + 1$), we restructure the forest of trees for $C := C + 1$. Create the working list L containing the indexes of the leaf states of the forest. First, scan all the passed states once to set the button indexes to zero ($P(i) := 0$) and add the indexes of the uncompleted states to L ($O(n)$ operations). Furthermore, for the element i at the head of the list L, we scan the list $T(i)$ and, for each state j in this list, check if it already belongs to the forest of trees. If it does not ($P(j) = 0$ & $B(j) = b + 1$), the state j is placed at the tail of L and $P(j) := M(j, i)$. The state i is removed from L. The construction completes when L is empty. The number of the transitions checked by this procedure does not exceed $O(btn)$. Therefore, the forest is constructed using $O(n) + O(btn) = O(btn)$ operations. The forest is constructed when an uncompleted state becomes completed (it remains completed up to the end). Hence, the forest is constructed not more than n times, which gives the estimate $O(btn^2)$ of the complexity of constructing all the forests.

The total estimate for the block *input + observation* is $O(btn^2)$.

The block **G** contains an uncompleted state? checks the condition $C \ne N$, which requires $O(btn)$ operations.

The block *transition to an uncompleted state*. To perform a transition, we press the button $P(i_c)$ and query the poststate. Determine the index of the poststate i' using $O(n)$ operations, change the current state $i_c := i'$, and check if it is completed. If it is not ($B(i_c) < b + 1$), the operation is repeated. Taking into account the number of test inputs in this block, we obtain the total estimate of the amount of computations $O(nbt^n)$ for $t > 1$ and $O(bn^3)$ for $t = 1$.

For $t > 1$, we have $n \le t^{n-1}$. Therefore, we have the following estimate of the amount of computations: $O(btn) + O(btn^2) + O(btn) + O(nbt^n) = O(nbt^n)$ for $t > 1$ and $O(btn) + O(btn^2) + O(btn) + O(bn^3) = O(bn^3)$ for $t = 1$.

3.5. Strongly Δ-Connected LTSs

The traversal algorithm just described can be used for any finite strongly connected t-nondeterministic LTS. When $t = 1$, we have the deterministic case with the estimate of the traversal length $O(bn^2)$ and the amount of computations $O(bn^3)$. However, for $t > 1$, both estimates are exponential, which is hardly suitable for practice. These estimates can be improved if we restrict ourselves to certain subclasses of LTS. One such subclass consists of strongly Δ-connected LTSs.

The concept of strongly Δ-connectivity was introduced in [7] to solve the problem of traversing the transition graph of a nondeterministic FMS. In that case, the test input is interpreted as sending a stimulus to the machine and an observation is interpreted as getting a response from it. The FMS does not contain unlabeled transitions (τ-transitions in the LTS). Here, we reformulate the basic definitions and results of [7] in terms of the LTS and $\Re$-semantics.

A Δ-transition (s, P) is defined as the set of transitions $s \xrightarrow{z} s'$, where $z \in P \in \Re$; the $[u, V]$-Δ route is the set D of routes such that u is their common beginning, V is the set of their endpoints, and, for any prefix of any route in D ending in the state s, there is a button P such that the set of transitions continuing this prefix in D coincides with the Δ-transition (s, P). If $V = \{v\}$, then we have a $[u, v]$-Δ route. The length of a [Delta]-route is the maximum of the lengths of its routes. A route is called a button traversal if it contains at least one transition from each nonempty Δ-transition. A Δ-traversal is a Δ-route in which all the routes are button traversals. A Δ-traversal algorithm performs a button traversal for arbitrary nondeterministic behavior of the LTS; in total, all these traversals form a Δ-traversal. A Δ-path is defined as a Δ-route in which all the routes are paths. If there exists a $[u, v]$-Δ path, then v is Δ-reachable from u. An LTS is strongly Δ-connected if any state is Δ-reachable from any other state. Strongly Δ-connected LTSs do not contain τ-transitions because an unstable state cannot be Δ-reachable. Note that Δ-reachability is considered with regard for restarts.

In [7], a Δ-traversal algorithm for strongly Δ-connected LTSs was proposed in terms of LTS. That algorithm has the sharp estimate $O(nm)$ of the traversal length and the estimate $O(n^2m)$ for the amount of computations, where n is the number of states and m is the number of Δ-transitions. The algorithm is based on a local approximation of the Δ-distance of the state u from the set of states V, where the Δ-distance is the minimal length of the $[u, V]$-Δ route in the passed LTS. The set of states in which not all the buttons were yet pressed was used as V, In the case of t-nondeterminism, the set of uncompleted states is used as V for the traversal (but not Δ-traversal). Every Δ-transition contains $O(t)$ transitions and $m \le bn$; therefore, the estimates are as follows: $O(btn^2)$ for the number of test inputs and $O(btn^3)$ for the amount of computations.

4. TESTING ALGORITHM

First, we note that, if we are sure that the destruction and divergence are not reachable in the implementation under test (in particular, when they are not included in the specification), then testing is reduced to a traversal of the implementation and the subsequent analytical verification of the passed LTS. In the general case, the traversal algorithms are modified, and the verification is performed simultaneously with the traversal.

4.1. Safely Reachable LTS

Denote the LTS implementation by **I** and the LTS specification by **S**. When performing the tests, we traverse the safely reachable LTS **G** in which the transitions are induced by pressing the buttons in the state i that are safe after a trace that is safe in the specification and ends in the implementation in the state i (rather than all the buttons in the state i). If strong Δ-connectivity is assumed, this property must be satisfied only for the safely reachable LTS of the implementation rather than the entire implementation.

As before, $c(P, i)$ is the number of clicks of the button P in the state i (see Subsection 3.1). Also, for the state i, we store the set $\mathbf{S}(i)$ consisting of the sets $S = (\mathbf{S}\ \textit{after}\ \sigma)$, where $\sigma \in \textit{Safe}(\mathbf{S})$ and $i \in (\mathbf{I}\ \textit{after}\ \sigma)$. The set s is added to $\mathbf{S}(i)$ when the trace σ is obtained and the implementation is in the state i. The button P is *admissible* in i if it is safe at least in one of the sets $S \in \mathbf{S}(i)$. In the state i, only admissible buttons are pressed. The definition of the completed state changes: now we require that every admissible in this state (rather that every) button be completed in this state.

The implementation **I** and the completely constructed LTS **G** have the same set of safely testable traces of the form $\sigma \cdot \langle o \rangle$, where $\sigma \in \textit{Safe}(\mathbf{S}) \cap T(\mathbf{I})$ and the observation o is enabled by a certain button P *safe* **S** *after* σ. Therefore, **I** *saco* **S** ⇔ **G** *saco* **S**. Moreover, **I** and **G** have the same set of states that are reachable by safely testable traces.

4.2. Start of the Algorithm

At the start of the testing, **G** contains a single state $i \in (\mathbf{I}\ \textit{after}\ \epsilon)$ and $\mathbf{S}(i) = \{\mathbf{S}\ \textit{after}\ \epsilon\}$. All the buttons P *safe* **S** *after* ε are enabled, and $c(P, i) := 0$.

If the initial state of the implementation i_0 is unstable, we need to obtain all the states from the set **I** *after* ε in the course of testing. To this end, it is necessary and sufficient that the restart be defined at least at one state of the implementation that is reachable by a safe specification trace.

If this condition is not fulfilled and the state i_0 is unstable, the strong connectivity guarantees the traversal but not the completeness of testing. An example is shown in Fig. 4. Here, **I** ~~*saco*~~ **S**; indeed, from the very beginning, **I** contains the action y, which is not allowed by the specification (there must be the refusal $\{y\}$). If we find ourselves in state 1 at the beginning of testing, then, without the restart, we can reach state 0 only after pressing the button $\{x\}$. According to the specification, the button $\{y\}$ is unsafe after the traces containing x—it may not be pressed. Therefore, the bug will not be detected.

4.3. General Scheme of the Algorithm

The general scheme of the algorithm is shown in Fig. 5. In distinction from the traversal scheme shown in Fig. 2, there are three extra blocks here; they are enclosed in a wide grey frame. In addition, only admissible buttons can be pressed in the block *input + observation*. This guarantees that only the traces that are safe in the specification are passed because only the buttons that were earlier pressed in the *transition to*

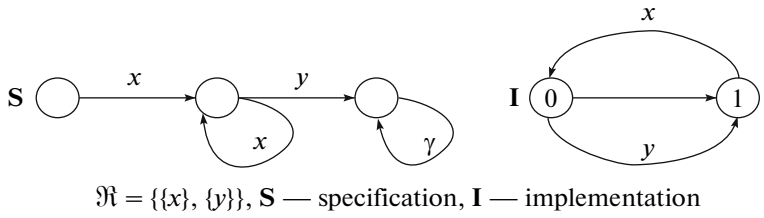

Fig. 4. Example of a specification and nonconformal implementation.

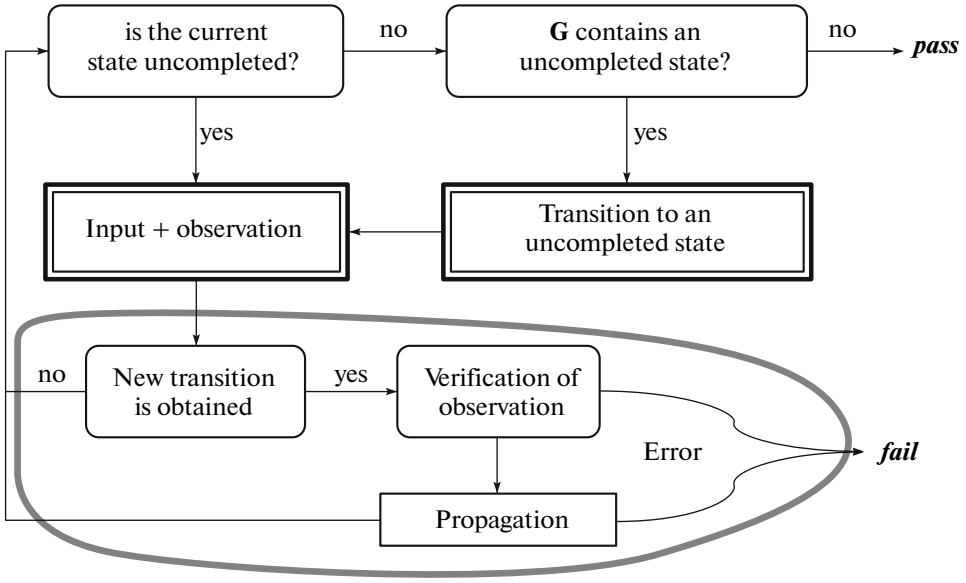

Fig. 5. General scheme of the algorithm.

an uncompleted state block can be pressed in the block *input + observation* (and they are admissible).

Define the procedure *Post*($i \xrightarrow{o} i'$, S) that combines the verification of an observation (oracle) and the computation of the set of the specification poststates. Here, $i \xrightarrow{o} i'$ is a transition in the LTS **G** and $S \in \mathbf{S}(i)$. If an error (nonconformance) is detected, the testing terminates with the verdict *fail*. Otherwise, the computed set of poststates S' is added to the set $\mathbf{S}(i')$ provided that $S' \neq \emptyset$ and $S' \notin \mathbf{S}(i')$; the new state is labeled in $\mathbf{S}(i')$ by a special flag *added*. Three cases are possible.

(1) o is an external action. If there is a button P *safe* S such that $o \in P$, then S' consists of all the states s' that are reachable from **S** by τ-transitions from the endpoints of the transitions $s \xrightarrow{z} s'$, where $s \in S$. If $S' = \emptyset$, then $o \notin obs(S, P)$, and an error is registered. If there is no such a button P, nothing is done.

(2) $o = P$ is a refusal (a virtual loop). If P *safe* S, then S' consists of all the states $s' \in S$ that contain a refusal P. If $S' =$ ststates S' is added to the set $\mathbf{S}(i')$ provided that $S' \neq \emptyset$, then $o \notin obs(S, P)$, and an error is registered. If P ~~*safe*~~ S, nothing is done.

(3) $o = \tau$ is an internal action. $S' = S$ is computed, and no error is registered.

The block *verification of observation* goes on when a *new* (possibly, virtual) transition $i \xrightarrow{o} i'$ is added. If o is the restart, the set **S** *after* ϵ is added to $\mathbf{S}(i')$ if it is not yet included there and label it by the flag *added* in $\mathbf{S}(i')$. If o is the refusal of the restart, nothing is done. Otherwise, call the procedure *Post*($i \xrightarrow{o} i'$, S) for each $S \in \mathbf{S}(i)$. If no errors are registered, the block *propagation* is invoked.

In this block, for every transition $i \xrightarrow{o} i'$, where o is not the restart, the refusal of the restart, and an *added* set of states $S \in S(i')$, we call the procedure *Post*($i \xrightarrow{o} i'$, S) and then remove the flag *added* from the set S in $\mathbf{S}(i)$. These operations are repeated until there are added sets. Since the sets of implementation and specification states are finite (therefore, the num-

ber of sets of states of the specification is also finite), the block *propagation* terminates in a finite amount of time.

It is important to note that not only the observations obtained after *actual* traces passed in the course of testing are verified but also possible observations after *potential* traces are verified as well. More precisely, the observations are verified for which it is established that they can be realized in the implementation after the known traces that are safe in the specification. This yields considerable savings in the number of test inputs needed to verify the conformance; indeed, we perform many checks without actual testing on the basis of the acquired knowledge about the implementation's behavior.

Let us prove that testing using the algorithm just described is complete; that is, we prove that it is significant and exhaustive.

First, we prove that, for every state i in the LTS $\mathbf{G}$, every set $S \in \mathbf{S}(i)$ is an endpoint of a certain trace that is safe in the specification: $\exists\sigma \in \mathit{Safe}(\mathbf{S}) \cap T(\mathbf{I})\ S = (\mathbf{S}\ \mathit{after}\ \sigma)\ \&\ i \in (\mathbf{I}\ \mathit{after}\ \sigma)$. The proof is by induction on the sequence of additions $\mathbf{S}(i) := \mathbf{S}(i) \cup S$, where $S \notin \mathbf{S}(i)$ for any S and any i. Such an addition is made in three cases: (1) at the start of the work; (2) in the block *verification and observation* for the restart; and (3) in the procedure *Post*. In cases (1) and (2), the assertion is true for the empty trace $\sigma = \epsilon$. The procedure $\mathit{Post}(i \xrightarrow{o} i', S)$ adds the set of states S' for the transition $i \xrightarrow{o} i'$ to $\mathbf{S}(i')$ in such a way that, if $\exists\sigma \in \mathit{Safe}(\mathbf{S}) \cap T(\mathbf{I})$ such that $S = (\mathbf{S}\ \mathit{after}\ \sigma)\ \&\ i \in (\mathbf{I}\ \mathit{after}\ \sigma)$, then o *safe* $(\mathbf{S}\ \mathit{after}\ \sigma)\ \&\ S' = (\mathbf{S}\ \mathit{after}\ \sigma \cdot \langle o\rangle)\ \&\ i' \in (\mathbf{I}\ \mathit{after}\ \sigma \cdot \langle o\rangle)$for $o \neq \tau$ and, for $o = \tau$, $S' = S\ \&\ i' \in (\mathbf{I}\ \mathit{after}\ \sigma)$. This proves the proposition.

Significance. If the test detects an error, the implementation is not conformal. Taking into account the proposition proved above, the procedure *Post* returns *fail* when it detects that, after a trace $\sigma \in \mathit{Safe}(\mathbf{S}) \cap T(\mathbf{I})$, a button P *safe* $\mathbf{S}$ *after* σ enables an observation $o \in \mathit{obs}(\mathbf{I}\ \mathit{after}\ \sigma, P)$ that is not included in the specification ($o \notin \mathit{obs}(\mathbf{S}\ \mathit{after}\ \sigma, P)$), that is, when the implementation is not conformal.

Exhaustiveness. If the test's verdict is *pass*, the implementation is conformal. It is sufficient to prove that, at the end of the test, for every trace $\sigma \in \mathit{Safe}(\mathbf{S}) \cap T(\mathbf{I})$, each state $i \in (\mathbf{I}\ \mathit{after}\ \sigma)$ belongs to the LTS $\mathbf{G}$ and $(\mathbf{S}\ \mathit{after}\ \sigma) \in \mathbf{S}(i)$. The proof is by induction on traces. For the empty trace ϵ, the assertion is true; indeed, at the start of testing after the restarts, we add $\mathbf{S}$ *after* ϵ to $\mathbf{S}(i)$ for every state $i \in (\mathbf{I}\ \mathit{after}\ \epsilon)$.

Consider the nonempty trace $\sigma \cdot \langle o\rangle \in \mathit{Safe}(\mathbf{S}) \cap T(\mathbf{I})$, where o is distinct from the restart and the refusal of restart and is enabled by a button P *safe* $\mathbf{S}$ *after* σ. Assume that the assertion is true for σ and prove it for $\sigma \cdot \langle o\rangle$. Let the state $i' \in (\mathbf{I}\ \mathit{after}\ \sigma \cdot \langle o\rangle)$. Then, the implementation includes a state $i \in (\mathbf{I}\ \mathit{after}\ \sigma)$ and the transition $i \xrightarrow{o} i'$ (if o is a refusal, this is the virtual loop). By the inductive assumption, the state i is included in $\mathbf{G}$ and $(\mathbf{S}\ \mathit{after}\ \sigma) \in \mathbf{S}(i)$. In the case of the verdict *pass*, all the states are completed; therefore, the transition $i \xrightarrow{o} i'$ must be obtained and the procedure $\mathit{Post}(i \xrightarrow{o} i', S)$ must be performed in the state i after pressing the button P. In this case, $(\mathbf{S}\ \mathit{after}\ \sigma \cdot \langle o\rangle) \in \mathbf{S}(i)$, which was to be proved.

4.4. Estimation of the Number of Test Inputs

In contrast to traversal, only admissible buttons are pressed in the course of testing. The admissibility of a button depends on the trace. It may happen that the state $i \in (\mathbf{I}\ \mathit{after}\ \sigma)$ is completed and the button P *safe* $\mathbf{S}$ *after* σ is inadmissible in i because the trace σ was not obtained (actually or potentially). After obtaining the trace σ, i becomes uncompleted again.

Now, we cannot assume that a completed state always remains in this position. When estimating the amount of computations in the block *transition to an uncompleted state*, we can no more assume that the number of completed states is monotonically nondecreasing, and this number affects the estimate of the number of test inputs for a single execution of this block. We give an estimate based on the number of executions of this block. After going to an uncompleted state, we press an uncompleted button. Therefore, the block is executed not more than btn times and not more than $f(n-1)$ test inputs are applied. This gives the estimate $O(bnt^n)$ for $t > 1$ and $O(bn^2)$ for $t = 1$.

For the deterministic case ($t = 1$), the estimate did not change, but it increased by a factor of n for $t > 1$. One can suppose that this estimate is too high and the exact estimate is the same—$O(bt^n)$. This conjecture is true for the case $b = 1$, which is identical to traversal. Indeed, a completed state becomes uncompleted again if a new button admissible in this state appears; however, in this case, there is a single button.

This conjecture can also be proved if the restart is defined in every state that is reachable in the test. The transition to an uncompleted state is possible from the initial state after the restart (or several restarts if the initial state is unstable; this does not increase the estimate). Select a tree that is oriented from the initial state and contains all the passed states. Let us move on this tree while pressing the corresponding buttons. If we do not reach the desired transition because of nondeterminism, we again make a restart and begin from the beginning. Denote by a_r the number of states at the distance r from the initial state in the tree. In order to surely reach this state, not more than $O(t^r)$ test inputs are needed. In order to reach every state one time, not more than $\sum_r (a_r t^r) \le t + t^2 + \ldots + t^{n-1} =$ $O(t^{n-1})$ test inputs are needed. We need to reach each state not more than bt times; therefore, the ultimate estimate is $O(bt^n)$.

4.5. Estimation of the Amount of Computations

Consider the differences between the testing and the traversal that affect the amount of computations.

1. More test inputs. The number of test inputs is multiplied by n, which is the complexity of finding the index of a state by its identifier; this yields $O(bn^2t^n)$ for $t > 1$, or $O(bnt^n)$ if $b = 1$ or the restart is defined everywhere, and $O(bn^3)$ for $t = 1$.

2. The forest of trees is constructed more times, namely, $O(bn)$ times instead of $O(n)$ because an uncompleted state becomes completed when a certain button is pressed for the last time. Therefore, the estimate increases by a factor of b: $O(b^2tn^2)$.

3. Additional blocks of the algorithm. The greatest contribution is from the procedure $Post(i \xrightarrow{o} i', S)$—its one-time execution requires $O(1)$ operations. A specification with k states and an implementation with n states can be constructed for which the LTS **G** includes all the pairs (i, S); that is, it includes $n2^k$ pairs. For each pair, not more than bt transitions are checked. This gives the estimate $O(btn2^k)$.

All the checks performed by the procedure *Post* are needed to verify the conformance: all the observations that are possible in the implementations after all safe specification traces are checked. Test inputs are required only for a part of these checks ($O(btn)$); the other checks are performed at the stage of propagation.

The final estimate is $O(bn^2t^n) + O(b^2tn^2) + O(btn2^k)$. If $b = 1$ or the restart is defined everywhere, the first term is $O(btn^n)$; if $t = 1$, then the first term is $O(bn^3)$.

4.6. Strongly Δ-Connected Implementations

The number of test inputs is mainly determined by the transitions to uncompleted states. The algorithm based on a local approximation of Δ-distances in the LTS **G** assumes that every state becomes completed only once. However, exact Δ-distances can be calculated. Again, the number of test inputs on the way to an uncompleted state is not greater than $O(n)$. The number of passes is not greater than btn; therefore, the final estimate is $O(btn^2)$.

Consider the three terms of the amount of computations.

1. Finding state indexes from their identifiers. The estimate of the number of test inputs is $O(btn^2)$; therefore, the estimate of the amount of computations is $O(btn^3)$.

2. Computation of Δ-distances. Let us label the Δ-transitions (buttons in the states) used to calculate the minimal Δ-distances. This is similar to the construction of the forest of trees and the required number of operations is of the order of the number of transitions $O(btn)$. The Δ-distances are recalculated when an uncompleted state becomes completed. A state becomes completed when a certain button is pressed for the last time; that is, not more than b times. The total estimate is $O(b^2tn^2)$.

3. Computations in the procedure *Post*. As in the general case, the estimate of the number of operations is $O(btn2^k)$.

The ultimate estimate is $O(btn^3) + O(b^2tn^2) + O(btn2^k)$.

REFERENCES

1. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Formalization of Test Experiments, *Programmirovanie,* 2007, no. 5, pp. 3–32 [*Programming Comput. Software* (Engl. Transl.), 2007, vol. 33, no. 5, pp. 239–260].
2. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., *Teoriya sootvetstviya dlya system s blokirovkami i razrusheniem* (Conformance Theory for Systems with Refusals and Destruction), Moscow: Nauka, 2008.
3. Bourdonov, I.B., Conformance Theory for the Functional Testing of Software Systems Based on Formal Models, *Doctoral (Math.) Dissertation,* Moscow: Institute for System Programming, Russian Academy of Sciences, 2008; http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf.
4. van Glabbeek, R.J., The Linear Time—Branching Time Spectrum, *Proc. of CONCUR'90*, Baeten, J.C.M. and Klop, J.W., Eds., *Lect. Notes Comput. Sci.*, 1990, vol. 458, pp. 278–297.
5. van Glabbeek, R.J., The Linear Time—Branching Time Spectrum II: The Semantics of Sequential Processes with Silent Moves, *Proc. of CONCUR'93,* Hildesheim, Germany, 1993, Best, E., Ed., *Lect. Notes Comput. Sci.*, 1993, vol. 715, pp. 66–81.
6. Milner, R., Modal Characterization of Observable Machine Behavior, *Proc. CAAP*, 1981, Astesiano, G. and Bohm, C. Eds., *Lect. Notes Comput. Sci.,* 1981, vol. 112, pp. 25–34.
7. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case, *Programmirovanie*, 2004, no. 1, pp. 4–24 [*Programming Comput. Software* (Engl. Transl.), 2004, vol. 30, no. 1, pp. 2–17].
8. Blass, A., Gurevich, Y., Nachmanson, L., and Veanes, M., Play to Test Microsoft Research, *Techn. Report*, MSR-TR-2005-04, 2005, *5th Int. Workshop on Formal Approaches to Testing of Software (FATES 2005)*, Edinburgh, 2005.
9. Fujiwara, S. and Bochmann, G.V., Testing Nondeterministic Finite State Machine with Fault Coverage, in *Proc. of the Fourth IFIP TC6 Int. Workshop on Protocol Test Systems, 1991*, Kroon, J., Heijing, R.J., and Brinksma, E., Eds., North-Holland, 1992, pp. 267–280.
10. Milner, R., Calculus of Communicating Processes, *Lect. Notes Comput. Sci.*, 1982, vol. 92.
11. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case, *Programmirovanie*, 2003, no. 5, pp. 11–30 [*Programming Comput. Software* (Engl. Transl.), 2003, vol. 29, no. 5, pp. 245–258].
12. Kuliamin, V.V., Petrenko, A.K., Kossatchev, A.S., and Bourdonov, I.B., The UniTesK Approach to Designing

Test Suites, *Programmirovanie*, 2003, no. 6, pp. 25–34 [*Programming Comput. Software* (Engl. Transl.), 2003, vol. 29, no. 6, pp. 310–322].

13. Petrenko, A., Yevtushenko, N., and Bochmann, G.V., Testing Deterministic Implementations from Nondeterministic FSM Specifications, in *Selected Proc. of the 9th IFIP TC6 Int. Workshop on Testing of Communicating Systems*, 1996.
14. Goodenough, J.B. and Gerhart, S.L., Toward a Theory of Test Data Selection, *IEEE Trans. Software Eng.*, 1975, vol. SE-1, no. 2, pp. 156–173.
15. Grochtmann, M. and Grimm, K., Classification Trees for Partition Testing, *Software Testing, Verificcation and Reliability*, 1993, no. 3, pp. 63–82.
16. Zhu, Hall, May, Software Unit Test Coverage and Adequacy, *ACM Comput Surveys*, 1997, vol. 29, no. 4.
17. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Application of Finite Automatons for Program Testing, *Programmirovanie*, 2000, no. 2, pp. 12–28 [*Programming Comput. Software* (Engl. Transl.), 2000, vol. 26, no. 2, pp. 61–73].
18. Vasilevskii, M.P., On the Detection of Faults in an Automaton, *Kibernetika,* 2973, vol. 9, no. 4, pp. 93–108.
19. Aho, A.V., Dahbura, A.T., Lee, D., and Uyar, M.Ü., Optimization Technique fpr Protocol Conformance Test Generation Based on UID Sequences and Rural Chinese Postman Tours, *IEEE Trans. Commun.*, 1991, vol. 3, no. 11, pp. 1604–1615.
20. Lee, D. and Yannakakis, M., Testing Finite State Machines: State Identification and Verification, *IEEE Trans. Comput.*, 1994, vol. 43, no. 3, pp. 306–320.
21. Lee, D. and Yannakakis, M., Principles and Methods of Testing Finite State Machines—A Survey, in *Proc. of the IEEE 84*, 1996, no. 8, pp. 1090–1123.
22. Legeard, B., Peureux, F., and Utting, M., Automated Boundary Testing from Z and B, *Proc. of the Int. Conf. on Formal Methods Europe, FME'02*, Lect. Notes Comput. Sci., 2002, vol. 2391, pp. 21–40.
23. Bourdonov, I.B., Traversal of an Unknown Directed Graph by a Finite Robot, *Programmirovanie*, 2004, no. 4, pp. 11–34 [*Programming Comput. Software* (Engl. Transl.), 2004, vol. 30, no. 4, pp. 188–203].
24. Bourdonov, I.B., Backtracking Problem in the Traversal of an Unknown Directed Graph by a Finite Robot, *Programmirovanie,* 2004, no. 6, pp. 6–29 [*Programming Comput. Software* (Engl. Transl.), 2004, vol. 30, no. 6, pp. 305–322].
25. Bourdonov, I.B., Examination of Unidirectional and Bidirectional Distributed Networks by a Finite Robot, *Trudy Vserossiiskoi konferentsii nauchnyi servis v seti Internet* (Proc. of the All-Russia Conf. on the Research Services on the Internet), 2004.
26. Edmonds, J. and Johnson, E.L., Matching, Euler Tours, and the Chinese Postman, *Math. Programm.*, 1973, no. 5, pp. 88–124.
27. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., "Security, Verification, and Conformance Theory," in *Materialy vtoroi mezhdunarodnoi nauchnoi konferentsii po problemam bezopasnosti i protivideistviya terrorizmu* (Proc. of the Second Int. Conf. on Security Problems and Terrorism Counteractions), Moscow: MNTsMO, 2007.