ISSN 0361-7688, Programming and Computer Software, 2013, Vol. 39, No. 6, pp. 269–278. © Pleiades Publishing, Ltd., 2013.
Original Russian Text © I.B. Bourdonov, A.S. Kossatchev, 2013, published in Programmirovanie, 2013, Vol. 39, No. 6.

Agreement between Conformance and Composition

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 June 12, 2013

Abstract—In our previous paper [1], a new model of a Labeled Transition System (LTS)-type implementation was proposed. In ordinary LTSs, transitions are labeled by actions; therefore, they can be called LTSs of actions. The new model is an LTS of observations; in this model, observations and test actions (*buttons*) are used instead of actions. This model generalizes many testing semantics that are based on the LTS of actions but use additional observations (refusals, ready sets, etc.). Moreover, systems with priority, which are not described by the LTS of actions, are simulated uniformly. In the present paper, we develop this approach by focusing on the composition of systems. The point is that, on observation traces, one cannot define a composition with respect to which a composition of LTSs would possess the property of *additivity*: the set of traces of a composition of LTSs coincides with the set of all pairwise compositions of traces of LTS operands. This is explained by the fact that an observation in a composition state is not calculated based on observations in states–operands. In this paper, we propose an approach that eliminates this drawback. To this end, we label the transitions of LTSs by symbols (*events*) that, on the one hand, can be composed to guarantee the property of additivity, and, on the other hand, can be used to generate observations under testing: a transition by an event gives rise to an observation related to this event. This model is called an LTS of events. In this paper, we define (1) a transformation of an LTS of events into an LTS of observations to conform with the principles of our previous paper [1]; (2) a composition of LTSs of events; (3) a composition of specifications that preserves conformance: a composition of conformal implementations is conformal to a composition of specifications; and (4) a uniform simulation of LTSs of actions in terms of the LTSs of events, which allows one to consider an implementation in any interaction semantics admissible for LTSs of actions. In this case, a composition of LTSs of events obtained as a result of simulation of the original LTSs of actions is equivalent to the LTS of events obtained as a result of simulation of the composition of these LTSs of actions.

DOI: 10.1134/S0361768813060029

1. INTRODUCTION

A Labeled Transition System (LTS) is the basic model of interacting processes. For this model, operations of the algebra of processes and, first of all, the operation of parallel composition of LTSs have been defined. Testing is understood as the interaction of an implementation with a test system formalized as a parallel composition of an LTS implementation with an LTS test. Originally, it was assumed that the only observations over LTSs are external (observable) actions, which label the transitions of LTSs. Unobservable internal activity is also admitted, which is represented in an LTS by transitions by internal (unobservable) actions, which only change the state of the LTS. Since such actions are unobservable, they are indistinguishable under testing and are denoted by the same symbol τ. Such a model can be called an LTS of *actions*, or an action transition system (ATS). The result of testing is a sequence of observable actions—a trace of actions. On the traces of actions, one can define a composition of traces so that the testing and a composition of LTSs turn out to be consistent: the set of traces of a composition of LTSs coincides with the set of all pairwise compositions of traces of LTS operands. This property can be called *additivity* of traces with respect to composition.

Another formalization of testing (closely related to the algebra of processes) is based on the so-called testing machine [7–9], which fixes some capabilities of the test system to control and observe the behavior of implementation. It is assumed that, using a testing machine, one can allow or forbid an implementation to execute transitions by one or other external actions, while τ activity is always allowed. It is clear that other observations are possible under testing except for actions. Anyway, all of them are related to the observation of stopping of an LTS implementation that arises when a current state is stable (there is no τ activity, i.e., τ transitions, in this state), and one cannot execute transitions by external actions that are defined in this state, because all of them are forbidden at a given instant of time by the testing machine. These observations include, in addition to the very fact of stopping, a refusal set and a ready set. The refusal set is the set of actions that are allowed by the testing machine, but there are no transitions by these actins at a current state of implementation. If we observe the stopping of a machine for one or other set of allowed external actions, then we observe the corresponding refusal set. The ready set is the set of all external actions by which there are transitions at a current state of implementation. Such an observation is characteristic, for example, of systems with graphic interface, when a menu of possible actions is displayed on the screen and the system waits for one or other choice made by the user. Now the result of testing is an observation trace, which may include not only external actions, but also other observations. The collection of admissible actions and observations formalizes the capabilities of the test system and is defined by the interaction semantics. For example, the well-known conformance relation *ioco* (*Input Output COnformance*) [11,12], which is based on the division of actions into stimuli (*input*) and reactions (*output*), admits two types of test actions—sending a stimulus or reception of all reactions, and the only type of refusal—the absence of reactions (*quiescence*).

A drawback of this formalization is that all observations, except for external actions, are described by an LTS model implicitly. Moreover, a test action actually reduces to a set of external actions allowed by this test action. All this, first, complicates the interpretation of the model during testing, second, does not give freedom of considering new types of observations and test actions for one or other interaction semantics, and, third, does not allow one to explicitly define in the model the test actions to which it reacts by some or other explicitly defined observations.

In our previous paper [1], we tried to get rid of this drawback and defined a new generalized model of an LTS. In this model, transitions are labeled by observations or test actions, called *buttons*, rather than by external actions. In addition to the freedom concerning buttons and observations in an implementation, this allowed us to uniformly simulate systems with priorities, which are not described in terms of LTSs of actions. This new model can be called an observation transition system (OTS).

However, there remains another significant drawback of both LTSs of observations and LTSs of actions when considering the traces of not only actions but also of other observations. The point is that a transition from actions to observations leads to a mismatch between testing and a composition of LTSs: on observation traces, it is impossible to define a composition so that the property of additivity would hold. This is associated with the fact that, in general, an observation in a composition state is not calculated by observations in the operand states. The only exception, besides an external action, is the ready set.

In this paper, we propose an approach that eliminates this drawback. The idea consists in labeling transitions of LTSs by symbols (called *events*) such that, on the one hand, these symbols can be composed to guarantee the property of additivity, and, on the other hand, they can be used to generate observations when testing: a transition by an event gives rise to one or other observation related to this event. The latter property can be called *generativity* of event traces: all observation traces of an implementation are calculated by these traces. This model can be called an event transition system (ETS).

We define (1) a transformation of an ETS into an OTS to comply with the concepts of our previous paper [1]; (2) a composition of ETSs; (3) a composition of specifications that preserves conformance: a composition of conformal implementations is conformal to a composition of specifications; and (4) uniform simulation of ATSs in terms of ETSs that allows one to consider an implementation in any interaction semantics admissible for an ATS. Here we aim at guaranteeing the following important property of *consistency* between a composition and simulation: a composition of ETS operands obtained as a result of simulation of ATS operands is equivalent to an ETS obtained as a result of simulation of a composition of these ATS operands. Here equivalence is understood as the equality of the sets of event traces.

2. AN LTS OF EVENTS AND GENERATION OF OBSERVATIONS BY EVENTS

By an LTS is meant a directed graph with nonempty set of vertices, which are called states; one of the states is distinguished as the initial state, and the arcs are called transitions and labeled by symbols from a certain universe of symbols L or by a special symbol of an internal action $\tau \neq\in L$. We also introduce a special additional symbol of destruction $\gamma \in L$ that simulates any behavior of an LTS that is undesirable when interacting with this LTS [2, 3, 6]. An LTS S is defined by a quadruple, $S = Lts(V_S, E_S, s_0)$, where V_S is the set of states, $E_S \subseteq V_S \times (L \cup \{\tau\})$ is the set of transitions, and s_0 is the initial state. The expression $a \xrightarrow{x} a'$ denotes that $(a, x, a') \in E_S$. A trace of an LTS is a sequence of labels on a path starting from the initial state, the symbol τ being omitted. Denote the set of traces of an LTS S by $tr(S)$.

When defining an ATS (an LTS of actions), we assume that a universe of external actions $A \subseteq L$ is defined. By a destruction is meant an external action $\gamma \in A$. An ATS is an LTS $S = Lts(V_S, E_S, s_0)$ in which each transition is labeled either by a symbol τ or by an action from A, i.e., $E_S \subseteq V_S \times (A \cup \{\tau\}) \times V_S$. We will write $S = Ats(V_S, E_S, s_0)$. The traces of an ATS are called traces of actions; they contain only actions.

To define an OTS (or an LTS of observations) introduced in our previous paper [5], we assume that two disjoint universes are defined: a universe of buttons (test actions), $B \subseteq L$, and a universe of observations, $O \subseteq L$, such that $B \cap O = \varnothing$. By a destruction is meant an observation $\gamma \in O$. An OTS is an LTS $S = Lts(V_S, E_S, s_0)$ in which each transition is labeled either by a symbol

τ, by a button from B, or by an observation from O; i.e., $E_S \subseteq V_S \times (B \cup O \cup \{\tau\}) \times V_S$. In addition, the absence of a transition by a button in a state is interpreted as a loop transition by this button. For an OTS S, we write $S = Ots(V_S, E_S, s_0)$. The traces of an OTS are called observation traces; they contain only observations and buttons.

To define an ETS (an LTS of events), we will assume that, in addition to the universe of buttons $B \subseteq L$, a universe of events $E \subseteq L$ $B \cap E = \varnothing$ is defined that is disjoint from the universe of buttons. By a destruction is meant an event $\gamma \in E$. An ETS is an LTS $S = Lts(V_S, E_S, s_0)$ in which each transition is labeled either by a symbol τ, by a button from B, or by an event from E; i.e., $E_S \subseteq V_S \times (B \cup E \cup \{\tau\}) \times V_S$. In addition, the absence of a transition by a button in a state is interpreted as a loop transition by this button. For an ETS S, we write $S = Ets(V_S, E_S, s_0)$. The traces of an ETS are called event traces; they contain only events and buttons.

In [1], a pair B/O defines a testing machine such that its "black box" contains an OTS, each test action corresponds to a button from B on the keyboard of the machine, and an observation from O is displayed on the screen. If no button is pressed, then an implementation can execute any τ transition, a γ transition, or a transition by an observation $x \in O$. In the latter case, an observation x is displayed on the screen. A button $p \in B$ is displayed on the screen when it is pressed. When a button p is pressed, an implementation can execute a finite number of τ-transitions, after which it executes either a γ transition or a p transition.

In this paper, we consider an implementation as an ETS, rather than an OTS; therefore, we assume that the black box of the testing machine contains an ETS, rather than an OTS. A transition by a button in an ETS corresponds to a transition by the same button in an OTS, and a τ transition in an ETS corresponds to a τ transition in an OTS. To a transition by an event in an ETS, we assign a transition by one or other observation in an OTS. We will assume that a mapping of events into observations is defined by a universal partially defined single-valued function f: $E \longrightarrow O$. Let us require that the event of destruction be mapped into the observation of destruction: $\gamma \in Dom(f)$ and $f(\gamma) = \gamma$. An event $a \in Dom(f)$ is called an *observable* event.

The operation of an ETS is determined not only by its structure but also by the function f. When a button $p \in B$ is pressed, then an implementation can execute a finite number of τ transitions, after which either a γ transition or a p transition is executed. After a p transition (before pressing the next button), as well as at the beginning of testing before pressing the first button, the implementation can execute any τ transition or a transition by an observable event $a \in Dom(f)$; in the latter case, we have an observation $f(a)$. Notice that a transition by destruction, just as a τ transition, can always be executed; however, in contrast to a τ transition, which does not give observations, one observes γ during a γ transition. We also note that transitions by unobservable events (outside the domain of the function f) are not executed under testing. However, we will need such transitions in order that the composition of an ETS and event traces be additive. Moreover, a composition of two events one or each of which is unobservable can be an observable event.

Note that ETSs with a single set of events can be assumed equivalent because there are a sufficient number of traces of events to calculate traces of observations and, in view of the additivity property (which we demonstrate in the next section), they are sufficient for a composition.

The function f defines a natural mapping of an ETS into an OTS under which a transition by an event either is replaced by a transition by the corresponding observation or is removed, if there is no such an observation. We will denote such a mapping by f: $ETS \longrightarrow OTS$. For an ETS S, this mapping is defined as follows. Every transition $s \xrightarrow{a} t$, where $a \in Dom(f)$, is replaced by a transition $s \xrightarrow{f(a)} t$, while every transition $s \xrightarrow{a} t$, where $a \in E \backslash Dom(f)$, is removed; transitions by buttons and τ transitions are preserved.

Testing an ETS implementation S is equivalent to testing an OTS implementation $f(S)$.

It is natural to extend the mapping f: $E \longrightarrow O$ to the traces of events f: $(E \cup B)^* \longrightarrow (O \cup B)^*$ and to the sets of traces such that f: $2^{(E \cup B)^*} \longrightarrow 2^{(O \cup B)^*}$. Let a trace of events be given by $\sigma = u_1, u_2, \ldots, u_n$, where $u_i \in E \cup B$. Then $\sigma \in Dom(f)$ if, for every $i = 1, \ldots, n$, either $u_i \in B$ or $u_i \in Dom(f)$. In this case, $f(\sigma) = v_1, v_2, \ldots, v_n$, where $v_i = u_i$ if $u_i \in B$ and $v_i = f(u_i)$ if $u_i \in E$. For a set $\sum$ of event traces, $f(\sum)$ is defined as the set of appropriate observation traces: $f(\sum) \triangleq \{f(\sigma) | \sigma \in Dom(f)\}$.

It is obvious that any ETS S satisfies the relation $tr(f(S)) = f(tr(S))$.

3. PARALLEL COMPOSITION

We define a parallel composition of ETSs in the spirit of communicating sequential processes (CSPs) [10]. In CSPs, the result of composition of two ATSs is an ATS whose states are given by pairs of states of operands, the initial state is given by a pair of initial states, and a transition is either asynchronous and corresponds to a transition in one of the operands (the state of only this operand is changed), or synchronous and corresponds to a pair of transitions by the same, synchronous, action in both operands (the states of both operands may be changed). In this case, the synchronism or asynchronism of transitions is uniquely defined by the actions by which these transitions are labeled; i.e., all external actions are classified under synchronous and asynchronous actions, whereas the internal action τ is always assumed to be asynchronous. We will also assume that destruction is always asynchronous. In either case, an action on the resulting transition coincides with an action on a single operand transition (asynchronous case) or on both operand transitions (synchronous case). After that, some[1] synchronous actions on transitions are "hidden" by the operator *hide*; i.e., they are replaced by the symbol τ.

For a composition of ETSs, we will always assume that buttons from B are synchronous, while an event γ is asynchronous. Other events from $E\backslash\{\gamma\}$ may be either synchronous or asynchronous, depending on the alphabet of synchronous events $U \subseteq E\backslash\{\gamma\}$. Moreover, an event on a synchronous transition of the composition does not necessarily coincide with events on operand transitions. We will assume that a partially defined composition of synchronous events is defined, which also results in a synchronous event $|_U| : U \times U \longrightarrow U$, $Dom(|_U|) \subseteq U \times U$. This composition should be commutative: if $(a, b) \in Dom(|_U|)$, then $(b, a) \in Dom(|_U|)$ and $a|_U|b = b|_U|a$.

Let us formally define a composition of ETSs $|_U|$: $ETS \times ETS \longrightarrow ETS$. For $S = Ets(V_S, E_S, s_0)$ and $T = Ets(V_T, E_T, t_0)$, we have $S|_U|T = Ets(V_S \times V_T, E, s_0, t_0)$, where E is a minimum set of transitions that is generated by the following inference rules: $\forall s, t, s', t', a, b, p$,

[1] The reason for which only a part of synchronous symbols can be hidden is the possibility of simulation of broadcasting: a transition in the composition of several ATSs corresponds to transitions in several (not necessarily two) operands.

1.1: $a \in (E\backslash U) \cup \{\tau\}$ & $s{-}a\longrightarrow s'$ $\vdash st{-}a\longrightarrow s't$;
1.2: $b \in (E\backslash U) \cup \{\tau\}$ & $t{-}b\longrightarrow t'$ $\vdash st{-}b\longrightarrow st'$;
1.3: $p \in B$ & $s{-}p\longrightarrow s'$ & $t{-}p\longrightarrow t'$ $\vdash st{-}p\longrightarrow s't'$;
1.4: $(a, b) \in Dom(|_U|)$ & $s{-}a\longrightarrow s'$ & $t{-}b\longrightarrow t'$ $\vdash st{-}ab\longrightarrow s't'$.

On the basis of the composition of events, we define a composition of traces of events $|_U|$: $(B \cup E)^* \times (B \cup E)^* \longrightarrow (B \cup E)^*$ as a minimum set of traces obtained by the following inference rules: $\forall \sigma, \sigma_1, \sigma_2 \in (B \cup E)^*$ and $\forall a, b, p$,

2.0: $\vdash \epsilon \in \epsilon|_U|\epsilon$;
2.1: $\sigma = \sigma_1|_U|\sigma_2$ & $a \in E\backslash U$ $\vdash \sigma \cdot a \in (\sigma_1 \cdot a)|_U|\sigma_2$;
2.2: $\sigma = \sigma_1|_U|\sigma_2$ & $a \in E\backslash U$ $\vdash \sigma \cdot b \in \sigma_1|_U|(\sigma_2 \cdot b)$;
2.3: $\sigma = \sigma_1|_U|\sigma_2$ & $p \in B$ $\vdash \sigma \cdot p \in (\sigma_1 \cdot p)|_U|(\sigma_2 \cdot p)$;
2.5: $\sigma = \sigma_1|_U|\sigma_2$ & $(a, b) \in Dom(|_U|)$ $\vdash \sigma \cdot (a|_U|b) \in (\sigma_1 \cdot a)|_U|(\sigma_2 \cdot b)$.

If $\sigma_1|_U|\sigma_2 = \varnothing$, we will say that these traces are not composed. The composition of traces of events is naturally extended to the composition of the sets of traces of events, which is defined as a union of the sets of all pairwise compositions:

$$\sum\nolimits_1 |_U| \sum\nolimits_2 \triangleq \cup\{\sigma_1|_U|\sigma_2 \mid \sigma_1 \in \sum \ \& \ \sigma_2 \in \sum\}.$$

This composition possesses the property of additivity: the relation $tr(S|_U|T) = tr(S)|_U|tr(T)$ holds for any two ETSs S and T.

The composition of ATSs and traces of actions is a particular case of the composition of ETSs and traces of events if we assume that $A \subseteq E$, $U \subseteq A\backslash\{\gamma\}$, $Dom(|_U|) = \{(a, a) | a \in U\}$ and, if $a \in U$, then $|_U|a = a$.

After a composition, some transitions by synchronous events are hidden by the operator *hide*, while transitions by buttons are not hidden. Denote the set of hidden symbols by $H \subseteq U$.[2]

For $S = Ets(V_S, E, s_0)$, formally $hide(S, H) = Ets(V_S, E, s_0)$, where E is a minimum set generated by the following inference rules: $\forall s, s'$, and a,

3.1: $a \notin H$ & $s - a \longrightarrow s'$ $\vdash s - a \longrightarrow s'$ and
3.2: $a \in H$ & $s - a \longrightarrow s'$ $\vdash s - \tau \longrightarrow s'$.

For a trace of events $\sigma \in E^*$, a trace $hide(\sigma, H)$ is defined that is obtained by the removal from σ of all events belonging to H: $\forall \sigma \in E^*$ and $\forall a \in E$,

4.0: $\vdash hide(\epsilon, H) = \epsilon$,
4.1. $a \notin H$ $\vdash hide(\sigma \cdot a, H) = hide(\sigma, H) \cdot a$,
4.2: $a \in H$ $\vdash hide(\sigma \cdot a, H) = hide(\sigma, H)$.

The hiding operator for traces is naturally extended to the sets of traces: for $H \subseteq U$ and the set of traces $\sum \subseteq E^*$, we have $hide(\sum, H) \triangleq \{hide(\sigma, H) | \sigma \in \sum\}$.

The hiding operator possesses the following important property: the relation $tr(hide(S, H)) = hide(tr(S), H)$ holds for any ETS S and any $H \subseteq U$.

Along with the property of additivity of the composition of ETSs, we have $hide(tr(S|_U|T), H)) = tr(hide(S|_U|T, H)) = hide(\cup(tr(S)|_U|tr(T)), H)$.

It is obvious that the hiding operator for ATSs and traces of actions is a particular case of the hiding operator for ETSs and traces of events.

4. COMPOSITION OF SPECIFICATIONS

In the general form, the problem of composition is the problem of matching the specification of a compo-

[2] Actually, the condition $H \subseteq U$ for the hiding operator is redundant; however, after a composition, the hiding of asynchronous events hardly makes sense. Moreover, after a composition, it suffices to hide only synchronous events from $Im(|_U|) \subseteq U$, because other synchronous events in the composition are certainly missing.

sition system to the specifications of its components. The specification of a system is said to be *correct* [3, 6] if a composition of implementations conformal to the specifications of the components is conformal to the specification of the system. This property is called preservation of conformance under composition [13, 14], or monotonicity of a composition [2, 3, 6, 15]. How can one check if a given specification of the system is correct? It is clear that if a specification does not impose any requirements on implementations, then any implementation is conformal to this specification, and, hence, this is a correct specification of any composition system. Therefore, of interest is to seek a correct specification of the system that would impose the most stringent requirements on implementations compared with all the other correct specifications of the system. This the "maximum correct" specification of the system is called a composition of specifications of the components [3, 6]. Thus, the main problem of composition is the construction of a composition of specifications.

In [1], a specification in the *B*/*O* semantics was defined as the set of finite traces of observations considered as errors (of the first kind). An implementation is conformal if it does not contain errors. More rigorously: an OTS implementation I is conformal to a specification S if $tr(I) \cap S = \varnothing$. Denote the set of OTS implementations conformal to the specification S by $Oconf(S)$.[3] Accordingly, an ETS implementation I is conformal to the specification S if $tr(f(I)) \cap S = \varnothing$. Denote the set of ETS implementations that are conformal to the specification S by $Econf(S)$. It is obvious that $f(Econf(S)) \subseteq Oconf(S)$.

For any LTS S (in particular, OTS and ETS), there exists a prefix relation between its traces: a set of LTS traces contains, together with each trace, all its prefixes. In addition, for any OTS and ETS, there is a transition by every button in each state, because the absence of a transition by a button in a state is interpreted as the presence of a loop transition by this button. This gives an additional relation between traces: if there is a trace σ, then there also exists a trace $\sigma \cdot p$ for every button p. The presence of these two trivial relations defines, in addition to errors of the first kind, which are directly defined by a specification, other errors—*nonconformal* traces, i.e., traces that are not encountered in conformal implementations. Errors that are not errors of the first kind are called errors of the second kind. *Primary* errors are errors all of whose strict prefixes are conformal. This allows one to either minimize or maximize specifications. The minimization procedure[4] constructs a set of primary errors: (1) adds an error σ if an error is given by the trace $\sigma \cdot p$, where p is a button and (2) removes from the specification every error some strict prefix of which is also an error. After the systematic application of these two rules to the specification S, one obtains a minimal specification, which we denote by $min(S)$. The maximization procedure constructs the set of all errors; it also systematically applies two rules: a rule similar to 1 and the inverse rule 2; i.e., it adds every extension of every error to the specification. For the specification S, we denote a maximal specification by $max(S)$.

Consider a composition system obtained as a result of composition of two components. Suppose given the specifications $S_1 \subseteq O^*$ and $S_2 \subseteq O^*$ of the components, a synchronous alphabet $U \subseteq E\backslash\{\gamma\}$, and a hiding set $H \subseteq U$. A formal definition of the correctness of the specification $S \subseteq O^*$ is as follows:

S *is correct* $\triangleq \forall I_1 \in Econf(S_1) \forall I_2 \in Econf(S_2)$ $hide(I_1|_U|I_2, H) \in Econf(S)$.

If operand implementations are taken from fixed subclasses of implementations I_1 and I_2, then this definition is appropriately changed:

S *is correct* $\triangleq \forall I_1 \in Econf(S_1) \cap I_1 \forall I_2 \in Econf(S_2) \cap$ I_2 $hide(I_1|_U|I_2, H) \in Econf(S)$.

A formal definition of a composition of specifications $S_1|_U|S_2 \subseteq O^*$ is as follows:

$S = S_1|_U|S_2 \triangleq S$ *is correct* $\& \forall S'$ (S' *is correct* $\Rightarrow$ $max(S') \subseteq max(S)$).

Note that a specification of a system that is correct on the classes of all implementations is obviously correct on any subclasses of implementations. However, a specification of a system that is correct on subclasses of implementations may be incorrect on the classes of all implementations and, hence, may impose more stringent requirements on the implementation of the system than those imposed by the composition of specifications on the classes of all implementations.

The following assertion is rather obvious: the set of observation traces that are not generated by the traces of events encountered in the compositions of conformal implementations is a composition of specifications. This set of traces is a complement of $(B \cup O)^*$ of the set of observation traces that are generated by the set of traces of events encountered in compositions of conformal implementations. The last set of event traces, as shown in Section 3, coincides with the set of pairwise compositions of event traces that are encountered in conformal implementations of the first and second operands, respectively. A trace of events is encountered in conformal implementations of the ith operand if and only if an observation trace generated by this event trace is conformal with respect to the specification S_i. Such traces of events are obtained from the set of conformal observation traces by using the function f^1, and this set of conformal observation traces is a complement of $(B \cup O)^*$ of the set of all errors of $max(S_i)$.

These arguments (when read in reverse order) give the following procedure for constructing a composi-

[3] In [1], where we considered only OTS implementations, we wrote C_S instead of $O(S)$.

[4] In [1], this procedure was called normalization.

tion of specifications on the class of all implementations.

1. Construct a set A_i of observation traces that are not encountered in conformal OTS implementations of the ith component: $A_i = max(S_i)$, where S_i is the specification of the ith component. Note that this set is postfix-closed (it contains, along with each trace, all its extensions).

2. Construct a set B_i of observation traces that are encountered in conformal OTS implementations of the ith component: $B_i = (B \cup O)^* \backslash A_i$. Note that this set is prefix-closed (it contains, along with each trace, all its prefixes).

3. Construct a set C_i of traces of events that are encountered in conformal ETS implementations of the ith component: $C_i = \cup\{f^1(\sigma) \,|\sigma \in B_i\}$.

4. Construct a set D of pairwise compositions of traces of events that are encountered in conformal ETS implementations of the first and second operands: $D = C_1|_U|C_2$.

5. Perform a hiding: $E = hide(D, H)$.

6. Construct a set F of observation traces that are generated by the traces of events encountered in compositions of conformal ETS implementations of operands: $F = f(E)$. Note that this set is prefix-closed.

7. Construct a set G of observation traces that are not generated by the traces of events encountered in compositions of conformal implementations: $G = (B \cup O)^* \backslash F$. Note that this set is postfix-closed.

The set G is a maximal composition of specifications; i.e., $G = max(S_1|_U|S_2)$. Further, it can be minimized in order that the required specification of the system contain only primary errors: $S = min(G)$.

Note that, if we consider subclasses of operand implementations I_1 and I_2, then the maximization and minimization procedures are changed, because they should be based on a nontrivial relation between errors that is available for these subclasses. This applies to item 1, where the maximization of a specification is performed, and to the minimization of the composition of specifications constructed. But even after this the composition of specifications may contain "redundant" errors, i.e., traces that are not encountered in the compositions of implementations from the classes I_1 and I_2. Of course, such errors do not affect the result of testing; however, the elimination of these errors could optimize the generation of tests: there is no need in the tests that detect "redundant" errors. In order to do this in the procedure of composition of specifications, one should change only items 2 and 7, in which a complement of the set of traces is constructed. In item 2, the set A_i should be completed to the set of observation traces that are encountered in implementations from I_i: $\cup\{tr(f(I))|I \in I_i\}$, rather than to the set of all observation traces $(B \cup O)^*$. In item 7, the set F should be completed to the set of observation traces that are encountered in the compositions of implementations from I_1 and I_2: $\cup\{tr(f(hide(I_1|_U|I_2, H))) \,|I_1 \in I_1 \,\&\, I_2 \in I_2\} = \cup\{f(hide(tr(I_1)|_U|(I_2), H)) \,|I_1 \in I_1 \,\&\, I_2 \in I_2\}$, rather than to the set of all observation traces $(B \cup O)^*$.

For practical application, the procedure described has a drawback that some sets of traces obtained are infinite. For example, if the specification S_i of a component is finite, then, after the maximization, the set $\max(S_i)$ is infinite (except for a degenerate case when the universes of buttons and observations are empty). At the same time, if the set of traces is regular, it can be represented in a finite form as a finite generating graph. Such a generating graph differs from the LTS $S = Lts(V_S, E_S, s_0)$ only in that[5] a subset of finite states $T_S \subseteq V_S$ is distinguished. We will denote this graph by S^g, and the set of traces generated by this graph, by $tr(S^g)$. An LTS is a generating graph all of whose states are finite.

It is known that any generating graph S^g can be made into a deterministic graph preserving the set of traces $tr(S^g)$ generated by this graph. This is made by the procedure of determinization, which constructs a new deterministic generating graph $d(S^g)$. The states of this graph are given by all nonempty sets of the states of the graph S^g, and the initial state is the set of states that can be reached from the initial state of the graph S^g by empty transitions. A set of states U is declared finite if at least one of its elements $u \in U$ is a finite state of the graph S^g. A transition $U \xrightarrow{a} U'$ is performed if and only if there exists an $u \in U$ such that there is a transition $u \xrightarrow{a} u'$ in the graph S^g and U is the set of ends u' of all such transitions.

Suppose that universes B, O, and E are finite. Let us rewrite our procedure for the case when the specifications of the components S_1 and S_2 are defined by finite generating graphs S_1^g and S_2^g: $tr(S_1^g) = S_1$ and $tr(S_2^g) = S_2$.

1. Construct a graph A_i^g that generates a set $A_i = max(S_i)$.

a. To this end, we declare that, in S_i^g, the initial state of each transition by a button that leads to a finite state is also finite.

b. Add a new finite set ω.

c. In each finite state (including ω), execute a transition to the state ω by all buttons and observations. Notice that this corresponds to the postfix-closedness of the error set.

2. Construct a graph B_i^g that generates a set $B_i = (B \cup O)^* \backslash A_i$.

a. To this end, we first apply the procedure of determinization; i.e., we construct a graph $d(A_i^g)$.

[5] Another difference is purely formal: in the generating graph, a τ transition corresponds to an empty transition (an unlabeled arc of the graph.

b. Add a new finite state ω' to $d(A_i^g)$, in which we execute loop transitions by all buttons and observations.

c. In each nonfinite state, we execute a transition to the state ω' by all buttons and observations by which there were no transitions from this state.

d. Remove all finite states together with the transitions that lead to these states.

e. All the remaining states are declared finite.

Notice that this corresponds to the prefix-closedness of the set of conformal observation traces.

3. Construct a graph C_i^g that generates a set $C_i = \cup\{f^1(\sigma) | \sigma \in B_i\}$. To this end, in B_i^g, we transform every transition by observation x from state s into a set of multiple transitions by every event a such that $x = f(a)$. If there are no such events a, then we just remove a transition by observation x.

4. Construct a graph D^g that generates a set $D = C_1|_U|C_2$. To this end, we apply the composition $D^g = C_1^g|_U|C_2^g$ described in Section 3.

5. Construct a graph E^g that generates a set $E = hide(D, H)$. To this end, we apply the hiding operation $E^g = hide(D^g, H)$ described in Section 3.

6. Construct a graph F^g that generates a set $F = f(E)$. To this end, in E^g, we transform every transition by event $a \in Dom(f)$ from the state s into a transition by observation $f(a)$. If $a \neq\in Dom(f)$, then we just remove a transition by such an event.

7. Construct a graph G^g that generates a set $G = (B \cup O)^*\backslash F$.

a. To this end, we first apply the procedure of determinization, i.e., construct a graph $d(F^g)$.

b. Add a new state ω' to $d(F^g)$, in which we execute loop transitions by all buttons and observations. This state, and only it, is declared finite.

c. In every state, we perform a transition to the state ω' by all buttons and observations by which there were no transitions from this state. Notice that this corresponds to the postfix-closedness of the error set. Note also that we can obtain states from which the state ω' is not reachable. Such states can be removed.

The graph G^g generates a maximal composition of specifications, i.e., $tr(G^g) = \max(S_1|_U|S_2)$. This graph can be minimized to obtain a graph S^g that generates a minimal composition of specifications (the set of primary errors), i.e., $tr(S^g) = \min(S_1|_U|S_2)$. To this end, we systematically apply the following two rules: (1) if a transition by button leads from a state s to a finite state s', then the state s is declared finite, and (2) a transition from a finite state is removed. At the end, one can remove the isolated states obtained.

5. SIMULATION OF OTHER SEMANTICS

In [1], we considered various testing semantics defined by appropriate testing machines and a simulation of each such semantics T_i in the B/O semantics, that is defined as an implementation transformation M_i: $ATS \longrightarrow OTS$ corresponding to this semantics. In the present paper, we simulate these semantics in the B/E semantics; i.e., we consider a transformation W: $ATS \longrightarrow ETS$, after which we can transform an ETS into an OTS by defining an appropriate function f, as described in Section 2.

In these semantics, an implementation is defined by an ATS, that is, by an LTS in which transitions are labeled by external actions from the universe A or by a symbol τ. A test action allows the implementation to execute external actions from a certain set of allowed actions; a destruction γ and an internal τ action are always allowed. A machine is said to be generative or reactive depending on whether the implementation can execute a sequence of allowed external actions or only one such action, after which the execution of external actions is blocked until the arrival of the next (maybe, the same) test action. If there are no allowed external actions and no τ activity in a current state of implementation, the implementation stops. This stop can be observed if a so-called green lamp is activated: it is on while the implementation executes some actions and is turned off when the implementation stops. After the stop, the testing either ends or can be continued by one or other test action. Finally, when the machine stops, one can observe actions transitions by which are defined at the current state of implementation. There are menu lamps corresponding to actions for this purpose. If such a lamp is activated, it is on at the time of stop if there is a transition by the appropriate action in the implementation. All these additional observations are classified in [8], and semantics with constraints on the set of allowed actions were considered in [2, 3, 6, 11, 12].

In this paper, we will consider all these possibilities as attributes of a button. This allows us to consider a wide class of semantics based on an ATS, including "mixed" semantics, when, for example, the green lamp is activated for some buttons and does not work for other buttons. Formally, we will assume that a subset of the universe of buttons $P \subseteq B$ is fixed for which the following functions are defined (which are defined everywhere on P):

• *acts*: $P \longrightarrow 2^A$, allowed external actions except for the always allowed destruction.

• *reactive*: $P \longrightarrow Bool$ if *true*, the button is reactive; i.e., the next button can be pressed only after the execution of an external action or after an observed stop of the implementation; if *false*, the button is generative, which does not impose restrictions on pressing the next button.

• *green*: $B \longrightarrow Bool$ if *true*, then one can observe the stop of implementation.

• *continue*: $P \longrightarrow Bool$ if *true*, then one can continue testing after the stop.

• *menu*: $P \longrightarrow 2^A$, activated menu lamps.

It is said that there are no priorities in the system if the nondeterministic choice rule prescribes the implementation to choose any action a for execution (either an external, $a \in A$, or an internal, $a = \tau$, action) that is defined in this implementation; in other words, there is a transition by a in the current state, and this transition is allowed by a pressed button p; i.e., $a \in acts(p) \cup \{\tau, \gamma\}$. In [1], we also considered the systems with priorities that were introduced in [4, 5], in which the feasibility of an action a depends on a pressed button $p \in P$, rather than only on the condition $a \in acts(p) \cup \{\tau, \gamma\}$. Such systems are simulated by ATSs in which a transition is labeled by a pair (a, p), where $p \in P$ and $a \in acts(p) \cup \{\tau, \gamma\}$, rather than by an action a. Such a transition is executed with an observation of action a only when a button p is pressed. It is clear that a system without priorities can be understood as a special case of the system with priorities if we replace each transition by an action a in this system by a set of multiple transitions by pairs (a, p), where p runs over the set $\{p \in P | a \in acts(p) \cup \{\tau, \gamma\}\}$. In what follows, we will consider only systems with priorities. For such systems, the menu lamp should correspond to a pair (action, button), rather than to an action. Now, the τ activity is also controlled by buttons; i.e., instead of τ transitions, we have (τ, p) transitions, where $p \in P$. Therefore, one should speak not of stability, but of p-stability of a state, which means the absence of transitions labeled by a button p, i.e., by a pair of the form (a, p), in the state. Accordingly, pairs of the form (τ, p) also correspond to menu lamps. Therefore, we assume that $menu$: $P \longrightarrow 2^{(A \cup \{\tau\}) \times P}$.

The table presents the semantics considered in [1–3, 6, 8, 11, 12] together with the buttons admissible for these semantics. We will consider all the actions from A as events: $A \subseteq E$. In addition, we introduce events of the form (r, p), where $r \in 2^{(A \cup \{\tau\}) \times P}$ is a ready set and $p \in P$ is a button. Denote the ready set in a state s by $r(s) = \{(a, p) | s \xrightarrow{a,p}\}$. Thus, we introduce the set of events $A \cup (2^{(A \cup \{\tau\}) \times P} \times P) \subseteq E$.

To simulate these semantics in the B/E semantics, we define an implementation transformation W: $ATS \longrightarrow ETS$. Just as in the transformation M: $ATS \longrightarrow OTS$, for every state s of the original ATS and every button $p \in P$, a new state s_p and a transition $s \xrightarrow{p} s_p$ are added. In every state s_p, a transition by an action $a \in acts(p) \cup \{\tau, \gamma\}$ is defined if there was a transition $s \xrightarrow{a,p} t$ in the state s. If $reactive(p) = true$ and $a \in acts(p)$, then the added transition also leads to the state t. If $reactive(p) = false$ or $a \in (\tau, \gamma)$, then the added transition leads to the state t_p. In every state s_p where $reactive(p) = false$, for every button $q \in P$, a transition $s_p \xrightarrow{a} s_q$ is defined. For every state s and every button $p \in P$, a transition from s_p by event (r, p) is added if s is p-stable, $green(p) = true$, and $r = r(s)$, where $r(s)$ is calculated by the original ATS. The condition of p-stability of the state s is as follows: $(\tau, p) \notin r(s)$ & $(\tau, p) \notin r(s)$. This transition leads either to the state s_p, if $continue(p) = true$ and $reactive(p) = false$, or to the state s, if $continue(p) = true$ and $reactive(p) = true$, or to some terminal state (if necessary, such a state is added), if $continue(p) = false$. After that, all (a, p) transitions are removed from the old states in which $a \in A$ and $a \neq \gamma$, and $(\tau, \varnothing)$ and $(\gamma, \varnothing)$ transitions are replaced by τ and γ transitions, respectively. The initial state remains the same; it is assumed that no button was pressed at the beginning of testing (all external actions except for destruction are prohibited), and the green lamp is not activated.

Now, we define the function f for events from $A \cup (2^{(A \cup \{\tau\}) \times P} \times P)$. For actions, we assume that $A \subseteq Dom(f)$ and $f(a) = a$ for every $a \in A$. For the event $(r, p) \in 2^{(A \cup \{\tau\}) \times P} \times P$, the observation condition, i.e., $(r, p) \in Dom(f)$, is as follows: $\{(a, x) \in r | x = p\} = \varnothing$. This is the condition of observable stopping of the machine in a state with a ready set r under a pressed button p. If $(r, p) \in Dom(f)$, then $f(r, p) = (r^+, r^-)$, where $r^+ = menu(p) \cap r$ and $r^- = (menu(p) \backslash r) \cup (acts(p) \times \{p\})$. Here r^+ is the set of pairs (a, p) for which menu lamps are activated and turned on. Regarding every pair $(a, p) \in r^+$ at the stopping time of implementation, we certainly know that there is a transition labeled by this pair and this transition can be executed when a button p is pressed. Accordingly, r^- is the set of all pairs (a, p), where p is a pressed button, as well as pairs (a, p) for which the menu lamps are activated but are not turned on. Regarding every pair $(a, p) \in r^-$ at the stopping time of implementation, we certainly know that there are no transitions labeled by this pair.

In the semantics that contain an event (r, p), an observation (r^+, r^-) can be interpreted in one or other way if the semantics (as a collection of buttons) imposes constraints on the possible values of the sets r^+ and r^-. For example, in the failure trace (FT) semantics, there are no menu lamps (they are not activated); therefore, for an event (r, p), we always have $r^+ = \varnothing$ and $r^- = acts(p)$, which is uniquely defined by the set $acts(p)$, which is called a refusal set at the stopping time of implementation under a pressed button p. In a semantics with finite sets of allowed actions and finite sets of activated menu lamps (RT^- and R^-), the sets r^+ and r^- are finite. In the completed trace semantics $T0$, only one type of refusal is possible, which arises in terminal states. This observation is denoted as 0, which coincides with a refusal set containing all external actions. An observation S arises for any observable stopping of implementation, i.e., precisely when an empty refusal is observed. In the R/Q semantics, observed refusals are elements of the set R, whereas, in the input output conformance (*ioco*) semantics, there is a single refusal, which is denoted by the symbol δ and implies the absence of transitions (from the set Y)

Table

Semantics in alphabet $A \subseteq A$		Admissible buttons from P					Full observation (r^+, r^-)	Interpretation of observation in semantics
		reactive =	*acts* $\in$	*green* =	*menu* $\in$	*continue* =		
trace	T	× (*false*)	2^A	*false*	×	×	–	–
	T^-	× (*false*)	$\{A\}$	*false*	×	×		
failure trace	FT	× (*false*)	2^A	*true*	$\{\varnothing\}$	*true*	$r^+ = \varnothing$ $r^- = acts(p) \times \{p\}$	refusal *acts*(p)
failure	F	× (*false*)	2^A	*true*	$\{\varnothing\}$	*false*		
failure trace	FT^-	× (*false*)	2^{A-}	*true*	$\{\varnothing\}$	*true*		
failure	F^-	× (*false*)	2^{A-}	*true*	$\{\varnothing\}$	*false*		
ready trace	RT	× (*false*)	2^A	*true*	$\{(A \cup \{\tau\}) \times P\}$	*true*	$r^+ = r$ $r^- = (A \times P)\backslash r$	ready r
readiness	R	× (*false*)	2^A	*true*	$\{(A \cup \{\tau\}) \times P\}$	*false*		
ready trace	RT^-	× (*false*)	2^{A-}	*true*	$Fin((A \cup \{\tau\}) \times P)$	*true*	r^+, r^-	(r^+, r^-)
readiness	R^-	× (*false*)	2^{A-}	*true*	$Fin((A \cup \{\tau\}) \times P)$	*false*		
completed traces	$T0$	× (*false*)	$\{A\}$	*true*	$\{\varnothing\}$	×	$r^+ = \varnothing$ $r^- = A \times \{p\}$	refusal $0 = A = acts(p)$
stability	$T0S$	× (*false*)	$\{A\}$	*true*	$\{\varnothing\}$	×		
			$\{\varnothing\}$			*true*	$r^+ = \varnothing$ $r^- = \varnothing \times \{p\} = \varnothing$	refusal $S = \varnothing = acts(p)$
$R \cap Q = \varnothing$ and $\cup R \cup\cup Q = A$	R/Q	× (*true*)	R	*true*	$\{\varnothing\}$	*true*	$r^+ = \varnothing$ $r^- = acts(p) \times \{p\}$	refusal *acts*(p)
			Q	*false*	×	×	–	–
$X \cap Y = \varnothing$, $X \cup Y = A$ and $\delta = Y$	*ioco*	× (*true*)	$\{\delta\}$	*true*	$\{\varnothing\}$	*true*	$r^+ = \varnothing$ $r^- = \delta \times \{p\} = \{(\delta, p)\}$	refusal $\delta = Y = acts(p)$
			$\{\{x\} \mid x \in X\}$	*false*	×	×	–	–

Note: *reactive* = *false* in the generative machine of van Glabbeek, and *reactive* = *true* in the reactive R/Q machine and in the *ioco* machine. For a set Z, $Fin(Z)$ denotes a family of *finite* subsets of the set Z.

by reactions in the implementation at the time of stopping.

For a specific semantics T_i, define a transformation D_i: $ETS \longrightarrow ETS$, which removes all transitions by buttons that are not admissible in this semantics. Taking into account the above-described interpretation of a general observation (r^+, r^-) in one or other semantics T_i, we can easily see that the simulation M_i described in [1] gives the same result as some general simulation M: $ATS \longrightarrow OTS$, after which transitions by buttons that are not admissible in the semantics A_i are removed, i.e., the transformation D_i is applied. This universal transformation M differs from the transformation W only in that, instead of an event (r, p), a transition is labeled by an observation $f(r, p) = (r^+, r^-)$. It is easily seen that observation traces are preserved under simulation: in an ATS S, the set of observation traces and buttons corresponding to the semantics T_i is equal to $tr(f(D_i(W(S)))) = tr(D_i(M(S)))$.

It remains to define a composition of the events we have introduced. Only external actions can be asynchronous; a destruction is always asynchronous, while events of the form (r, p) are always synchronous. After the composition, we will hide all synchronous transitions by actions and only them. Thus, $H \subseteq A \backslash \{\gamma\}$ and $U = H \cup (2^{(A \cup \{\tau\}) \times P} \times P)$.

The definition of a composition is based on the following basic properties of the ready set. If the states s and t of two ATS operands are p-stable, then the composition state st is p-stable if and only if there is no pair of synchronous transitions, i.e., transitions by the same button q and the same synchronous action a, in the states s and t a: $\{(a, q) \in r_1 \cap r_2 | a \in H\} = \varnothing$. This yields the following definition of the domain of composition of events:

$Dom(|_U|) = \{(a, a) \mid a \in A\} \cup \{((r_1, p), (r_2, p)) \mid r_1, r_2 \in 2^{(A \cup \{\tau\}) \times P}\ \&\ p \in P\ \&\ \{(a, q) \in r_1 \cap r_2 | a \in H\} = \varnothing\}$.

If a composition state st is p-stable, then its ready set is calculated by the ready sets of the operand states; namely, it contains a pair (a, q) if and only if this pair is contained in the ready set of at least one of the operand states and the action a is asynchronous: $r(st) = \{(a, q) \in r(s) \cup r(t) | a \in H\}$. This yields the following definition of the value of a composition of events:

$$\forall (a, a) \in Dom(|_U|) \qquad a|_U|a = a,$$
$$\forall ((r_1, p), (r_2, p)) \in Dom(|_U|) \qquad (r_1, p)|_U|(r_2, p) = (\{(a, q) \in r_1 \cup r_2 | a \notin H\}, p).$$

Now, based on the definitions of a composition of an ATS and an ETS, we can easily see that the property of agreement between composition and simulation W: $ATS \longrightarrow ETS$ is satisfied: for any ATSs S and T, the set of event traces of an ETS that simulates a composition of these ATSs coincides with the set of event traces of the composition of ETSs that simulate these ATSs: $tr(W(S|_U|T)) = tr(W(S)|_U|W(T))$.

REFERENCES

1. Bourdonov, I.B. and Kossatchev, A.S., Formalization of a test experiment – II, *Program. Comput. Software*, 2013, vol. 39, no. 4, pp. 163–181.

2. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Formalization of test experiments, *Program. Comput. Software*, 2007, vol. 33, no. 5, pp. 239–260.

3. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., *Teoriya sootvetstviya dlya sistem s blokirovkami i razrusheniem (*Conformance Theory for Systems with Blockings and Destruction), Moscow: Nauka, 2008.

4. Bourdonov, I.B. and Kossatchev, A.S., Systems with priorities: Conformance, testing, and composition, *Tr. Inst. Syst. Program.*, 2008, no. 14.1.

5. Bourdonov, I.B. and Kossatchev, A.S., Systems with priorities: Conformance, testing, and composition, *Program. Comput. Software*, vol. 35, no. 4, pp. 198–211].

6. Bourdonov, I.B., *Teoriya konformnosti (Funktsional'noe testirovanie programmnykh system na osnove formal'nykh modelei)* (Conformance Theory: Functional Testing of Software Systems on the Basis of Formal Models), Saarbrucken: LAP LAMBERT Academic Publ., 2011.

7. 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.*, Springer, 1990, vol. 458, pp. 278–297.

8. 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.*, Springer, 1993, vol. 715, pp. 66–81.

9. Milner, R., Modal characterization of observable machine behavior, *Proceedings CAAP 81*, Astesiano, G. and Bohm, C., Eds., *Lect. Notes Comput. Sci.*, Springer, 1981, vol. 112, pp. 25–34.

10. Hoare, C.A.R., Communicating sequential processes, in *On the Construction of Programs—An Advanced Course,* McKeag, R.M. and Macnaghten, A.M., Eds., Cambridge: Cambridge Univ. Press, 1980, pp. 229–254.

11. Tretmans, J., Conformance testing with labelled transition systems: Implementation relations and test generation, *Comput. Networks ISDN Syst.*, 1996, vol. 29, no. 1, pp. 49–79.

12. Tretmans, J., Test generation with inputs, outputs and repetitive quiescence, in *Software-Concepts and Tools*, 1996, vol. 17, issue 3.

13. van der Bijl, M., Rensink, A., and Tretmans, J., Compositional testing with ioco, *Formal Approaches to Software Testing, Third International Workshop, FATES 2003*, Montreal, Quebec, Canada, October 6th, 2003, Petrenko, A. and Ulrich, A., Eds., *Lect. Notes Comput. Sci.*, Springer, 1981, vol. 2931, pp. 86–100.

14. van der Bijl, M., Rensink, A., and Tretmans, J., Component Based Testing with ioco, *CTIT Technical Report TR-CTIT-03-34*, University of Twente, 2003.

15. Jard, C., Jeron, T., Tanguy, L., and Viho, C., Remote testing can be as powerful as local testing, *Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII/ PSTV XIX' 99*, Beijing, China, Wu, J., Chanson, S., and Gao, Q., Eds., October 1999, pp. 25–40.

Translated by I. Nikitin