ISSN 0361-7688, Programming and Computer Software, 2010, Vol. 36, No. 5, pp. 247–263. © Pleiades Publishing, Ltd., 2010.
Original Russian Text © I.B. Bourdonov, A.S. Kossatchev, 2010, published in Programmirovanie, 2010, Vol. 36, No. 5.

Interaction Semantics with Refusals, Divergence, and Destruction

I. B. Bourdonov and A. S. Kossatchev

Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia

e-mail: kos@ispras.ru

Received December 17, 2009

Abstract—Formal methods for testing the conformance of a software system to its specification are considered. The interaction semantics determines the testing capabilities, which are reduced to the observation of actions and refusals (absence of actions). The semantics is parameterized by the families of observable and unobservable refusals. The concept of destruction as a prohibited action that should be avoided in the course of interaction is introduced. The concept of safe testing, the implementation safety hypothesis, safe conformance, and generation of a complete test suite based on the specification are defined. Equivalences of traces, specifications, safety relations, and interaction semantics are examined. A specification completion is proposed that can be used to remove from the specification irrelevant (not included in the safely testable implementations) and nonconformal specification traces is proposed. The concept of total testing that detects all the errors in the implementation (rather than at least one error as is the case in complete testing) is introduced. On the basis of the analysis of dependences between errors, a method for the minimization of test suites is proposed. The problem of preserving the conformance under composition (the monotonicity of conformance) is investigated, and a monotone transformation of the specification solving this problem is proposed.

DOI: 10.1134/S0361768810050014

1. INTERACTION SEMANTICS AND SAFE TESTING

This paper is devoted to the problems and methods concerning the verification of "correctness" of software systems, where correctness is interpreted as the conformance of the system to the given requirements. In the model world, the system is mapped to its implementation model (implementation), the requirements are mapped to a specification model (specification), and their conformance is mapped to a binary conformance relation. A specification is always given, and the existence of an implementation (as a model of a real-life system) is assumed (this is the testing hypothesis). If the implementation is also given explicitly, the conformance can be verified analytically. For an implementation whose structure is unknown (a black box) or that is too complicated to analyze, one has to use testing as a means of the verification of conformance in the course of test experiments. It is clear that, in this case, the requirements must be functional; that is, they must be represented in terms of the interaction of the system with its environment, which is replaced by a test suite in the course of testing. For that reason, the conformance relation and its testing are based on certain interaction semantics.

The interaction semantics formalizes the available set of testing capabilities for control and observation of the behavior of the system under test. In the course of testing, we can only observe the behavior of the implementation that is, first, induced by a test input (control) and, second, can be observed in an external interaction. Such an interaction can be simulated using the so-called testing machine [1–5]. This machine is a black box containing the implementation (see Fig. 1). The control is reduced to the following. An operator performs a test (which is interpreted as an instruction for the operator) by pressing buttons on the machine's keyboard thus enabling the implementation to perform certain actions that can be observed. Observations on the machine's display are classified into two types. An observation of an external *action* that is enabled by the operator and is performed by the implementation, and an observation of a *refusal*, which means that no actions enabled by the buttons pressed by the operator are observed.

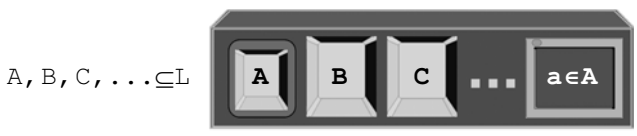


Fig. 1. Testing machine.

We emphasize that the operator allows the implementation to perform a set of actions (not necessarily only a single action). We assume that the operator may press only one button simultaneously. Each button is assigned an individual set of enabled 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 on the machine's keyboard and by the set of buttons for which a refusal can be observed. Thus, the interaction semantics is determined by the alphabet of external actions $\mathbf{L}$ and by two sets of the testing machine buttons—those for which the corresponding refusals are observable (the family $\mathbf{R} \subseteq 2^{\mathbf{L}}$) and not observable (the family $\mathbf{Q} \subseteq 2^{\mathbf{L}}$). We assume that $\mathbf{R} \cap \mathbf{Q} = \varnothing$ and $\cup\mathbf{R} \cup \cup\mathbf{Q} = \mathbf{L}$. Such kind of semantics is called the $\mathbf{R}/\mathbf{Q}$ semantics. If $\mathbf{Q} = \varnothing$ and $\mathbf{R} = 2^{\mathbf{L}}$, this is the well-known *failure trace semantics* [3, 4, 6–9]. Another example is the semantics of the popular *ioco* relation [9, 10] when the actions are divided into inputs (stimuli) and outputs (responses): $\mathbf{L} = \mathbf{I} \cup \mathbf{U}$. There is only one button for receiving all the outputs $\mathbf{R} = \{\delta\}$, where $\delta = U$, and the corresponding refusal δ is called *quiescence*. Every input x is sent to the implementation using the $\mathbf{Q}$-button $\{x\}$ so that $\mathbf{Q} = \{\{x\} | x \in \mathbf{I}\}$.

In addition to the external actions, the implementation can execute internal (unobservable) actions, which are denoted by τ. These actions are always enabled (when any button is pressed or no buttons are pressed at all).

For an action (external or internal) 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 (that is, if any action satisfying these conditions can be chosen for execution), the system is said to have no priorities. In this paper, we consider only the systems without priorities.

It is assumed that any finite sequence of arbitrary actions (both external and internal) terminates in a finite amount of time and an infinite sequence of actions terminates in an infinite amount of time. We also assume that a test action (pressing a button) is passed from the testing machine to the implementation in a finite amount of time; furthermore, an observation is passed from the implementation to the machine's display in a finite amount of time. This guarantees that the time passed between pressing the button enabling the external action and the observation of this action is finite.

A stronger form of this assumption is often used to implement the observation of the $\mathbf{R}$-refusal; namely, it is assumed that the time taken by the execution of every action enabled by a button along with all the corresponding preceding internal actions is not only finite but also bounded. In this case, a timeout is introduced; if this time has expired without the observation of the action, this is interpreted as a refusal. Note that this is only one of possible methods to implement the observation of a refusal.

After pressing an $\mathbf{R}$-button, the operator either observes an external action enabled by this button in a finite amount of time or observes the corresponding refusal. However, assume that a refusal is possible in the implementation and a $\mathbf{Q}$-button is pressed. Since this refusal is not observable, the operator does not know whether he should wait for the observation of an external action or there is no such action because there was a refusal. Therefore, the operator cannot continue testing, nor can he finish it.

An infinite sequence of τ-actions (an infinite loop) is called *divergence*; it is denoted by Δ. The divergence per se is not harmful, but, when the operator presses any ($\mathbf{R}$- or $\mathbf{Q}$-) button to escape the divergence, he does not know whether an observation (of an external action or an $\mathbf{R}$-refusal) should be expected or the implementation will endlessly continue its internal activity. Therefore, the operator cannot continue testing, nor can he finish it.

We also define a special action called *destruction* that is not controlled by the buttons; this action is denoted by γ. It models any undesirable behavior of the system including its actual destruction, which cannot be allowed in the interaction (for example the self-destruction button in military systems). Destruction is one possible way to interpret the unspecified behavior. The exclusion from the consideration of the implementation destroying interactions is often justified by the requirement for the implementations to check the correctness of the input parameters [9, 11]. If the parameters are incorrect, the implementation either ignores the call or reports an error. This requirement is quite natural if this is a public system; indeed, such a system must be fool-tolerant. However, in the case of the interaction between internal components of restricted access subsystems, the mutual correctness verification is redundant. Such a situation often occurs when the parameters of a complex structure are handled and the correctness conditions are not trivial; then, the overhead of correctness verification unreasonably increases the development effort, the size of the system, and the execution time. In this case, an alternative is a strict specification of operation preconditions [12]. For example, an attempt to free a memory chunk that was not earlier allocated is a violation of the corresponding precondition. It is the correctness of the access of components to each other that must verified rather than the behavior of the components in response to incorrect calls from other components. In other words, since we want to verify the correctness of the implementation rather than the correctness of the environment, we are not interested in

the behavior of the implementation when it is handled incorrectly. The destruction marks the situations in the specification in which an arbitrary behavior of the implementation is admitted including the actual destruction. The semantics of destruction assumes that it must not occur when the behavior of the environment is correct.

Testing under which no destruction, attempts to exit from divergence, and unobservable refusals occur is said to be *safe*.

2. MODELS OF IMPLEMENTATIONS AND SPECIFICATIONS

For an interaction based on observations, the only result of a test experiment is an alternating sequence of buttons (test inputs) and observations, which will be called (*test*) *history*. Due to the semantics of divergence and destruction, it suffices to consider only the histories in which the symbols Δ and γ either never appear or are the last symbols in the history. Since the divergence and destruction are always enabled, they are not preceded by any button in the history. Any other observation u (an external action or **R**-refusal) is enabled by the button P that immediately precedes it; that is, $u \in P$ or $u = P$ for $P \in \mathbf{R}$. A subsequence of a history consisting of observations (including Δ and γ) is called a trace.

For systems without priorities, only traces are important because the possibility or impossibility of the appearance of an observation after a trace depends only on whether or not the given button enables this observation, and it is independent of the other observations enabled by this button. For a given system under test, the set of its histories can be uniquely reconstructed from the set of its traces. Therefore, the *trace model* considered as the set of traces that can be observed when working with the system is the most natural model of this system.

If the traces may include as refusals arbitrary subsets of the alphabet of external actions **L**, then such traces and such a trace model are said to be complete. The corresponding $2^{\mathbf{L}}/\varnothing$-semantics is well known in the literature under the name *failure trace semantics*, and complete traces without divergence and destruction are called *failure traces*. A trace in which all the refusals are observable for the given **R**/**Q**-semantics (i.e., belong to **R**) is called an **R**-trace, and the subset of **R**-traces of the complete trace model is called the **R**-model. Under safe testing, only the **R**-traces that do not contain Δ and γ can be observed. For the *ioco* semantics, the **R**-traces (without Δ and γ) are called *suspension traces*. The other traces in the complete trace model are needed to indicate which **R**-traces are safe (can be observed under safe testing) and which are not. For that purpose, it suffices to take into account only the traces in which the refusals belong to the set $\mathbf{R} \cup \mathbf{Q}$.

The interaction semantics prohibits some traces in the trace model. For example, the trace must be *consistent*; namely, an action $z \in P$ as well as the symbols Δ and γ cannot follow the refusal P. Therefore, we can consider as a model only a set of traces that satisfies certain necessary and sufficient conditions. A formal definition of such conditions and a proof of their mutual independence, necessity, and sufficiency can be found in [2].

Another equivalent model of implementations and specifications is a labeled transition system (LTS), which is defined as a directed graph in which the vertices are called states and the arcs are labeled by external actions or the symbol τ or γ; the arcs are called transitions. The transition from the state s to the state s' by the symbol z is denoted by $s \xrightarrow{z} s'$. In this graph, an initial state (vertex) is distinguished; the implementation starts executing from the initial state after each restart. Having been restarted, the implementation performs a sequence of adjacent transitions; that is, the implementation moves along a route that begins at the initial state, and each transition in this route is labeled by an action z that is enabled by the currently pressed button P of the testing machine; that is, $z \in P \cup \{\tau, \gamma\}$.

In an LTS, a refusal P is induced in a *stable* state with no outgoing τ- and γ-transitions provided that there are not outgoing transitions by the actions $z \in P$ from this state. A state is *divergent* if there is an infinite τ-route starting at this state; this is a route in which all the transitions are labeled by the symbol τ.

In order to define traces in an LTS, we add in every stable state virtual loops corresponding to the induced refusals, and transitions by Δ are added in the divergent states. Then, a trace in the LTS is defined as a sequence of labels on the transitions of the route starting at the initial state that does not extend beyond a Δ- or γ-transition and in which the symbols τ are omitted. The LTS is the most illustrative model because the set of traces in any LTS is the trace model.

However, the LTS model has a considerable inconvenience because it is nondeterministic. The nondeterminism in the LTS manifests itself as the existence of τ-transitions and (or) a fan of transitions $s \xrightarrow{z} s'$ leading from the state s to different states s' by the same external action z. Due to this fact, a trace in the LTS generally ends in a set of states rather than in a single state. The authors of this paper proposed one more model that does not have this inconvenience. It is called the refusal transition system (RTS). It is a deterministic LTS in the alphabet $\mathbf{L} \cup \mathbf{R} \cup \{\tau, \Delta, \gamma\}$ rather than in the alphabet $\mathbf{L} \cup \{\tau, \gamma\}$; that is, such a model may include explicit transitions by **R**-refusals, and the infinite chain of τ-transitions $s \xrightarrow{\tau} \ldots$ is replaced by a

single transition $s \xrightarrow{\Delta}$. A downside of this model is the lack of clearness; indeed, not every deterministic LTS in such an alphabet is a model. In order for a deterministic LTS in such an alphabet to be a model, it must satisfy a special set of conditions determined by the interaction semantics. When an LTS is transformed into an RTS, the states of the RTS are the sets of the LTS states that end the traces; this is similar to the known "determinization" algorithm of a generating automaton (or graph).

The class of models in the alphabet **L** is denoted by *MODEL*(**L**); here, by the model we mean an LTS or RTS trace model depending on the context.

3. SAFETY HYPOTHESIS AND SAFE CONFORMANCE

How is it possible to perform safe testing if the implementation is not known? For example, if a transition by destruction is determined in the initial state, such an implementation cannot be tested; moreover, it should not be executed at all because it can break down even before the first test input, that is, before any button has been pressed. The solution is to restrict oneself to the implementations that can be safely tested to verify their conformance to the given specification. This restriction of the class of implementations that can be tested is formulated in the form of the safety hypothesis; conformance is defined only for the implementations that satisfy this hypothesis. Due to the equivalence of the trace, LTS, and RTS models, it suffices to formulate the safety hypothesis and safe conformance only for trace models of the implementation and specification. Such a safety hypothesis and safe conformance can be called *trace* ones because they depend only on the implementation and specification traces but not on their states and state correspondence.

First, safe testing assumes a formal definition on the level of the model of the safety relation "the button P is safe in the model M after the trace σ." In the course of safe testing, only safe buttons are pressed. This relation is different for the implementation and specification models. In a complete trace implementation **I**, the safety relation (*safe in*) assumes that the button P after the trace σ is *nondestructive*—pressing this button does not entail an attempt of exiting from the divergence (there is no divergence after the trace) and does not entail destruction (after an action enabled by the button). Moreover, pressing a button cannot lead to an unobservable refusal (if this is a **Q**-button): $\forall P \in \mathbf{R} \cup \mathbf{Q}\ \forall \sigma \in \mathbf{I}$, we have

$$P\ safe_{\gamma\Delta}\ \mathbf{I}\ after\ \sigma$$
$$=_{def} \sigma \cdot \langle \Delta \rangle \notin \mathbf{I}\ \&\ \forall u \in P\sigma \cdot \langle u, \gamma \rangle \notin \mathbf{I}.$$
$$P\ safe\ in\ \mathbf{I}\ after\ \sigma =_{def} P\ safe_{\gamma\Delta}\ \mathbf{I}\ after\ \sigma$$
$$\&\ (P \in \mathbf{Q} \Rightarrow \sigma \cdot \langle P \rangle \notin \mathbf{I}).$$

In a complete trace specification **S**, the safety relation (*safe by*) is different only for the **Q**-buttons. We do not require that there is no **Q**-refusal Q after the trace σ; however, we require that that there is at least one action $z \in Q$. In addition, if an action is enabled by at least one nondestructive button, it must be enabled by a safe button. If this is a nondestructive **R**-button, it is also safe. However, if all the nondestructive buttons that enable an action are **Q**-buttons, at least one of them must be declared safe. Such a safety relation always exists—it is sufficient to declare safe every nondestructive button that enables an action extending the trace. However, these requirements do not uniquely determine the relation *safe by*; for that reason, a specific relation is indicated when the specification **S** is determined. The requirements for the relation *safe by* are as follows: $\forall R \in \mathbf{R}\ \forall z \in \mathbf{L}\ \forall Q \in \mathbf{Q}\ \forall \sigma \in \mathbf{S}$

$$R\ safe\ by\ \mathbf{S}\ after\ \sigma \Leftrightarrow R\ safe_{\gamma\Delta}\ \mathbf{S}\ after\ \sigma,$$
$$\exists P \in \mathbf{R} \cup \mathbf{Q}\ P\ safe_{\gamma\Delta}\ \mathbf{S}\ after\ \sigma\ \&\ z \in P$$
$$\&\ \sigma \cdot \langle z \rangle \in \mathbf{S} \Rightarrow \exists P' \in \mathbf{R} \cup \mathbf{Q}\ z \in P'$$
$$\&\ P'\ safe\ by\ \mathbf{S}\ after\ \sigma,$$
$$Q\ safe\ by\ \mathbf{S}\ after\ \sigma \Rightarrow Q\ safe_{\gamma\Delta}\ \mathbf{S}\ after\ \sigma$$
$$\&\ \exists v \in Q\ \sigma \cdot \langle v \rangle \in \mathbf{S}.$$

The safety of buttons determines the safety of observations. An **R**-refusal R is safe if the button R is safe after the trace. An action z is safe if it is enabled by a button that is safe after the trace:

$$z\ safe\ in\ \mathbf{I}\ after\ \sigma =_{def} \exists P \in \mathbf{R} \cup \mathbf{Q}\ z \in P$$
$$\&\ P\ safe\ in\ \mathbf{I}\ after\ \sigma.$$
$$z\ safe\ by\ \mathbf{S}\ after\ \sigma =_{def} \exists P \in \mathbf{R} \cup \mathbf{Q}\ z \in P$$
$$\&\ P\ safe\ by\ \mathbf{S}\ after\ \sigma.$$

Now, we can define *safe* **R**-*traces*. An **R**-trace σ is safe if it belongs to the model and the following two conditions are fulfilled. (1) The model does not break down at the very beginning immediately after the machine is started (before the first button is pressed); that is, the model does not include the trace $\langle \gamma \rangle$. (2) Every symbol in the trace is safe immediately after the preceding trace prefix:

$$\langle \gamma \rangle \notin \mathbf{I}\ \&\ \forall \mu\ \forall u\ (\mu \cdot \langle u \rangle \le \sigma$$
$$\Rightarrow u\ safe\ in\ \mathbf{I}\ after\ \mu),$$
$$\langle \gamma \rangle \notin \mathbf{S}\ \&\ \forall \mu\ \forall u(\mu \cdot \langle u \rangle \le \sigma$$
$$\Rightarrow u\ safe\ by\ \mathbf{S}\ after\ \mu).$$

The sets of safe traces of the implementation **I** and the specification **S** are denoted by *SafeIn*(**I**) and *SafeBy*(**S**), respectively.

Note that the empty **Q**-button is unsafe both after any safe (with respect to the relation *safe in*) trace of any implementation and after any safe (with respect to any relation *safe by*) trace of any specification. Such a button may never be pressed in the course of safe testing. For that reason, we everywhere assume that the semantics does not contain such a button— $\varnothing \notin \mathbf{Q}$. At the same time, the empty **R**-button is meaningful. Indeed, it is safe after any safe trace that is not

extended with the divergence, and the observation of the empty **R**-refusal indicates that the implementation turned out to be in a stable state.

It is seen from the definition of the safety relation *safe by* that the set *SafeBy*(**S**) of the safe traces of the specification **S** is uniquely determined by the set of its **R**-traces. Furthermore, it is seen from the definition of the safety relation *safe in* that the set *SafeIn*(**I**) of the safe traces of the implementation **I** depends, in addition to the set of its **R**-traces, on the extension of the **R**-traces with **Q**-refusals in the set **I**.

The requirement for testing safety defines the class of *safe* implementations *SafeImp*(**R**, **S**, *safe by*) that can be safely tested to check their conformance or nonconformance to the given specification **S** with the given relation *safe by* in the given **R**/**Q**-semantics. This class is defined by the following *safety hypothesis*. An implementation **I** is *safely testable* for the specification **S** if the following holds. (1) The implementation does not break down 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:

$$\mathbf{I}\ \textit{safe for}\ \mathbf{S} =_{def} (\langle\gamma\rangle \notin \mathbf{S} \Rightarrow \langle\gamma\rangle \notin \mathbf{I})$$
$$\& \ \forall\sigma \in \textit{SafeBy}(\mathbf{S}) \cap \mathbf{I}\ \forall P \in \mathbf{R} \cup \mathbf{Q}$$
$$(P\ \textit{safe by}\ \mathbf{S}\ \textit{after}\ \sigma \Rightarrow P\ \textit{safe in}\ \mathbf{I}\ \textit{after}\ \sigma).$$

Note that, for the *ioco*-semantics, we allow safe input refusals in safely testable implementations; these are input refusals after the traces that are not extended with these inputs in the specification. This requirement is less strict than the requirement that the implementation must be defined everywhere with respect to inputs, which is proposed by *ioco*'s author Tretmans in [10]. Thus, we remove the inconsistency of the relation *ioco*, which manifests itself as follows. Suppose that we have an implementation defined everywhere with respect to inputs and an implementation differing from the former one only in that it contains "safe" refusals that are indistinguishable under *ioco*-testing. However, the former implementation can be conformable and the latter one is certainly nonconformable because it is not defined everywhere with respect to inputs and, therefore, does not belong to the *ioco* domain.

Now, we can define the *safe conformance* relation. An implementation **I** is *safely conformable* (or simply *conformable*) to the specification **S** if it is safe and the following testable condition is fulfilled: any observation possible in the implementation in response to pressing a safe (in the specification) button is enabled by the specification:

$$\mathbf{I}\ \textit{saco}\ \mathbf{S} =_{def} \mathbf{I}\ \textit{safe for}\ \mathbf{S}\ \&\ \forall\sigma \in \textit{SafeBy}(\mathbf{S}) \cap \mathbf{I}$$
$$\forall P\ \textit{safe by}\ \mathbf{S}\ \textit{after}\ \sigma\ \textit{obs}(\sigma, P, \mathbf{I}) \subseteq \textit{obs}(\sigma, P, \mathbf{S});$$

here $\textit{obs}(\sigma, P, M) =_{def} \{u | \sigma \cdot \langle u\rangle \in M\ \&\ (u \in P \vee u = P\ \&\ P \in \mathbf{R})\}$ is the set of observations over the complete trace model M that can be obtained by pressing the button P after the trace σ.

This relation determines the class of conformable implementations *ConfImp*(**R**/**Q**, **S**, *safe by*).

Note that the safety hypothesis cannot be checked in the course of testing; this hypothesis is the testing precondition, while testing checks the testable conformance condition.

4. TEST SUITE GENERATION

In terms of the testing machine, a test is an instruction for the machine's operator. Each item of this instruction specifies a button that must be pressed by the operator and, for each observation that is possible after pressing this button, it includes the instruction item that must be executed next, or the verdict *pass* or *fail* (if the testing must be stopped) is indicated. In the test, only an observation u is allowed after the button P that is enabled by P; that is, $u \in P \vee u = P \in \mathbf{R}$.

A test can be interpreted as a prefix-closed set of finite histories in which (1) every maximal history terminates in an observation and is assigned a verdict; (2) every maximal history that terminates in a button can be extended in this set only with the observations that are enabled by this button, and it is inevitably extended with the observations that can occur in the safely testable implementations after a subtrace of this history.

A test is safe if and only if, in every its history, each button is safe in the specification after the subtrace of the history prefix that immediately precedes this button. In other words, a test is safe if and only if the subtraces of all its histories are test traces, where a *test trace* is an empty safe trace or a trace $\sigma \cdot \langle u\rangle$, where σ is safe in the specification and the observation u is safe in the specification **S** after σ (but it does not necessarily extends σ in **S**): $\sigma \in \textit{SafeBy}(\mathbf{S})$ & u *safe by* **S** *after* σ. It is clear that it suffices to consider only the test traces that can appear in implementation models. Such traces must satisfy the conditions imposed on the traces in the trace model. In particular, if a safe trace σ ends with a refusal P, then any action $u \in P$ is safe after σ; however, the trace $\sigma \cdot \langle u\rangle$ is nonconformable and it cannot appear in the model.

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 implementation passes it. A test suite is *complete* if it is both significant and exhaustive. To determine whether a safely testable implementation is conformable, a complete test suite should be generated on the basis of the given specification.

A complete test suite always exists; in particular, the set of all *primitive* tests is complete [2]. A primitive test is constructed on the basis of a distinguished nonmax-

imal (with respect to the relation ≤) safe **R**-trace of the specification. To this end, before every refusal R, the button R is included, and before every action z, an arbitrary safe (after the corresponding trace prefix) button P is included that enables the action z; after the trace, an arbitrary button P' that is safe after this trace is included. The safety of the trace guarantees the safety of the button R and the existence, for every action z, of a safe button P that enables this action; the nonmaximality of the safe trace guarantees the existence of the final button P'. The buttons P and P' are not unique. Generally, given the same safe trace, several different primitive tests can be generated. However, the sets of tests generated from different traces do not overlap.

If an observation after pressing a button extends the trace, the test continues (a nonmaximal history in the test). An observation extending the trace (that is, an observation obtained after the final button has been pressed) and any observation that branches off the trace always finish the testing (a maximal history in the test). The verdict *pass* is assigned if the resulting **R**-trace (a subtrace of the maximal history) belongs to the specification; the verdict *fail* is assigned if it does not belong to the specification. 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 (considered as a set of histories) is a union of a set of primitive tests; that is, these tests detect the same bugs. Therefore, we may consider only primitive tests.

We have already mentioned that an implementation may at any time 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 our testing observation and control capabilities; that is, the nondeterminism depends on the interaction semantics. In other words, the behavior of the implementation is nondeterministic because depends on certain "weather conditions" 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, that is, if all the possible weather conditions can be reproduced in an infinite sequence of test runs, the testing is said to be *global* [5]. 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 arbitrary behavior of the operator. Otherwise, we cannot be sure that we have executed every test for all possible weather conditions (i.e., for any possible nondeterministic behavior of the implementation).

If the global testing hypothesis is valid, then, in order for a complete test suite could be executed under all possible weather conditions, this test suite must be enumerable. Then, the tests are executed in the order of this enumeration so that every test is run an infinite number of times in the "diagonal" process of enumerating the tests and their runs. Since all the tests are finite, each of them terminates in a finite amount of time; then, the system is restarted, and the next test or one of the earlier executed tests is run. If the implementation is nonconformal, the completeness of the test suite guarantees that a bug will be detected in a finite amount of time. However, the conformance of the implementation cannot be ascertained in a finite amount of time if the test suite is infinite or the global testing requires an infinite number of test runs. This is the problem of practical testing.

Note that a sequence of test runs alternating with restarts can be considered as a single (unified) test in which restart can be used along with test inputs. Instead of the finiteness of each test in the test suite, we can require the finiteness of every segment of the unified test between the restarts; that is, we can require that the unified test does not contain an infinite postfix without restarts. In addition, we assume everywhere that the system restart is always correct (the system is always reset to its initial state) and, therefore, the restart needs not to be tested. Thus, the difference between an enumerable test suite and a test with restarts is conditional, and it is determined by the convenience of the organization of the testing system.

5. EQUIVALENCE OF TRACES, SPECIFICATIONS, SAFETY RELATIONS, AND INTERACTION SEMANTICS

The safety hypothesis and conformance are specified by the triple consisting of an interaction semantics $\mathbf{R}/\mathbf{Q}$, a specification model $\mathbf{S}$, and a safety relation *safe by*. Two such triples $(\mathbf{R}_1/\mathbf{Q}_1, \mathbf{S}_1, \textit{safe by}_1)$ and $(\mathbf{R}_2/\mathbf{Q}_2, \mathbf{S}_2, \textit{safe by}_2)$ in the same alphabet $\mathbf{L} = \cup\mathbf{R}_1 \cup\cup \mathbf{Q}_1 = \cup\mathbf{R}_2 \cup\cup \mathbf{Q}_2$ can be considered to be equivalent if they determine identical classes of safely testable implementations $\textit{SafeImp}(\mathbf{R}_1/\mathbf{Q}_1, \mathbf{S}_1, \textit{safe by}_1) = \textit{SafeImp}(\mathbf{R}_2/\mathbf{Q}_2, \mathbf{S}_2, \textit{safe by}_2)$ and identical classes of conformal implementations $\textit{ConfImp}(\mathbf{R}_1/\mathbf{Q}_1, \mathbf{S}_1, \textit{safe by}_1) = \textit{ConfImp}(\mathbf{R}_2/\mathbf{Q}_2, \mathbf{S}_2, \textit{safe by}_2)$. Respectively, a transformation of semantics, specification model, and (or) safety relation is said to be an *equivalent transformation* if its result is a triple that is equivalent to the original triple.

The requirements for the *safe by* relation uniquely determine the safety of the **R**-buttons, but they leave considerable freedom for the declaration of safe and unsafe **Q**-buttons. If the semantics and the specifica-

tion model are fixed, we can consider equivalent *safe by* safety relations as the relations that determine the same classes of safely testable and conformal implementations. The question of when the class of conformal implementations changes and when it remains the same under a transformation of the *safe by* safety relation that preserves only the class of safely testable implementations is a subject for a specific study.

In some case, one can speak of the "inconsistency" of the relation *safe by* in the following sense: although a **Q**-button is declared to be safe after a trace of the specification and is unsafe after another trace, this button is safe with respect to the relation *safe in* after these two traces in any implementation. Such a situation occurs if these two traces terminate in the same set of states in any LTS implementation. In turn, this occurs if and only if the traces are *equivalent* in the following sense: at the corresponding places in these traces are the same external actions or the same sequences of refusals with the same set of refused external actions. These sequences of refusals are associated with the stable states that do not include transitions by all the actions belonging to some refusals in this sequence. This enables us to define a *normal safe by* relation that determines the same safe buttons after equivalent traces. Any *safe by* relation can be *normalized* if we declare to be safe after every given trace the buttons (and only those buttons) that are declared to be safe by the original relations after *some* trace that is equivalent to the given trace. Normalization is an equivalent transformation.

Another useful restriction that can be imposed on the *safe by* relation is the requirement that the sets of buttons that are safe after the traces having identical trace extensions are identical. If the traces in the LTS model terminate in the same set of states, they have identical extensions; however, the converse is generally not true. Nevertheless, for LTS specifications, one can restrict oneself by the *safe by* relation under which it is required that the sets of the buttons that are safe after the traces terminating in the same sets of states (for the RTS specifications, this is a single state) are identical. Such a *safe by* relation is said to be *2-normal*. In the general case, for a fixed semantics and specification model, a 2-normal *safe by* relation that is equivalent to the original relation does not necessarily exist. However, this does not decrease the specification capacity; indeed, if we fix only the semantics, then any specification model with any *safe by* relation can be equivalently transformed into another model with another, 2-normal, relation *safe by*. The question of when such a transformation preserves the finiteness of the specification model remains open, while this question is important for practical applications. 2-Normal relations *safe by* are convenient for practical applications because, for specifications with a finite set of states, the family of sets of states after the traces is also finite, and the 2-normal relation *safe by* is defined for such sets.

Testing significantly depends on the available set of test capabilities concerning the control and observation of the system under test, which are formalized in the interaction semantics. For that reason, the comparison of different semantics from the viewpoint of testing capabilities is of interest.

An $\mathbf{R}_1/\mathbf{Q}_1$-semantics is said to be not stronger than an $\mathbf{R}_2/\mathbf{Q}_2$-semantics in the same alphabet **L** (this fact is denoted by $\mathbf{R}_1/\mathbf{Q}_1 \le \mathbf{R}_2/\mathbf{Q}_2$) if, for every specification model **S** with the relation *safe by*$_1$ in the $\mathbf{R}_1/\mathbf{Q}_1$-semantics, there is a relation *safe by*$_2$ in the $\mathbf{R}_2/\mathbf{Q}_2$-semantics with the same classes of safe and conformal implementations, that is, if the triples ($\mathbf{R}_1/\mathbf{Q}_1$, $\mathbf{S}_1$, *safe by*$_1$) and ($\mathbf{R}_2/\mathbf{Q}_2$, $\mathbf{S}_2$, *safe by*$_2$) are equivalent. Two semantics are equivalent if none of them is stronger than the other. It was proved in [13] that the relation *not stronger* is a preorder (i.e., it is reflexive and transitive). Therefore, the equivalence of semantics is reflexive, transitive, and symmetric. There is a necessary and sufficient condition for the validity of the relation $\mathbf{R}_1/\mathbf{Q}_1 \le \mathbf{R}_2/\mathbf{Q}_2$. This condition is as follows: (1) Every $\mathbf{Q}_1$-button also is a $\mathbf{Q}_2$-button; (2) every $\mathbf{Q}_2$-button can be represented as the union of an $\mathbf{R}_1$- and $\mathbf{Q}_1$-buttons; (3) every $\mathbf{R}_1$-button can be represented as the union of a finite number of $\mathbf{R}_2$-buttons and, conversely, every $\mathbf{R}_2$-button can be represented as the union of a finite number of $\mathbf{R}_1$-buttons. Two semantics are equivalent if the families of their **Q**-buttons are identical and every **R**-button of the one semantics can be represented as the union of a finite number of **R**-buttons of the other semantics.

In connection with the concept of equivalence, it is interesting to consider the equivalent semantics that are minimal and smallest in terms of the embedding of the families of buttons. A button in **R** is said to be *finitely decomposable* if it can be represented as the union of a finite number of buttons from **R** different from it (otherwise, the button is said to be *finitely indecomposable*). It is known that if, for a given **R**/**Q**-semantics, there is an equivalent minimal $\mathbf{R}_0/\mathbf{Q}$-semantics, then the family $\mathbf{R}_0$ coincides with the set of all finitely indecomposable buttons in **R**. For the existence of such an equivalent minimal $\mathbf{R}_0/\mathbf{Q}$-semantics, it is necessary and sufficient that any finitely decomposable **R**-button is decomposable into a union of a finite number of finitely indecomposable **R**-buttons. If the number of infinite **R**-buttons is finite (in particular, if the family **R** is finite), a minimal equivalent semantics exists. If a minimal equivalent semantics exists, it is also the smallest one.

The relation *not stronger* for semantics can be generalized if the specification model is not fixed but it is only required that both the testing capacity and the

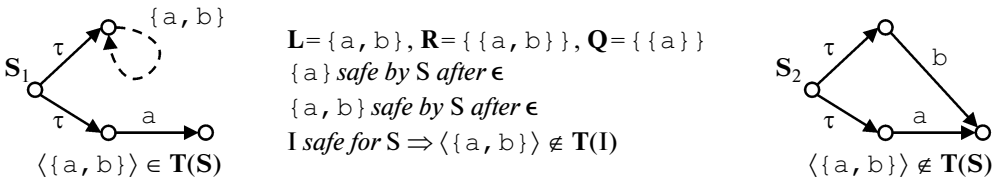


Fig. 2. Irrelevant safe and testable traces specification traces.

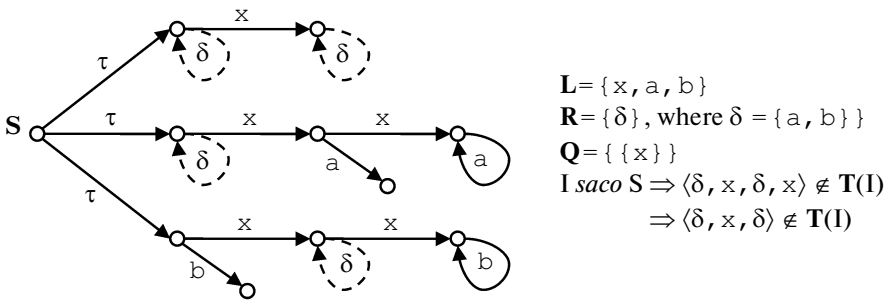


Fig. 3. A nonconformal safe specification trace.

specification capacity be preserved: an $\mathbf{R}_1/\mathbf{Q}_1$-semantics is *not stronger* than an $\mathbf{R}_2/\mathbf{Q}_2$-semantics in the same alphabet $\mathbf{L}$ if, for every specification model $\mathbf{S}_1$ with the *safe by*$_1$ relation in the $\mathbf{R}_1/\mathbf{Q}_1$-semantics, there exists a specification model $\mathbf{S}_2$ with the *safe by*$_2$ relation in the $\mathbf{R}_2/\mathbf{Q}_2$-semantics with the same classes of safe and conformal implementations. The equivalence of semantics is generalized in a similar fashion.

A further generalization enables us to relate by equivalence and by a *not stronger* relation semantics in different alphabets $\mathbf{L}_1$ and $\mathbf{L}_2$. It is natural to consider only the implementations that are defined in the intersection of the alphabets $\mathbf{L} = \mathbf{L}_1 \cap \mathbf{L}_2$. Here, we use the fact that any LTS implementation in the alphabet $\mathbf{L}$ can be considered as an LTS implementation in a larger alphabet $\mathbf{L}_i$ ($i = 1, 2$). The extended implementation just has no transitions by the actions belonging to the sets $\mathbf{L}_i \backslash \mathbf{L}$. For the $\mathbf{R}_i/\mathbf{Q}_i$-semantics, the implementation is considered in the alphabet $\mathbf{L}_i$, and their $\mathbf{R}_i$ traces are used.

Such extended relations between semantics are a subject for future studies; in particular, the case when the transformation of the specification model preserves it finiteness is of interest. A special case of semantics equivalence is considered below when the extension of the specification is discussed in Section 7.

6. IRRELEVANT AND NONCONFORMAL SPECIFICATION TRACES

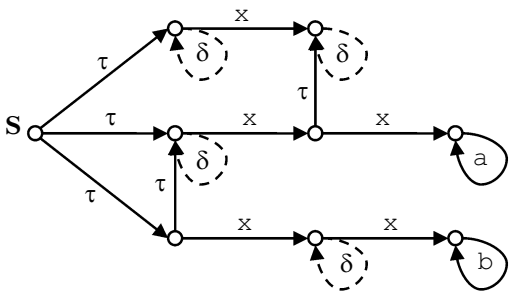
A specification trace is said to be relevant if it appears in safely testable implementations. It was proved in [2] that, if a semantics does not include **Q**-buttons, then the specification satisfies its own safety hypothesis. Therefore, all the safe specification traces for such semantics are relevant. In the presence of **Q**-buttons, not every safe specification trace is relevant in the general case.

An example is shown in Fig. 2. In that figure, the button $\{a\}$ is safe after the empty trace in the LTS specification $\mathbf{S}_1$ depicted on the left; the trace $\langle\{a, b\}\rangle$ is also safe. According to the safety hypothesis, the safely testable implementation cannot have a **Q**-refusal $\{a\}$ after the empty trace. However, if an implementation contains the trace $\langle\{a, b\}\rangle$, then at least one state has an **R**-refusal $\{a, b\}$ after the empty trace; consequently, this state also includes the **Q**-refusal $\{a\}$. Therefore, safely testable implementations cannot include the trace $\langle\{a, b\}\rangle$ although it is a safe specification trace.

It is clear that it is sufficient to generate test only for relevant safe specification traces, which enables us to optimize test generation.

In the same way, a test trace that is not included in the specification can be irrelevant. For one thing, all the inconsistent test traces are irrelevant. But consistent test traces can also be irrelevant. In the LTS specification $\mathbf{S}_2$ shown in Fig. 2 on the right, the test trace $\langle\{a, b\}\rangle$ is not included in the specification, is consistent but irrelevant. The test histories involving such irrelevant test traces are removed from every test generated on the basis of a relevant safe trace.

A trace that appears in conformal implementations is said to be *conformal*. It was shown in [2] that, if the semantics does not include **Q**-buttons, then the specification is testable and safe on its own account. Therefore, all the safe specification traces are confor-



L = {x, a, b}
R = {δ}, where δ = {a, b}
Q = {{x}}
I *saco* S ⇒ ⟨δ, x, δ, x⟩ ∉ **T**(I)
⇒ ⟨δ, x, δ⟩ ∉ **T**(I)
⇒ ⟨δ, x⟩ ∉ **T**(I)
⇒ ⟨δ⟩ ∉ **T**(I)
⇒ ϵ ∉ **T**(I)

Fig. 4. A specification that has no conformal traces.

mal for such semantics. In the presence of **Q**-buttons, not every safe specification trace is generally conformal.

Figure 3 reproduces an example from [1]. Here, the semantics of the *ioco* relation is used. This relation assumes that the implementation is defined everywhere with respect to inputs; that is, it assumes that the implementation does not include **Q**-refusals. The relation *saco* is less stringent; however, even for this relation, the button $\{x\}$ in the LTS specification **S** must be safe after the trace $\langle x\rangle$ according to the second rule of the relation *safe by*. If an implementation includes the trace $\langle \delta, x, \delta\rangle$, then it also includes the trace $\langle x\rangle$ after which the button $\{x\}$ must be safe with respect to *safe in* by the safety hypothesis. Hence the button $\{x\}$ is safe with respect to *safe in* after the trace $\langle \delta, x, \delta\rangle$. Since this is a **Q**-button, its safety after the trace $\langle \delta, x, \delta\rangle$ implies that the implementation includes the trace $\langle \delta, x, \delta, x\rangle$. Since δ is an **R**-button and the specification does not include destruction, this button is safe after any safe specification trace; in particular, it is safe after the trace $\langle x, x\rangle$; therefore, it is safe in the implementation with respect to *safe in* after the trace $\langle x, x\rangle$; hence, it is also safe with respect to *safe in* after the trace $\langle \delta, x, \delta, x\rangle$. Consequently, the implementation must include at least one of the traces $\langle \delta, x, \delta, x, a\rangle$, $\langle \delta, x, \delta, x, b\rangle$, or $\langle \delta, x, \delta, x, \delta\rangle$. Then, the implementation contains at least one of the traces $\langle x, \delta, x, a\rangle$, $\langle \delta, x, x, b\rangle$, or $\langle x, x, \delta\rangle$. Each of these traces is an extension of a safe trace by an observation enabled by the safe button δ; however, this observation is not included in the specification, which contradicts its conformance. Therefore, the trace $\langle \delta, x, \delta\rangle$ is safe in the specification and nonconformal.

The existence on nonconformal safe traces in the specification is not a problem per se, but it enables one to optimize testing. An ordinary *ioco* test terminates after the trace $\langle \delta, x, \delta\rangle$ with the verdict *pass* (it is redundant to press the button δ for the second time). The modified test that recognizes nonconformal traces terminates with the verdict *fail* in this case, which makes it possible to detect nonconformance faster. Such recognition of nonconformal traces can significantly reduce the size of the generated tests.

Figure 4 is a modification of Fig. 3. In order to eliminate the refusal δ in the states of interest, instead of the first transitions by the actions *a* and *b*, we use τ-transitions. As before, the conformal implementation cannot include the trace $\langle \delta, x, \delta\rangle$. The button δ is safe after the trace $\langle \delta, x\rangle$, but the actions *a* and *b* cannot occur. Therefore, a nonconformal implementation cannot include the trace $\langle \delta, x\rangle$. If a conformal implementation contained the trace $\langle \delta\rangle$, it also had to include the nonconformal trace $\langle \delta, x\rangle$ because $\{x\}$ is a **Q**-button, which is safe after the trace $\langle \delta\rangle$. Hence, a conformal implementation cannot include the trace $\langle \delta\rangle$. At the same time, it cannot have the actions *a* and *b* after the empty trace. Therefore, a conformal implementation cannot include the empty trace. Hence, the empty trace is also nonconformal and, consequently, all the traces in the specification are nonconformal.

This somewhat strange example shows that the analysis of nonconformal traces can be very useful. In this example, we have an infinite set of safely testable implementations but there are no conformal implementations. The complete testing without additional assumptions or testing capability is infinite because there is a cycle by the action *a* after the safe trace $\langle \delta, x, x\rangle$; therefore, there is an infinite set of safe traces of the form $\langle \delta, x, x, a, \ldots, a\rangle$ and relevant but erroneous traces of the form $\langle \delta, x, x, a, \ldots, a, b\rangle$ or $\langle \delta, x, x, a, \ldots, a, \delta\rangle$. Furthermore, the testing that recognizes nonconformal traces will be not performed at all because it will be discovered that all the specification traces are nonconformal.

7. COMPLETION OF SPECIFICATION

The cause by which a specification can contain safe irrelevant or nonconformal traces is the difference in the definition of safety of **Q**-buttons in the relations *safe in* and *safe by*. Moreover, due to this difference, a specification in which there are **Q**-refusals after safe traces does not satisfy its own safety hypothesis; therefore, it is nonconformal. For example, in the *ioco* semantics, any specification in which certain traces are extended both with the input *x* and with the refusal $\{x\}$ do not satisfy their own safety hypothesis *safe for*

(much less does is it satisfy the requirement to be defined everywhere with respect to inputs); hence, such a specification is nonconformal. This implies that the *saco* relation (and, in particular, *ioco*) is generally not reflexive. In the general case, this relation is also not transitive [2]. The nonreflexivity of a specification and the presence of irrelevant and nonconformal safe traces in it are counterintuitive. Indeed, if we "copy" the implementation from such a specification, we certainly obtain an incorrect implementation, which sometimes cannot be safely tested.

At the same time, if all the refusals are observable (there are no **Q**-buttons), the relation *saco* is reflexive and transitive; that is, it is a preorder [2]. In this case, the specification is conformal in itself; therefore, it cannot contain irrelevant or nonconformal safe traces. This suggests the idea to move from the **R**/**Q**-semantics to the $\mathbf{R} \cup \mathbf{Q}/\varnothing$-semantics in which all the refusals are observable. The transition is performed using a transformation of the specification called *completion*. In the completed specification, the safety relation *safe by* = *safe in* is defined. The completed specification contains no **Q**-refusals.

The completed specification is equivalent to the original one in the **R**/**Q**-semantics; in particular, the classes of safely testable and conformal implementations are preserved. At the same time, if the completed specification is considered in the $\mathbf{R} \cup \mathbf{Q}/\varnothing$-semantics, the class of conformal implementations is preserved while the class of safely testable implementations is not restricted and, therefore, the possibility to test the implementations that could be tested against the original specification is retained; however, this class can become larger, which makes it possible to test additional implementations.

It is clear that, in order to test an implementation against the completed specification in the $\mathbf{R} \cup \mathbf{Q}/\varnothing$-semantics rather than in the original **R**/**Q**-semantics, the corresponding testing capabilities must be available; namely, all the refusals must be observable. If this is not the case, the **R**/**Q**-semantics is used for testing as before, and the usefulness of the completion in this case is that the irrelevant and nonconformal traces are removed from the specification (or, at least, they are somehow labeled).

The completion of the specification also is the first step in solving the problem of conformance monotonicity, which is discussed in Section 9.

More precisely, the completed specification cannot, in the general case, be constructed in the same **R**/**Q**-semantics; it is constructed in the equivalent $\mathbf{R}^{\#}/\mathbf{Q}^{\#}$-semantics in the extended alphabet $\mathbf{L}^{\#}$. The idea is to add, for each refusal $P \in \mathbf{R} \cup \mathbf{Q}$, add a special dummy action *nonrefusal* P denoted by $\not P$ to the alphabet of the external actions and to the refusal P itself. The new refusal is denoted by $P^{\#} = P \cup \{\not P\}$. It is assumed that $P \neq P' \Rightarrow \not P \neq \not P'$. The specification model **S** is transformed into the specification model $\mathbf{S}^{\#}$ in which the new safety relation *safe by* = *safe in* is defined. As a result, the triples (**R**/**Q**, **S**, *safe by*) and ($\mathbf{R}^{\#}/\mathbf{Q}^{\#}$, $\mathbf{S}^{\#}$, *safe in*) are equivalent on the class of implementations in the original alphabet $\mathbf{L} = \mathbf{L} \cap \mathbf{L}^{\#}$—they determined identical classes of safely testable implementations $SafeImp(\mathbf{R}/\mathbf{Q}, \mathbf{S}, safe\ by) = SafeImp(\mathbf{R}^{\#}/\mathbf{Q}^{\#}, \mathbf{S}^{\#}, safe\ in) \cap MODEL(\mathbf{L})$ and identical classes of conformal implementations $ConfImp(\mathbf{R}/\mathbf{Q}, \mathbf{S}, safe\ by) = ConfImp(\mathbf{R}^{\#}/\mathbf{Q}^{\#}, \mathbf{S}^{\#}, safe\ in) \cap MODEL(\mathbf{L})$. Thus, the **R**/**Q**- and $\mathbf{R}^{\#}/\mathbf{Q}^{\#}$-semantics are equivalent (on the intersection of the alphabets **L**).

A generic algorithm for completing specifications is described in [2]. Its idea dates to the method of determinization of a generating automaton (or graph) but has some significant differences. In the trace model, $\mathbf{S}^{\#}$ is defined as the closure of the set of *final* traces $\mathbf{S}_{final}$ with respect to the operation d of the removal of a refusal from the trace. A trace σ is final if the operations of removing refusals d, permutation of adjacent refusals t, and repetition of refusals r can transform this trace into a trace σ' that is safe in the original specification **S**. Such a trace σ' is said to be a *drt*-subtrace of σ. Furthermore, every final trace σ is extended with a nonrefusal $\not P$ if the refusal P does not appear in the postfix of refusals of σ and is further extended with the divergence Δ if the button P is safe after a certain *drt*-subtrace of σ or with the destruction γ otherwise.

For finite alphabets, the completion transformation can be algorithmized. One can construct an LTS completion $\mathbf{S}_{final}$ in which the states are final traces without repeated refusals (in any subsequence of refusals). However, such an LTS completion and the corresponding RTS completion are generally infinite even if the original specification LTS model **S** is finite, which is unacceptable in practice. A solution is to use the 2-normal relation *safe by*. In this case, one can construct an RTS completion $\mathbf{S}_{final}$ using as a state, instead of the final trace σ not containing nonrefusals, the family of sets of states terminating the *drt*-subtraces of σ that are safe in **S** plus the set of actions prohibited by the postfix of refusals of the trace σ (the union of refusals of this postfix). For the LTS **S** with a finite number of states, the number of such states in the RTS $\mathbf{S}_{final}$ is also finite.

In the resulting RTS completion $\mathbf{S}_{final}$, all the irrelevant and nonconformal traces can be recognized.

The state s of the RTS $\mathbf{S}_{final}$ is irrelevant if, for a certain **Q**-button Q, every action $z \in Q$ belongs to an **R**-refusal $z \in R$ by which there is a loop transition $s \xrightarrow{R} s$ in s and the transition by the nonrefusal $\not Q$, if it exists, is extended with the divergence: $s \xrightarrow{\not Q} \xrightarrow{\Delta}$. This implies that the button $Q^{\#}$ is safe after a certain

trace σ that ends at the state s and has in its postfix of refusals all such **R**-refusals R. However, the only possible observation is the nonrefusal $\not Q$, which does not exist in the implementation in the alphabet **L**. Therefore, if such an implementation contains the trace σ, then, when the button Q is pressed after the this trace, an unobservable refusal occurs; hence this implementation is not safely testable.

Removing from the RTS $\mathbf{S}_{final}$ all the irrelevant states and the transitions ending at them, we obtain an equivalent specification $\mathbf{S}_{rel}$ in which all the traces are relevant.

For example, for the specification $\mathbf{S}_1$ shown in Fig. 2, the irrelevant safe trace $\langle\{a, b\}\rangle$ disappears after completion, but it remains an irrelevant test trace. No tests will be generated on the basis of this trace; furthermore, in any test that begins with pressing the button $\{a, b\}$, the observation of $\{a, b\}$ is removed from the set of possible observations; only the conformal observation of a (the verdict *pass* or the continuation of testing) and the nonconformal observation of b (the verdict *fail*) remain.

The state s is *nonconformal* if, for a certain **R**- or **Q**-button P, the only possible observation is the nondestructive nonrefusal $\not P$, that is, if $s \xrightarrow{\not P} \xrightarrow{\Delta}$, there are no transitions $s \xrightarrow{z}$ for $z \in P$, and there is no transition $s \xrightarrow{P}$ for $P \in \mathbf{R}$. The trace ending in a nonconformal state is nonconformal because, if a safe button P is pressed after this trace, no conformal observations are possible in the **R**/**Q**-semantics.

Upon all the nonconformal states and all the transitions leading into them are removed from $\mathbf{S}_{rel}$, we can obtain new nonconformal states, and so on. If the RTS $\mathbf{S}_{rel}$ is finite, it is clear that this process terminates in a finite number of steps, and we obtain an equivalent specification $\mathbf{S}_{conf}$ in which all the traces are conformal.

For example, for the specification illustrated in Fig. 3, the safe nonconformal trace $\langle\delta, x, \delta\rangle$ disappears after completion, but it remains a test trace after which the verdict *fail* rather than *pass* (as is the case for the original specification) is assigned.

Note that the completed specification can still contain traces that are not included in the original specification. For example, if we replace in the specification depicted in Fig. 3 the transitions by the action b with the transitions by the action a, then the trace $\langle\delta, x, \delta\rangle$ becomes conformal; however, the completion also contains the new traces $\langle\delta, x, \delta, x\rangle$ and $\langle\delta, x, \delta, x, a\rangle$. After these traces, the bugs $\langle\delta, x, \delta, x, b\rangle$, $\langle\delta, x, \delta, x, \delta\rangle$, $\langle\delta, x, \delta, x, a, b\rangle$, and $\langle\delta, x, \delta, x, a, b\rangle$ can appear. However, the presence or absence of tests generated on the basis of these traces and detecting these bugs does not affect the testing completeness. Therefore, the newly created traces can be labeled so as to avoid the generation of tests based on them. Such an optimization can be performed in the RTS $\mathbf{S}_{conf}$ by labeling certain transitions (in the example presented in Fig. 3, the transition by x after the trace $\langle\delta, x, \delta\rangle$ is labeled).

8. COMPLETE AND TOTAL TESTING AND DEPENDENCES BETWEEN BUGS

The goal of complete testing is to verify the conformance or nonconformance of an implementation. If the implementation is nonconformal, it suffices to detect at least one bug. However, testing is only one stage in the lifecycle of the target system development [14]. Testing is typically followed by bug fixes (in the implementation and, sometimes, also in the specification) and retesting. For that reason, in order to reduce the number of iterations of the lifecycle, it is desirable that as many as possible bugs be detected in the course of testing; it is also desirable to detect situations that are free of bugs. The testing that detects all the bugs in the implementation and the corresponding test suite are said to be *total*. Complete testing is performed "up to the first bug," and total testing is performed until all the bugs are detected. It is clear that the total testing is complete, but the converse is generally not true.

The fact that specifications can contain safe nonconformal traces enables us to subdivide bugs into three kinds. A *bug of the first kind* is the extension of a safe conformal trace with a safe observation not included in the specification. A *bug of the second kind* is a safe nonconformal trace in the specification. A *bug of the third kind* is the extension of a safe nonconformal trace with a safe observation not included in the specification.

Note that the kind of a bug is not invariant under equivalent transformations of the specification. Moreover, the set of relevant test traces can be not preserved under equivalent transformations; namely, some relevant bugs determined by a specification can be not test traces (and, therefore, bugs) for another equivalent specification.

For instance, in the example depicted in Fig. 3, the trace $\langle\delta, x, \delta\rangle$ in the original specification **S** formally is a bug of the second kind; however, since the nonconformal traces are not labeled in the original specification, tests cannot detect this bug. In the specifications $\mathbf{S}_{final}$ or $\mathbf{S}_{rel}$, in which the nonconformal traces are labeled, this is a bug of the second kind, while this is a bug of the first kind in the specification $\mathbf{S}_{conf}$. In the original specification **S** and in $\mathbf{S}_{conf}$, the trace $\langle\delta, x, \delta, x, a\rangle$ is not an error because these specifications do not contain its prefix $\langle\delta, x, \delta, x\rangle$; however, this is a bug of the third kind in the specifications $\mathbf{S}_{final}$ and $\mathbf{S}_{rel}$ with nonconformal traces already detected and labeled.

For that reason, claiming that the total testing must detect all the bugs, we must interpret bugs as the traces that are errors for some specification in the class of

equivalent specifications. The best way to detect all such bugs is to construct the "largest" equivalent specification that could determine all those bugs; this is the equivalent specification with the largest set of relevant test traces. To that end, in distinction from the complete testing, we should enhance the completion $\mathbf{S}_{rel}$ rather than reduce it to $\mathbf{S}_{conf}$ by adding the lacking relevant traces but preserving the labeling of nonconformal traces. This yields the specification $\mathbf{S}_{total}$ that determines all the bugs.

To optimize the test suite in the case of complete or total testing, we must take into account all possible dependences between the bugs (erroneous test traces). We say that an error μ_1 *implies* an error μ_2 if any safely testable implementation containing the trace μ_1 also contains the trace μ_2. Obviously, the implication of errors is a preorder (i.e., it is reflexive and transitive). Since this implication is not asymmetric in the general case, it determines a nontrivial equivalence (reflexive, transitive, and symmetric relation) of errors. This equivalence determines the factor relation of error implication, which is a partial order (reflexive, transitive, and antisymmetric relation).

It is clear that, for the completeness of testing, it suffices to detect only the errors that are minimal with respect to the implication preorder up to their equivalence; in other words, it suffices to detect at least one error in every *minimal* equivalence class. For the testing to be total, it suffices to detect up to their equivalence; in other words, it suffices to detect at least one error in *every* equivalence class.

Note that, if a class M_1 implies a class $M_2 \neq M_1$, then the total test suite detecting all the errors must contain tests that detect both the errors in M_1 and the errors in M_2. Indeed, if an implementation contains an error from M_1 but has no errors from M_2, then tests for M_1 are needed to detect errors in M_1 (it is insufficient to have tests for M_2). On the other hand, if an implementation does not contain errors from M_1 but contains an error from M_2, then tests for M_2 are needed (it is insufficient to have tests for M_1). It is quite another matter that, if an error from M_1 is detected in the course of the actual test runs, there is no need to run tests designed for M_2 because the implementation surely contains such errors. Conversely, if we are sure that the implementation does not contain errors from M_2, then there is no need to run tests designed for M_1 because the implementation surely does not contain such errors.

9. THE PROBLEM OF CONFORMANCE MONOTONICITY AND MONOTONE TRANSFORMATION OF SPECIFICATIONS

The problem of conformance monotonicity occurs in connection with the composition of systems. A composite system is assembled from components using certain composition rules. In this paper, the components are modeled by LTSs and the composition rules are modeled by the operator $\uparrow\downarrow$ of the parallel composition of LTSs in the vein of the CCS (Calculus of Communicating Systems) [15, 16]. We assume that the involution (the bijection that is inverse of itself) "underline" is defined on the universe of external actions that assigns to every external action z the *opposite* action $\underline{z}$ such that $\underline{\underline{z}} = z$. For two LTS operands in the alphabets $\mathbf{A}$ and $\mathbf{B}$, the composition is an LTS in the alphabet $\mathbf{C} = (\mathbf{A}\backslash\underline{\mathbf{B}}) \cup (\mathbf{B}\backslash\underline{\mathbf{A}})$. The states in the composition are defined as the pairs of states of the LTS operands initial state–pair of initial states. The transitions in the composition are defined as the minimum set generated by the following inference rules: for any states a, a' of the first LTS operand and any states b, b' of the second LTS operands,

(1) $z \in (\mathbf{A} \cup \{\gamma\tau\})\backslash\underline{\mathbf{B}}$ & $a \xrightarrow{z} a' \vdash ab \xrightarrow{z} a'b$,

(2) $z \in (\mathbf{B} \cup \{\gamma\tau\}\backslash\underline{\mathbf{A}}$ & $b \xrightarrow{z} b' \vdash ab \xrightarrow{z} ab'$,

(3) $z \in \mathbf{A} \cap \underline{\mathbf{B}}$ & $a \xrightarrow{z} a'$ & $b \xrightarrow{\underline{z}} b' \vdash ab \xrightarrow{\tau} a'b'$.

The main problem of composite systems is as follows: if the components work correctly, why is the system as a whole incorrect? In conformance theory, the correctness of the implementation of a particular component is defined as the conformance of the implementation of this component to its specification, while the correctness of the implementation of the system is defined as the conformance of the system implementation to the system's specification.

The correctness of a component's implementation is verified by testing this component while the test interacts with the component thus replacing its environment. Such a testing is said to be *autonomous* (an individual component is tested) or *synchronous*. It is clear that a possible reason of the incorrect system operation is incorrect operation of its components that were not completely tested in the course of autonomous testing. Taking into account the fact that the complete test suite is infinite and the implementation may be nondeterministic, this situation is quite probable, and it is often encountered in practice.

However, the problem of composite systems is broader. The component's implementations can be conformal to their specifications but the system assembled from these components can be not conformal to the specification of the entire system. What can be the reason and what should we do in such a situation?

The cause may be in that the very relationships between the components' specifications and the specification of the system is incorrect. This is an error in the decomposition of the system specification into the specifications of the components [2, 17–20]. Such errors are much more serious than the errors in specific components because they are more difficult to detect and have more serious implications. Therefore,

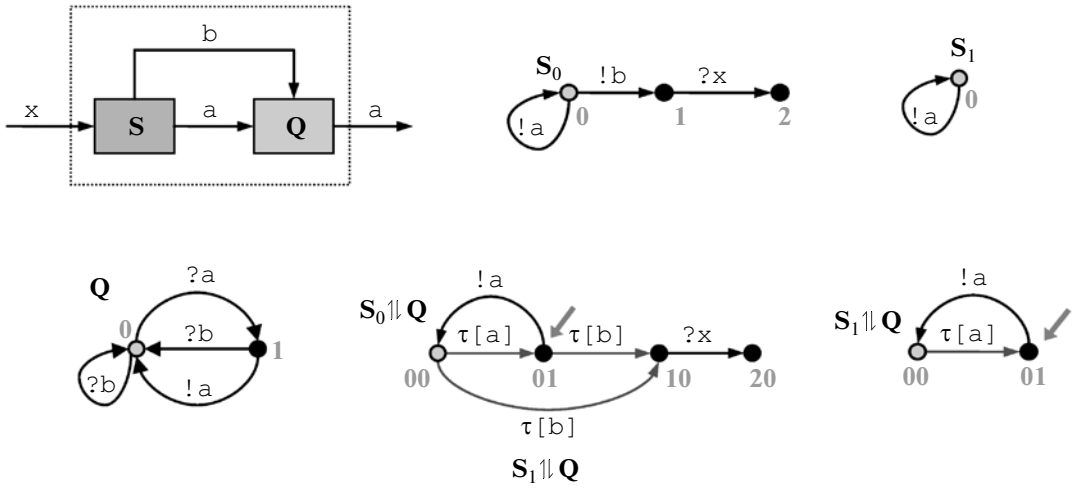


Fig. 5. Nonpreservation of conformance under asynchronous testing.

the system or complex testing not only continues the unfinished testing of components, but it is also designed for detecting architectural errors that cannot be detected by autonomous tests. Such errors can lead to drastic changes in the specifications, which requires a modification of the implementations of all or some of the components or even their reimplementation.

A correct relationship between the specifications of the components and the specification of the system must satisfy the *monotonicity condition*: the composition of the components that are conformal to their specifications is conformal to the specification of the system. A system specification is correct if it satisfies the monotonicity condition for the given specifications of the components. The strongest (that imposes the strongest requirements) correct specification of the system is determined by the monotonicity condition itself. However, this definition is implicit; it is not clear if such the strongest specification exists and if it is possible to construct it algorithmically.

The point is that the strongest correct specification of the system, if it exists, is weaker than the composition of the specifications performed according to the same rules as those used for the composition of the components' implementations. If the composition of implementations $\uparrow\downarrow$ is called *direct*, then we need another—*skew*—composition of specifications $\nearrow\swarrow$ coinciding with the strongest correct specification of the system if such a specification exists. This situation is caused by different levels of abstraction used in the definitions of conformance and direct composition. Conformance is based on the traces of observation of the behavior of the implementation model; the composition is additionally based on directly unobservable states and unobservable actions (τ-actions).

A special case of composition is asynchronous testing or context testing [21]. This kind of testing can be considered as testing a system consisting of two components of which one is an implementation and the other is a fixed interaction environment. In distinction from synchronous testing, asynchronous testing causes two problems [16]. One of them is *permissiveness* (the situation when asynchronous tests do not detect an error found by synchronous tests) and the other is *nonpreservation of conformance* (when asynchronous tests detect a spurious error. It seems that the first problem should be put up with; indeed, asynchronous (and, in general, composition) testing is more "indirect" (through a context) testing of the implementation. It does not always allow one to reproduce all the interaction situations and observe all the behaviors of the implementation that are possible under synchronous testing when the test and the implementation interact directly. The other problem is more serious—it is a special case of the monotonicity problem.

Thus, we want to construct a skew composition for the given conformance. The specification completion is the first step in the solution of this problem. We acquire the possibility to solve this problem for the **R**/∅-semantics in which there are no **Q**-buttons. Then, we may mean by a specification the completed specification in which we have already moved from the **R**/**Q**-semantics to the **R** ∪ **Q**/∅-semantics. Recall that the relations *safe by* and *safe in* are identical in the semantics without **Q**-buttons, and a button is unsafe if and only if it enables a destructive action.

First, we note that the monotonicity problem cannot be solved using the completion only. Figure 5 shows an example in which the conformance is not preserved under asynchronous testing for a semantics with inputs and outputs in which all the refusals are observable, namely, quiescence (the absence of all outputs !*a*, !*b*, ...) and, for every input ?*x*, its refusal {?*x*}. The environment **Q** is the single bounded output queue (in Fig. 5 it has length 1) with the additional

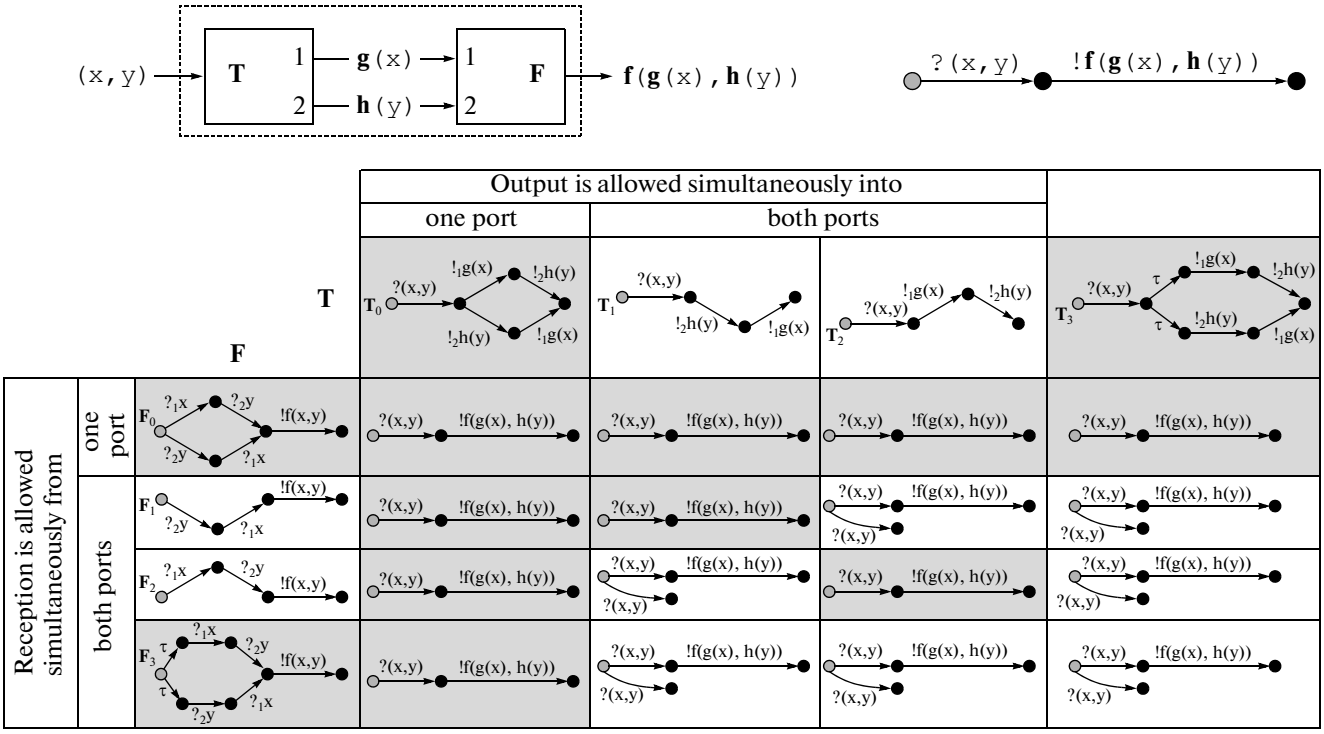


Fig. 6. Verification of the decomposition of system requirements.

reset instruction (*b*). The specification $\mathbf{S}_0$ describes the following requirements for the system: first, the input is refused, but a chain of outputs may be produced; then, it is allowed (but not required) to reset the queue (*b*), accept the input *x*, and finish in the terminal state. The implementation $\mathbf{S}_1$ is conformal—it does not reset the queue and, correspondingly, does not accept the input. When the queue is assembled with the specification, the first input is not refused. However, when the implementation $\mathbf{S}_1$ is composed, such a refusal appears, and this behavior is considered as an error (shown in arrows).

In [2], a formal definition of the skew composition for the relation *saco* is given in the **R**/∅-semantics. It is shown that the skew composition coincides with the direct composition of *transformed* specifications. Such a transformation is called monotone for this conformance, and the conformance is said to be *monotone* with respect to this transformation.

We emphasize that the monotone transformation is determined not uniquely. In [2], two such transformations are described. One of them is based on the union of conformal implementations, and it is always applicable. The other can be used only under certain conditions, but its advantage is that it is defined constructively (inductively), which allows one to implement it by an algorithm.

We also note that different components may have different alphabets, but even in the same alphabet their conformance may be defined in different **R**/∅-semantics, that is, for different sets of **R**-buttons of the testing machine. Hence the question: in which semantics should we consider the composition of components? Fortunately, this problem is resolved by the fact that the direct composition of monotonically transformed specifications is the skew composition in *any* **R**/∅-semantics (naturally, for the same composition alphabet that is uniquely determined by the alphabets of the operands and is independent of the **R**-semantics of the operands in the same alphabets).

Under asynchronous testing, it is assumed that the environment does not contain errors and is known. For that reason, the monotone transformation is applied only to the implementation specification, while the environment remains invariable. In this case, we deal with a left-monotone transformation meaning that the environment determined as the right operand of the composition is preserved (this term is conditional because the composition is commutative). When the monotonicity problem is solved, this case is also taken into account.

The skew composition of specifications of the components of the composite system allows us to perform two tasks: (1) verify the given specification of the system (its conformance to the specifications of the components), and (2) generate the system specification (if it is not given) from the specifications of its components.

The first problem is sometimes called the verification problem of the decomposition of system require-

ments; this is the verification of correctness of the decomposition of the system requirements into the requirements for the components. To solve this problem, the skew composition of the system must be constructed and its conformance to the given system specification must be checked considering it as an implementation. Such a verification can be performed analytically, for example, by modeling the testing of the skew composition considered as an implementation using the complete test suite generated from the given system specification.

By way of example, consider the composition of a receiver with two input ports and a transmitter with two output ports depicted in Fig. 6. The specification of the composite system is very simple: the system must receive the pair of arguments (x, y) and produce the value $f(g(x), h(y))$. Intuitively, this is exactly what the assembly of the implementations of the components must do: the transmitter **T** with the specification $\mathbf{T}_0$ computes the functions g and h, and the receiver **F** with the specification $\mathbf{F}_0$ computes the function f. Every component is considered in two semantics. In one of them, only the specification itself is conformal; in the other semantics, all the depicted implementations are conformal. For the transmitter **T**, the first semantics allows simultaneous output only into one output port; the other semantics allows the output into both ports simultaneously. For the receiver **F**, the first semantics allows simultaneous reception only from one of the ports; the other semantics allows the reception from both ports simultaneously. The shaded cells correspond to the cases in which the composition of the components' implementations conforms to the system specification, and the other cells correspond to the nonconformal case—the system receives the arguments but does not produce the result. We see that not every combination of the semantics of the components **T** and **F** guarantees the conformance of the components' specifications to the system specification. Among four variants, one combination is incorrect and the three others are correct.

How can a decomposition be verified? In Fig. 6, for each component **T** and **F**, the shaded cells illustrate the monotone transformation of the specification depending on the semantics in which the original specification was considered. The cells in which the composition of monotonically transformed specifications is presented (in shortened form without τ-transitions) have a double frame. The shaded cells with a double frame depict the composition of specifications that conforms to the given specification; the light-colored cell with the double frame shows the nonconformal composition of specifications. In the last case, the specifications of the components do not match the system specification.

The solution of the second problem (generation of the system specification) pursues three goals. (1) The resulting specification can be used as a document describing the system from the user point of view. (2) Such a specification can be considered as a requirement specification for the system developer including future modifications of the system (its components and even the decomposition scheme). (3) System (complex) tests can be generated from this specification.

Let us discuss the third goal in more detail. Theoretically, if the components are completely tested for the conformance to their specifications, there is no need for testing the system as a whole. Instead, it suffices to verify the correct relationships between the system specification and the specifications of its components (the first problem (task)) or simply generate the lacking system specification from the given specifications of the components (the first two goals of the second problem). However, due to the fact that the complete test suite is typically infinite and the implementation can be nondeterministic, any practical testing is incomplete; therefore, we cannot be sure that the autonomous testing detected all the errors in the components. For that reason, there is always need for system tests in practice. In particular, system testing reproduces such modes of interaction between the components that are closer to the modes used in the real-life operation of the system and that are rarely reproduced in the course of autonomous testing (or this is very difficult to do).

System testing is associated with two problems. First, the skew composition of specifications must satisfy the requirements that make the algorithmic generation of tests possible. To this end, one has to impose additional constraints upon the original specifications. Second, system testing must be safe. It was shown in [2] that, if no additional assumptions about possible implementations of the components are made or no special means for checking the safety are used, the composition of the component implementations can be unsafe for almost any system specification if these components interact. As a result, system testing is either unsafe or is reduced to the autonomous testing of the individual components that do not interact with each other.

In monotonicity theory, we deal with conformance and composition. Conformance is defined in the theory of observation traces, and composition is defined for LTS models. The observation traces (**R**-traces) are insufficient for determining the composition because the same model of the observation traces can be assigned many different LTSs that can give considerably different results from the viewpoint of conformance.

Nevertheless, it is possible to define in trace theory the composition of models that has the same sense by using different traces. In [2], the so-called ϕ-traces are introduced, and the composition of ϕ-traces is defined

in such a way that the compositions of LTS models coincide with the composition of the ϕ-traces of these models. This property is called *additivity* of ϕ-traces with respect to composition. Note that, in the general case, ϕ-traces are not observation traces (at least, in the **R**/**Q**-semantics). At the same time, given the ϕ-traces of an LTS model, all its **R**-traces can be obtained, although the converse is not true. This property is called the *generativity* of ϕ-traces.

ϕ-traces are similar to *ready traces*; however, they have two distinctions. (1) Instead of the *ready set*, its complement—*ref*-set—is used; this is the set of actions that cannot be executed at the current stable state. This is done by analogy with the *refusal set* in *failure traces* and is not of prime importance. (2) The concept of γ-set is introduced. This is the set of (directly) destructive actions defined in the current stable state. This is done to facilitate the construction of the theory. The pair (*ref*-set, γ-set) is called a ϕ-symbol, and a ϕ-trace is defined as a sequence of external actions and ϕ-symbols. It is clear that, given ϕ-traces of an LTS model, its ready traces can be easily calculated.

The theory of ϕ-traces makes it possible to define a ϕ-model as a set of ϕ-traces having a certain set of properties; this model is equivalent to the observation trace model, as well as to the LTS and RTS models. It also enables us to define the composition of ϕ-models and construct a monotone transformation of ϕ-models.

10. CONCLUSIONS

The parameterized **R**/**Q** interaction semantics, the safety hypothesis, safe conformance, and the test generation method proposed in this paper provide a basis for conformance theory that generalizes many practical and theoretical conformances and methods for their testing. In particular, *failure trace semantics* [3, 4, 6–9] and the semantics of the popular *ioco* relation [9, 10] are special cases of the **R**/**Q**-semantics.

The theory develops in several directions. Some of them are as follows. (1) Introduction of priorities into the model and the corresponding modification of conformance and test generation [22]. (2) Extension of the proposed approach to simulations (conformances based not only on observation traces but also on the correspondence between implementation and specification states) [23]. (3) Generalized interaction semantics that allow the observation of refusals not necessarily coinciding with the set of enabled actions (and even not necessarily embedded in this set) [24].

Another direction of research is the application of the theory in practice. Here, the main problem is to find practically acceptable constraints on the semantics, implementation, and (or) specification, as well as additional testing capabilities that could enable one to perform complete testing in a finite amount of time. These are constraints on the size of the implementation, on the nondeterminism of the implementation, the possibility of observing the current state of the implementation in the course of testing, and the use of mediator programs that transform test actions and observations [3, 25, 26].

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., 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.
3. 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.
4. 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.
5. 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.
6. Baeten, J.C.M., *Procesalgebra. Programmatuurkunde*, Deventer: Kluwer, 1986.
7. Langerak, R., A Testing Theory for LOTOS Using Deadlock Detection, in *Protocol Specification, Testing, and Verification IX*, Brinksma, E., Scollo, G., and Vissers, C.A., Eds., North-Holland, 1990, pp. 87–98.
8. Phillips, I., Refusal Testing, *Theor. Comput. Sci.*, 1987, vol. 50, no. 2, pp. 241–284.
9. Tretmans, J., Test Generation with Inputs, Outputs and Repetitive Quiescence, *Software Concepts and Tools*, 1996, vol. 17, no. 3.
10. Tretmans, J., Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation, *Comput. Networks ISDN Systems,* 1996, vol. 29, no. 1, pp. 49–79.
11. Heerink, L., Ins and Outs in Refusal Testing, *PhD Thesis*, Enschede, Netherlands: Univ. of Twente, 1998.
12. Hoare, C.A.R., An Axiomatic Basis for Computer Programming, *Commun. ACM*, 1969, vol. 12, no. 10, pp. 576–585.
13. Bourdonov, I.B. and Kossatchev, A.S., Equivalent Interaction Semantics, in *Trudy ISP RAN,* 2008, no. 14.1, pp. 55–72.
14. Kuliamin, V.V., *Software Development Technologies: Component-based Approach*, Moscow: BiINOM, 2007.
15. Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989.

16. Jard, C., Jéron, 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*, Wu, J., Chanson, S., and Gao, Q., Eds., Beijing, 1999, pp. 25–40.
17. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., *Teoriya sootvetstviya dlya sistem s blokirovkami i razrusheniem* (Conformance Theory for Systems with Refusals and Destruction), Moscow: Nauka, 2008.
18. Lipaev, V.V., *Testing Large Software Systems for Conformance to Requirements*, Moscow: Globus, 2008.
19. van der Bijl, M., Resnik, A., and Tretmans, J., Compositional Testing with ioco, in *Formal Approaches to Software Testing, Third Int. Workshop FATES 2003*, Montreal, Quebec, 2003; *Lect. Notes Comput. Sci.*, 2003, vol. 2931, pp. 86–100.
20. van der Bijl, M., Resnik, A., and Tretmans, J., Component Based Testing with ioco, CTIT Technical Report TR-CTIT-03-34, University of Twente, 2003.
21. Revised Working Draft on "Framework: Formal Methods in Conformance Testing," JTC1/SC21/WG1/Project 54/1, in *ISO INterim Meeting/ITU-T*, Paris, 1995.
22. Bourdonov, I.B. and Kossatchev, A.S., Systems with Priorities: Conformance, Testing, and Composition, *Programmirovanie*, 2009, no. 4, pp. 24–40 [*Programming Comput. Software* (Engl. Transl.), 2009, vol. 35, no. 4, pp. 198–211].
23. Bourdonov, I.B. and Kossatchev, A.S., Testing Conformance Based on State Correspondence, *Trudy ISP RAN*, 2010, no. 18.
24. Bourdonov, I.B. and Kossatchev, A.S., Generalized Semantics of Test Interaction, in *Trudy ISP RAN*, 2008, no. 15, pp. 69–106.
25. Bourdonov, I.B. and Kossatchev, A.S., Complete Open-State Testing of Limitedly Nondeterministic Systems, *Programmirovanie*, 2009, no. 6, pp. 3–18 [*Programming Comput. Software* (Engl. Transl.), 2009, vol. 35, no. 4, pp. 301–313].
26. Bourdonov, I.B. and Kossatchev, A.S., Testing with Semantics Transformation, in *Trudy ISP RAN*, 2009, no. 17, pp. 193–208.