ISSN 0361-7688, Programming and Computer Software, 2013, Vol. 39, No. 4, pp. 163–181. © Pleiades Publishing, Ltd., 2013.
Original Russian Text © I.B. Bourdonov, A.S. Kossatchev, 2013, published in Programmirovanie, 2013, Vol. 39, No. 4.

Formalization of a Test Experiment-II

I. B. Bourdonov and A. S. Kossatchev

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

Received January 12, 2013

Abstract—The paper develops the approach to testing considered in [1]. A formal model of test interaction of the most general type and reduction-type conformance are proposed for which there is hardly any dependence between errors. It is shown that many known types of conformance in various interaction semantics are particular cases of this general model. The paper is devoted to the problem of dependence between errors defined by specification and to the related problem of optimization of tests. There is dependence between errors if there exists a strict subset of errors such that any nonconformal implementation (i.e., implementation that contains some error) contains an error from this subset. Accordingly, it is sufficient that the tests detect errors only from this subset. In the general model proposed, the dependence between errors may arise when one chooses, as a class of implementations under test, some strict subset of the class of all implementations. Partial interaction semantics and/or various implementation hypotheses (in particular, a safety hypothesis) precisely suggest that an implementation under test is not arbitrary but belongs to some subclass of (safe) implementations.

DOI: 10.1134/S0361768813040026

INTRODUCTION

In the widest sense, by the correctness of a system under study is understood its correspondence to given requirements. To verify this correspondence by formal methods, one represents the objects and relations of the real world as model, mathematical objects and relations. A model of a system under study is called an implementation, a model of requirements is called a specification, and the correspondence to the requirements is represented as a model conformance. The latter is understood as an ordinary mathematical correspondence, i.e., a subset of a Cartesian product of the sets of implementations and specifications.

A specification and a conformance are assumed to be defined. As for the implementation as a model of the system under study, a *test hypothesis* assumes that such a model exists for every system under study [8].

If an implementation, as a model of the system under study, is known, then a static (analytic) verification is possible, which reduces to the verification of the fact that a pair of formal models <implementation, specification> belongs to an admissible set of such pairs that is defined by a conformance relation.

What should one do if the implementation is unknown (or if it is too difficult to construct it by the system under study)? In this case, one needs testing, which is understood as dynamical verification of conformance, i.e., the verification of conformance during experiments. Of course, for the testing to be possible, the conformance itself should be expressed in terms of the interaction of the implementation with the environment. A test interacts with the system, playing the role of the environment.

In this paper, we consider a testing based on three assumptions, rather than an arbitrary testing.

The first assumption. We consider only a *discrete* test interaction, which reduces to a sequence of discrete events of two kinds: test actions on the implementation and observations of the behavior of the implementation. This sequence is called a *trace*. Note that, in the general case, not any behavior of implementation can be observed in a test experiment; i.e., the implementation may contain events that are unobservable and, hence, indistinguishable between each other and are not included in a trace. Such unobservable behavior of implementation is conventionally denoted by a symbol τ and is called τ-activity.

A discrete interaction is modeled by a so-called testing machine, which includes an implementation. A test action corresponds to pressing some button on the keyboard of the machine, and an observation corresponds to the appearance of the symbol of the test action on the display screen. Thus, a trace represents a sequence of buttons and observations. A test is understood as an instruction for the operator in which it is indicated what should the operator do after one or other trace: press buttons (and which buttons) and/or wait for observations.

All that we can learn about an implementation by means of testing is the set of its traces. The observation of a certain trace in an experiment suggests that all its prefixes are observed in the same experiment at earlier instants of time. Therefore, the set of traces of an

implementation is prefix-closed. A test experiment may be empty: no button is pressed, and observations are not expected. In this case, it is natural to assume that an empty trace is observed. Thus, every implementation contains an empty trace; i.e., the set of traces of an implementation is not empty. Since a test action (pressing a button) does not depend on the implementation (the operator can press any button at any instant of time), an extension of the implementation trace with a button also gives a trace of the implementation. This means that the set of traces of an implementation contains, together with every trace σ, all traces of the form $\sigma\rho$, where ρ is a sequence of buttons.

Now, a specification can be understood as the description of which sets of implementation traces are correct (conformal) and which are not. In the general case, if an implementation has a set of traces I, then the entire set I of traces, rather than each individual trace $\sigma \in I$, is either conformal or nonconformal.

A test (as an instruction for the operator) is also defined by a nonempty prefix-closed set of traces. A test experiment runs a test that ends when one obtains either the maximum trace in the test or a "branching off"; i.e., after a certain trace of the test, one obtains an observation that does not extend this trace in the experiment. The test run results in a trace of implementations $\sigma \in I$. Various runs of the same test may give different results if the implementation and /or test are nondeterministic.

A test is nondeterministic if, after a certain trace of the test, the behavior of the operator of the testing machine is nondeterministic: he may either wait for an observation or press buttons, or he may only press buttons; but there are a few such buttons. In other words, a test is nondeterministic if some of its nonmaximal traces is extended in the test either with both observations and buttons or with several buttons. A nondeterministic test is equivalent to a collection of deterministic tests in the sense that they make it possible to observe the same set of implementation traces. Therefore, as a rule, one considers only deterministic tests.

As regards the implementation, it is assumed that its indeterminism results from the abstraction from some unaccountable external factors—weather conditions—which determine the choice of some or other behavior deterministically. The *global testing hypothesis* suggests that any weather conditions can be reproduced in a test experiment. To this end, even a deterministic test should be run a few times in order to observe all implementation traces that are possible for this test.

While testing, one runs some tests from a certain collection of tests under some weather conditions. The result of testing is a set X of traces that are observed in all these test experiments. The verdicts *pass* or *fail* are given. A collection of tests is *significant* if each conformal implementation passes this test, *exhaustive* if each nonconformal implementation fails it (an error is detected), and *complete* if it is significant and exhaustive. Notice that X is not necessarily the set of all implementation traces. If a specification states that any implementation with a large set of traces $I \supseteq X$ is nonconformal, then a significant collection of tests can (although ought not to), while a complete set of tests must, give the verdict *fail* when observing the set of traces X (or any of its supersets). If a specification states that any implementation with a large set of traces $I \supseteq X$ is, conversely, conformal, then an exhaustive (and a complete) collection of tests can (although ought not to), while the compete set of tests must, give the verdict *pass* when observing the set of traces X (or any of its supersets).

The second assumption. In this paper, we restrict ourselves to those conformances that satisfy the *principle of independence of traces*: any trace of an implementation is either conformal or nonconformal irrespective of other traces of this implementation. Conformances of this type are called *reductions*. For example, a conformance that allows an implementation to have either a trace σ_1 or a trace σ_2, but not both traces simultaneously, is not a reduction. The independence principle rules out simulation-type conformances that are based on the correspondence between the states of implementation and specification, as well as conformances that require that the implementation should contain one or other observations after one or other traces.

For a reduction, one can assume that a specification S (directly or indirectly) defines a set of *solvable* traces $tr(S)$. If an implementation has a set of traces I, then the conformance implies the inclusion $I \subseteq tr(S)$ and represents a partial (nonstrict) order (reflexive, symmetric, and transitive relation). A trace $\sigma \notin tr(S)$ is called an error.

When testing a reduction, the detection of any error $\sigma \in I \backslash tr(S)$ implies that the implementation is nonconformal. Therefore, a significant collection of tests (a significant test) can (although ought not to) give the verdict *fail* as soon as such a trace σ is observed. A collection of tests is exhaustive if, for every nonconformal implementation, at least one error $\sigma \in I \backslash tr(S)$ contained in this implementation can be detected by some test from the collection; i.e., it is a trace of this test. This means that the nonconformance of an implementation is always detected in finite time, whereas a conclusion on the nonconformance of the implementation can be generally made only after all runs under all possible weather conditions of all tests of the complete collection (the number of such runs may be infinite).

The third assumption. In practice, naturally, one applies only finite tests, more precisely, tests that are completed in finite time. Under a discrete interaction, this means that, for given specification and test generation, only finite traces are used. Note that, for a specification based on finite traces, infinite test experiments add nothing. However, this is not so in the gen-

eral case. For example, consider two implementations in which only two observations x and y are possible. Buttons are not used. One implementation contains an infinite chain x. The other implementation does not contain such an infinite chain, but has an infinite "fan" of finite chains x. Both implementations do not contain a transition by y. In finite test experiments, these two implementations are indistinguishable. For a specification in which the observation y is considered as an error after any number of x, both implementations are conformal. At the same time, an infinite test experiment allows one to distinguish between these implementations; the first implementation contains an infinite trace x, while the second does not. If infinite traces are admitted, then a specification can interpret the infinite chain x as an error. Naturally, such an error cannot be detected in finite time.

By definition, any implementation that contains an error is nonconformal. At the same time, in addition to errors detected by a specification, i.e., traces that do not belong to the set of solvable traces $tr(S)$, there may exist other traces (that belong to $tr(S)$) that, nevertheless, are not encountered in conformal implementations. Such traces are said to be nonconformal. Then, by an error we will mean any nonconformal trace, while errors detected by a specification (traces that do not belong to $tr(S)$) will be called errors of the first kind. An error of the second kind is a nonconformal trace that is not an error of the first kind, i.e., that belongs to $tr(S)$.

In this paper, we consider the problem of dependence between errors and the closely related problem of optimization of a complete collection of tests. We will say that an error set A implies an error set B and denote this by $A \longrightarrow B$ if any implementation that contains an error from A contains an error from B. If $A \longrightarrow B$, then, instead of the tests that detect errors from A, one can use tests that detect errors from B. If A is the set of all errors, then $B \subset A$, and obviously $B \longrightarrow A$. If, in addition, $A \longrightarrow B$, then the sets A and B are equivalent (which is denoted by $A \sim B$). One of such subsets of errors that are equivalent to the set of all errors is naturally given by the set of errors of the first kind. However, there may also exist other error sets that are equivalent to the set of errors of the first kind, including its strict subsets. It also happens that the set of errors of the first kind is infinite, but there exists an equivalent finite error set. This allows one to significantly optimize the tests.

A type of such dependence between errors is inherent in any reduction-type conformance for any discrete interaction. First, any implementation that contains a trace σ also contains a trace $\sigma\rho$, where ρ is a sequence of buttons. Therefore, if $\sigma\rho$ is an error, then σ is also an error. Therefore, if $\sigma\rho \in A$, then $A \longrightarrow A \cup \{\sigma\}$. Second, the set of implementation traces is prefix-closed. Therefore, if an error μ is a prefix of a trace σ (we denote this by $\mu \le \sigma$), then σ is also an error. Therefore, if $\mu \in A$, then $A \longrightarrow A\backslash\{\sigma\}$. This allows for the following optimization of tests: for the completeness of testing, it suffices to detect only such errors that are prefix-minimal in the set of all errors (rather than in a set of only errors of the first kind); such errors do not end with buttons. The set of such errors is equivalent to the set of errors of the first kind and, hence, is equivalent to the set of all errors.

At the same time, there exist a variety of conformances for which there are other dependences between errors. Finding such dependences and the related optimization of tests sometimes presents a difficult problem (see, for example, [6, 7]).

The goal of the present study is to determine the general nature of dependences between errors. To this end, we formally define a general model of discrete interaction and a general reduction-type conformance. We will show the following. First, for such a general reduction, there is no other dependence between errors except that pointed out above. Second, other reduction-type conformances are a particular case of the general reduction; i.e., they are reduced to it. In this case, the class of specifications and implementations under test narrows down. Third, the restriction of the class of specifications does not affect the dependence between errors, whereas the restriction of the class of implementations under test gives rise to additional dependences between errors (see, for example, [6, 7]). It is important that each such partial reduction defines a certain natural class of implementations under test. However, in practice, one often uses additional restrictions on the implementations under test, which, in turn, also gives rise to additional dependences between errors (see, for example, [16, 17]). In other words, we reduce the problem of the dependence between errors in the class of implementations under test, which is natural for some partial reduction, to the general problem of dependence between errors that arises as a result of restricting the class of implementations under test.

2. GENERAL MODEL

In this section, we formally define a general model of discrete interaction and a general reduction-type conformance.

2.1. Interaction Semantics

For many partial kinds of interaction semantics, there exits some preset relationship between buttons and observations. Some of such semantics are considered below. In the general case, we do not assume any preset relationship between buttons and observations. However, such a relationship may exist in some specific implementation.

Suppose given two disjoint universes of symbols: B, a universe of test actions (buttons) and O, a universe of observations. We will call such a semantics the B/O semantics. A trace is a sequence in the alphabet $B \cup O$.

A semantics is said to be finite if the total numbers of buttons and observations are finite.

2.2. Testing Machine

The *B/O* semantics is modeled by a testing machine, which represents a "black box" containing an implementation. The machine is equipped with a keyboard for control and a display screen for observation.

The keyboard represents a set of buttons B. A test action is made by pressing a button on the keyboard. When a button is pressed, the testing machine transmits a single signal on an appropriate test action to the implementation and waits for an answer signal indicating that the implementation "takes into consideration" this test action; after that the machine can transmit a signal on the next test action to the implementation. A button is not fixed (it is automatically released); this allows the operator to press the next (another or the same) button. One can press only one button at a time, which corresponds to a single test action.

The screen of the machine successively displays the symbols of buttons and observations, i.e., the symbols from $B \cup O$. In order that the operator could distinguish between consecutive identical symbols, the screen is turned off for a short period of time between these symbols. It is the sequence of symbols appearing on the screen that represents a trace observed during a test experiment. The symbol of a button appears on the screen at the moment when the operator presses the corresponding button. An observation appears on the screen when the corresponding observable event occurs in the implementation. Unobservable τ-activity of the implementation is displayed in no way on the screen.

In order that one could carry out several test experiments, the machine is equipped with a *restart* button. This button resets the implementation to the initial state and turns off the screen. Each new restart of the machine may cause a change of weather conditions, which determine the behavior of the implementation. The global testing hypothesis suggests that a sequence of restarts reproduces all possible weather conditions.

At the same time, a restart allows one to carry out an at most countable number of experiments, for an at most countable number of weather conditions. To evade this constraint, a testing machine can be equipped with a *replication* button instead of the restart button. A single press of such a button creates a set of copies of the testing machine of arbitrary cardinality. A testing occurs independently with each copy of the machine; i.e., an independent test experiment is carried out with each copy. For each copy, its own variant of weather conditions is fixed. The global testing hypothesis implies that at least one copy of the machine is produced under replication for each variant of weather conditions.

It is important to note that, for reduction-type conformances, it suffices to make a replication once, before starting the testing, rather than repeatedly, after obtaining one or other traces. A multiple replication (after every step of testing, i.e., after every observation and after pressing every button) is needed for simulation-type conformances.

2.3. Implementation

For a reduction-type conformance, an implementation actually reduces to the set of traces of this conformance. Such a *trace model of implementation* is formally defined as a set $I \subseteq (B \cup O)^*$ that is (1) nonempty, (2) prefix-closed, and (3) together with each trace σ, contains all traces of the form $\sigma\rho$, where ρ is a sequence of buttons.

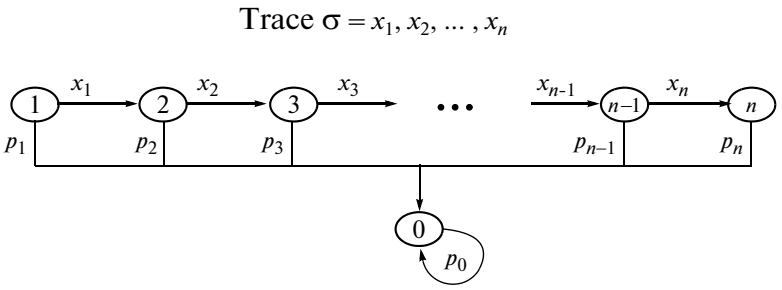
To compactly define a set of traces, in particular, to define an infinite set of traces in a finite way, one applies a *label transition system* (LTS). It represents a directed graph whose vertices are called states, one state is distinguished as the initial state, and the arcs are labeled by symbols from $B \cup O$ and are called transitions. Unobservable τ activity is understood as a chain of elementary τ events each of which is represented by a transition labeled by symbol τ. An LTS implementation is said to be *finite* if the number of its states that can be reached from the initial state is finite.

Since a test action (pressing a button of the testing machine) on the implementation is executed outside the implementation and is independent of it, a transition by a button only implies that the implementation "has learned" about the test action executed. As a result of such a transition, the implementation changes its state, which subsequently may lead to a change in its behavior, i.e., may give rise to other observations. If, in some state, the implementation ignores a test action, then this is equivalent to the fact that there is a loop transition by the given button in this state. Therefore, the absence of a transition by a button in a state of the implementation is interpreted as the presence of a loop transition by this button in this state.

A *path* is a sequence of adjacent transitions when the beginning of any transition, except for the first, coincides with the end of the preceding transition. An implementation trace is a sequence of labels of transitions of a path starting in the initial state, in which the symbol τ is omitted.

Assume that transitions are executed instantaneously.[1] In every state, the implementation waits for some finite time, after which it should execute one of outgoing transitions. This assumption is usually called a *progress assumption* [11]. We also assume that the

[1] It suffices to assume that a transition is executed in finite time; however, if this is a transition by observation, then the symbol of observation is displayed on the screen of the testing machine at the beginning (not in the middle or at the end) of this transition.



For $i = 1..n$, the symbol p_i runs through all buttons from **B**, except for x_i (if $x_i \in$ **B**). The symbol p_0 runs through all buttons from **B**.

implementation may pass only a finite path in finite time. This means that the implementation is considered as a discrete system [11]. To this end, it suffices to assume, for example, that the waiting time in every state is bounded from below by some constant greater than zero.

The set of traces of an LTS implementation is a trace model of implementation, and, conversely, for any trace model of implementation, there exists an LTS with the same set of traces.

It is obvious that, for every trace σ, there exists a minimal implementation with respect to this set that contains the trace σ; this is the set of traces $\{\mu\rho \mid \mu \le \sigma \ \& \ \rho \in B^*\}$. The corresponding LTS implementation is shown in the figure.

2.4. Interaction with Implementation

During testing in any observable trace, a subsequence of buttons contains exactly those buttons that have been pressed by the operator of the machine and precisely in that order in which the operator pressed them. From the external point of view, a test action affects only those observations that appear in the trace. In other words, pressing a button just regulates the flow of observations of the behavior of the implementation. This is achieved by transitions by buttons, which change the state of the implementation when pressing a button and thus change the further flow of observations.

As regards the unobservable behavior of the implementation, we proceed from the *main assumption of* τ*-activity*: between any two observations (and before the first observation in the observation trace), there may be any finite τ-activity in the implementation [11]. In terms of an LTS, this means that, between two transitions by observation (and before the first such transition), the implementation may execute any finite number of τ-transitions. We extend this assumption over test actions: the implementation may execute any finite number of τ-transitions between any two transitions by observations or buttons (and before the first such transition).

A specific feature of our model of interaction is the *priority of a test action over the behavior of implementation*, both observable and unobservable. If, after obtaining a trace σ in a test experiment, the operator waits for observations instead of pressing a button, then any observation can be obtained that is contained in the implementation after the trace σ, i.e., any trace σ*u* available in the implementation can be obtained, where *u* is an observation. Moreover, if there is infinite τ-activity (*divergence* as an infinite chain of τ-transitions) after the trace σ in the implementation, then it may occur that there are no observations. If, immediately after the observation of the trace σ, the operator presses a button *p*, then the implementation must execute a transition by button *p*, and one obtains a trace σ*p*. However, by the main assumption of τ-activity, between a transition by the last symbol (observation or button) of the trace σ and a transition by the next observation *u* in the trace σ*u* or the next button *p* in the trace σ*p*, the implementation can execute any finite number of τ-transitions. Thus, the implementation must take into consideration a test action in finite time after pressing an appropriate button. At the same time, we do not specify what does "taking into consideration" mean; one may even simply ignore a test action, which is modeled in the LTS by a loop transition by this button (the absence of a transition by button is interpreted as the presence of such a loop).

As a result, we obtain the following interaction protocol. If there is no test action, i.e., no button is pressed, then, depending on weather conditions, the implementation can execute any chain of transitions by observations and τ-transitions that starts in the current state of the implementation. If a test action is executed, i.e., if the operator pressed a button *p*, then, depending on weather conditions, the implementation can execute any finite chain of τ-transitions, after which it must execute any transition by the button *p*. While executing a *p*-transition, the implementation, as if "informs" the testing machine that it has received a test action. After that, the implementation is ready to receive the next test action (pressing one or other button).

Note that, under such an interaction protocol, the appearance of the symbol of a button on the screen during pressing a button is actually equivalent to the appearance of this symbol when the implementation

executes a transition by this button. That is why the trace displayed on the screen coincides with the trace of the path that the implementation passes during this time.

When there are several transitions in a state that can be executed by the implementation, one chooses one of these transitions in the indeterministic way. Under a pressed button, the number of τ-transitions that are executed before a transition by button is also chosen indeterministically. Both these choices are understood as choices depending on weather conditions. The global testing hypothesis guarantees that all possible weather conditions can be enumerated.

In any interaction session with implementation through a testing machine, the screen displays an implementation trace, as well as all of its prefixes (at earlier instants of time).

2.5. Operator of Testing Machine

The operator of a testing machine simulates the operation of the test system. We assume that the operator executes a test understood as an instruction for the operator. This instruction points out what can the operator do after obtaining a trace: should he wait for observations and/or press buttons, and which buttons. Naturally, if the test determines the behavior of the operator after a trace μ, then, in order that one could obtain this trace μ, the test must determine the behavior of the operator even after any prefix of the trace μ.

If the test allows the operator to press a button p after the trace μ, then it is assumed that the operator can press this button any time after obtaining the trace μ. Thus, the operator is *not prohibited* to sustain any pause after obtaining some traces, including pauses before pressing the next button: any time he can make a "tea break." This means that, when the operator presses the button p some time after the trace μ and, after pressing the button waits for some time, then one actually obtains the trace $\mu\pi_1 p\pi_2$, where π_1 and π_2 are observation sequences, rather than the trace μp.

At the same time, for the completeness of testing, it is necessary that any implementation trace of interest should be observable during its interaction with the implementation through the testing machine. For this purpose, the operator must *be able* to press buttons sufficiently quickly after obtaining traces. This means that the delay between obtaining a trace and pressing the next button may turn out to be less than the waiting time of the implementation in the state after the trace at least in one testing session. Then, pressing the button p immediately after the trace μ, the operator will observe precisely the trace μp. Of course, if the operator does not turn off the machine and does not press buttons for some period of time after this trace, then one can obtain the extension of this trace with a sequence of observations, i.e., the trace $\mu p\pi_2$.

2.6. Specification and General Reduction

As pointed out in the Introduction, a specification defines, either explicitly or implicitly, a set of allowed traces and, simultaneously, its complement—a set of errors of the first kind. In this study, by a specification we mean an arbitrary set of finite traces that is understood simply as a set of errors of the first kind. An implementation is conformal if it does not contain errors of the first kind, i.e., specification traces. Below, we will call such a conformance a *general reduction*.

A specification as a set of errors of the first kind can be defined by a generating graph, i.e., by an LTS with distinguished terminal vertices. An LTS specification is said to be *finite* if the number of its states that can be reached from the initial sate is finite. For some infinite sets of errors of the first kind, an LTS specification can be finite. As is known, for any generating graph, there exists a determinization procedure that constructs a deterministic graph generating the same set of sequences. Therefore, for any specification, there exists a deterministic LTS specification that defines the same set of traces. Here the determinacy implies that every reachable state does not contain τ-transitions and, for any symbol $x \in B \cup O$, at most one transition by x from this state is defined. If an LTS specification is finite, then it remains finite after determinization.

A specification S defines a class of conformal implementations, which we denote by C_S. If a specification contains an empty trace, i.e., if an empty trace is assumed to be an error of the first kind, then all implementations are nonconformal, because any implementation contains an empty trace. If a specification is an empty set, then all implementations are conformal.

2.7. Test

A *test* is a set T of finite traces. Obtaining any of these traces during testing leads to the verdict *fail*, while obtaining any other trace, to the verdict *pass*. A test is understood as an instruction for the operator of the testing machine. The aim of a test is to verify whether there is at least one of the traces of the test in the implementation: if this is so, then the testing ends and the general verdict *fail* is given.

Denote the prefix-closure of the set T of sequences by $pre(T) = \{\mu \mid \exists\sigma \in T\mu \le \sigma\}$.

Pressing a button. During testing, the operator may press a button p after obtaining a trace μ if the trace μp is a prefix of some trace of the test, $\mu p \in pre(T)$. It is assumed that, in this case, after obtaining the trace μ, the operator presses the button p at least in one testing session, and does this sufficiently quickly after obtaining the trace μ. It is also assumed that if the trace μu, where u is an observation, is a prefix of some trace of the test, $\mu u \in pre(T)$, then the operator waits for observations after obtaining the trace μ at least in one testing session. If these assumptions are satisfied, the global

testing hypothesis guarantees that if a certain trace $\sigma \in T$ is encountered in the implementation, then it will be obtained in at least one testing session.

Turning off the machine. During testing, the following three cases are possible after obtaining the trace μ:

1. $\mu \in pre(T)\backslash T$; i.e., μ is a strict prefix of some trace from T;

2. $\mu \in T$;

3. $\mu \notin pre(T)$; i.e., μ is not a prefix of any trace from T.

In case 1, the operator should continue the testing session; i.e., he should not turn off the testing machine. In cases 2 and 3, the operator should complete the testing session; i.e., he should turn off the testing machine. Then, the following verdicts are given: *fail* in case 2 and *pass* in case 3.

A trace that can be obtained in some testing session with a given test (not only at the end of the session) has either the form $\mu\pi$, where μ is a prefix of some trace $\sigma \in T$ and π is a sequence of observations, or the form $\mu\pi_1 p\pi_2$, where μp is a prefix of some trace $\sigma \in T$, p is a button, and π_1 and π_2 are observation sequences. We will call the set of such traces an expansion of a test and denote $exp(T) = \{\mu\pi \mid \mu \in pre(T) \& \pi \in O^*\} \cup \{\mu\pi_1 p\pi_2 \mid \mu p \in pre(T) \& \pi \in B \& \pi_1 \in O^* \& \pi_1 \in O^*\}$. The set of traces that can be obtained at the end of the testing session is given by $(exp(T)\backslash pre(T)) \cup T = exp(T)\backslash(pre(T)\backslash T)$.

An implementation *passes* a test if the verdict *pass* is given for any testing session (under any weather conditions). An implementation *passes* a collection of tests if it passes each test from this collection. For a given class of implementations (in particular, for the class of all implementations), a collection of tests (a test) is significant if every conformal implementation from this class passes this collection of tests (this test), exhaustive if every nonconformal implementation from this class fails it, and complete if this collection of tests (test) is both significant and exhaustive.

A test is *deterministic* if it uniquely defines the behavior of the operator. This means that any trace from the prefix-closure of the test is either continued by a button and is not continued by observations or is not continued by buttons in this prefix-closure: $\forall \mu \in pre(T)\ (|\{p \in B \mid \mu p \in pre(T)\}| = 1 \ \& \ \{u \in O \mid \mu u \in pre(T)\} = \varnothing) \vee \{p \in B \mid \mu p \in pre(T)\} = \varnothing$.

A test is *primitive* if it contains a single trace. It is obvious that a primitive test is deterministic. Any test T is equivalent to the union of the set of primitive tests in the sense that they give the verdict *fail* for the same implementations: $T = \cup\{\{\sigma\} \mid \sigma \in T\}$. It is also obvious that a specification (as a set of errors of the first kind) is a complete test on the class of all implementations. Hence, a collection of primitive tests constructed over all errors of the first kind, i.e., over all specification traces, is complete on the class of all implementations.

2.8. Normalization of a Specification and Optimization of Tests

As already pointed out in the Introduction, in addition to errors of the first kind, i.e., specification traces, there may also exist other errors—nonconformal traces, i.e., traces that are not encountered in conformal implementations. The errors that are not errors of the first kind are called errors of the second kind. For the completeness of testing, it suffices to detect only those errors that are prefix-minimal in the set of all errors (not only errors of the first kind). Such errors are called *primary* errors; a *secondary* error is an error that has a strict prefix, which is an error. Primary errors do not end with buttons. The set of primary errors is equivalent to the set of errors of the first kind and, hence, to the set of all errors. Obviously, this set is the least by inclusion subset of errors that is equivalent to the set of all errors. It can be considered as a specification, which we will call a *normalized* specification.

For every specification S, the set of all errors is constructed by the systematic application of the following operations:

- If p is a button and $\sigma p \in S$, then we add a trace σ to S.
- If $\mu \in S$ and $\mu < \sigma$, then we add a trace σ to S.

If S^* is the set of all errors for specification S, then the normalization procedure reduces to the elimination of prefix-nonminimal errors: if $\mu \in S$, $\sigma \in S$, and $\mu < \sigma$, then we remove the trace σ from S.

The normalization can also be carried out directly by the original specification S as a systematic application of the following operations:

- If p is a button and $\sigma p \in S$, then we add a trace σ to S.
- If $\mu \in S$, $\sigma \in S$, and $\mu < \sigma$, then we remove the trace σ from S.

Normalized specifications are in one-to-one correspondence with their classes of conformal implementations: $A = B \Leftrightarrow C_A = C_B$.

Suppose that a specification S is normalized. As pointed out above, for every trace σ, there exists an implementation minimal with respect to the set of traces that contains the trace σ; this is the set of traces $\{\mu\rho \mid \mu \le \sigma \ \& \ \rho \in B^*\}$. Hence, a collection T of tests is significant if and only if each trace of every test from the collection has an error from S as a prefix; i.e., if S is coinitial with $\cup T$. A collection T of tests is exhaustive if and only if every trace from S has a prefix that is a trace of some test from the collection; i.e., if $\cup T$ is coinitial with S.

A collection T of tests is complete if and only if S and $\cup T$ are mutually coinitial. Since S is normalized, the condition that $\cup T$ is coinitial with S can be replaced by the inclusion condition $S \subseteq \cup T$. Indeed, otherwise there exists a trace $\mu \in S\backslash\cup T$, and then, since $\cup T$ is coinitial with S, there exists a trace $\mu_1 < \mu$ such that $\mu_1 \in \cup T$; then, since S is coinitial with $\cup T$, there exists a trace $\mu_2 \le \mu_1$ such that $\mu_2 \in S$; hence, S

contains two errors $\mu_2 < \mu$, which contradicts the normality of S.

In other words, a collection T of tests is complete if and only if all traces of all of its tests are given by all primary errors (traces form the normalized specification S) and some of their extensions. An obvious optimization is the elimination of such extensions; this results in a collection T' of tests whose set of all traces of all tests is a normalized specification: $S = \cup\{\{\sigma\} \mid \sigma \in S\} = \cup T'$. Thus, an optimized complete collection T' of tests is the covering of S, while the collection of primitive tests $\{\{\sigma\}|\sigma \in S\}$ is one of possible partitions of S.

3. CLASS OF IMPLEMENTATIONS

For various reasons, as implementations under test, one considers implementations belonging to some class I, rather than arbitrary implementations. This leads to additional dependences between errors (including dependences between primary errors) and, accordingly, allows for additional optimization of the tests.

The first definition of equivalence of specifications. Two specifications A and B are said to be equivalent on the class of implementations I if they define the same conformance of implementations on this class: $I \cap C_A = I \cap C_B$. If I is the class of all implementations, then $C_A \subseteq I$, and $C_B \subseteq I$, ($C_A = C_B$) coincides with the equality ($A = B$). On other subclasses of implementations, this is not generally the case.

A trace encountered in the implementations of the class I is said to be *actual* on the class I. If I is a set of trace implementations, then the set of actual classes is equal to $\cup I$. On the class of all implementations, all traces are actual. Other classes of implementations may contain both actual and nonactual traces. A trace that is encountered in conformal implementations from the class I is said to be conformal on the class I. This is a conformal class that is actual on the class I. Errors (including errors of the first and second kind) are classified into actual and nonactual ones. When testing implementations from the class I, it obviously suffices to detect only errors that are actual on this class. The set of traces that are conformal on the class I is equal to $\cup(I \cap C_S)$ for the specification S. Accordingly, the set of errors that are actual on the class I is equal to $\cup I \setminus \cup (I \cap C_S)$.

This gives **the second definition of equivalence of specifications**: Two specifications A and B are said to be equivalent on the class of implementations I if they define the same set of actual errors: $\cup I \setminus \cup(I \cap C_A) = \cup I \setminus \cup(I \cap C_B)$.

In fact, the two definitions of equivalence of specifications are equivalent: $I \cap C_A = I \cap C_B \Leftrightarrow \cup I \setminus \cup(I \cap C_A) = \cup I \setminus \cup(I \cap C_B)$. Let us prove this. First, we show that $\cup(I \cap C_A) = \cup(I \cap C_B) \Leftrightarrow \cup I \setminus \cup(I \cap C_A) = \cup I \setminus \cup(I \cap C_B)$. Indeed, if $\cup(I \cap C_A) = \cup(I \cap C_B)$, then, obviously, $\cup I \setminus \cup(I \cap C_A) = \cup I \setminus \cup(I \cap C_B)$. Let us show that if $\cup I \setminus \cup(I \cap C_A) = \cup I \setminus \cup(I \cap C_B)$, then $\cup(I \cap C_A) = \cup(I \cap C_B)$. Suppose that this is not the case: for example, a trace $\sigma \in \cup(I \cap C_A) \setminus \cup(I \cap C_B)$. Then this trace belongs to some implementation from I that is conformal for A. But then this implementation does not belong to C_B; i.e., it contains an error from B. This error belongs to $\cup(I \cap C_A)$; hence, it does not belong to $\cup I \setminus \cup(I \cap C_A)$. This error also belongs to $\cup I$ but does not belong to $\cup(I \cap C_B)$; hence, it belongs to $\cup I \setminus \cup(I \cap C_B)$, which contradicts the equality $\cup I \setminus \cup(I \cap C_A) = \cup I \setminus \cup(I \cap C_B)$. Now, let us show that $I \cap C_A = I \cap C_B \Leftrightarrow \cup(I \cap C_A) = \cup(I \cap C_B)$. If $I \cap C_A = I \cap C_B$, then, obviously, $\cup(I \cap C_A) = \cup(I \cap C_B)$. Let us show that if $\cup(I \cap C_A) = \cup(I \cap C_B)$, then $I \cap C_A = I \cap C_B$. Suppose that this is not the case. Then there exists an implementation that belongs, say, to $I \cap C_A \setminus I \cap C_B$. Then this implementation belongs to I but does not belong C_B; hence, it contains some error from B. Thus, this error belongs to $\cup(I \cap C_A)$. But this error cannot belong to $\cup(I \cap C_B)$, which contradicts the equality $\cup(I \cap C_A) = \cup(I \cap C_B)$.

Now, suppose that I is a mapping that defines, for every specification S, a class I_S of implementations under test. We say that, for the mapping I, a specification B *can be used instead of specification* A if (1) $I_A \subseteq I_B$ and (2) $I_A \cap C_A = I_A \cap C_B$. The first condition says that any implementation that could be tested to verify the conformance of the specification A can be tested to verify the conformance of the specification B. The second condition (equivalence of specifications on the class I_A) says that the specifications A and B define identical conformance of implementations on the class of implementations under test for the specification A.

An error is detected by a collection of tests if it is a trace of one of the tests of the collection. Any collection of tests that is complete on the class I of implementations under test obviously defines a set of detectable errors (the set of all traces of all of its tests) that is equivalent to the set of all errors on the class I.

4. SAFETY HYPOTHESIS

In [1, 2, 5, 6], we introduced the concept of safe testing. This is a testing under which implementation traces that are assumed to be unsafe are not passed. The safety hypothesis defines a class of implementations that can be safely tested to verify the conformance of a given specification.

4.1. General Form of the Safety Hypothesis

We say that a *safety hypothesis* is defined if, for any implementation I, a prefix-closed subset $SafeTraces(I) \subseteq I$ of traces is defined that are called safe traces. A testing of a given implementation is said to be safe if one can obtain only safe traces of this implementation during this testing. Implementation I is safe for a test T if testing by this test is safe for this implementation: $exp(T) \cap I \subseteq SafeTraces(I)$. Every test T defines a class of safe implementations $SafeImpl(T) = \{I|exp(T) \cap I \subseteq SafeTraces(I)\}$. A collection of tests T

defines a class of implementations that are safe for each test from this collection: $SafeImpl(T) = \cap\{SafeImpl(T)|T \in T\}$. A specification S defines a class of safe implementations as a class of implementations that are safe for the complete test for S, or, which is the same, for a collection of primitive tests constructed by specification errors: $SafeImpl(S) = SafeImpl(\{\{\sigma\}|\sigma \in S\})$. In the general case, if a safe testing of implementations from a given class I is assumed, then safe implementations from the class I, i.e., implementations from the class $I \cap SafeImpl(S)$ are tested.

4.2. Hypothesis on a Finite Waiting Time for Observation

In order that every testing session be finite in time, it is necessary that the waiting times of buttons and observations on the screen be finite: (1) a button should appear on the screen in a finite time after its pressing, and (2) if the operator waits for observations, then an observation should appear on the screen in a finite time.

The first condition is satisfied for sure in this model of interaction with implementation. The second condition may even not hold. Let us formulate a requirement on the implementation for this second condition to hold for a given test.

Hypothesis on observations—λ-hypothesis:[2] if an implementation trace is a prefix of a trace of a test and is extended to the prefix-closure of the test with observation, then, in the implementation, it should also be extended with some (not necessarily the same) observation for any behavior of the implementation. This means that (1) in each stable state (a state in which no τ-transitions start) in the implementation, there is a transition by some observation after this trace, and (2) there is no divergence after this trace. The λ-hypothesis is a particular case of the safety hypothesis.

Define, formally, a set $SafeTraces(I)$ of safe traces of the implementation I for the λ-hypothesis. A λ-trace of an implementation is an implementation trace that ends either in a stable state, where there are no transitions by observations, or in a divergent state. An implementation trace is said to be safe if any of its strict prefixes that is followed by an observation in the trace is not a λ-trace.

The λ-hypothesis does not change the actuality of traces: all traces are actual. The λ-hypothesis changes the conformance of traces: on the class of safe implementations defined by this hypothesis, a trace μ is nonconformal if, for every observation u, the trace μu is an error. The following *procedure of λ-normalization of a specification* is applied to determine the primary errors of a specification in the case of the λ-hypothesis: we systematically apply three actions:

(1) If, for any observation u, the trace σu belongs to S, then we add the trace σ to S. (2) If p is a button and $\sigma p \in S$, then we add the trace σ to S. (3) If $\mu \in S$, $\sigma \in S$, and $\mu < \sigma$, then we remove the trace σ from S. The set of traces obtained is the set of primary errors in the case of the λ-hypothesis.

4.3. Destruction Hypothesis

Another variant of the safety hypothesis is the so-called destruction hypothesis. By destruction is meant any behavior of an implementation that is undesirable during testing [1, 2, 5]. The reasons of undesirability of some behavior may be quite diverse; here we do not impose any constraints. To represent a destruction in the LTS model of implementation, we replace some of its transitions by observation or τ-transitions (unobservable behavior) by γ-transitions, i.e., by transitions labeled by a special symbol of destruction γ.

Now, by an implementation model is meant an LTS in the alphabet with an added symbol γ. Since we are not interested in the behavior of implementation after destruction, by a trace we mean a sequence of buttons and observations that may end with a destruction.

Destruction hypothesis—γ-hypothesis: for a specification S, any trace from $exp(S)$ is not extended in the implementation with destruction. The γ-hypothesis is a particular case of the safety hypothesis.

Define, formally, a set $SafeTraces(I)$ of safe traces of implementation I for the γ-hypothesis: an implementation trace is said to be safe if any of its prefixes is not extended in the implementation with destruction.

The γ-hypothesis changes the actuality of traces: a trace is actual if it is not represented as $\mu\gamma$, where $\mu \in exp(S)$. The γ-hypothesis does not change the conformance of actual traces: on the class of safe implementations defined by this hypothesis, only that actual trace is nonconformal whose prefix is an error. Since the γ-hypothesis does not change the conformance of actual traces, it does not require the normalization procedure: all specification errors are primary.

Combined, λ and γ-hypotheses define a class of safe implementations $SafeImpl_{\lambda,\gamma}(S) = SafeImpl_{\lambda}(S) \cap SafeImpl_{\gamma}(S)$.

5. SIMULATION OF OTHER SEMANTICS

In this section, we show how some known reduction-type conformances are reduced to a general reduction in the above-described *B/O*-semantics. In 1993, van Glabbeek published a generalizing and systematizing article [11] in which he defined 30 types of observations. Some or other combination of these types of observation corresponds to some or other interaction semantics. Not all the combinations are admissible; van Glabbeek distinguished 155 possible semantics and corresponding conformances. We consider only those observations that correspond to a reduction-type conformance (i.e., not to simulation-type conformances) and show how these observations

[2] The symbol λ is used to denote a situation when a deadlock or divergence arise [10].

and the corresponding semantics are represented in our *B/O*-model and in the LTS implementation.

It is assumed that an implementation can execute external, observable actions from some alphabet A. As regards the unobservable behavior of an implementation, van Glabbeek proceeds from the main assumption of τ-activity: τ-activity may occur between any two actions (and before the first observable action) in the implementation. In the LTS model of implementation, transitions are labeled by symbols from the set $A \cup \{\tau\}$.

The keyboard of van Glabbeek's testing machine consists of switches—one switch for each external action. The switch has two positions: *free* and *blocked*. An implementation can execute an external action a only when the switch "a" is in the position *free*; τ-actions are always allowed, irrespective of the position of the switches. It is assumed that, at any instant of time, the operator of the machine can set any switch in any position and is sufficiently fast to do this rapidly, i.e., immediately after some observation. The external actions executed by the implementation are displayed on the screen. Van Glabbeek's machine is generative: an implementation executes external actions and τ-actions allowed by the switches as long as it contains such actions. If, at a given instant of time (in a given state), the implementation can execute several actions, then the action to be executed is chosen indeterministically, depending on weather conditions.

A test action in van Glabbeek's machine consists in changing the positions of switches, which corresponds to the set of switches in the position *free*, i.e., a subset $\rho \subseteq A$. In the *B/O*-machine, such a test action corresponds to a separate button "p",[3] and every action $a \in A$ is an observation from O. In other words, we will assume that $\{\text{“}p\text{”} \mid p \subseteq A\} \subseteq B$ and $A \subseteq O$.

If there are no other observations except for external actions, then such a semantics is called a *trace semantics* [12]. To simulate this semantics in the *B/O*-semantics, one should perform the following transformation of the original LTS implementation S. For every state s and every button "p," where $p \subseteq A$, we introduce a new state s_p. In this state, we execute a transition $s_p \xrightarrow{a} t_p$, where $a \in A \cup \{\tau\}$, if and only if $a \in p \cup \{\tau\}$ and the transition $s \xrightarrow{a} t$ has been executed. We also execute a transition $s_q \xrightarrow{q} s_q$ for every button "q", where $q \subseteq A$ and $q \neq p$ (one may not show loop transitions by buttons in the *B/O*-semantics); this transition corresponds to a change in the position of switches. A new initial state is the state $s_{0\varnothing}$, where s_0 is the initial state in the original LTS S (it is assumed that, immediately after turning on the machine, the switches are in the position *blocked*). Note that if there have been no transitions by actions from $p \cup \{\tau\}$ in the state s, then there are no transitions by actions from $A \cup \{\tau\}$ in the position s_p (only transitions by buttons are defined). The transformation results in a new LTS S_T.

Note that a change in the position of switches for a *trace semantics* is redundant: it suffices to set all the switches to the position *free* from the very beginning and not to change them. This is equivalent to the absence of switches under the assumption that all actions are thereby allowed (that is how the testing machine is described for the *trace semantics* in [10]). Such an operation mode is characterized by the same testing capacity; i.e., one obtains the same set of traces as for all possible changes of the position of switches. To simulate in the *B/O*-semantics, it suffices to leave only states of the form s_A in the LTS S_T; the new initial state is the state s_{0A}, rather than the state $s_{0\varnothing}$. As a result of transformation, we obtain an LTS S_{T^*}. It is obvious that the LTS S_{T^*} is isomorphic to the LTS S.

In this testing mode, when buttons are not used, a transformation of the LTS implementation is not needed.

In addition, the van Glabbeek machine may have a *green lamp*, which is turned on when the implementation has some activity (a transition is executed either by external action or by τ-action). If the green lamp is turned off, this means that no activity goes on the implementation: it has no τ-activity, while all external actions that it could execute are blocked by the positions of appropriate switches. This gives new observations—a *refusal set*. A refusal $p \subseteq A$ is defined in a state s when there are no transitions by actions from $p \cup \{\tau\}$ in this state. Traces that contain refusals in addition to actions are called *failure traces*, and the corresponding interaction semantics is called a *failure trace semantics*, which is denoted as *FT*. To simulate the *FT* semantics in the *B/O*-semantics, it suffices to add, in the LTS implementation S_T, a loop transition by refusal p to each state s_p in which transitions by actions from $A \cup \{\tau\}$ are not defined. As a result of transformation, we obtain an LTS S_{FT}.

The van Glabbeek machine may also contain so-called menu lamps, one for each action $a \in A$. A lamp 'a' for an action a is turned on if a transition by the action a is defined in the implementation at a current instant of time (irrespective of whether or not this action is allowed by the switches). It is clear that if the implementation executes some actions (either external or internal), then the menu lamps are constantly blinking. Therefore, they give reliable information only when no activity goes on the implementation, which is indicated by the green lamp. In this case, we obtain a new observation—a *ready set*. A ready set $r \subseteq A$ is defined in state s when this state is stable (there are no τ-transitions) and the set of external actions by which transitions are defined in the state is equal to r. The traces that contain actions and ready sets are called *ready traces*, and the corresponding interaction

[3] We denote a button with the use of quotation marks, "p" for $p \subseteq A$, to distinguish between a button "p" and a refusal p, which is introduced below.

semantics is called a *ready trace semantics*, which is denoted as RT. It is obvious that, by a ready set r, one can calculate all refusals in this state: these are all the subsets $A\backslash r$. Therefore, in the RT-semantics, refusals are not considered as separate observations, and one does not consider mixed traces that contain both ready sets and refusals. To simulate the RT-semantics in the B/O-semantics, it suffices to add a loop transition by the ready set for the state s in the original LTS S to every state s_p in which transitions by actions from $A \cup \{\tau\}$ are not defined in the LTS implementation S_T. Notice that the ready set for the state s in the original LTS S is equal to the union of the ready sets for all states in the LTS S_T of the form s_q, where q runs over all buttons. In other words, this is the set of all external actions that the LTS implementation S_T in the state s_p can execute after some change in the position of the switches of the van Glabbeek machine (after pressing some button of the B/O-machine). As a result of transformation, we obtain an LTS S_{RT}.

Van Glabbeek also considers the operation mode of the machine when the switches cannot be switched from the position *blocked* to the position *free*, except for the initial setting of the switches when turning on the machine. In this mode, there cannot be any continuation of the operation of the machine after stopping the implementation; therefore, a refusal or a ready set can be observed only at the end of a trace. Usually it is assumed that either a *failure pair* (a trace of actions and a refusal) or a *ready pair* (a ready set) is observed at the end of a testing session. The corresponding semantics are called *failure semantics* (F) and *readiness semantics* (R). To simulate the F and R semantics in our model, one replaces loop transitions by refusals or ready sets in the LTS implementations S_{FT} or S_{RT} by transitions to the terminal state. As a result of transformation, we obtain LTSs S_F or S_R.

If only a finite number of switches can be set to the position *free*, then the corresponding semantics with refusals are denoted as FT^- and F^-. To simulate such semantics in the B/O-semantics, one removes transitions by infinite buttons from the LTS implementations S_{FT} or S_F. As a result of transformation, we obtain and LTSs S_{FT^-} or S_{F^-}.

In the presence of menu lamps, one considers semantics when, in addition to the fact that only a finite number of switches can be set in the position *free*, only a finite number of menu lamps can be activated. In this case, the menu lamps are interpreted as buttons–lamps; to activate such a button–lamp, it should be released; a nonactivated lamp indicates nothing. Now, if no activity goes on the implementation, the question of whether there is an action $a \in A$ in the current state of this implementation can have three answers: (1) yes; then the switch "a" is in the position *blocked*, and a button–lamp 'a' is released and turned on; (2) no; then either the switch "a" is in the position *blocked* and a button–lamp 'a' is released but is turned on, or the switch "a" is in the position *free* (a released button–lamp 'a' cannot be turned on); and (3) unknown; then the switch "a" is in the position *blocked*, and the button–lamp 'a' is not released. Therefore, full information on the actions in a current state is described by a pair of sets, by the set r^+ of actions with the answer "yes" and by the set r^- of actions with the answer "no," rather than by a single ready set r. In contrast to the case when all menu lamps are always activated, these two sets r^+ and r^-, when combined, may not make up the set A of all actions. Now, a test action consists not only in setting some switches in the position *free*, but also in the activation of some menu lamps. The corresponding semantics with ready sets are denoted by RT^- and R^-. To simulate such semantics in the B/O-semantics, instead of the state s_p in the LTS implementation S_{RT} or S_R, one creates, instead of the state s_p, a set of states of the form s_{px}, where x is a finite set of activated menu lamps. Instead of a transition by an action $s_p \xrightarrow{a} t_p$, one executes transitions of the form $s_{px} \xrightarrow{a} t_{px}$. Instead of a transition by a button $s_p \xrightarrow{q} s_q$, one executes transitions of the form $s_{px} \xrightarrow{qy} s_{qy}$, where y is a finite set of activated menu lamps, only for a finite button q. Instead of a loop transition by a ready set $s_p \xrightarrow{r} s_p$, one executes loop transitions by pairs of finite sets $s_{px} \xrightarrow{r^+r^-} s_{px}$; it is obvious that $x\backslash r^- = r^+$ and $p \subseteq r^-$. As a result of transformation, we obtain LTSs S_{RT^-} or S_{R^-}.

Van Glabbeek also considers two special observations 0 and S. The observation 0 arises when the green lamp is turned off, which means that no activity goes on the implementation. The observation S arises when the implementation passes to a stable state. In the $FT^-_!$ or F-semantics, the observation 0 arises every time when some refusal arises, while the observation S arises every time when the refusal $\varnothing$ arises. Therefore, both these observations are redundant in these semantics.

The observation 0 is useful when there are no switches, which is interpreted as setting all switches to the position *free* (just as for the *trace semantics*). In this case, the observation 0 implies a transition of the implementation to the terminal state, which corresponds to the refusal A in the FT- or F-semantics. It is clear that there cannot be any observations after the observation 0; therefore, the observation 0 can only complete a trace. Such traces are called *completed traces*. To simulate in the B/O-semantics, it suffices (just as for the *trace semantics* in the absence of switches) to add a loop transition by the observation 0 to the LTS implementation S_{T^*} in each terminal state. As a result of transformation, we obtain an LTS S_{T0}.

The observation S is useful when, instead of switches for every external action, there is a single switch that either allows or blocks all external actions

at once. If all the actions are prohibited, while the green lamp is turned off, this means that the implementation is in a stable state, which is represented by the observation S. This observation is equivalent to the observation of refusal $\varnothing$ in the FT- or F-semantics. If all actions are allowed, while the green lamp is turned off, this means that the implementation is not only in a stable but also in the terminal state; i.e., we have observation 0 that "absorbs" the observation S. In addition to actions, traces may include the observation S; such traces may end with the observation 0. To simulate in the B/O-semantics, it suffices to leave only states of the form s_A and $s_\varnothing$ in the LTS implementation S_T and then add loop transitions by the observation 0 to all states of the form s_A where there are no transitions by actions from $A \cup \{\tau\}$, and loop transitions by the observation S to all stable states of the form $s_\varnothing$. As a result of transformation, one obtains an LTS S_{T0S}.

Except for the restriction by the finiteness in the FT^-, F^-, RT^-, and R^- semantics and the two special cases above with the observations 0 and S, van Glabbeek does not consider conformances that are obtained under various restrictions on what switches can be set in the position *free*, i.e., restrictions on the set of allowed external actions. It is also assumed that if there is a green lamp, then there are no constraints on its operation: it can operate always, irrespective of the position of the switches. In other words, one either observes all possible refusals or does not observe any refusals. At the same time, many conformances are based precisely on these kinds of constraints.

One of such conformances, which is not included in van Glabbeek's classification, is the now popular *ioco* (*input–output conformance*) relation, which was proposed by Tretmans in 1996 [14, 15]. It is assumed that the alphabet A of external actions is divided into two disjoint subsets of stimuli (*input*) X and reactions (*output*) Y. Either one stimulus or all reactions can be allowed. It is said that the operator can either send one stimulus to the implementation or wait for any reaction from the implementation. In this case, the green lamp is turned on only when waiting for reactions; thus, there is only one refusal, which means the absence of reactions and is called quiescence; it is denoted by the symbol $\delta = Y$. In addition, the *ioco* semantics requires a reactive, rather that generative, machine. Instead of switches, there are buttons, which are automatically released after the implementation executes an allowed external action. To obtain the next external action, one should press once again other or the same buttons. In fact, the difference between generative and reactive machines is insignificant, as van Glabbeek showed in the same paper [11].

A generalization of such an approach is the R/Q semantics proposed by the present authors [1, 5, 6]. It is defined by two disjoint families of sets of actions $R \subseteq 2^A$ and $Q \subseteq 2^A$ that cover the whole alphabet: $(\cup R) \cup (\cup Q) = A$. The R/Q testing machine is reactive. Each set $p \in R \cup Q$ corresponds to a button "p" that allows all actions from p. The green lamp is turned on only if $p \in R$; in other words, only refusals from R are observed. Note that the R/Q-semantics also admits transitions by destruction γ in the implementation.

The *ioco* relation is a particular case of the R/Q-semantics when $R = \{\{x\} | x \in X\}$ is the family of all sets each of which consists of a single stimulus and $Q = \{\mathbf{Y}\}$ is a family consisting of a single set of all reactions. In addition, it is assumed for *ioco* that the implementation has no destruction.

To simulate the R/Q-semantics in the B/O-semantics, one should transform the original LTS implementation S as follows. For every button "p", where $p \in R \cup Q$, and every state s, we add a new state s_p and a new transition $s \xrightarrow{\text{"}p\text{"}} s_p$. In each new state s_p, we execute a transition $s_p \xrightarrow{a} t$ if $a \in p$ and there is a transition $s \xrightarrow{a} t$. We also execute a transition $s_p \xrightarrow{a} t_p$ if $a \in \{\tau, \gamma\}$ and there is a transition $s \xrightarrow{a} t$. If $p \in R$ and there are no transitions by actions from $A \cup \{\tau\}$ in the state s_p, then we execute a transition $s_p \xrightarrow{a} s$. This transition is obviously executed if and only if there is an R-refusal p in the state s. After that, we remove all transitions by external actions from the old states, while retaining τ and γ-transitions.

We obtain an LTS $S_{R/Q}$ whose states are classified into "old" and "new" ones; this is necessary to simulate the "reactance" of the R/Q-machine on the generative B/O-machine. A transition by a button "p" leads from an old state to a new state: from s_p to t. A transition by an external action leads from the new state to the old: from s_p to t, and only by an action a that is allowed by the button "p", i.e., $a \in p$. A transition by refusal $p \in R$ leads from the state s_p to the state s, while the τ and γ-transitions lead both from old to old states (from s to t) and from the corresponding new to the corresponding new states (from s_p to t_p).

A more detailed account of the simulation of the R/Q-semantics in the B/O-semantics is given in [8].

Thus, we can see that all the above-considered semantics are simulated in the B/O-semantics by an appropriate transformation of the original LTS implementation. Hence, the original conformance in the B/O-semantics is considered on a subclass of transformed LTS implementations, rather than on the class of all LTS implementations admitted by the B/O-semantics. Therefore, an implication of error sets and equivalent sets of errors arise that do not coincide with the specification; i.e., various equivalent specifications arise. Each such specification is obtained from some complete collection of tests if one takes all *fail* traces of tests of the collection as errors. In other words, there exist complete collections of tests such that the difference between the tests cannot be removed by a trivial optimization similar to the normalization of specifica-

tions. That is what the problem of optimization of tests for various semantics consists in, which turns out to be a particular case of the problem of optimization of tests on some or other class of implementations.

6. PRIORITIES

As pointed out by van Glabbeek himself [11], his testing machine assumes the absence of priorities between actions: an implementation may execute an action a if the switch "a" is in the position *free*, irrespective of the position of other switches, i.e., irrespective of what other actions are allowed. The rule of nondeterministic choice suggests that the implementation should choose any action for execution that is defined in this implementation and is allowed by the position of switches. In this case, the τ-activity can always be performed irrespective of the position of the switches.[4] At the same time, for real software and hardware systems, this rule does not always adequately reflects the required behavior of a system. Below we consider a few examples of such systems.

In order to introduce priorities into the van Glabbeek machine, one should label every transition in the LTS implementation not only by an action a but also by a set of allowed actions p, that is, by the pair (a, p). In this case, it is assumed that $a \in p \cup \{\tau\}$. What does occur under the change of the position of switches when the set of allowed actions is changed from p to q? We suppose that the effect of this change on the behavior of the implementation occurs in two steps. At the first step, transitions by actions are blocked, but τ-transitions labeled by the "old" set p, i.e., transitions labeled by the pair (τ, p), still remain allowed. At the same time, it is assumed that the implementation can execute only a finite number of such transitions at this step. After that, at the second step, only transitions labeled by the "new" set q, i.e., transitions labeled by the pair (a, q), where $a \in q \cup \{\tau\}$, are allowed.

Let us explain the meaning of the first step. In the absence of priorities, by the main assumption on τ-activity, τ-transitions are always allowed irrespective of the position of the switches. Therefore, under the change of the position of switches, there is no difference between the τ-transitions executed immediately before this change and immediately after it. However, in the presence of priorities, such a difference arises. If the first step were missing, then the change of the set of allowed actions from p to q would immediately prohibit (τ, p) transitions. When the implementation passes through a certain path M with a trace σ, it may occur in the state s in which a chain of (τ, p) transitions is defined. For the completeness of testing, it is necessary that this chain of transitions be passed in at least one testing session. However, for the implementation to be able to pass this chain, the operator should change the position of switches only after executing this chain. To this end, the operator should wait for some time interval depending on the length of the chain and the waiting times in the states of this chain. Since the implementation is unknown to the operator (it is hidden in the black box), the value of this interval is unknown to the operator. Therefore, we should have required that, in various testing sessions, the operator made all possible time delays before the next change of the position of switches. However, we make the only requirement to the operator: at least in one testing session, this delay should be small enough, i.e., less than the waiting time of the implementation in the state s. The presence of the first step guarantees that all chains of (τ, p) transitions after the path M with trace σ can be passed without additional requirements to the operator. In other words, it is guaranteed that every path with trace σ is passed before the set of allowed actions is changed from p to q and only (τ, q) transitions start to be executed.

In the presence of priorities, the concepts of refusal and divergence are changed. A refusal p arises for the allowed set of actions p when there are now transitions labeled by a pair of the form (a, p), where $a \in p \cup \{\tau\}$, in a current state. Divergence for the allowed set of actions p arises when an infinite τ path with an infinite postfix of (τ, p) transitions starts in a current state. Accordingly, we can speak of p divergence and on p-divergent and p-convergent states.

As pointed out above, in the B/O-semantics, there is a priority of a testing action over the behavior of implementation, both observable and unobservable. This allows one to represent any semantics of the van Glabbeek machine with priorities as a particular case of the B/O-semantics for reduction-type conformances. For such a simulation, one should execute a modified transformation of the original LTS implementation S with priorities into an appropriate LTS S_i, where i denotes a semantics; i.e., i is T, T^*, FT, RT, F, R, FT^-, RT^-, F^-, R^-, $T0$, or $T0S$. The modification consists in the following: a transition $s_p \xrightarrow{a} t_p$, where $a \in A \cup \{\tau\}$, is defined if and only if $a \in p \cup \{\tau\}$ and there was a transition $s \xrightarrow{(a, p)} t$ in S, rather than the transition $s \xrightarrow{a} t$, as before. Then, the transitions by buttons, refusals, ready sets (or pairs of sets), and the observations 0 and S are executed as usual.

For the R/Q-semantics with priorities, an LTS implementation is defined analogously: each transition is labeled by a pair (a, p), where $a \in p \cup \{\tau, \gamma\}$ and $p \subseteq A$. Naturally, for given R and Q, only those (a, p) transitions can be executed during testing by the R/Q-machine in which $p \in R \cup Q$. To simulate the R/Q-semantics with priorities in the B/O-semantics, one should perform a modified transformation of the original LTS implementation S with priorities into the LTS $S_{R/Q}$. The modification consists in the following:

[4] In the R/Q semantics this also applies to γ-transitions.

the transitions $s_p \xrightarrow{a} t$, where $a \in p$, and $s_p \xrightarrow{a} t_p$, where $a \in \{\tau, \gamma\}$, are defined if there is a transition $s \xrightarrow{(a,p)} t$ rather than the transition $s \xrightarrow{a} t$, as before, in S. The transitions by refusals and buttons are executed as usual. Notice that the transition by a refusal $s_p \xrightarrow{p} s$, where $p \in R$, is executed when there is the R refusal p in s, i.e., when there are no transitions labeled by pairs (a, p) in the state s, where $a \in p \cup \{\tau, \gamma\}$. Finally, not all τ- and γ-transitions are left in the old states, but only those that are labeled by an empty set, i.e., by the pair $(\tau, \varnothing)$ or $(\gamma, \varnothing)$.

Note a specific feature of such simulation of the R/Q-semantics with priorities by means of the B/O-semantics. In the R/Q-semantics, which contains an empty button ($\varnothing \in R \cup Q$), τ- and γ-transitions under pressed empty button and in the absence of the pressed button are indistinguishable: in either case the set of allowed external actions is the same—an empty set. Therefore, one cannot require that such a transition be activated only when no button is pressed, or, conversely, it be activated when the empty button is pressed but could not be executed if no button was pressed. When simulating in the B/O-semantics, the τ-transitions $s_\varnothing \xrightarrow{\tau} t_\varnothing$ and $s \xrightarrow{\tau} t$ arise always simultaneously (provided that there is a transition $s \xrightarrow{(\tau,\varnothing)} t$). However, after such a simulation in the B/O-semantics, we can resolve this problem by leaving only one of these τ-transitions.

A more detailed description of the R/Q-semantics with priorities is given in [3, 4], and its simulation in the B/O-semantics is given in [8].

Since all the semantics with priorities considered above are simulated in the B/O-semantics by an appropriate transformation of the original LTS implementation, the original conformance in the B/O-semantics is considered on a subclass of transformed LTS implementations, rather than on the class of all LTS implementations admitted by the B/O-semantics. Therefore, just as in the case of semantics without priorities, there arise an implication of error sets and equivalent sets of errors that do not coincide with the specification; i.e., there arise various equivalent specifications.

It may turn out that there are many multiple transitions in the LTS with priorities, that are labeled by pairs (a, p_i) with the same action a and different sets of allowed actions $p_1, p_2, \ldots$. To represent such transitions more compactly, we introduce Boolean variables: one variable for each action $a \in A$; the variable 'a' takes the value *true* if either the switch "a" in the van Glabbeek machine is in the position *free* or the button "p" is pressed in the R/Q-machine and $a \in p$. The set of allowed actions p can be defined by an elementary conjunction 'p' of these variables in which each variable a appears only once, either without negation, if $a \in p$, or with negation otherwise. The conjunction 'p' takes the value *true* if and only if the switches in the van Glabbeek machine allow the set of actions p or the button "p" in the R/Q-machine is pressed. After that, one can represent the set of multiple transitions by a single transition labeled by the pair (a, π), where π is a predicate equivalent to the perfect disjunctive normal form $'p_1' \vee 'p_2' \vee \ldots$.

Consider a few characteristic examples of using priorities.

Exit from divergence. A request incoming from outside can be ignored by the system for an infinitely long time if it has the same priority as infinite internal activity, i.e., divergence. Note that internal activity can be initiated by a previous request. If one deals with a composite system assembled from several components, then divergence can naturally result from the interaction of components with each other. In this case, to process a request incoming to the system (to one of its components) from outside, this request should have higher priority than the internal interaction.

In the B/O-semantics, such a request can be understood as a (test) action, i.e., as pressing a button.[5] A transition by a button is executed for sure in finite time, i.e., only after finite τ-activity. Thus, an exit from divergence can be implemented, provided, of course, that a transition by button does not lead it to a convergent state.

In both the van Glabbeek machine and the R/Q-machine, requests correspond to some external actions from an alphabet A. Then a τ-transition from state s is labeled only by a set p of allowed actions that does not contain requests transitions by which are defined in s. If the implementation is in the state s and the set of actions whose switches are in the position *free* is equal to p, then divergence can arise only when the state s is p-divergent.

Exit from oscillation (priority of reception over output). By oscillation is meant an infinite chain of message output by a system. In order that such a chain may be interrupted, by making the system to process a request incoming from outside, the latter should have higher priority than the output of messages. Usually, it is also assumed that the internal activity has lower priority than the reception of a request.

In the B/O-semantics, it is natural to assume that the output of a message is an observation, while a request is a (test) action (pressing a button). Since pressing a button blocks observations before a transition by the button, an exit from oscillation is performed if such a transition by button leads to a state where there is no infinite chain of message outputs and no divergence.

In the van Glabbeek machine and in the R/Q-machine, requests and an output of messages correspond to two disjoint subsets of the alphabet A. Then transitions on the output of messages and τ-transitions from the state s are labeled only by those sets p of

[5] If we can observe the reception of a request by the implementation, then, in addition to the button, a separate observation corresponds to such a request (see Subsection 8.1).

allowed actions that do not contain requests transitions by which are defined in s.

For brevity, we consider other examples only for the van Glabbeek machine.

Priority of output over reception in unbounded queues. This inverse example is characteristic of an unbounded queue that is used as a buffer between interacting systems, in particular, when testing in a context [13]. Here it is necessary that a sample from the queue should have higher priority than queuing. Otherwise the queue is only allowed to receive messages and never to deliver them. When testing in a context, for the input queue this means that all input messages sent by the test do not reach implementation, being infinitely accumulated in the queue. Accordingly, for the output queue this means that the test may receive no response messages from the implementation, although the queue outputs them, because these messages accumulate in the queue.

Suppose that the elements of the queue belong to an alphabet Z. The queuing of an element $x \in Z$ corresponds to an action $!x \in A$ and a switch "$!x$." For the queuing of elements to be deterministic, at most one switch is allowed to be set in the position *free*. The sampling of an element $y \in Z$ from the queue corresponds to an action $?y \in A$ and a switch "$?y$." The priority of the sampling from the queue over the queuing implies that a transition by queuing, i.e., by the action $!x$, can be executed only when the switch "$!x$" is in the position *free* and either there is nothing to take from the queue, i.e., the queue is empty, or it is prohibited to take an element from the queue, i.e., the switch "$?y$," where y is the first element of the queue, is in the position *blocked*. The state s of implementation is a state of the queue, i.e., $s \xrightarrow{(!x,\, p)} Z^*$. A transition $s \cdot !x$ is defined if and only if $p \subseteq A$, $!x \in p$, and either the queue s is empty or the first element of this queue, $y = s(1)$, cannot be selected from the queue because the switch "$?y$" is in the position *blocked*, i.e., $?y \notin p$. A transition $?y \cdot s \xrightarrow{(!y,\, p)} s$ is defined, as usual, for every $p \subseteq A$ provided that $?y \in p$.

Interruption of a chain of actions. The command *cancel* should interrupt the chain of actions initiated by the previous request and call the chain of completion actions. In the absence of priorities, such a command, even if it is given immediately after the output of the request, can only be executed after the whole processing is completed; i.e., in fact, this command does not change anything.

We will consider the command *cancel* as one of external actions. If a transition by *cancel* is defined in a state s, then all the other transitions from the state s are labeled by the same sets p of allowed actions that do not contain *cancel*. A transition by *cancel*, as well as all the other transitions in other states, are labeled by all admissible sets of allowed actions. If a transition by *cancel* is defined in the state s and the switch "*cancel*" is in the position *free*, then only a transition by *cancel* will be executed.

Priority processing of requests. If several requests are simultaneously incoming to the system, it is often required to process them according to some priorities between them. This is implemented in the form of a queue of requests with priorities, or in the form of several queues of requests with priorities between the queues. The processing of hardware interruptions in an operating system also belongs to this type of priorities.

A set of requests is partitioned into disjoint subsets $X_1, X_2, \ldots$ of the alphabet A so that requests from a subset with greater index have higher priority. A transition from the state s by a request $x \in X_i$ is labeled by a set $p \subseteq A$ that does not contain any request $y \in X_j$ such that $j > i$ and there is a transition by y in the state s.

7. OPTIMIZATION OF TESTS FOR VARIOUS CLASSES OF IMPLEMENTATIONS

In the previous sections, we have shown that, for the *B/O*-semantics and the general reduction on the class of all possible implementations, there are only trivial dependences between errors, which can easily be removed by the normalization of specification. We have also considered two safety hypotheses: λ- and γ-hypotheses, which narrow down the class of implementations under test. Moreover, the λ-hypotheses gives rise to an additional dependence between errors, which, however, is easily removed by additional λ-normalization, while the γ-hypotheses does not lead to an additional dependence between errors, and there is no need in additional normalization. Next, we have considered examples of semantics and conformances that are reduced to the *B/O*-semantics and the general reduction but are considered on restricted classes of implementations. Because of such restriction, there arise nontrivial dependences between errors, which requires a nontrivial optimization of tests [6, 7].

All this can be considered as a particular case of the general problem of restriction of the class of implementations under test, that gives rise to dependences between errors and allows one to optimize tests. Consider a few examples of such restriction of the class of implementations under test that are not directly related to the choice of one or other semantics or safety hypothesis. In theses examples, we will show that such a restriction allows one to apply finite complete collections of tests. All these examples suggest that the *B/O*-semantics and the LTS specification S are finite. We will assume that the total number of buttons and observations is not greater than m, while the number of states of the deterministic LTS specification is not greater than k.

The first example is a class of LTS implementations with a limited number of states. If the number of states of the implementation is not greater than n, then, for the completeness of testing, it suffices to restrict one-

self to tests of length at most nk. Then this collection of tests contains at most $O(m^{nk})$ tests.

To prove this assertion, it suffices to construct a composition of an LTS implementation and a specification by the following rules. The states of the compositional LTS are given by pairs of states of implementation and specification, and the initial state is given by the pair of initial states. A transition $(s, t) \xrightarrow{a} (s', t')$ is defined if and only if there is a transition $s \xrightarrow{a} s'$ in the implementation and a transition $t \xrightarrow{a} t'$ in the specification. The implementation is nonconformal if and only if it contains an error of the first kind, i.e., a specification trace. In a deterministic specification, such a trace ends in a state that is declared finite. There is such a trace in the implementation if and only if, in the compositional LTS, a state of the form (s, t), where t is the terminal state of the specification, is reachable from the initial state. Such a state can be reached by a simple path (that passes through each state at most once) whose length is not greater than the number of reachable states of the compositional LTS, which, in turn, does not exceed the total number of states equal to nk. Thus, an implementation is nonconformal if and only if there is an erroneous trace of length at most nk in this implementation. In other words, the collection of all primitive tests of length at most nk is complete. The number of such sequences in an m-symbol alphabet is obviously equal to $O(m^{nk})$.

The second example is a finite (up to an isomorphism) class of implementations under test. For a finite semantics, the class of LTS implementations with a limited number of states is obviously finite up to an isomorphism. Therefore, the first example is a particular case of the second example. If a semantics and a specification are finite, then, for any finite class of implementations, there exists a finite complete collection of tests. To prove this assertion, it suffices to notice that any finite class of implementations I is a subclass of the class of implementations the number of whose states is bounded by a number n, where n is the maximal number of states in the implementations from the class I.

The third example is a finite subclass of nonconformal implementations from the class of implementations under test. In [16, 17], such a subclass is called a *failure class*. For the completeness of a test collection to be finite, it suffices that a subclass $I\backslash C_S$ of failures of the class I, rather than the class I itself, be finite. Indeed, in every nonconformal implementation from the class $I \in I\backslash C_S$, there is some error of the first kind; let us choose one of such errors σ_I. The collection of errors $S_I = \{\sigma_I \backslash I \in I\backslash C_S\}$ is finite and, obviously, is a complete test, whereas the collection $\{\{\sigma_I\} | I \in I\backslash C_S\}$ of primitive tests constructed by these errors is a complete collection of tests for the class I.

Thus, while testing, we actually try to find a finite subset $S_I \subseteq S$ of the set of errors of the first kind rather than all errors of the first kind defined by the specification S. This is equivalent to that, instead of the specification S, we use the specification S_I. In other words, on the class of implementations I, the specifications S and S_I are equivalent. To tell the truth, while carrying out a testing by the specification S, we can detect an error faster than when testing by the specification S_I. This is attributed to the fact that the nonconformal implementation $I \in I\backslash C_S$ may contain not only an error σ_I, but also some errors that do not belong to the collection S_I. For example, the specification S may define some observations as erroneous from the very beginning (before pressing buttons); a, b_1, b_2, b_3, ..., and S_I contains only one such error a. While testing, we can wait for observations from the very beginning and, based on the specification S, we give the verdict *fail* if we obtain any of the errors a, b_1, b_2, b_3, ...; however, based on the specification S_I, we give the verdict *fail* only for the error a.

These arguments give the fourth example—a finite subset of errors $S_I \subseteq S$ such that every nonconformal implementation (i.e., implementation containing at least one error from S_I) from the class I contains at least one error from S_I. Instead of a finite class of failures, it suffices to simply use a finite subcollection of the collection of errors defined by the specification.

Next, recall that the class of implementations I defines errors of the second kind: traces that are not encountered in conformal implementations of the class I but are encountered in some of its nonconformal implementations. Such an error of the second kind σ may not be an error of the first kind, i.e., $\sigma \notin S$. Therefore, the fourth example is a particular case of the last, fifth, example when, for the class I of implementations under test, a finite collection of errors (of the first and second kind) S_I is defined such that every nonconformal implementation (i.e., an implementation containing at least one error from S) from the class I contains at least one error from S_I. This is the last example, because its condition is simply equivalent to the condition of the existence of a finite complete collection of tests. If such a collection of tests exists, then the set of errors S_I is given precisely by the set of traces of all tests of the collection.

8. SUBSTANTIATION OF THE CHOSEN INTERACTION MODEL

In this concluding section, we substantiate the interaction model chosen. We consider six questions that arise in connection with this model: (1) When does a button is inserted into a trace: when it is pressed and/or when the LTS implementation executes a transition by a button? (2) Why does pressing a button block observations? (3) Why the operator should be able to press a button sufficiently quickly after obtaining a trace? (4) Why pressing a button does not block τ-activity? (5) Why pressing a button blocks divergence, i.e., why it allows only finite τ-activity? and (6)

Why a transition by every button is defined in every state of implementation?

8.1. When Is a Button Inserted into a Trace?

Pressing a button is executed by the operator of the testing machine, and, at this moment, he knows what trace is already obtained. Therefore, there is no reason why the operator should not notice the fact that he pressed a given button after the observation of a given trace. On the other hand, the behavior of the implementation generally depends on the trace after which the operator presses a button. Therefore, in any case, when a button is pressed, the implementation is inserted into a trace.

If, when the LTS implementation executes a transition by a button p, the button p also appears on the screen, then this is similar to the situation when, while the implementation executes a transition by observation, this observation is displayed on the screen. This means that the execution of a transition by the button p is in fact an observation, whose appearance in a trace is denoted by p' to distinguish it from p, which means pressing the button p.

In principle, such an observation is no different from other observations; therefore, the operation mode with an observable transition by a button can be considered as a particular case of the general model of interaction. In order to reproduce such an operation mode in this model, it suffices to replace every transition $s \xrightarrow{p} t$ by a button in the LTS implementation by two transitions by introducing an additional intermediate state: $s \xrightarrow{p} s' \xrightarrow{p'} t$.

8.2. Why Does Pressing a Button Block Observations?

If pressing a button does not block observations, then additional dependence between implementation traces (and, hence, between errors) arises. Let us explain this by an example. Suppose that, during interaction with an implementation, one can observe a trace up, where u is an observation and p is a button. Then, since one observes the trace up, one also observes its prefix—the trace u. If the operator presses the button p before the observation u but observations are not blocked by this pressing, then the implementation can anyway execute a transition by u. Therefore, one observes the trace pu. Hence, if one can observe the trace up during interaction with the implementation, then one can also observe the trace pu.

In the interaction model chosen, there is no such additional dependence between traces. At the same time, the operation mode with missing blocking of observations under pressing a button is easily simulated in our model. To this end, it suffices to systematically perform the following transformation of the implementation, while this is possible: if there are transitions $s \xrightarrow{p} s_p$, $s \xrightarrow{u} t$, and $t \xrightarrow{p} t_p$ in the implementation, we add a transition $s_p \xrightarrow{u} t_p$. Thus, if, in the state s, a trace up starts that ends in the state tp, then there also is a trace pu that ends in the same state.

Thus, the interaction model with blocking of observations under pressing a button is a more general model. The class of all implementations for a model without blocking corresponds to a subclass of implementations, obtained by the above-described procedure, for a model with blocking. Just as in the general case, such a restriction of the class of implementations gives rise to dependences between implementation traces (in particular, between errors).

Moreover, the blocking of observations is a consequence of the priority of a test action over observations. This priority is needed in order that one could simulate the behavior of systems with priorities, in particular, to interrupt a chain of external actions by the command *cancel*.

8.3. Why the Operator Should Quickly Press Buttons?

First of all, notice that we proceed from the basic assumption of τ-activity: the implementation can have τ-activity before or after any observation, as well as before or after any transition by a button. It is clear that any constraints imposed on the τ-activity could only restrict the class of implementations considered, which would give rise to additional dependences between errors. The presence of τ-activity does not yet mean that it should certainly manifest itself; however, it is naturally assumed that it manifests itself at least for some interaction. Naturally, a τ-activity may manifest itself when no button is pressed. In the next subsection, we will consider whether or not pressing a button blocks the τ-activity.

In addition we want that any reachable transition in the LTS implementation could be executed under some or other interaction with this implementation (depending on the behavior of the operator and weather conditions that simulate the nondeterministic behavior of the implementation). If this is not the case and some transition is not executed for any interaction, then this is equivalent to the absence of this transition in the implementation. This, in turn, leads to the restriction of the class of implementations considered, which is also fraught with the rise of additional dependences between errors. Only safety hypotheses prohibit the execution of some "unsafe" transitions in the implementation; however, as we have considered above, this also leads to the restriction of the class of implementations and may give rise to additional dependences between errors.

Why do we require that the operator be able to press buttons sufficiently quickly after obtaining a trace, although he should not do this always? If the operator cannot press a button sufficiently quickly after a trace, then the implementation can have time for executing

one or several τ-transitions after this trace. Hence, a transition by a button that starts in the state before these τ-transitions will never be executed.

8.4. Why Pressing a Button Does not Block τ-Activity?

Here we again proceed from the requirement of executability of every reachable transition. If pressing a button blocks τ-activity, then, in order that the implementation could execute a certain chain of τ-transitions (and, after them, a transition by a button), the operator should not press a button until this chain is executed and should press the button immediately after the execution of this chain. Since the τ-activity is unobservable, the operator should just wait for some period of time before pressing a button. Thus quite nontrivial requirements are made to the operator concerning the efficiency of his work: after obtaining a trace, he should sustain a pause before pressing a button; in general, the duration of this pause should be arbitrary in different testing sessions.

Instead, we have chosen a variant when pressing a button does not block the τ-activity. Then a single requirement is made to the operator, which was considered in the previous subsection: the operator should be able to press a button sufficiently quickly after obtaining a trace, although he should not do this always. The implementation will have a choice: to execute either a τ-transition or a transition by a pressed button. As usual, this choice is nondeterministic and is determined by weather conditions.

8.5. Why Pressing a Button Does not Block Divergence?

Although pressing a button does not block τ-activity, it allows only a finite τ-activity, i.e., it allows one to execute only a finite number of τ-transitions. Thus, pressing a button blocks divergence. This is necessary to implement "exit from divergence, i.e., the priority of a test action over divergence.

8.6. Why A Transition by Every Button Is Define in Each State of Implementation?

Until now we assumed that a transition by a button is defined in each state of the LTS implementation (by default, the absence of such a transition is interpreted as the presence of a loop transition). In fact, this requirement is not too essential; if we omit it, this will affect only the execution condition of τ-activity under a pressed button. The new condition is as follows: the implementation may execute no transition by a pressed button p only if, after a finite number of τ-transitions, it will move infinitely along an infinite τ-path that passes only through those states where there are no transitions by the button p. Otherwise the implementation executes a finite number of τ-transition and then a transition by the button p.

An LTS implementation in which transitions by buttons are not defined in all states can be simulated by an LTS in which such transitions exist in all states. To this end, the following changes are made in the original LTS implementation.

1. If a transition by the button p is missing in the stable state s, then a *deadlock* arises: the implementation cannot execute a transition by p or a τ-transition because there are no such transitions and cannot execute a transition by observation because such transitions are blocked by the pressed button p and can be unblocked only after a transition by p. Outwardly (for the operator of the testing machine), such a *deadlock* looks like the absence of observations. One can exit from such a *deadlock* by pressing a different button, by which there is a transition in the stable state s. In our model, this is implemented by adding a transition $s \xrightarrow{p} s'$ leading to a new state s' in which loop transitions $s' \xrightarrow{q} s'$ are defined by all the buttons q by which there are no transitions from the state s, as well as transitions by buttons by which there are transitions from the state s that lead to the same state to which they lead from the state s: a transition $s_p \xrightarrow{r} t$ is executed when there is a transition $s \xrightarrow{r} t$.

2. If there is no transition by the button p in an unstable state s in which an infinite τ-path does not start that passes only a finite number of times through the states in which there is a transition by the button p, then some transition by p will be executed after a finite number of τ-transitions. It suffices to add an arbitrary transition $s \xrightarrow{p} t'$ if the state t can be reached from the state s by τ-transitions and there is (or is added by the first change) a transition $t \xrightarrow{p} t'$.

3. If a transition by the button p is missing in a divergent state s in which an infinite τ-path starts that passes only a finite number of times through the states in which there is a transition by the button p, then it is possible that the implementation will pass precisely through this infinite path. In this case, no transition by p may be executed, and transitions by observations remain blocked. Outwardly (for the operator of the testing machine), such divergence looks like the absence of observations. One can exit from such divergence by pressing another button for which the condition of this subsection is not fulfilled. This is completely analogous to Subsection 1: when simulating in our model, the same changes in the implementation are valid that are described in the first change.

REFERENCES

1. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Formalization of test experiments, *Programmirovanie*, 2007, no. 5, pp. 3–32 [*Program. Comput. Software* (Engl. Transl.), vol. 33, no. 5, pp. 239–260].

2. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., *Teoriya sootvetstviya dlya sistem s blokirovkami i razrusheniem* (Conformance Theory for Systems with Blockings and Destruction), Moscow: Nauka, 2008.
3. Bourdonov, I.B., and Kossatchev, A.S., Systems with priorities: Conformance, testing, and composition, *Tr. Inst. Syst. Program*, 2008, no. 14.1.
4. Bourdonov, I.B., and Kossatchev, A.S., Systems with priorities: Conformance, testing, and composition, *Programmirovanie,* 2009, no. 4, pp. 24–40 [*Program. Comput. Software* (Engl. Transl.), vol. 35, no. 4, pp. 198–211].
5. Bourdonov, I.B., *Teoriya konformnosti (Funktsional'noe testirovanie programmnykh system na osnove formal'nykh modelei)* (Conformance Theory: Functional Testing of Software Systems on the Basis of Formal Models), Saarbrucken: LAP LAMBERT Academic Publ., 2011.
6. Bourdonov, I.B., and Kossatchev, A.S., Elimination of Nonconformal Paths from Specification, *Preprint of Inst. of System Programming*, Moscow, 2011, no. 23.
7. Bourdonov, I.B., and Kossatchev, A.S., Specification completion for *ioco*, *Programmirovanie,* 2011, no. 1.
8. Bourdonov, I.B., and Kossatchev, A.S., Dependence between errors on the classes of tested implementations, *Tr. Inst. Syst. Program.*, 2013, no. 23.
9. Bernot, G., Testing against formal specifications: A theoretical view, *TAPSOFT'91*, 1991, vol. 2, pp. 99–119, Abramsky, S. and Maibaum, T.S.E., Eds., *Lecture Notes in Computer Science,* Springer, 1991, vol. 494.
10. van Glabbeek, R.J., The linear time—branching time spectrum, *Proc. of CONCUR'90*, Baeten, J.C.M. and Klop, J.W., Eds., *Lect. Notes Comput. Sci.*, Springer, 1990, vol. 458, pp. 278–297.
11. van Glabbeek, R.J., The linear time—branching time spectrum II: The semantics of sequential processes with silent moves, *Proc. of CONCUR'93* (Hildesheim, Germany, 1993), Best, E., Ed., *Lect. Notes Comput. Sci.*, Springer, 1993, vol. 715, pp. 66–81.
12. Hoare, C.A.R., Communicating sequential processes. in *On the Construction of Programs—An Advanced Course*, McKeag, R.M. and Macnaghten, A.M., Eds., Cambridge: Cambridge Univ. Press, 1980, pp. 229–254.
13. Revised Working Draft on "Framework: Formal Methods in Conformance Testing," JTC1/SC21/WG1/ Project 54/1, in *ISO Interim Meeting /ITU-T*, Paris, 1995.
14. Tretmans, J., Conformance testing with labelled transition systems: Implementation relations and test generation, *Comput. Networks ISDN Syst.*, 1996, vol. 29, no. 1, pp. 49–79.
15. Tretmans, J., Test generation with inputs, outputs and repetitive quiescence, in *Software-Concepts and Tools*, 1996, vol. 17, issue 3.
16. da Silva Simao, A., Petrenko, A., and Yevtushenko, N., Generating Reduced Tests for FSMs with Extra States, *TestCom/FATES 2009: 129–145.*
17. Petrenko, A., and Yevtushenko, N., Testing from partial deterministic FSM specifications, *IEEE Trans. Comput*, 2005, vol. 54, no. 9, pp. 1154–1165.

Translated by I. Nikitin