ISSN 0361-7688, Programming and Computer Software, 2007, Vol. 33, No. 5, pp. 239–260. © Pleiades Publishing, Ltd., 2007.
Original Russian Text © I.B. Bourdonov, A.S. Kossatchev, V.V. Kuliamin, 2007, published in Programmirovanie, 2007, Vol. 33, No. 5.

Formalization of Test Experiments

I. B. Bourdonov, A. S. Kossatchev, and V. V. Kuliamin

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

Received February 20, 2007

Abstract—Formal methods for testing conformance of the system under examination to its specification are examined. The operational interaction semantics is specified by a special testing machine that formally determines the testing capabilities. A set of theoretically powerful and practically important capabilities is distinguished that can be reduced to the observation of external actions and refusals (the absence of external actions). The novelties are as follows. (1) Parameterization of the semantics by the families of observable and not observable refusals, which makes it possible to take into account various constraints on the (correct) interactions. (2) Destruction as a forbidden action, which is possible but should not be performed in the case of a correct interaction. (3) Modeling of the divergence by the Δ-action, which also should be avoided in the case of a correct interaction. On the basis of this semantics, the concept of safe testing, the implementation safety hypothesis, and the safe conformance relation are proposed. The safe conformance relation corresponds to the principle of independent observations: a behavior of an implementation is correct or incorrect independently of its other possible behaviors. For a more narrow class of interactions, another version of the semantics based on the ready traces may be used along with the corresponding conformance relation. Some propositions concerning the relationships between the conformance relations under various semantics are formulated. The completion transformation that solves the problem of the conformance relation reflexivity and a monotone transformation that solves the monotonicity problem (preservation of the conformance under composition) are defined.

DOI: 10.1134/S0361768807050015

1. INTRODUCTION

In the broadest sense, the correctness of a system is defined as its conformance to the requirements. Such a conformance will be called real. To verify the conformance using formal methods, the objects and relationships of the real world are mapped to model mathematical objects and relations. The model of the system under examination is called its implementation, the model of the requirements is called the specification, and the conformance is mapped to the model conformance. The model conformance is interpreted as an ordinary mathematical relation, that is, as a subset of the Cartesian product of the sets of implementations and specifications.

The verification of conformance is based on the assumption that the model is "correct." This enables us to assume that the real and the model conformances are equivalent: a system conforms to the requirements if and only if its implementation considered as a model of the system conforms to the specification considered as a model of the requirements. Certainly, this assertion is tautological in the sense that if we wanted to give a formal definition of the model correctness, it would be reduced to the equivalence of the real and the model conformances. Thus, the verification of conformance would verify exactly the same thing that is must assume. To break this vicious circle, we must understand the nature of modeling.

Modeling of the requirements is the process of their refinement and formalization. Although the result is a formal description of the requirements (the specification), the very process can hardly be formalized because it establishes a relationship between the objects of different nature—informal and formal ones. Only intuition can decide whether or not the informal requirements are correctly written in the form of a formal specification. If the intuition lets us down, it can lead (and often does) to the detection of phantom errors in the system under examination. Then, the model of the requirements must be revised; i.e., an error should be found in the specification rather than in the system itself. After the specification has been revised, the verification of conformance must be repeated.

Modeling the conformance relation has the same nature: this is a refinement and formalization of the intuitive understanding of the claim that the "system conforms to the requirements." Formalization also assumes that we abstract ourselves from the details of the system, the requirements, and their relationships that are immaterial from the viewpoint of the system correctness interpreted intuitively.

To be able to verify the conformance, we must be given the specifications and the conformance relation in some formal form. If the implementation considered as a model of the system is also available, then the static analytical verification is possible. Such a verification is

reduced to checking if the pair <implementation, specification> belongs to the admissible set of such pairs defined by the model conformance relation.

In contrast to the requirements and the real conformance, the system under examination is usually an ordinary software and (or) hardware system, which can be interpreted quite formally at a certain level of abstraction. Thus, a mapping of the system to its implementation relates two formal objects, which theoretically enables one to define (and perform) this mapping formally. Sometimes, such a mapping is actually possible and is used in practice. However, the system and its implementation are usually described at very different levels of abstraction; this makes it practically impossible to perform the analysis and the formal transformation into the implementation for complex systems. When constructing the model, we abstract ourselves from the greater part of the details of the system organization and its behavior and distinguish between important and unimportant details from the viewpoint of the given specification and the model conformance. When the model is constructed informally or half formally, we obtain one more source of phantom errors. For this reason, one often has to abandon the system analysis and acknowledge that no implementation is available and, at best, we know only a *class of possible implementations*. Moreover, it often happens in practice that the system cannot be formally analyzed at all. For example, these are remote systems that can be only accessed through a communication channel, or programs whose source code is unavailable (formally, the machine code can be analyzed, but it is rarely possible in practice), or hardware considered as a black box.

Here, we face the question: how the conformance can be verified if the implementation is unavailable? This is possible only if the requirements for the system are represented in terms of its interaction with the environment. Then, it becomes possible to perform testing (dynamic verification) as an experimental verification of conformance. A test replaces the environment and observes the system behavior while interacting with it. By practical considerations, the test must terminate in a finite time, which is not necessarily the case for the arbitrary environment. For this reason, a set of tests is used to simulate an arbitrary behavior of the environment; theoretically, such a set may be infinite. Certainly, in practice only finite test suites are used; such a testing is complete (that is, it correctly determines the conformance or inconformance) only under certain constraints on the system being tested. These constraints represented in the model form refine the class of possible implementations; they are called *implementation hypotheses* or *error models* if they reduce to listing the errors (types of inconformances) that can be and cannot be encountered in the system.

Now, we naturally face the problem of formalizing the testing process. For this purpose, we must first assume that a model of the system (its implementation) exists (even when it is not known). This is the so-called *test hypothesis* [1]. Furthermore, we must formalize the interaction and simulate real tests by model tests. In the model, the formal relation *an implementation passes a suite of (model) tests* is defined. A suite of model tests is said to be *complete* for the given specification if it satisfies the following condition: an implementation conforms to the specification if and only if this implementation passes the given suite of tests. In the model world, we first must *prove* the existence of a complete suite of tests for each specification (at least, for each specification in the given class of specifications) and a certain class of possible implementations. Second, we must find a method for *test generation*.

For practical purposes, we must determine a procedure for *translating* the model tests into real tests. Here, we face the situation that is converse of the situation of modeling the system or the requirements: we do not determine a model of a real object but rather construct a real object using its model. Then, the complete suite of real tests (that is, the tests that are obtained by translating the complete suite of model tests) is performed on the system under examination. In real world, the relation *the system passes the suite of (real) tests* is also defined. Finally, we conclude that the system conforms to the requirements if and only if it passes the complete suite of real tests. It is clear that this conclusion is based on the assumption that not only the requirements and the conformance relation are modeled correctly but also the translation of the tests and the mapping of the real relation to the model one are *performed* correctly.

Below, we assume that the test model and the translation are correct. We rather focus on the model world while keeping in mind the real world, which is the source of all the practical restrictions that must be taken into account.

Formalization of the interaction is the key point in conformance testing. The conformance relation and the admissible classes of implementations and specifications directly depend on the kind of interactions that is examined. We define the operational semantics of the interaction using the so-called testing machine. The interaction of an implementation with the environment always satisfies some constraints. Of concern to us are only the interactions that are admissible in practice. This depends, first, on the operational situation in which the interaction occurs and, second, on the requirements for the environment. The former means that the environment *cannot* and the latter means that the environment *must not* interact with the implementation in an arbitrary way. In other words, some interactions do not occur, and some other interactions are of no concern to us because we verify the correctness of the implementation but not its environment. Details of the operational situation and the requirements apart, we describe the constraints on the interaction using an appropriate testing machine. Such a machine determines the abstraction level and the constraints on the

interactions under which the conformance relation is formulated in terms of such an interaction. From the viewpoint of testing, the machine determines the available capabilities for controlling the interaction and for observing the implementation behavior. In other words, the conformance relations that are of concern to us are the relations that can be verified using the testing capabilities provided by the testing machine; moreover, the conformance relations of interest are such that all the available testing capabilities are necessary for the verification (there are no redundant capabilities).

We will analyze the testing capabilities from three viewpoints:

(1) what do *we want* to verify;

(2) what *can be theoretically* verified using the available testing capabilities (the testing power);

(3) which testing capabilities and under which conditions *can we count on in practice*.

Let us consider the first viewpoint in some detail. It directly determines the type of the conformance relations of interest under the given testing capabilities. The case in point is which behaviors of the implementation are considered correct by the specification and which are not. We assume that the implementation satisfies the principle of independence: any behavior of the implementation in a certain situation is either correct or incorrect independently of its other behaviors. In this case, we may assume that, in each situation t, the specification determines a set of *admissible* behaviors $\Sigma_p(t)$. If the implementation has the set of behaviors $\mathbf{I}(t)$, then the conformance is a preorder (a transitive reflective relation); therefore, we have $\mathbf{I}(t) \subseteq \Sigma_p(t)$. For each observable behavior of the implementation $i \in \mathbf{I}(t)$, we must verify that $i \in \Sigma_p(t)$. Such a behavior is observed in a single test run; therefore, we may conclude after the test run has completed whether it *passed* or *failed*. An implementation passes a test if any run of this test leads to the verdict *pass* (the test never fails). An implementation is said to pass a suite of tests if it passes each test in the suite.

Thus, we do not consider the conformances that cannot be verified by analyzing every behavior individually but require an analysis of the set of observable behaviors. In particular, we do not consider the conformances that require that some behaviors are present in the implementation. If a specification requires a set of *mandatory* behaviors $\Sigma_r(t)$ in the situation t, then the conformance requires the reverse inclusion $\mathbf{I}(t) \supseteq \Sigma_r(t)$. The verification of such a conformance requires that the entire set of the observable behaviors $\mathbf{I}(t)$ be analyzed and that, for each $s \in \Sigma_r(t)$, the inclusion $s \in \mathbf{I}(t)$ be verified. Equivalences provide an example of such conformances: each admissible behavior must be present in the implementation; i.e., $\Sigma_r(t) = \Sigma_p(t)$. For such kind of conformances, it is insufficient to make a conclusion after each test run: the conformance predicate is not a conjunction of the verdicts returned at the end of each test run (if we assume that *pass* = *true* and *fail* = *false*).

We consider two known testing machines proposed by Milner in [2] and by van Glabbeek in [3, 4] with various modifications. Simultaneously, we define a machine with a constrained control, which provides the minimal testing capabilities that are available in a greater part of practical cases. The other testing capabilities are either redundant (they enable us to verify something that does not need to be verified), or are not theoretically studied, or are impractical.

As an example, we will use a particular case of interactions that is reduced to the exchange of discrete portions of information (messages) between the implementation and the environment. Such systems are called input–output systems (see [5–7]). The messages that are sent from the environment to the system are called *inputs*, and the messages sent from the system to the environment are called *outputs*. Using the notation of the calculus of communicating systems (CCS) [8, 9], we denote the inputs by ?m and the outputs by !m, where m is the message symbol.

2. IMPLEMENTATION AS A MODEL OF THE SYSTEM BEING TESTED

Our purpose is to formalize the interaction of the implementation with the environment. Therefore, the internal structure of the implementation is of no interest. We are only interested in its external interface that makes it possible to apply test actions and observe its external behavior (that is, the behavior that manifests itself in the interactions). In accordance with this approach, the testing machine can be thought of as a black box with the implementation being tested inside. This box is equipped with various devices for performing test actions and for observing the external behavior of the implementation. These actions and observations are performed by the machine operator whose behavior simulates the behavior of the environment. A test may be interpreted as a variant of the behavior of the environment; therefore, we may assume that the operator's behavior is determined by the test. The difference between the test and the corresponding environment is that the test is succeeded by the verdict *pass* or *fail*.

For the illustration purposes, we will use the model of the system and of its environment that is called the labeled transition system (LTS). This is a directed graph with a distinguished initial vertex in which the arcs are labeled by certain symbols. Formally, LTS is the collection $S = LTS(V_S, L, E_S, s_0)$, where V_S is a nonempty set of states (graph vertices), L is the alphabet of symbols called external actions, τ is the symbol called the internal action, $E_S \subseteq V_S \times (L \cup \{\tau\}) \times V_S$ is the set of transitions (labeled arcs), and $s_0 \in V_S$ is the initial state (the initial vertex of the graph). The transition from the state s to the state s' under the action z is denoted by $s \xrightarrow{z} s'$. Introduce the notation $s \not\xrightarrow{z} =_{def} \nexists s'\ s \xrightarrow{z} s'$. Note that different LTSs can be indistinguishable from

the viewpoint of testing if they demonstrate the same external behavior under any interaction.

The external behavior is considered as a sequence of discrete external actions such that the operator can observe the system's response to these actions. The alphabet L of the *external* actions is specified for the machine. One may assume that the external action $z \in L$ is a notation for the set of actions of the machine that are observed by the operator as z; i.e., all these actions are indistinguishable from the operator's viewpoint. However, the execution in the same situation of one or another action that are perceived as z from outside does not mean that the results of performing these actions are also indistinguishable. We also note that fixing the alphabet L does not imply that the system under test cannot execute some other actions that can be observed from outside. We just are not interested in these actions and do not include them into L for this reason. For example, when testing the methods of an object that evaluate numerical functions, we may not be interested in the other methods defined in the object's class. In the LTS model with the alphabet L, the execution of the external action $z \in L$ is associated with the transition $s \xrightarrow{z} s'$. Being in the state s, the LTS performs this transition, which is observed by the operator as the execution of the action z, and goes to the state s'. The states s and s' are not observable.

In addition to the external observable actions, the machine may have some *internal activity* that is not observable externally (in the interactions). It is denoted by τ. The internal activity can be finite (when it terminates in a certain time) or infinite. The infinite internal activity is called *divergence*. Without loss of generality, we can assume that the τ-activity is a sequence of discrete τ-actions (naturally, also internal and unobservable). Thus, τ denotes the set of all internal actions that are not observed by the operator and, therefore, are indistinguishable for him. The finite τ-activity is a finite sequence of τ-actions and the infinite activity is an infinite sequence of such actions. In the LTS model, the performance of the τ-action is associated with the transition $s \xrightarrow{\tau} s'$.

We assume that the execution time of any external or τ-action is finite and bounded below by a nonzero value. Then, the execution of any finite sequence of actions takes a finite time, while the execution of any infinite sequence of actions (in particular, any divergence) takes an infinite time.

Interaction always assumes that both parts—the implementation and the environment—are involved. For example, in the input–output systems, sending a message implies that the environment sends an input and the implementation receives it or the implementation sends an output and the environment receives it. Sometimes, the implementation is said to initiate an output and, when it is sent, the environment must receive it. In other cases, this is formulated in another way: the implementation cannot reject the input sent by the environment. This can be considered as a constraint imposed on the admissible behavior of the environment or the implementation. In general, such constraints do not violate the general principle of the reciprocity of interactions.

As applied to the testing machine, this principle means that the machine can execute only the actions that are defined in the implementation and are allowed by the operator. In the LTS implementation, the action z is defined if the current state s contains at least one transition $s \xrightarrow{z} s'$. The test action assumes that the operator specifies which actions are enabled. The τ-actions, which do not take part in the interaction, are always enabled. The permission to execute external actions can also be interpreted such that the operator instructs the machine to perform one action at its choice. For example, the environment receives all the outputs, but the implementation decides which output is sent.

Although only defined and enabled actions can be performed, not all such actions are generally executable. The executability of the action z is a predicate on the set of defined actions and the set of enabled actions. This predicate may be different at different time instants (at different states of the LTS implementation) and for different actions (for different transitions in the LTS implementation). Thus, there may be priorities for performing actions in the machine. In this paper, we usually consider only the *machines without priorities*: every defined and enabled action is executable.

The rule of nondeterminstic choice. If there are several executable actions, only one of them is actually executed; this action is chosen nondeterministically. This is the reason of the possible system nondeterminism: we abstract ourselves from the factors ("weather conditions") that determine the choice of one or another action. Note that even when all the executable actions are labeled by the same symbol ($z \in L$ or τ), they may be different; in particular, they may have different observable consequences. In the LTS implementation, this implies that there may be several transitions $s \xrightarrow{z} s'$ defined for the same current state s; these transitions differ only by there final state s'.

For a machine without priorities, the choice of the action to be performed is modeled in the LTS using the composition operator that is borrowed from an algebra of processes. It is convenient for us to use the composition operator from the calculus of communicating systems (CCS) [8, 9]. For this purpose, an involution (a bijection that is inverse of itself) is defined on the set of external actions. It is denoted by the underscore. This involution assigns to each external action z a *reverse* action $\underline{z}$ such that $\underline{z} = z$. Note that $z \in L$ does not necessarily imply $\underline{z} \in L$. The result of the composition of two LTSs $S = LTS(V_S, L, E_S, s_0)$ and $T = LTS(V_T, M, E_T, t_0)$ is a third LTS $S \upharpoonright\downarrow T = LTS(V_S \times V_T, L \upharpoonright\downarrow M, E, s_0t_0)$. Its states are the pairs of the states of the LTS operands,

and the initial state is the pair of the initial states. The alphabet of the composition consists of the actions belonging to one of the LTS operands for which there are no inverse actions in the alphabet of the other LTS operand: $L \pitchfork M = (L\backslash\underline{M}) \cup (M\backslash\underline{L})$. The transitions in the composition are classified as synchronous and asynchronous. An asynchronous transition corresponds to a transition in one of the LTS operands that is labeled by a symbol in the composition alphabet or by τ. Only the state of this LTS operand can be changed. A synchronous transition corresponds to a pair of transitions in the LTS operands corresponding to mutually inverse actions that becomes a τ-transition in the composition. In this case, the states of both LTS operands can be changed. Formally, the set of composition transitions E is the least set generated by the following inference rules:

$$l \in (L\backslash\underline{M}) \cup \{\tau\} \quad \&s \xrightarrow{l} s' \quad \vdash st \xrightarrow{l} s't,$$

$$m \in (M\backslash\underline{L}) \cup \{\tau\} \quad \&t \xrightarrow{m} t' \vdash st \xrightarrow{m} st',$$

$$z \in L \cap \underline{M} \quad \&s \xrightarrow{z} s' \,\&t \xrightarrow{\underline{z}} t' \vdash st \xrightarrow{\tau} s't'.$$

If the operator in a machine with the LTS implementation S enables the action $z \in L$, then this fact corresponds to the transition $t \xrightarrow{\underline{z}} t'$ in the LTS test T. If the operator can *change its mind* and enable another set of external actions without waiting for the external action to be performed, then this fact corresponds to the τ-transition $t \xrightarrow{\tau} t'$ in the LTS test T. It is usually assumed that the system <implementation–test> is closed during testing; i.e., the implementation and the test interact only with each other and do not interact with the rest of the external world. In this case, $M = \underline{L}$ and $L \pitchfork M = \varnothing$; i.e., the composition $S \pitchfork T$ includes only τ-transitions (inherited from the operands or synchronous transitions).

Note that, in practice, a real test (more precisely, a tester that performs the test) can interact with the system being tested via a certain communication environment rather than directly. A standard example is the environment consisting of a queue of inputs and a queue of outputs in input–output systems. In addition, the system under test can interact not only with the test and (or) communication environment but also with another part of the environment. For example, we can be unable to intercept all the external interactions of the system. In this case, we have to assume that, in the model world, the composition of the implementation with the communication environment and with another part of the environment is tested. In this case, it is said that the implementation is embedded in a test context, and the testing is said to be *asynchronous* or *testing in the context*. In fact, the very conformance relation is changed in this case. The tester can also receive commands from *above* (change the testing mode or abnormally terminate the process). In the model world, this can be interpreted as the replacement of the test by the composition of this test with a "higher authority." Below, we assume that the system <implementation–test> is closed.

3. CONTROL AND OBSERVATION OF EXTERNAL ACTIONS

In terms of the testing machine, a test action reduces to the specification of the set of enabled external actions. The way used to specify this set depends on the machine organization.

The van Glabbeek machine is called *generative*: each external action $z \in L$ is assigned a switch that has two states—*free* (the action is enabled) and *blocked* (the action is disabled). Any switch may be set to any state. The set of switches in the state *free* is the set of enabled actions $P \subseteq L$.

The Milner machine is called *reactive*. It works by orders: for each external action $z \in L$, there is a button; when this button is pressed, only the action z is enabled to be executed. This imposes a restriction on the operation of the environment: in the LTS model of the test T, not more than one transition $t \xrightarrow{\underline{z}} t'$ is allowed, where $z \in L \cap \underline{M}$. The absence of transitions means that at the moment (at the current state of the test) the operator does not press any buttons. Upon the execution of an external action, the machine *stops* until the next button is pressed.

We also define the testing machine with *restricted control*, which will also be called the *parameterized* machine. As the reactive machine, it operates by orders; however, when a button is pressed, a set of external actions $P \subseteq L$ is enabled. Such a button is denoted by "P." It is assumed that each external action $z \in L$ is enabled at least by one button; that is, there exists a button "P" such that $z \in P$. This machine is called a machine with *restricted* control because not every set $P \subseteq L$ is associated with a corresponding button "P." The machine is parameterized by a set of buttons, that is, by a family of subsets of the external actions $\mathcal{R} \subseteq \mathcal{P}(L)$. A pressed button is automatically released when the machine executes an external action or the operator presses another button.

Let us compare the three machines described above. A parameterized machine can be interpreted as a modification of a generative machine in which not every state of the switches is allowed. If a reactive machine is modified by allowing several buttons to be pressed simultaneously, such a machine will operate as a generative machine. Taking into account these modifications, all the three machines can be considered equivalent with a single exception.

This exception is the behavior of the machine after an external action is executed and before another button is pressed or the states of the switches are changed. The reactive machine stops until the same or other buttons are pressed. The generative machine continues to operate by executing internal actions and the external

actions enabled by the switches in the *free* state. In the parameterized machine, one more variant is implemented: the machine continues to operate but only the internal actions can be executed (the button is not pressed). Let us discuss how the behavior of one machine can be simulated by another machine.

Van Glabbeek showed that the stop of a reactive machine can be simulated in a generative machine by a special on/off switch that blocks the execution not only of all the external but also internal actions. To imitate such a behavior, the operator must press the blocking switch immediately after observing an external action and remove the block after setting the main switches. Note that the operator imitates the environment that can be very quick or very slow. Therefore, the imitation of the behavior of a reactive machine does not actually add anything and leaves the behavior inherent in generative machines. It is clear that the on/off switch does not increase the power of testing.

Conversely, the behavior of a generative machine when at least one external action is enabled is simulated in the reactive machine by a sequence of presses of one and the same set of buttons. Van Glabbeek proposed to simulate the ban of all the external actions (all the switches are in the *blocked* state) in a reactive machine by pressing the button of a special odd action that is never executed. This odd action is added to the alphabet L and the corresponding button is added to the machine.

It is sufficient to show how the behavior of the generative machine can be simulated in a parameterized machine and conversely. The behavior of the generative machine is simulated in a parameterized machine by quickly pressing the same button after the corresponding external action is executed. Note that it does not matter if the operator fails to press the button sufficiently quickly because the machine (*without priorities*) has only time to execute one or several τ-actions, which it also can execute if the button is pressed immediately. In other words, we require that the operator is able to work quickly but do not make him work quickly. This is in accordance with the requirement that the operator must be able to simulate arbitrary rate of the environment operation.

To simulate the behavior of a parameterized machine in a generative machine, a modification of the generative machine mentioned by van Glabbeek can be used. In this modification, the switches are automatically set to the *blocked* state each time an external action is executed. This exactly corresponds to the property of the parameterized machine requiring that only τ-actions may be executed between the execution of an external action and pressing a new button.

In the generative and parameterized machines, there is a display used to observe the external actions. When the machine executes the external action z, the symbol z is shown on the display. In the parameterized machine, the display is cleared when the next button is pressed. Since the generative machine operates continuously, it is possible to observe the sequence $\langle z, z, z, \ldots\rangle$ without changing the states of the switches. To enable the operator to differentiate between different executions of the same action, the display must die away for a short period of time t_0 between two successive actions. There is no display in the reactive machines, but the button corresponding to the action being executed sinks, the machine stops, and, in order to continue testing, the operator must relieve this button, and then press the same or another button (or several buttons in the modified machine). Obviously, these differences are insignificant.

Summing up, note two important points (for definiteness, we use the terminology of the parameterized machine).

Toggling without observation. In a machine without priorities, the possibility to execute τ-actions is independent of the set of the enabled external actions (the states of the buttons and the switches). Therefore, it is senseless to press buttons without observation: this does not increase the power of testing. Indeed, if the button "P" was pressed and then another button "Q" was pressed without observing the effect of the first button, the machine could execute only τ-actions in this interval of time. However, the same τ-actions could be executed if the button "Q" was pressed once without pressing "P." Thus, any behavior (the trace of external actions) that can be observed in the first case can also be observed in the second case.

In the presence of priorities, toggling without observation is necessary for the completeness of testing because different sets of enabled actions differently affect the execution of τ-actions, which results in the behaviors that look differently from outside.

Testing log. To accumulate information about testing, the operator could write down the sequence of his actions and observations. Such a sequence of pressed buttons and observed external actions will be called the *testing log*; its subsequence consisting of the observed external actions will be called the *observation trace* or simply the *trace*. In the general case, everything we can learn from testing is reduced to the set of all possible testing logs. In the presence of priorities, traces are not sufficient, and one has to use logs.

However, in a machine without priorities, the possibility to execute an external action does not depend on which other external actions are enabled. If the action $z \in P$ was observed after the button "P" was pressed, it also could be observed after pressing any other button "Q" such that $z \in Q$. Therefore, the set of all testing logs can be unambiguously reconstructed from the set of observation traces. Such a set of traces will be called the *trace model of the implementation* or simply the *trace implementation*. In the LTS model I, a trace is defined as a sequence of external actions that label the transitions in an LTS route that begins at the initial state (the τ-actions are omitted in this sequence). LTS implementations are distinguishable in the interaction only

accurate to their trace model $\mathbf{I} = traces(I)$. Therefore, a specification can be considered as a description of the requirements for the set of implementation traces. For the conformance type used in this paper, the specification determines the set of admissible traces that must include all the traces of any conformal implementation. The specification itself can be described as the set of traces of the corresponding LTS $\Sigma = traces(S)$. This set will be called the *trace model of the specification* or simply the *trace specification*. It is assumed that the trace specification unambiguously determines the set of admissible traces, although it may be not identical to this set, which will be seen later.

The trace model (both the model of implementation and the model of specification) is a (*prefix-closed*) tree, which means that if a trace is observed (enabled), then its any prefix is also observed (enabled). Therefore, the testing may proceed up to the first error: if, upon the observation of an admissible trace σ, an external action z is observed and the trace $\sigma \cdot \langle z \rangle$ is not admissible, then the testing may be finished with the verdict *fail*. Indeed, any continuation $\sigma \cdot \langle z \rangle \cdot \lambda$ is also not admissible. Moreover, we will see later that the continuation of testing in this situation can be unsafe.

4. MACHINE STOP AND OBSERVATION OF REFUSALS

When there are no executable actions, the machine stops. If the operator can detect this fact, he gets a new kind of observations: the machine in the state in which no actions can be executed and the enabled actions form the set *P*. This set *P* is called the refusal set. From now on, we will include in the testing logs and the observation traces not only the actions from the alphabet *L* but also the observable refusal sets as subsets of the set of actions $P \subseteq L$.

A machine without priorities can stop only if there is no internal activity. An LTS implementation stops at the state s when there are no τ-transitions $s \overset{\tau}{\not\longrightarrow}$; such a state is called stable (quiescent). A refusal P can also be observed in a quiescent state s such that $\forall z \in P$ $s \overset{z}{\not\longrightarrow}$. The rule *observation without toggling* is corrected accordingly: not only external actions but also refusal sets are considered to be observations. The possibility to reconstruct testing logs from traces with refusals also remains open because the refusal set *P* can be observed only when the button "*P*" is pressed. To obtain traces with refusals (*failure traces*) from an LTS implementation, it is sufficient to add the refusal sets to the alphabet of external actions of the LTS, add loop transitions corresponding to the observable refusals in each quiescent state, and take the set of traces of external actions of the resulting LTS: $\mathbf{I} = Ftraces(I)$.

To observe refusals, there is a green light in the generative and reactive machines that is on when the machine executes some external or internal action; the light goes out when the machine stops. In a generative machine, the refusal set is calculated as the set of actions written on the switches having the state *free*; in a reactive machine, this set is calculated as the set of actions written on the buttons pressed by the operator. There is another mode of operation of the green light. In this mode, it is on only when an internal action is executed. If the green light goes out, this is either a stop or the execution of an external action. In a reactive machine, the stop is detected if no buttons are selected. In a generative machine, the stop is detected if the display is blank for a time $t > t_0$; this is required to make certain that this is a stop rather than an interval between the execution of two external actions.

Although both modes of operation of the green light can be used to detect the stop, they differ by the power of testing. The second mode makes it possible to separately observe the internal activity (a nonempty sequence of internal actions) when the green light is on and the display is blank. Note that the internal actions are indistinguishable; in particular a single τ-action and any finite sequence of τ-actions cannot be distinguished. If we denote the observation of the internal activity by τ, then the traces $\langle z, \tau, z' \rangle$ and $\langle z, z' \rangle$ are distinguishable, and the traces $\langle z, \tau, P \rangle$ and $\langle z, P \rangle$, where z, $z' \in L$ and $P \subseteq L$, are also distinguishable, although they are indistinguishable in the first mode. The practical efficiency of this possibility depends on the conditions of the interaction. For example, when a program is run on a local computer and does not respond for a long time, we can look if it uses the processor or modifies some variables or files. On the other hand, if the program is intended for remote interaction, there is no such possibility when the testing is remote. However, a more important question is if we really need such a testing possibility. It is difficult to imagine a situation when the presence or absence of the internal activity between an external action (or the start of the program) and the next external action or refusal must be interpreted as an error.

To detect the stop, it is not necessary to be able to observe the (internal) activity of the implementation. For example, assume that the time of performing any external action and any finite sequence of internal actions is bounded above by a known quantity t_1. Then, when the timeout $t > t_1$ expires in the absence of external actions, we may conclude that there is either the divergence or the stop. If we are sure that there is no divergence, then this is the stop. In the input–output systems, such a timeout can be established by receiving all the outputs without sending any inputs. The corresponding refusal set is called the *quiescence*; it is denoted by δ [7, 10]. In the LTS implementation, such a refusal is observed in the *quiescent state* in which there are no output transitions. If sending the input $?x$ assumes a reliable (error-free and, therefore, not requiring testing) delivery notification that must arrive not later than in certain known time and there is no divergence, we can observe the *input refusal* $\{?x\}$. In an LTS implementation, the input refusal $\{?x\}$ can be observed

in a quiescent state in which there are no transitions labeled by ?x. Note that the expiration of the timeout in itself does not make it possible to distinguish between the divergence and the machine stop. One can observe the stop only if there is no divergence. This problem will be discussed below.

There is another possibility. An implementation may inform the machine from within the black box not only about the execution of an external action but also about the stop. For example, in order to observe a quiescent state, one also can use a reliable output absence notification instead of the timeout expiration. To observe the input refusal, one can use the reliable nondelivery notification along with the delivery notification instead of the timeout expiration.

In a parameterized machine, we propose to abstract ourselves from the method used for stop detection. We assume that, if the stop can be detected when the button "P" is pressed, then this is done by the machine itself: it outputs the special symbol θ on the display. In the LTS test, θ must be added to the alphabet, and the capability of defining θ-transitions must be provided. In a composition, the θ-transition is executable if and only if no other transitions can be executed. In a machine without priorities, this requirement is formulated by the following additional inference rule:

$$Deadlock \,\&t \xrightarrow{\theta} t' \vdash st \xrightarrow{\tau} st', \text{ where}$$

$$Deadlock =_{def} s \overset{\tau}{\not\longrightarrow} \,\&t \overset{\tau}{\not\longrightarrow}$$

$$\&\forall z \in L \cap \underline{M}\,(s \overset{z}{\not\longrightarrow} \vee t \overset{z}{\not\longrightarrow}).$$

For a closed <implementation–test> system, we have $M = \underline{L}$ and $L \pitchfork M = \varnothing$. Therefore, when the implementation stops at the state s, any refusal $P \subseteq \{z \in L | s \overset{z}{\not\longrightarrow}\}$ is possible. In the machine that is parameterized by the family $\mathcal{R}$, it also must hold that $P \in \mathcal{R}$.

Now, let us refine the parameterization of the machine taking refusals into account. We assume that some refusals can be observed when the machine is at stop and the others can not. For example, in the input–output systems, we can have a timeout for outputs but no delivery or nondelivery notification of the input. In this case, the quiescent state is observed, but the input refusal is not observed. Such input–output systems are considered, for example, for the conformance relations *iot*, *ioconf*, *ior*, and *ioco*, which will be considered below.

Thus, the family of buttons is subdivided into two subfamilies: the buttons with observable refusals $\mathcal{R} \subseteq \mathcal{P}(L)$, and the buttons with unobservable refusals $\mathcal{Q} \subseteq \mathcal{P}(L)$; together, they cover the entire alphabet: $(\cup(\mathcal{R})) \cup (\cup(\mathcal{Q})) = L$. We assume that $\mathcal{R} \cap \mathcal{Q} = \varnothing$ because, if there is the button "P" with the observable refusal, the addition of the button "P" with the unobservable refusal does not enhance the power of testing. The machine parameterized by the families $\mathcal{R}$ and $\mathcal{Q}$ will be called the $\mathcal{R}/\mathcal{Q}$-machine; we say that this machine determines the $\mathcal{R}/\mathcal{Q}$ – interaction semantics. The traces with refusals in $\mathcal{R}$ will be called $\mathcal{R}$-traces, and the set of such LTS traces will be called the $\mathcal{R}$-model.

The $\mathcal{P}(L)$-model is also called the F-model. For an implementation, its F-model is the set of observation traces on the $\mathcal{P}(L)/\varnothing$-machine, which will be called the F-machine. For the F-model T, the subset of its $\mathcal{R}$-traces is and $\mathcal{R}$-model (!),[1] which will be denoted by $T_{\mathcal{R}} = T \cap (L \cup \mathcal{R})^*$. The converse is also true: any $\mathcal{R}$ – model is a subset of the $\mathcal{R}$-traces of an F-model (!). In what follows, we will interpret the implementation and specification as F-models and denote them by $\mathbf{I}$ and Σ, respectively. Given LTS models, the corresponding F-models are constructed by taking all the traces with refusals: $\mathbf{I} = Ftraces(I)$ and $\Sigma = Ftraces(S)$. Below, we will discuss how the set of admissible traces can be determined from Σ.

Note that neither van Glabbeek nor Milner considered such variants of their machines. In their considerations, either there is a green light and all the refusals are observable, or there is no green light and all the refusals are not observable. The set of observable traces is either the F-model (all the subsets of the alphabet are observable refusals) or the $\varnothing/\{L\}$-model (there are no observable refusals and the traces contain only external actions).

5. INTERACTION PROTOCOL AND SAFE TESTING

The machine stop can sometimes lead to a deadlock in the interaction. For definiteness, we will consider the $\mathcal{R}/\mathcal{Q}$-machine. In the following two cases, there is no deadlock.

(1) The environment can continue working without waiting if the result of the last interaction step is an external action, a refusal, or the last step does not terminate at all. The operator presses a button when there are no observations. There is a τ-transition in the state of the LTS environment/test.

(2) At stop, there is an observable refusal, and the environment responses to it. The operator observes the refusal and then presses (another) button. The state of the LTS environment/test is quiescent and there is a θ-transition defined in it.

A deadlock is a normal completion of the interaction if this situation coincides with the completion of the environment operation. The operator does not and will not press any buttons: all the external actions are disabled, there will be no observations. The LTS environment (test) is in a terminal state (without transitions). The testing completes with the verdict *pass* or *fail*.

[1] The symbol (!) denotes that the proof of the corresponding proposition is omitted due to space limitations.

A deadlock in a nonterminal state is considered to be an interaction error: the LTS environment or test "wants" a continuation, but it is impossible. Note that, in this case, some button is pressed. Such an error can be caused by the implementation or by the environment. Assume that the implementation is not to "blame." Then, if the refusal is observable, then the fault of the environment is that it does not response although it could do that. If the refusal is not observable, then the fault of the environment is that it allowed this situation to happen: the operator must not have pressed the button "*P*" if the refusal *P* is not observable and can occur in this situation. If all the actions of the environment are correct but an interaction error does occur, then the implementation is to "blame:" the pressed button does not make it possible to observe the refusal and the implementation stops. However, the problem is that the interaction errors cannot be found by testing by definition: the possibility to detect such an error would mean that the refusal is observable.

If pressing the button "*P*" after the trace σ cannot cause an interaction error (an unobservable refusal), such a button is said to be *safe in the implementation after the trace* σ. This means that, in the *F*-model of the implementation **I**, the trace $\sigma \in \mathbf{I}_{\mathcal{R}}$ is not continued by the refusal *P* if $P \in \mathcal{Q}$ (in the LTS implementation, the trace σ does not terminate in a quiescent state in which there are no transitions corresponding to the actions belonging to *P*):

$$P \text{ safe in } \mathbf{I} \text{ after } \sigma =_{def} P \in \mathcal{R} \vee \sigma \cdot \langle P \rangle \notin \mathbf{I}.$$

The interaction protocol is a rule that determines which $\mathcal{Q}$-buttons may be and may not be pressed after particular observation traces. The former buttons are called safe, and the latter are said to be *unsafe after the trace in the framework of this protocol*. The $\mathcal{R}$-buttons are declared safe (in the sense that there are no unobservable refusals) after any trace. Many protocols may exist. Of our concern are not arbitrary interactions but rather those that comply to a given *i*th protocol. We say that an implementation corresponds to the *i*th *safety hypothesis* if there are no interaction errors when any environment complying to the *i*th protocol interacts with this implementation. Each *i*th safety hypothesis determines the *i*th class of *safely testable* implementations. It is impossible to verify by testing if an implementation is safely testable within the given *i*th protocol; indeed, such a verification would imply the verification of the presence or absence of interaction errors. Therefore, the corresponding safety hypothesis is a precondition of testing. In turn, the test (the machine operator executing the test) must adhere to the *i*th protocol; such testing will be said to be *safe*.

The conformance or nonconformance of an implementation can be discussed only in this context: each *i*th interaction protocol determines the *i*th class of conformal implementations, which is a subclass of the *i*th class of safely testable implementations. The purpose of safe testing in the framework of the *i*th protocol is to verify the conformance of an implementation provided that it belongs to the *i*th class of safely testable implementations. Note that the implementation that is safely testable in the framework of one protocol can have no such property in the framework of another protocol.

We now give a more precise definition of the conformance as an embedding of unobservable implementation traces admitted by the specification. We want to parameterize the conformance by the *i*th given interaction protocol. First, we are only interested in the implementations that satisfy the *i*th safety hypothesis. Second, among all the implementation traces that can be observed on the given $\mathcal{R}/\mathcal{Q}$-machine, we are only interested in the traces that are observable in the interactions adhering to the *i*th protocol. The specification discriminates (distinguishes?) the subset of admissible traces in the set of all such traces. Now, the admissible traces are not all the traces that can be observed in conformal implementations but only those of them that can be observed under safe testing. We assume that the *F*-model Σ conforms to the specification and the set of admissible traces is the same for any interaction protocol: this is the set of all $\mathcal{R}$-traces $\Sigma_{\mathcal{R}}$. All the other traces that can appear in conformal implementations but are not observed under safe testing are determined by the protocol. In order to make the set of admissible traces identical to $\Sigma_{\mathcal{R}}$, we restrict ourselves to the protocols that satisfy the following rules.

First rule. All the $\mathcal{R}$-buttons are declared to be safe after any $\mathcal{R}$-trace.

Second rule. All the traces in $\Sigma_{\mathcal{R}}$ can be checked by testing. This means that, if the trace $\sigma \cdot \langle z \rangle \in \Sigma_{\mathcal{R}}$, then the protocol must declare after σ at least one button "*P*" that enables the action *z* (i.e., $z \in P$) to be safe. Note that the $\mathcal{R}$-buttons are safe according to any protocol, but we impose upper bounds on the set of the safe $\mathcal{Q}$-buttons. In particular, all the $\mathcal{Q}$-buttons that enable the action *z* may be declared safe after the trace σ. This rule makes it possible to check whether or not the trace σ is continued by the external action *z*. If *z* is observed after pressing the button "*P*," then the testing can be continued to compare the behavior of the implementation with that of the specification after the trace $\sigma \cdot \langle z \rangle$. If an action *z*' is observed that does not belong to the specification ($\sigma \cdot \langle z' \rangle \notin \Sigma$), then this is an error (nonconformance).

Third rule. The protocol does not impose unnecessary constraints on the implementation; that is, all the $\mathcal{Q}$-buttons that are not declared by the protocol to be safe according to the first or the second rule are declared unsafe; therefore, the safety hypothesis allows them to be unsafe in the implementation. In particular, if the trace σ is not continued by the actions that are enabled by the $\mathcal{Q}$-button "*Q*" (i.e., $\forall z \in Q\ \sigma \cdot \langle z \rangle \notin \Sigma$) in the specification, then the button "*Q*" is unsafe after σ according to any protocol. Note that, if a protocol

declared such a button to be safe, then any safely testable implementation that has the admissible trace σ would be not conformal.

In addition to the admissible trace $\sigma \in \Sigma_{\mathcal{R}} \cap \mathbf{I}$, a nonconformal implementation can have traces that are not observable for the given protocol. First, if the button "Q" is declared unsafe by the protocol and neither the safety hypothesis nor the conformance conditions are violated, then there can be an unobservable refusal $Q \in \mathcal{Q}$ after σ. Indeed, for any button "P" that is safe after σ according to the protocol, we have $\exists u\,(u \in P \backslash Q \vee u = P \in \mathcal{R}) \& \sigma \cdot \langle u \rangle \in \mathbf{I} \cap \Sigma$. Second, any trace of the form $\sigma \cdot \langle z \rangle \cdot \lambda$, where the external action z is enabled only by the buttons that are declared by the protocol to be unsafe after σ, is admissible. This immediately follows from the fact that the unsafe buttons are not pressed under safe testing and, therefore, the presence or absence of such continuations of σ cannot be checked. Certainly, these additional traces cannot be observed under safe testing according to the given protocol.

By way of example, consider the $\mathcal{R}/\mathcal{Q}$-machine with $\mathcal{R} = \emptyset$ and $\mathcal{Q} = \{\{a, b\}, \{b, c\}\}$ and the specification $\Sigma_{\mathcal{R}} = \{\epsilon, \cdot\langle b \rangle\}$. Depending on which buttons are considered to be safe at the very beginning (after the empty trace), three types of protocols are possible:

(1) only "$\{a, b\}$,"

(2) only "$\{b, c\}$,"

(3) arbitrary.

The implementation $\mathbf{I}$ is safely testable if

(1) $\langle b \rangle \in \mathbf{I} \vee \langle a \rangle \in \mathbf{I}$,

(2) $\langle b \rangle \in \mathbf{I} \vee \langle c \rangle \in \mathbf{I}$,

(3) $\langle b \rangle \in \mathbf{I} \vee \langle a \rangle \in \mathbf{I} \& \langle c \rangle \in \mathbf{I}$.

The implementation is conformal if

(1) $\langle b \rangle \in \mathbf{I} \& \langle a \rangle \notin \mathbf{I}$,

(2) $\langle b \rangle \in \mathbf{I} \& \langle c \rangle \notin \mathbf{I}$,

(3) $\langle b \rangle \in \mathbf{I} \& \langle a \rangle \notin \mathbf{I} \& \langle c \rangle \notin \mathbf{I}$.

In this example, any conformal implementation can contain an arbitrary continuation σ along with any nonempty trace $\sigma \cdot \lambda$.

6. SAFETY HYPOTHESIS AND SAFE CONFORMANCE

We will specify interaction protocols in the form of the relation *a button is safe after the $\mathcal{R}$-trace of the specification*: *P safe by Σ after σ*; this relation must satisfy the rules described above: $\forall \sigma \in \Sigma_{\mathcal{R}}\ \forall P \in \mathcal{R}\ \forall z \in L\ \forall Q \in \mathcal{Q}$,

(1) *P safe by Σ after σ*,

(2) $\sigma \cdot \langle z \rangle \in \Sigma \Rightarrow \exists P \in \mathcal{R} \cup \mathcal{Q}\ z \in P$ & *P safe by Σ after σ*,

(3) *Q safe by Σ after* $\sigma \Rightarrow \exists v \in Q\ \sigma \cdot \langle v \rangle \in \Sigma$.

An $\mathcal{R}$-trace of an implementation is said to be safe if any external action z appearing in this trace is enabled at least by one button that is safe in the implementation after the trace prefix that immediately precedes this action:

$$SafeIn(\mathbf{I}) =_{def} \{\sigma \in \mathbf{I}_{\mathcal{R}} | \forall \mu \forall z \in L$$
$$(\mu \cdot \langle z \rangle \le \sigma \Rightarrow \exists P \in \mathcal{R} \cup \mathcal{Q}$$
$$P \text{ safe in } \mathbf{I} \text{ after } \mu \& z \in P\}).$$

The safety hypothesis requires that, if the $\mathcal{R}$-trace σ of the specification is safe in a an implementation, then any button that is safe after σ in the specification is also safe after σ in the implementation

$$\mathbf{I} \text{ safe for } \Sigma =_{def} \forall \sigma \in \Sigma_{\mathcal{R}} \cap SafeIn(\mathbf{I})$$
$$\forall P \in \mathcal{R} \cup \mathcal{Q}\ (P \text{ safe by } \Sigma \text{ after } \sigma$$
$$\Rightarrow P \text{ safe in } \mathbf{I} \text{ after } \sigma).$$

Testing is performed until the first error is met, and it is stopped immediately after detecting a nonconformance. Now, the continuation of testing after obtaining an inadmissible trace is not only senseless (i.e., it does not affect the verdict), but is also unsafe because the safety hypothesis does not guarantee anything in this situation.

Denote the external actions and refusals that are observed in the model T after the trace σ when the button P is pressed as $obs(\sigma, P, T) =_{def} \{z | \sigma \cdot \langle z \rangle \in T \& (z \in P \vee z = P \& P \in \mathcal{R})\}$.

An implementation is said to *safely conform* to the specification if it is safely testable and any observation after pressing a safe button is allowed by the specification. Such a conformance will be denoted by *saco* (*SAfe COnformance*):

$$\mathbf{I} \text{ saco } \Sigma =_{def} \mathbf{I} \text{ safe for } \Sigma$$
$$\& \forall \sigma \in \Sigma \cap SafeIn(\mathbf{I}) \forall P \text{ safe by } \Sigma \text{ after } \sigma$$
$$obs(\sigma, P, \mathbf{I}) \subseteq obs(\sigma, P, \Sigma).$$

Generation of a complete set of tests for the *saco* relation reduces to pressing each button "P" that is safe in the trace σ after each trace σ that occurs in the specification and is observed. If an action $z \in P$ or a refusal $P \in \mathcal{R}$ that continue σ in the specification are observed after pressing this button, then testing either completes with the verdict *pass* or continues by pressing the next button. Otherwise, the test returns the verdict *fail*.

All the definitions of safety and conformance given above are applied to LTS models by using their F-models. For example,

$$I \text{ saco } S =_{def} Ftraces(I) \text{ saco } Ftraces(S).$$

7. COMPLETION OF SPECIFICATIONS

Now, let us consider the differences in the definitions of safe buttons in implementations and specifications *safe in* and *safe by*. The $\mathcal{Q}$-button "P" is safe after a trace in an implementation if this trace does not continue with the refusal P; in a specification, this button is safe if this trace is continued at least by one action $z \in P$ (even this is not a requirement if there are other safe

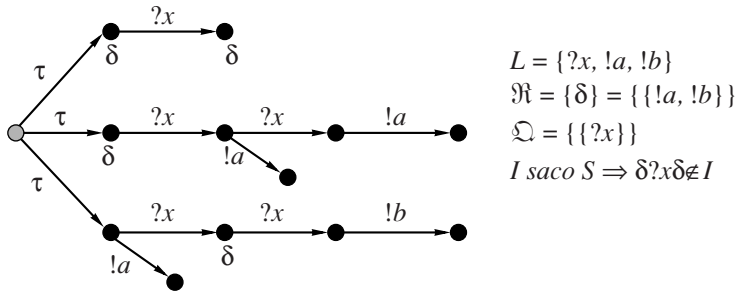

Fig. 1.

buttons that enable all the actions that continue the trace in the specification and are enabled by "*P*"). In the first case, the trace cannot be continued with the refusal *P*; in the second case, it must be continued by an action $z \in P$ and may also be continued by the refusal *P*. This means that we ignore the $\mathfrak{Q}$-refusals in the specification after the traces that are continued by the actions belonging to this refusal. Actually, in all the definitions, we use for implementations the $\mathfrak{R} \cup \mathfrak{Q}$-projection $\mathbf{I}_{\mathfrak{R} \cup \mathfrak{Q}}$; for specifications, we use the $\mathfrak{R}$-projection $\Sigma_{\mathfrak{R}}$. This difference is a cause of severe difficulties—*nonreflexivity* and *nonmonotonicity* of the conformance relation.

First, we consider the first difficulty: the same model considered as an implementation can be not safely testable and, therefore, nonconformal to itself considered as a specification. Moreover, the specification can contain $\mathfrak{R}$-traces that do not belong to any conformal implementation. An example is given in Fig. 1 for an input–output LTS. Here, the button "?*x*" is safe after the trace $\langle ?x \rangle$ in the LTS specification *S*; therefore, it is safe after the trace $\langle \delta, ?x, \delta \rangle$ (the refusal does not change the state of the LTS). Therefore, if an implementation contains the trace $\langle \delta, ?x, \delta \rangle$, it also contains the trace $\langle \delta, ?x, \delta, ?x \rangle$. Since receiving outputs is safe, at least one of the traces $\langle \delta, ?x, \delta, ?x, !a \rangle$, $\langle \delta, ?x, \delta, ?x, !b \rangle$, or $\langle \delta, ?x, \delta, ?x, \delta \rangle$ must be observed. Therefore, the implementation contains at least one of the traces $\langle ?x, \delta, ?x, !a \rangle$, $\langle \delta, ?x, ?x, !b \rangle$, or $\langle ?x, ?x, \delta \rangle$. Each of these traces is a continuation of a safe trace of the specification by an observation generated by the safe button "δ"; however, this continuation does not belong the specification, which contradicts the conformance.

The nonreflexivity of conformance contradicts intuition. Indeed, if the implementation is a "copy" of the specification, we obtain an invalid implementation. Note that, if $\mathfrak{Q} = \emptyset$, then *safe in* = *safe by*, and the relation *saco* is reflexive (and transitive; i.e., it is a preorder) (!). This fact suggests using the $\mathfrak{R} \cup \mathfrak{Q}/\emptyset$-semantics instead of the $\mathfrak{R}/\mathfrak{Q}$-semantics; in the former semantics, all the refusals are observable. To replace the latter semantics with the former one, a transformation of the specification $\mathscr{C}$ is made, which is called *completion*. It can be defined both for the trace and the LTS models. In both cases, conformance (the set of conformal implementations) must be preserved. For the trace model, we have $\forall \mathbf{I}\ \mathbf{I} saco_{\mathfrak{R}/\mathfrak{Q}} \Sigma \Leftrightarrow \mathbf{I}\ saco_{\mathfrak{R} \cup \mathfrak{Q}/\emptyset} \mathscr{C}(\Sigma)$. Here, the subscript indicates the test semantics in which the conformance relation is considered. The LTS completion is defined similarly; however, since several LTSs with the same set of traces can correspond to the same trace model, such a completion is not unique.

In the completed specification $\mathscr{C}(\Sigma)$, a new trace σ may appear if it is admissible in the conformal implementation. A necessary condition is that the set $D(\sigma)$ of the subtraces obtained from σ by deleting some (may be all) refusals contains a subtrace $\sigma' \in \Sigma$. Furthermore, in $\mathscr{C}(\Sigma)$, the trace σ is continued by the $\mathfrak{R}$-refusal *R* if this refusal is a continuation in Σ of every subtrace $\sigma' \in D(\sigma) \cap \Sigma$. Finally, in $\mathscr{C}(\Sigma)$, the trace σ is continued by the $\mathfrak{Q}$-refusal *P* if, for any subtrace $\sigma' \in D(\sigma) \cap \Sigma$ and any button "*R*" that is safe in Σ after σ', either there exists an action $z \in R \backslash P$ such that $\sigma' \cdot \langle z \rangle \in \Sigma$ or $R \in \mathfrak{R}$ and $\sigma' \cdot \langle R \rangle \in \Sigma$.

The completion described above is similar to the so-called "demonic" completion for input–output systems without observable input refusals [11–13]. The difference is that we allow arbitrary (including unobservable) refusals in the completed part of the model. In addition, the LTS transformation that does not completely preserve the conformance *ioco* (weakens it) is proposed in [12, 13]. In [11], the conformance preserving trace transformation is proposed, but this conformance is *ioconf* rather than *ioco*. This and some other conformance relations found in the literature will be discussed below. A review of various kinds of completions can found in [14, 15].

Note that the completed specification cannot be considered in the $\mathfrak{R}/\mathfrak{Q}$ – semantics because it is not equivalent to the initial specification in this semantics (see Fig. 2).

To prove the existence of a trace completion, it is sufficient to take the union of all the conformal trace implementations $\mathscr{C}(\Sigma) = \cup(\{\mathbf{I} | \mathbf{I}\ saco\ \Sigma\})$ (!). To construct the LTS completion, the union of LTSs is used: a new initial state is added along with the τ-transitions from it to the initial states of the underlying LTSs. The corresponding trace models are combined in the set-theoretic sense: $Ftraces \circ \mathscr{MC}(S) = \cup(\{\mathbf{I} | \mathbf{I}\ saco$

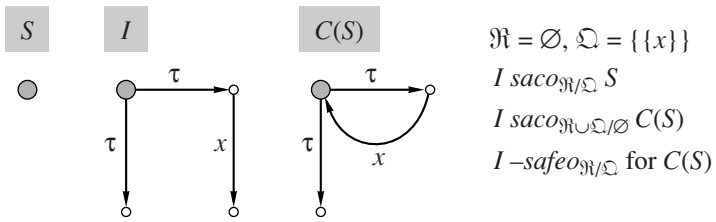

Fig. 2.

Ftraces(S)}). It is clear that this transformation cannot be constructed algorithmically. However, there exists a completion that can be constructed algorithmically (!).

8. MONOTONICITY OF CONFORMANCE

The second problem is the so-called *nonmonotonicity of conformance*; that is, the fact that the conformance is not preserved under the composition of LTSs. More precisely, the composition of the implementations that are conformal to their specifications may be not conformal to the composition of those specifications. This problem cannot be stated in the theory of $\mathfrak{R}$-traces because the composition of trace models is not defined in this theory. The monotonicity problem is more general than the completion problem: it persists even when there are no $\mathfrak{Q}$-buttons (!).

The *monotone* LTS transformation $\mathcal{M}$ that preserves the conformance of implementations solves this problem. The composition of the implementations that conform to their transformed specifications conforms to the composition of these transformed specifications. Naturally, the composition of the LTS completion and the monotone LTS transformation of $\mathcal{MC}$ is the LTS completion, which can be called the monotone completion. The union of conformal LTS implementations is exactly the completion of this type (!). Also, there exists a monotone transformation that can be constructed algorithmically (!).

9. EXAMPLES OF TEST SEMANTICS AND CONFORMANCE RELATIONS

Consider several test semantics (together with the corresponding conformance relations) encountered in the literature and interpret them as $\mathfrak{R}/\mathfrak{Q}$-semantics. Surveys of these semantics and conformances can be found in [3, 4, 7]. For each of these semantics interpreted as an $\mathfrak{R}/\mathfrak{Q}$-semantics, a unique safety hypothesis is uniquely distinguished because, in such semantics, each action z that is disabled by $\mathfrak{R}$-buttons is enabled by a single $\mathfrak{Q}$-button. All such actions that continue a specification trace determine all the $\mathfrak{Q}$-buttons that must be declared safe after this trace.

Trace semantics. Such a semantics is determined by the $\mathfrak{R}/\mathfrak{Q}$-machine with $\mathfrak{R} = \emptyset$ and $\mathfrak{Q} = \{L\}$. The traces only contain external actions. Usually, the *trace preorder* is the simple trace embedding. However, we interpret a stop with an unobservable refusal as an interaction error and consider only safe testing. The safety hypothesis requires that, if an observable trace in the specification is continued by an external action, then it cannot end at the terminal states of the LTS implementation. The operator is not allowed to press the button "L" only after the traces that end at the terminal states in the LTS specification.

Completed trace semantics. This semantics is determined by the $\mathfrak{R}/\mathfrak{Q}$-machine with $\mathfrak{R} = \{L\}$ and $\mathfrak{Q} = \varnothing$. The only refusal L is observable in the terminal state of the implementation when no other continuations are possible. The traces contain external actions and the refusal L; however, L can be followed only by the same refusal L. Any implementation is safely testable. The corresponding conformance is called the *testing preorder*.

Failure trace semantics or refusal trace semantics. This semantics is determined by the F-machine with $\mathfrak{R} = \mathcal{P}(L)$ and $\mathfrak{Q} = \varnothing$. The traces contain external actions and arbitrary refusals (*Ftraces*). Any implementation is safely testable. The corresponding conformance is called the *failure trace preorder* or *refusal preorder*. There is also a variant of this semantics in which only a finite set of external actions can be enabled. Such a semantics is determined by the $\mathfrak{R}/\mathfrak{Q}$-machine in which $\mathfrak{R}$ is the family of all *finite* subsets of the alphabet L.

Here, we do not consider the corresponding equivalences (*trace, completed trace*, and *failure* or *refusal equivalences*) because they do not satisfy the principle of independent behavior of implementations.

There are some specific semantics for the input–output systems. All of them are based on the following restrictions:

(1) The environment cannot choose which output to accept and which to reject. If outputs are accepted, then all of them are accepted.

(2) A single input can be sent without sending other inputs.

Semantics for input–output systems without input refusals. This semantics is determined by the $\mathfrak{R}/\mathfrak{Q}$-machine with $\mathfrak{R} = \{\delta\} = \{\{y!\}|!y \in L\}\}$ and $\mathfrak{Q} = \{\{?x\}|?x \in L\}$. The traces contain inputs, outputs, and the only refusal—the quiescence δ. The relation *ior*

(*input–output refusal relation* or *repetitive quiescence relation*) is equivalent to the simple embedding of traces $\mathbf{I}_{\mathcal{R}} \subseteq \Sigma_{\mathcal{R}}$. In order for the testing to be safe, each trace $\sigma \in \mathbf{I}_{\mathcal{R}} \cap \Sigma_{\mathcal{R}}$ in the LTS implementation must end at the quiescent states in which transitions corresponding to all the inputs are defined. However, if such a trace is not continued by an input $?x$ in the specification, the implementation is not conformal. For this reason, the specifications that contain, for example, only the trace $\langle ?x, !y\rangle$ and all its prefixes does not have conformal implementations.

The *ioco* (*input–output conformance*) relation is exempt of the drawback of *ior* discussed above. The *ioco* relation was proposed by Tretmans (see [7, 16]). It is based on the *input enabledness* hypothesis: in each reachable quiescent state, LTS implementations accept all the inputs. However, actually, the input $?x$ is not sent to the implementation after the trace σ that is not continued by $?x$ in the specification. In other words, the behavior of the implementation after receiving such an input does not matter for the *ioco* conformance. If $\mathbf{I}$ *ioco* Σ and $\sigma \cdot \langle ?x\rangle \in \mathbf{I}$, then the implementation $\mathbf{I}'$ that differs from $\mathbf{I}$ only in that it refuses the input $?x$ after the trace σ is also safely testable (there are no unobservable refusals). Both these implementations are safely testable, and they cannot be discriminated under safe testing. However, $\mathbf{I}$ is conformal while $\mathbf{I}'$ is not because it does not belong to the domain of *ioco*. In other words, the input enabledness hypothesis of the implementation with respect to inputs is too strong. The safety hypothesis proposed in this paper is more adequate to the testing precondition of implementations: if the trace σ is continued by the input $?x$ in the implementation, then σ must not end in the implementation in a state in which the input is refused. For the *saco* relation, there cannot be two distinct implementations of which one is conformal and the other is not and which are indistinguishable under safe testing.

Semantics for input–output systems with input refusals. This semantics is determined by the $\mathcal{R}/\mathcal{Q}$-machine with $\mathcal{R} = \{\delta\} \cup \{\{?x\} | ?x \in L\}$ and $\mathcal{Q} = \varnothing$. The traces contain inputs, outputs, and refusals (the quiescence δ and input refusals). Any implementation is safely testable. The corresponding conformance of the type *saco* is called $ioco_{3\delta}$ [17–20]. This semantics should be used when the specifications are completed in the semantics without input refusals.

Semantics for multi input-outputs transition systems (*MIOTS*). Such systems are studied in the works by Heerink and Tretmans [21, 22], where the *mioco* conformance relation is proposed. The set of inputs is divided into subsets called input channels L_I; for L_I, the implementation must either enable any input from L_I or disable all of them. The set of outputs is also divided into subsets called output channels L_U. In contrast to the inputs, no restrictions are imposed on the behavior of implementations with respect to the outputs. Each channel is assigned a specific θ-observation. For an input channel, it means the refusal for all the inputs of this channel; and for an output channel, it means the absence of response in this channel (partial quiescence). Note that, in this case, the test does not necessarily enable all the outputs or none of them; rather, it must enable all the outputs from each output channel or disable all of them. In an $\mathcal{R}/\mathcal{Q}$-machine, each input $?x$ is assigned the $\mathcal{R}$-button $\{?x\}$, each output channel L_U is assigned the $\mathcal{R}$-button L_U, and there are no $\mathcal{Q}$-buttons. The constraint imposed on output enabling is imposed on the implementations and specifications, but it does not concern the machine organization. Any implementation is safely testable.

For such a machine, the relations *mioco* and *saco* are almost identical with the only exception: *mioco* allows the implementation to enable an input regardless of whether or not it is enabled or refused in the specification. It seems that such a "liberalism" is explained by the fact that *mioco* is based on *ioco*, in which all the inputs are enabled. For *mioco*, this is not the case. An implementation may refuse an input in certain states after a certain trace if this is allowed by the specification; however, the implementation cannot refuse an input in all the states after a certain trace.

Sometimes, input–output systems impose a special constraint on the environment: it must not "block the outputs" produced by the implementation [23, 24]. In terms of LTS, this means that, in each quiescent state of the environment/test, transitions by all the outputs must be defined.

Semantics of the input–output systems without output blocking and without refusals. This semantics is determined by the $\mathcal{R}/\mathcal{Q}$-machine with $\mathcal{R} = \{\delta\}$ and $\mathcal{Q} = \{\{\delta, ?x\} | ?x \in L\}$. In contrast to the similar semantics with output blocking, the safety hypothesis is relaxed: if the trace σ is continued by the input $?x$ or by outputs in the specification, then, in the implementation, σ must not end in the quiescent state in which $?x$ is refused. While sending an input to a safely testable implementation, we expect to observe an output rather than the input reception. If the implementation has an infinite chain of outputs (*oscillations* [23]), then we cannot send the input $?x$ with the guaranteed reception by the implementation. Moreover, if this chain does not contain states in which $?x$ is received, we will never learn it by continuously pressing the same button $\{\delta, ?x\}$.

Semantics of the input–output systems without output blocking with refusals. This semantics is determined by the $\mathcal{R}/\mathcal{Q}$-machine with $\mathcal{R} = \{\delta\} \cup \{\{\delta, ?x\} | ?x \in L\}$ and $\mathcal{Q} = \varnothing$. Any implementation is safely testable. Input refusals are only observed together with quiescence (in the quiescent states of the LTS implementation). The oscillation problem also occurs in this case. Indeed, we can learn nothing about input refusals in a nonquiescent state.

For the majority of the semantics with $\mathcal{R}$-refusals described above, their variants in which any refusal or a refusal of a certain kind can be only observed at the

end of a trace were studied. After such a refusal has been observed, no other observations are possible. In an $\mathcal{R}/\mathcal{Q}$-machine (and in a reactive machine) this situation may correspond to keyboard locking after observing a refusal. For generative machines, van Glabbeek proposes to assume that the switches may not be toggled from the *blocked* state to the *free* state. On the whole, this is equivalent to keyboard locking in terms of the testing power. For the completed traces semantics, the situation is the same because the refusal of L implies that there will be no other actions anyway. For the other semantics, the conformance relations change as follows:

rf $\longrightarrow$ *conf* or *failure preorder* for the traces consisting of external actions and the pairs <trace of external actions, refusal> (*failure pairs*),

ior $\longrightarrow$ *iot* (*input–output testing relation*) for the traces consisting of external actions and for the traces consisting of external actions that end in a quiescent state (*quiescent traces*),

ioco $\longrightarrow$ *ioconf* (both stand for the *input–output conformance*).

The relation *rioco* in [24] is designed so that the input refusals (in contrast to the quiescence) can only occur at the end of traces.

All these variants reduce the testing power; however, it is not quite understood why the observation of a refusal prevents the continuation of testing in practice. Below, we do not consider this question.

10. DESTRUCTION

We introduce a new machine action called destruction; it is denoted by γ. γ-Action is any undesired behavior of the system including the system destruction, which must be avoided while testing (for example, the button of the immediate self-destruction in military systems). Destruction, as well as refusal, is one of possible interpretation of the system's unspecified behavior. The difference between them is that the refusal assumes the system protection from undesired action, while the destruction guarantees nothing.

The destructive actions are often eliminated from the consideration because the implementation must check the validity of the call parameters [7, 22]. If the parameters are incorrect, the implementation either ignores the call or reports an error. This requirement is quite natural if the system in question is a system of "common use." Then, it must be fool-tolerant. However, when internal components or subsystems with a restricted access are tested, mutual checks of the call correctness are not necessary. When the parameters have a complex internal structure and nontrivial correctness conditions, the verification overheads increase the time of the system development, its size, and also increase the execution time. In this case, an alternative is to have a thorough specification of the preconditions for the operation calls [25]. For example, freeing the memory that was not earlier allocated is the violation of a precondition. It is the correctness of calls of components made by other components that must be tested rather than the behavior of components caused by incorrect calls. Actually, this means that the behavior of each component must be tested (against its postcondition) only for the calls that satisfy the component's precondition. In other words, since we want to test the implementation rather than the environment, we are not interested in the component's behavior when it is called with the violation of the preconditions; moreover, this behavior is not regulated by the specification.

The semantics of the γ-action assumes that all the observations after this action are unreliable. Since we are only interested in reliable observations, we assume that no observations are possible after the destruction. The destruction is considered to be a conditionally observable action: considered as an event, it can occur during interaction; however, we restrict ourselves to testing in which no such events occur. In this sense, the destruction is similar to the unobservable refusal. Thus, we are only interested in the safe interaction of the environment with the implementation that does not cause destruction.

The destruction, as well as τ-actions, is not controlled by buttons, and it is always enabled. In the safe testing, the operator should not press the button if it can result in the execution of an external action that leads to the destruction of the implementation. Thus, the γ-action occurs only after an external action or in the beginning of the work. The latter case is a degenerate one: the implementation conforms only to the specification that enables the destruction at the very beginning; testing is not needed and even inadmissible (the machine is not allowed to be turned on). In the LTS model, a transition may be labeled not only by an external action or by the symbol τ but also by the symbol γ. When LTSs are composed, two new rules are added:

$$s \xrightarrow{\gamma} s' \vdash st \xrightarrow{\gamma} s't,$$

$$t \xrightarrow{\gamma} t' \vdash st \xrightarrow{\gamma} st'.$$

We have a new kind of traces that end by the destruction. Now, a button is safe after the trace $\sigma \in \mathbf{I}_{\mathcal{R}}$ if it does not cause an unobservable refusal and does not lead to the destruction (the condition that is doubly underscored): P *safe in* $\mathbf{I}$ *after* $\sigma =_{def} (P \in \mathcal{R} \vee \sigma \cdot \langle P \rangle \notin \mathbf{I})$ $\underline{\underline{\& \forall z \in P\ \sigma \cdot \langle z, \gamma \rangle \notin \mathbf{I}}}$.

The interaction protocol is accordingly modified (new doubly underscored conditions are added): $\forall \sigma \in \Sigma_{\mathcal{R}}\ \forall R \in \mathcal{R}\ \forall z \in L\ \forall Q \in \mathcal{Q}$,

(1) R *safe by after* $\sigma \Leftrightarrow \underline{\underline{\forall u \in R\sigma \cdot \langle u, \gamma \rangle \notin \Sigma}}$,

(2) $\sigma \cdot \langle z \rangle \in \Sigma$ & $\underline{\underline{\exists T \in \mathcal{R} \cup \mathcal{Q}\ z \in T\ \&\ \forall u \in T\ \sigma \cdot \langle u, \gamma \rangle \notin \Sigma}} \Rightarrow \underline{\underline{\exists P \in \mathcal{R} \cup \mathcal{Q}\ z \in P\ \&\ P\ safe\ by\ \Sigma\ after\ \sigma}}$,

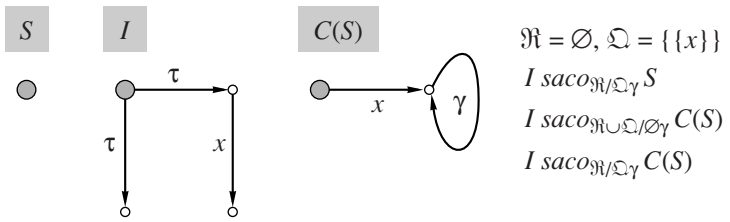

Fig. 3.

(3) *Q safe by* Σ *after* σ ⇒ ∃ϑ ∈ *Q* σ · ⟨ϑ⟩ ∈ Σ & ∀*u* ∈ *Q* σ · ⟨*u*, γ⟩ ∉ Σ.

The definition of a safe trace also changes. Now it (1) uses the relation *safe in* that takes account of the destruction; (2) a safe trace cannot contain unsafe $\mathcal{R}$-refusals. The new concept of a safe trace in the specification is introduced. Earlier, we had no need for it because safety was interpreted as the absence of unobservable refusals and the second rule in the protocol guaranteed that, for each external action in the trace, there was a button that enabled this action and did not cause an unobservable refusal after the preceding trace prefix. Now, we must take into account the possibility of destruction. If there is the trace ⟨γ⟩, then there are no safe traces.

$$SafeIn(\mathbf{I}) =_{def} \{\sigma \in \mathbf{I}_{\mathcal{R}} | \langle\gamma\rangle \notin \mathbf{I} \& \forall \mu \forall u$$
$$(\mu \cdot \langle u\rangle \le \sigma \Rightarrow \exists P \in \mathcal{R} \cup \mathcal{Q}$$
$$P \text{ safe in } \mathbf{I} \text{ after } \mu \& (u \in P \vee u = P)\}).$$
$$SafeBy(\Sigma) =_{def} \{\sigma \in \Sigma_{\mathcal{R}} | \langle\gamma\rangle \notin \Sigma \& \forall \mu \forall u$$
$$(\mu \cdot \langle u\rangle \le \sigma \Rightarrow \exists P \in \mathcal{R} \cup \mathcal{Q}$$
$$P \text{ safe by } \Sigma \text{ after } \mu \& (u \in P \vee u = P)\}).$$

The safety hypothesis now requires that there is no destruction in the implementation if it is impossible in the specification in the same situation, i.e., either at the very beginning (before any button is pressed) or after a safe continuation of a safe trace:

$$\mathbf{I} \text{ safe for } \Sigma =_{def} (\gamma \notin \Sigma \Rightarrow \gamma \notin \mathbf{I})$$
$$\& \forall \sigma \in SafeBy(\Sigma) \cap SafeIn(\mathbf{I}) \forall P \in \mathcal{R} \cup \mathcal{Q}$$
$$(P \text{ safe by } \Sigma \text{ after } \sigma \Rightarrow P \text{ safe in } \mathbf{I} \text{ after } \sigma).$$

The conformance relation is constructed only on the safe $\mathcal{R}$-traces of the specification:

$$\mathbf{I} \text{ saco } \Sigma =_{def} \mathbf{I} \text{ safe for } \Sigma$$
$$\& \forall \sigma \in SafeBy(\Sigma) \cup SafeIn(\mathbf{I})$$
$$\forall P \text{ safe by } \Sigma \text{ after } \sigma$$
$$obs(\sigma, P, \mathbf{I}) \subseteq obs(\sigma, P, \Sigma).$$

The assertions concerning the reflexivity and transitivity of the conformance relation in the absence of $\mathcal{Q}$-buttons and assertions concerning the existence of a completion $\mathcal{C}$ and a monotone transformation $\mathcal{M}$ (including those that can be described algorithmically) remain valid for the γ-semantics:

$$I \; saco_{\mathcal{R}/\mathcal{Q}} S \Leftrightarrow \mathbf{I} \; saco_{\mathcal{R} \cup \mathcal{Q}/\emptyset} \mathcal{C}(S)$$
$$\Leftrightarrow \mathbf{I} \; saco_{\mathcal{R}/\mathcal{Q}} \mathcal{C}(S)(!).$$

For the γ-semantics, there exists a completed specification that can be considered not only in the $\mathcal{R} \cup \mathcal{Q}$-semantics but in the $\mathcal{R}/\mathcal{Q}$-semantics as well (cf. Figs. 2 and 3).

Note that, in the completion $\mathcal{C}$, the γ-completion that continues the trace σ only by the trace σ · ⟨*z*, γ⟩ is used instead of the "demonic" completion of σ by all the traces σ · ⟨*z*⟩ · λ. Thus, no information about the unsafe buttons is lost: after the "demonic" completion, we have to test all the added traces σ · ⟨*z*⟩ · λ though it is senseless (everything is enabled), while the added trace σ · ⟨*z*⟩ is not tested in the case of the γ-completion because it leads to the destruction.

11. DIVERGENCE

The divergence (an infinite internal activity) is not necessarily an error of getting caught in an endless loop. In some cases, the divergence is a correct and inevitable behavior. For example, a test can replace not the entire environment but only a part of it. Actually, the composition of the implementation with the remaining part of the environment is tested. The interaction of the implementation with this part of the environment can be endless, but it is perceived as an endless internal activity from the test; that is, it is perceived as divergence (the synchronous transitions in the composition are τ-transitions). The purely internal divergence is also possible; for example, this is a waiting loop in an operating system that continues arbitrarily long until it is interrupted.

Actually, the problem is not in the divergence per se, but in the exit from it. If an external action has a higher priority than the internal activity, then the divergence stops. In a machine with priorities, the enabledness of the τ-actions depends on the selected button, and we can indirectly control these actions and, therefore, the divergence. In this case, we have the so-called *executable* divergence: when a certain button is selected (or when no buttons are selected), all the τ-actions of the endless chain are executable, but when another button is selected, they are not executable, therefore there is no endless loop. The divergence that starts after selected the button "*A*" can be exited when another button "*B*" is selected under which the divergence is not execut-

able. Note that that the exit from divergence requires that some buttons are toggled i.e., pressed without observation (recall that there may be no observation). The only case in which the divergence cannot be exited for sure is when the divergence is executable when any button is selected. For the machines without priorities, any divergence has this undesirable property.

Divergence is inconvenient because it makes it impossible to get an observation as a response to a test action in a finite time; therefore, testing cannot be continued after the observation. In this sense, the divergence is similar to an unobservable refusal. However, there is an important difference: an unobservable refusal occurs instead of an external action after a button is pressed, while the divergence occurs after an external action (or in the initial state of the machine before the first button is pressed) and manifests itself only when the operator presses a button once more. In other words, not the divergence per se is harmful but the exit from it.

In a machine without priorities, the behavior of the implementation in the case of divergence can be simulated using a special Δ-action. It is enabled by any button and is considered conditionally observable. Considered as an event, the divergence can occur in the course of interactions; however, we restrict ourselves to testing that is devoid of such events. The Δ-action is very similar to the γ-action; the only difference is that the destruction cannot be disabled, while the Δ-action is enabled by any button but is disabled if no buttons are pressed; the latter situation simulates the divergence when the attempt to exit it is made. The safety relations are changed as follows (the additions are doubly underscored):

$$P \textit{ safe in } \mathbf{I} \textit{ after } \sigma =_{def} (P \in \mathcal{R} \vee \sigma \cdot \langle P \rangle \notin \mathbf{I}) \,\&\, \forall z \in P\sigma \cdot \langle z, \gamma \rangle \notin \mathbf{I} \,\&\, \sigma \cdot \langle \Delta \rangle \notin \mathbf{I}.$$

$$\forall \sigma \in \Sigma^{\downarrow} L_{\mathcal{R}}\ \forall R \in \mathcal{R}\ \forall z \in L\ \forall Q \in \mathcal{Q},$$

(1) R *safe by* Σ *after* $\sigma \Leftrightarrow \forall u \in R\ \sigma \cdot \langle u, \gamma \rangle \notin \Sigma \,\&\, \sigma \cdot \langle \Delta \rangle \notin \Sigma$,

(2) $\sigma \cdot \langle z \rangle \in \Sigma \,\&\, \exists T \in \mathcal{R} \cup \mathcal{Q}\ z \in T \,\&\, \forall u \in T\ \sigma \cdot \langle u, \gamma \rangle \notin \Sigma \underline{\underline{\,\&\, \sigma \cdot \langle \Delta \rangle \notin \Sigma}} \Rightarrow \exists P \in \mathcal{R} \cup \mathcal{Q}\ z \in P \,\&\, P$ *safe by* Σ *after* σ,

(3) Q *safe by* Σ *after* $\sigma \Rightarrow \exists \vartheta \in Q\ \sigma \cdot \langle \vartheta \rangle \in \Sigma \,\&\, \forall u \in Q\ \sigma \cdot \langle u, \gamma \rangle \notin \Sigma \underline{\underline{\,\&\, \sigma \cdot \langle \Delta \rangle \notin \Sigma}}$.

Now, the safety hypothesis allows the divergence in the implementation only if it is allowed in the specification in the same situation, i.e., after the same trace that is safe in the specification.

In the composition, divergence can occur as an infinite chain of synchronous transitions even in the case when there is no divergence in the operands. Therefore, the class of models without divergence is not closed with respect to composition.

Sometimes, the testing is considered in which a direct (in contrast to conditional) observation of divergence (Δ-observation) or the so-called λ-observation is possible. The λ-observation assumes the divergence or an unobservable refusal. In both cases, either the possibility of infinite observation or an upper bound on the execution time of any external action and any finite chain of τ-actions is assumed. If the display is empty infinitely long or longer than a predefined timeout after an $\mathcal{R}$-button was pressed, then we have a divergence or, in the case when a $\mathcal{Q}$-button was pressed, a λ-observation. If the time constraint is used as a means for detecting refusals, the differences between $\mathcal{R}$ and $\mathcal{Q}$-buttons are leveled out and the expiration of the timeout is interpreted as a λ-observation in both cases. Below, we will assume that the divergence is modeled by the Δ-action and the testing is safe; i.e., we assume that the testing avoids unobserved refusals, divergence, and destruction. This assumption is based on the concept that no deadlocks, no infinite waiting of the environment for the interaction result, and no system destruction must occur in the case of correct interaction.

12. MENU LIGHTS AND READY TRACES

There may additional testing capabilities that are defined by the so-called *menu lights* in the machine. The menu lights correspond to external actions. The light corresponding to the action z is on if z is defined in the machine. If the machine is active, the state of the lights indicates the actions that are defined in the current state or were defined in a previous state. For this reason, it is usually assumed that the state of the lights is only reliable when the machine is in a quiescent state s. When the machine is at stop, the menu lights enable us to determine the *ready set*, which is defined as the set of all external actions defined in the state s, i.e., the set of actions that are ready to execute. This set is denoted by $ready(s)$. The traces that can include ready sets in addition to external actions are called *ready traces*; the corresponding semantics, preorder, and equivalence are called the *ready trace semantics*, *ready trace preorder*, and *ready trace equivalence*, respectively.

The ready traces possess the *generative* property; i.e., given such a trace, we can obtain all the refusal traces that could be observed when the implementation performs the same chain of transitions. In other words, the set of ready traces unambiguously determines the set of refusal traces. However, without using menu lights, we cannot decide by observing a refusal trace whether or not it is the complement (the replacement of the refusals by their complements to the alphabet) of a ready trace. When we have a refusal, we know that all the enabled actions are not defined in the implementation. However, we do not know whether or not the disabled actions are defined. Moreover, in a parameterized machine, not all the complements of the ready sets are observable refusals (correspond to $\mathcal{R}$-buttons) in the general case.

As for the refusal semantics, there are modifications in which the observation of readiness does not make it

possible to continue testing. In this case, we have the so-called *ready pairs* <trace of external actions, ready set>, and the *readiness semantics*, *readiness preorder*, and *readiness equivalence*. Similarly to the testing machine with finite refusals, one can consider a machine in which only finite ready sets of external actions can be recognized. Van Glabbeek proposes to interpret this situation in such a way that each menu light is simultaneously a button, and the light may be on only if the operator pressed this button. By pressing one light button after another after the machine stopped in a quiescent state (the green light is off), the operator can find out if the corresponding actions are defined. However, if the action alphabet is infinite, it remains unknown whether or not the resulting set of defined actions is the ready set, i.e., the set of all the defined actions.

The test capabilities modeled by menu lights increase the power of testing; however, their practical usefulness is doubtful. The fact that a refusal is observable implies that the environment can find out whether some action among those it demanded the implementation to perform was actually executed, and if it was, then exactly which action; later, this knowledge can be used to correct the subsequent interaction. In other words, the environment calculates not only on getting a confirmation about the execution of the required action but also on the notification of refusal to execute this action. This seems to be a natural behavior. However, in order to determine the ready set, the environment must find out which actions the implementation could execute in the given quiescent state. For this purpose, a special polling operation is needed, which is not always available in the implementation interface.

However, there are exceptions. For example, in graphical interfaces, such a menu of defined actions may appear on the display (sometimes, in the form of all or a part of the actions in which the disabled actions appear dimmed). Even in this example, the actions in the graphical menu are only inputs; however, it is not known in the general case which outputs the implementation will produce in response to pressing a button in the graphical menu. In other words, for this example, we must consider a testing machine in which the menu lights are available only for some of the actions. One may consider a more general testing capability of partial ready sets. In this case, when in a quiescent state, the system reports the status of each action. It can be (1) defined (the light is green), (2) undefined (the light is red), or (3) unknown (the light is off). Below, we consider only the complete ready sets, when there is no third possibility.

Now, let us discuss how the environment can use the ready sets. The readiness is observed when the machine is in a quiescent state. This enables the environment to calculate any refusal. Therefore, the environment can enable only the sets of actions in which at least one action can be executed by the implementation. With such an interaction protocol, unobserved refusals cannot occur after the machine has stopped, although they may still occur when a button is pressed after an external action or at the beginning of the work. Refusals do not provide any additional observations; for this reason, no mixed traces that contain both ready sets and refusals are considered. Naturally, we still must be able to detect the machine stop, which is made using the θ-observations in an $\mathcal{R}/\mathcal{Q}$-machine after pressing an $\mathcal{R}$-button or using the green light in a generative and reactive machine. After the stop, the observed ready set is added to the trace. In the $\mathcal{R}/\mathcal{Q}$-machines, the safety hypothesis is relaxed: it concerns the behavior of the implementation only after the traces that do not end by a ready set (the empty trace and the traces that end by an external action). After such a trace σ, the $\mathcal{Q}$-button "P" is safe in the specification Σ if it is safe after any continuation of this trace by the ready set R that is admissible in the specification: $\forall R\sigma \cdot \langle R\rangle \in \Sigma \Rightarrow R \cap P \neq \varnothing$.

In contrast to the external actions and refusals, it is natural to assume that, for the ready sets, the conformance implies that the implementation ready set R_i includes a specification ready set R_s ($R_i \supseteq R_s$) after the same trace rather than to assume that $R_i = R_s$. In other words, if, after a trace, the implementation can turn out to be in a state in which a certain set of actions R_i is defined, then this must be allowed by the specification in the sense that, after this trace, the specification allows a state in which the set of defined actions R_s is not greater than R_i. This reminds of the *mandatory* behavior requirement: the implementation cannot propose the menu of actions that is *less* than the menu allowed by the specification.

We will say that a trace in the implementation is *majorized* by a trace in the specification if they have the same length and the corresponding positions are occupied either by the same external actions or by embedded ready sets. If there is destruction, only the nondestructive external actions of the specification must be taken into account (the destructive actions of the specification are not checked and, therefore, cannot be defined in the implementation). For the sets of ready traces, the permissive principle is still used; however an implementation trace is allowed when the set of admissible specification traces includes a majorizing trace rather than the trace itself.

Such a conformance relation can be called the safe conformance with ready traces; it is denoted by *resaco*. In the $\mathcal{R}/\mathcal{Q}$-semantics for LTSs, it is related to the *saco* relation via a monotone completion $\mathcal{MC}$:

$$I\ saco_{\mathcal{R}/\mathcal{Q}}\ S \Leftrightarrow I\ saco_{\mathcal{R}\cup\mathcal{Q}/\varnothing}\ \mathcal{MC}(S)$$

$$\Leftrightarrow I\ saco_{\mathcal{R}/\mathcal{Q}}\mathcal{MC}(S)$$

$$\Leftrightarrow I\ resaco_{\mathcal{R}\cup\mathcal{Q}/\varnothing}\ \mathcal{MC}(S)$$

$$\Leftrightarrow I\ resaco_{\mathcal{R}/\mathcal{Q}}\mathcal{MC}(S).$$

This assertion is a hypothesis. The idea of its proof could be based on taking the union of the conformal implementations as the transformation $\mathcal{MC}$. Note that the equivalence $I\ resaco_{\mathcal{R}\cup\mathcal{Q}/\varnothing} S' \Leftrightarrow I\ resaco_{\mathcal{R}/\mathcal{Q}} S'$ is true only for the specification $S' = \mathcal{MC}(S)$ but not for arbitrary specifications S'.

The ready sets have an important property of *computability* under the composition of LTSs: the quiescence and the ready set of the composition state st are uniquely determined by the quiescence and the ready sets of the states s and t of the operands. Under the composition of LTSs, in the alphabets L and M, we have

$$st \text{ is quiescent} \Leftrightarrow s \text{ is quiescent}$$
$$\&ready(s) \cap ready(t) = \varnothing$$
$$\&t \text{ is quiescent } \&ready(s) \cap ready(t) = \varnothing,$$
$$st \text{ is quiescent: } ready(st) = (ready(s)\backslash\underline{M})$$
$$\cup(ready(t)\backslash\underline{L}).$$

This fact enables us to defined the composition of ready traces that, given the pair of traces λ and μ, returns a set of traces. This set consists of all the traces in which the subtrace of asynchronous actions from $\{\gamma\} \cup L\backslash\underline{M}(\{\gamma\} \cup M\backslash\underline{L})$ coincides with the subtrace of the same actions of the trace λ (μ) and each ready set is the composition of the corresponding ready sets in the operand traces. The operation *ReadyTraces* of taking the set of ready traces of an LTS is *additive* with respect to the composition of LTSs and of the ready traces:

$$ReadyTraces(S \pitchfork T) = \cup(ReadyTraces(S)$$
$$\pitchfork ReadyTraces(T))(!).$$

If the divergence is modeled by the Δ-action, we also must compose infinite ready traces. If the operand traces have infinite postfixes of inverse actions, then the composition trace ends by the Δ-action.

The generative and the additive properties of ready traces enable us to develop a closed trace theory that includes both the conformance relation (even if it is based on refusal traces rather than on ready traces) and the composition of trace models. We believe that this possibility makes the ready traces useful in the conformance theory.

13. REPLICATION AND SIMULATION

The observed behavior of the system under test depends on which buttons are pressed by the operator and when. In order to model various variants of the operator behavior, the so-called *replication button* is used in testing machines. It makes it possible to create several copies of the machine at the given time and continue working with each copy independently. In a parameterized machine, we assume that the replication is performed before the work starts. In this case, various copies of the machine imitate various testing "sessions:" after each session, the machine can be made to work from the beginning in the next session. Replication differs from such a session-by-session work in that it abstracts itself from the possible number of sessions: it can be infinite or even uncountable. Such a replication is necessary for the potential possibility to obtain all the finite traces (logs in a machine with priorities) using finite test experiments.

In van Glabbeek's and Milner's machines, a much stronger replication is used; it can be performed at an arbitrary moment. It enables one to obtain information about the system state after the trace σ in the form of the set of traces that can be observed in this state. Thus, the states are distinguished up to the set of such traces. If a replication is made in each copy after observing the trace σ, then we obtain the set of continuing traces Σ_i in the ith copy. In total, we have the family $\Sigma = \{\Sigma_i | i = 1, 2, \ldots\}$. If the replication is only made before the start of the work (only testing sessions), then we can only determine the union $\cup(\Sigma)$, i.e., the set of all traces that continue σ in all the states after this trace. However, we cannot find out the states of the machine associated with these traces; i.e., we cannot determine the summands $\Sigma_1, \Sigma_2, \ldots$.

The possibility to make a replication at any time enables us to define such conformance relations as *simulations* and *bisimulations*, which take into account the system sate in one way or another. These relations include *strong* and *weak* simulations, *delay* simulation, *branching* simulation, and others. Van Glabbeek also considers various modifications of replication that depend on various constraints imposed on the machine state at which the replication may be made (for example, only in the quiescent states) and on the number of machine copies that may be created by pressing the replication button one time. These modifications determine the corresponding modifications of simulations and bisimulations. To relate the trace σ observed before the replication with its continuations after the replication, special operations of the "conjunction" type are used. For the replication made before the start of the work, such operations are not needed because σ is always empty. In our opinion, the replication made at an arbitrary time is practically important only in rare special cases, which we do not consider in this paper.

14. GLOBAL TESTING

The observed traces are generally obtained nondeterministically; i.e., they do not uniquely determined by the sequence of the buttons pressed by the operator. Given the same sequence of buttons, the trace depends on the nondeterministic choice of one of several executable actions made by the machine. The nondeterminism of the action choice can be interpreted as a result of abstracting from some external factors—weather conditions—that are not taken into account; these conditions uniquely determine the choice.

Note that, for a machine with priorities, logs rather than traces must be stored; moreover, one must take

into account the time delays made by the operator between an observation and the subsequent pressing of a button or between two successive button pressings when they are toggled without observation. For this reason, we include into the weather conditions also the factors that affect the "free will" of the operator thus determining the time delays between button pressings. This is quite natural because the operator models the tester, which is a computer program. Such a program is nondeterministic only at a certain level of abstraction, when we neglect other programs or the hardware affecting its behavior.

If the machine uses no priorities, we may assume that the operator works sufficiently quickly. The weather conditions include only the factors that affect the machine but not the operator. Certainly, this does not mean that the operator cannot work slowly in some test experiments. This only means that any trace that can be obtained when the operator works slowly can also be obtained when he works quickly.

For the completeness of testing, we must assume that that any weather conditions can be reproduced in an experiment. In the formalism of the testing machine, this can be interpreted so that a sufficient number of copies for each variant of the weather conditions are created by replication. If there is such a possibility, the testing is said to be global. Note that we abstract ourselves from the number of variants of the weather conditions. Here, it is important to have a theoretical possibility to check the system's behavior under any weather conditions and arbitrary behavior of the operator.

While performing replication, at least one copy of each combination of the test considered as an instruction for the operator and a variant of the weather conditions must be made. Since it is known in advance which test is run on the machine, we may assume that the copies are labeled by the tests, even more so that we are only interested in the tests belonging to a certain complete test suite. Thus, for each test case belonging to the test suite under consideration, as many copies as there are variants of the weather conditions are created. A copy corresponding to a certain variant of the weather conditions for a certain model test provides a model of the run of a real test on the given system.

Certainly, only finite test cases are used in practice, as well as the number of test cases in the test suite and the number of runs of each test case must be finite. Since the test cases are finite, the complete test suite usually contains an infinite number of test cases. Moreover, without additional conditions, we cannot be sure that we have conducted the test experiments for each test case under all possible weather conditions. These problems can be resolved in a number of ways.

One of them is to provide special test capabilities for weather control. For this purpose, we must go beyond the framework of the model; indeed, the model does not take into account side external details; i.e., it does not take into account the weather. Therefore, testing becomes dependent not only on the specification but also on some implementation details, which can be called the operational environment of the implementation. For each variant of the operational environment, we have to create a special test suite. However, in some cases, practical advantages can be obtained along these lines.

Another solution is to make special implementation hypotheses. For a finite test suite, it is assumed that, if the implementation behaves correctly on its test cases, it will also behave correctly on all the test cases in the complete suite. For a finite number of runs, it is assumed that, if the implementation behaves correctly under certain weather conditions, it will also behave correctly under arbitrary weather conditions.

The third solution can be used if we know the probability distribution of the weather conditions. In this case, the testing is complete with a certain probability [26].

The fourth solution, which is close to the third one, assumes that only a finite number of weather conditions (up to the equivalence) is possible in each situation (after a trace) and there exists a number N such that, after N test runs, the behavior of the implementation is verified for all the weather conditions possible in the current situation [8, 27].

Finally, there is a more radical solution: to prohibit nondeterminism of the implementation; in this case, the implementation hypothesis restricts the class of implementations to the deterministic ones. Although rather naive, it is a commonly used practical method [28]. It is justified by the fact that it is often known in advance that the implementations of interest are deterministic.

Instead of complete but infinite test suites, in practice one has to resort to finite sound suites: if a test returns the verdict *fail*, the implementation is not conformal. Such a finite test suite is constructed using some *coverage criterion* so as to cover all the classes of situations (errors) of interest. Theoretically, a finite test suite can be obtained by filtering the complete enumerable test suite by the coverage criterion, although more direct methods of constructing the desired suite are more practical. A rather general approach is to use a specification model that is coarser than the initial model; it is called the test model. The test model is the result of the factorization of the initial LTS specification with respect to the equivalence of transitions, which is usually reduced to the equivalence of states and (or) actions [29]. Sometimes, nondeterminism disappears after factorization. In order to justify this approach, a motivated implementation hypothesis is needed that assumes that all possible implementation errors can be detected when testing against the factorized specification (more generally, using a finite test suite satisfying a coverage criterion) [29].

A practical example is testing a finite state automaton against a specification defined in the form of a finite state automaton. If we have a special operation that can

directly poll the current state of the implementation (*status message*), then the complete testing reduces to a traversal of the transition graph of the automaton and getting the status message in each state [15]. The traversal of the transition graph is also used in the case of black box testing when no states of the implementation are available. In this case, for the testing to be complete, an implementation hypothesis is required that makes up for the impossibility to determine the states of the implementation [30–32]. This approach can be extended to the general case of the LTSs for input–output systems [33]. In particular, this concerns the so-called quiescent testing when the inputs are sent to the implementation only in its quiescent states (in this case, the problem of output blocking is also removed) [34].

15. INFINITE AND NEGATIVE OBSERVATIONS

An infinite observation is the possibility to conduct a test experiment infinitely long and obtain an infinite trace. For the conformance relations based on finite traces, an infinite trace does not add anything to the set of its finite prefixes. An exception is the direct observation of divergence and the λ-observation discussed above. Infinite observations and the conformance relations based on infinite traces are not usually considered to be important for practice.

The negation $\neg\sigma$ means that the trace σ does not belong to the set of observable traces that can be obtained using the given testing machine to check the given conformance relation. A negative observation assumes that we can (at least, theoretically) obtain all the observable traces. To this end, replication (at least, before the start of the work) and global testing are needed. In essence, a negative observation is an observation that is calculated from positive, i.e., actual, observations. When the replication is only made before the start of the work, the set of observable traces is the union of the sets of traces for each test case.

Had we allowed deadlocks in the interaction of the environment with the implementation, we could calculate some refusals as negative observations when there is no divergence. Indeed, let the trace σ be not continued by any action from $P \in \mathcal{R} \cup \mathcal{Q}$ in the set of observable traces; i.e., assume that $\neg\sigma \cdot \langle z\rangle$ for each $z \in P$. Then, in the implementation, σ is continued either by divergence or by the refusal of P. Therefore, in the absence of divergence, the trace $\sigma \cdot \langle P\rangle$ can be calculated. However, this does not imply that all the refusals can be observed because σ can be continued in the implementation both by the refusal of P or by some actions $z \in P$. To be able to observe all the refusals, replication at an arbitrary time moment is required. Moreover, there is a problem concerning continuation after a refusal. In order to be able to find out if there exists the trace $\sigma \cdot \langle P\rangle \cdot \mu$ by observing the trace $\sigma \cdot \mu$, we must determine whether or not the trace μ begins at a quiescent state after σ. If σ ends by a refusal R, then this is the case. However, how can we determine this refusal R? Thus, we see that even in the case when deadlocks and negative observations of refusals are allowed, we need some means for observing the machine stop—a green light or $\mathcal{R}$-refusal in the $\mathcal{R}/\mathcal{Q}$-machine.

The situation with the calculation of ready sets is similar. Without replication at an arbitrary time moment, we can only calculate the set of actions that continue a trace; i.e., we can calculate the union of the ready sets in the quiescent states after a trace. To determine the summands, we need to be able to do replication at an arbitrary time moment and to observe the machine stop.

If we do not allow divergence and destruction in the case of correct interactions, then the negative observations add nothing to checking the conformances based on the independence principle, which states that the behavior of an implementation is correct or incorrect *independently* of its other behaviors. The final verdict is the conjunction of the verdicts returned by all the runs of all test cases from the complete test suite.

16. PRIORITIES

We have already discussed the advantages of priorities in implementations and the problems concerning the testing of such implementations. Here, we want to note that the θ-transition, which is usually allowed only in tests, could also be used in implementations as one of the methods of specifying priorities. Actually, this transition is internal but has the minimal priority: it is performed only when none of the other transitions can be performed. In an implementation, the θ-transition can be used as a method for specifying the alternative behavior in the case of deadlocks. In particular, blocking outputs by the environment (when the outputs sent by implementation are rejected) does not present difficulties if an alternative behavior can be specified in this situation. Moreover, in a quiescent state of the implementation, the θ-action makes it possible for the implementation to detect the absence of inputs. This variant was studied in [34], where such a θ-action was called the ε-action. Unfortunately, priorities are poorly studied in testing theory.

17. CONCLUSIONS

We considered various testing capabilities specified using a testing machine. Among them, we distinguished a set of theoretically powerful and practically important capabilities that determine the $\mathcal{R}/\mathcal{Q}$ testing semantics. This semantics is constructed on the basis of observing external actions and refusals. The novelties are as follows.

(1) The semantics is parameterized by the families of observable and unobservable refusals, which enables us to take into account various constraints on (correct) interactions.

(2) Destruction is considered as a disabled action that is possible but must not be performed under correct interactions.

(3) The Δ-action provides a model of divergence; this action is also possible but must not be performed under correct interactions.

On the basis of this semantics, the concept of safe testing, the implementation safety hypothesis, and the safe conformance relation (*saco*) corresponding to the independent observation principle are proposed.

For a more narrow class of interactions, the $\mathcal{R}/\mathcal{Q}$-*Ready*-semantics based on ready traces cab be used along with the corresponding *resaco* relation. In contrast to $\mathcal{R}$-traces, the composition for the ready traces can be defined so that the ready traces in the composition of LTSs are identical to the composition of the ready traces in the LTS operands. This additivity property, along with the generativity property ($\mathcal{R}$-traces can be obtained from ready traces) make it possible to develop the closed trace theory that includes both conformance relation (even if it is based on $\mathcal{R}$-traces) and the composition of trace models. This makes ready traces useful in conformance theory independently of the possibility to observe them in practice.

We also formulated some propositions concerning the relationships between the relations *saco* and *resaco* in $\mathcal{R}/\mathcal{Q}$-, $\mathcal{R} \cup \mathcal{Q}/\varnothing$-, $\mathcal{R}/\mathcal{Q}$-*Ready*-, and $\mathcal{R} \cup \mathcal{Q}/\varnothing$-*Ready*-semantics. Transition to the next semantics in this list is performed using a completion transformation, which solves the conformance reflexivity problem, while the monotone transformation solves the monotonicity problem (conservation under composition). The proof of the assertions labeled by (!) is beyond the scope of this paper. In [20], these assertions are proved for input–output systems in which the quiescence and input refusals are the only observable refusals. The assertion concerning the *resaco* relation remains a hypothesis.

REFERENCES

1. Bernot, G., Testing against Formal Specifications: A Theoretical View, *TAPSOFT'91*, Abramsky, S. and Maibaum, T.S.F., Eds., vol. 2, pp. 99–119, *Lect. Notes Comput. Sci.*, 1991, vol. 494.
2. Milner, R., Modal Characterization of Observable Machine Behavior, *Proc. CAAP*, 1981, Astesiano, G. and Bohm, C. Eds., *Lect. Notes Comput. Sci.*, 1981, vol. 112, pp. 25–34.
3. Van Glabbeek, R.J., The Linear Time—Branching Time Spectrum, *Proc. of CONCUR'90*, Baeten, J.C.M. and Klop, J.W., Eds., *Lect. Notes Comput. Sci.,* 1990, vol. 458, pp. 278–297.
4. Van Glabbeek, R.J., The Linear Time—Branching Time Spectrum II: The Semantics of Sequential Processes with Silent Moves, *Proc. of CONCUR'93,* Hildesheim, Germany, 1993, Best, E., Ed., *Lect. Notes Comput. Sci.*, 1993, vol. 715, p. 66.
5. Lynch, N.A. and Tuttle, M.R., Hierarchical Correctness Proofs for Distributed Algorithms, *Proc. of the 6th ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, 1987, pp. 137–151.
6. Phalippou, M. Relations d'Implantation et Hypotheses de Test des Automates a Entrees et Sorties, *PhD Thesis*, l'Universite de Bordeaux 1, 1994.
7. Tretmans, J., Test Generation with Inputs, Outputs and Repetitive Quiescence, *Software Concepts and Tools*, 1996, vol. 17, issue 3.
8. Milner, R., A Calculus of Communicating Systems, *Lect. Notes Comput. Sci.*, 1980, vol. 92.
9. Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989.
10. Vaandrager, F., On the Relationship between Process Algebra and Input/Output Automata, *Logic in Computer Science, Sixth Annual IEEE Symposium*, IEEE Computer Society, 1991, pp. 387–398.
11. Jard, C., Jéron, T., Tanguy, L., and Viho, C., Remote Testing Can Be as Powerful as Local Testing, *Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII/PSTV XIX'99* Wu, J., Chanson, S., and Gao, Q., Eds., Beijing, 1999, pp. 25–40.
12. Van der Bijl, M., Rensink, A., and Tretmans, J., Compositional Testing with *ioco*, *Formal Approaches to Software Testing: Third Int. Workshop FATES*, Petrenko, A. and Ulrich, A., Eds., Montreal, 2003, *Lect. Notes Comput. Sci.,* 2003, vol. 2931, pp. 86–100.
13. Van der Bijl, M., Rensink, A., and Tretmans, J., Component Based Testing with *ioco*, *CTIT Technical Report, Univ. of Twente*, 2003, no. TR-CTIT-03-34.
14. Von Bochmann, G.V. and Petrenko, A., Protocol Testing: Review of Methods and Relevance for Software Testing, *Proc. of ISSTA*, 1994, pp. 109–124.
15. Lee, D. and Yannakakis, M., Principles and Methods of Testing Finite State Machines: A Survey, *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, Berlin: IEEE Computer Society, 1996.
16. Tretmans, J., Formal Approaches to Conformance Testing, *PhD. Thesis*, Enschede, Netherlands: Univ. of Twente, 1992.
17. Bourdonov, I.B. and Kossatchev, A.S., Testing Components of a Distributed System, *Trudy Vserossiiskoi konferentsii "Nauchnyi servis v seti Internet"* (Proc. of the All-Russia Conf. on the Research Services on the Internet), Moscow: Mosk. Gos. Univ., 2005, pp. 63–65.
18. Bourdonov, I.B. and Kossatchev, A.S., Verification of the Composition of a Distributed System, *Trudy Vserossiiskoi konferentsii "Nauchnyi servis v seti Internet"* (Proc. of the All-Russia Conf. on the Research Services on the Internet), Moscow: Mosk. Gos. Univ., 2005, pp. 67–69.
19. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions, *Proc. of MBT*, Vienna, 2006.
20. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., *Teoriya sootvetstviya dlya sistem s blokirovkami i razrusheniyami* (Conformance Theory for Systems with Refusals and Destruction), Moscow: Nauka (in press).
21. Heerink, L. and Tretmans, J., Refusal Testing for Classes of Transition Systems with Inputs and Outputs, in *For-*

mal Description Techniques and Protocol Specification, Testing and Verification, Chapman & Hill, 1997.

22. Heerink, L. Ins and Outs in Refusal Testing, *PhD Thesis*, Enschede, Netherlands: Univ. of Twente, 1998.
23. Petrenko, A., Yevtushenko, N., and Huo, J.L., Testing Transition Systems with Input and Output Testers, *Proc. 15th Int. Conf. on Communicating Systems, TestCom'2003*, Sophia, Antipolis, France, pp. 129–145.
24. Lestiennes, G. and Gaudel, M.-C., Test de Systemes Reactifs non Receptifs, *J. Europ. des Systemes Automatises, Modelisation des Systemes Reactifs*, 2005, pp. 255–270.
25. Hoare, C.A.R., An Axiomatic Basis for Computer Programming, *Commun. ACM*, 1969, vol. 12, no. 10, pp. 576–585.
26. Blass, A., Gurevich, Y., Nachmanson, L., and Veanes, M., Play to Test Microsoft Research, *Technical Report*, 2005, no. MSR-TR-2005-04; *5th Int. Workshop on Formal Approaches to Testing of Software (FATES 2005)*, Edinburgh, 2005.
27. Fujiwara, S. and von Bochmann, G. Testing Nondeterministic Finite State Machine with Fault Coverage, *Proc. IFIP TC6 Fourth Int. Workshop on Protocol Test Systems*, 1991, Kroon, J., Heijink, R.J., and Brinksma E., Eds., North-Holland, 1992, pp. 267–280.
28. Petrenko, A., Yevtushenko, N., and von Bochmann, G., Testing Deterministic Implementations from Nondeterministic FSM Specifications, *Selected Proc. of the IFIP TC6 9th Int. Workshop on Testing of Communicating Systems*, 1996.
29. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Application of Finite Automatons for Program Testing, *Programmirovanie*, 2000, no. 2, pp. 12-28 [*Programming Comput. Software* (Engl. Transl.), 2000, vol. 26, no. 2, pp. 61–73].
30. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Deterministic Case, *Programmirovanie,* 2003, no. 5, pp. 11-30 [*Programming Comput. Software* (Engl. Transl.), 2003, vol. 29, no. 5, pp. 245–258].
31. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Irredundant Algorithms for Traversing Directed Graphs: The Nondeterministic Case, *Programmirovanie*, 2004, no. 1, pp. 4–24 [*Programming Comput. Software* (Engl. Transl.), 2004, vol. 30, no. 1, pp. 2–17].
32. Kuliamin, V.V., Petrenko, A.K., Kossatchev, A.S., and Bourdonov, I.B., The UniTesK Approach to Designing Test Suites, *Programmirovanie*, 2003, no. 6, pp. 25-43 [*Programming Comput. Software* (Engl. Transl.), 2003, vol. 29, no. 6, pp. 310–322].
33. Kuliamin, V.V., Kossatchev, A.S., Petrenko, A.K., Pakoulin, N.V., and Bourdonov, I.B., Integration of Functional and Timed Testing of Real-Time and Concurrent Systems, in *Perspectives of System Informatics, Lect. Notes Comput. Sci.*, 2003.
34. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Asynchronous Automata: Classification and Testing, *Tr. ISP RAN* (Proceedings of Institute for System Programming, Russian Academy of Sciences), 2003, vol. 4, pp. 7–84.
35. De Nicola, R. and Hennessy, M.C.B., Testing Equivalence for Processes, *Theor. Comput. Sci.*, 1984, vol. 34, pp. 83–133.
36. Langerak, R. A., Testing Theory for LOTOS Using Deadlock Detection, in *Protocol Specification, Testing, and Verification IX*, Brinksma E., Scollo, G., and Vissers, C.A., Eds. North-Holland, 1990, pp. 87–98.