ISSN 0361-7688, Programming and Computer Software, 2009, Vol. 35, No. 4, pp. 198–211. © Pleiades Publishing, Ltd., 2009.
Original Russian Text © I.B. Bourdonov, A.S. Kossatchev, 2009, published in Programmirovanie, 2009, Vol. 35, No. 4.

Systems with Priorities: Conformance, Testing, and Composition

I. B. Bourdonov and A. S. Kossatchev

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

Received July 15, 2008

Abstract—An approach to modeling the components of distributed systems whose interaction is based on handling events with regard for their priorities is considered. Although the priority-based servicing of requests or messages is widely used in practice, the mathematical models of the interaction of such programs often neglect the priorities thus introducing extra nondeterminism in the description of their behavior. The proposed approach attempts to avoid this drawback by defining the parallel composition that provides a model for the interaction of this kind. The subject matter of this paper is the development of a formal theory of testing the components that use priorities. Within this theory, the concept of a safe execution of the model and the conformance relation between the models are introduced, and the generation of test suites that check conformity is considered.

DOI: 10.1134/S0361768809040045

INTRODUCTION

In the existing conformance testing theories, it is assumed that no priorities are imposed on the actions the system may perform in a certain situation [1]. This is called the nondeterministic rule of choice of the action to be performed. However, this rule does not always correspond to the desired behavior of software and hardware systems. Consider several examples.

Divergence exit. By divergence, we mean endless internal activity of the system (when it gets caught in an endless loop). A request arriving from the outside of the system can be neglected arbitrarily long if it has the same priority as the system's internal activity. Note that the internal activity can be initiated by an earlier request. If the system consists of several components, such an internal activity can be a natural result of the interaction between the components. In this case, in order to handle the request arriving from the outside of the system to one of its components, this request must have a higher priority than the internal interaction.

Oscillation exit (the priority of the input over the output). By oscillation, we mean an endless sequence of output messages produced by the system. To interrupt such a sequence and make the system handle an outside request, this request must have a higher priority than the message output.

Priority of the output over the input in infinite queues. This opposite example is characteristic of an infinite queue used as a buffer between interacting systems (for example, in asynchronous testing or testing in a context). In this case, we need that the retrieval from the queue have a higher priority than the addition to the queue. Otherwise, the queue can only receive messages but never output them. In asynchronous testing, this means that the input messages sent by the test do not reach the implementation but are rather endlessly accumulated in the queue; for the output queue, this means that the test cannot receive messages from the implementation although it generates them because these messages are accumulated in the queue.

Interrupting a chain of actions. The cancel instruction must terminate the sequence of actions initiated by a preceding request and initiate a chain of terminating actions. If there are no priorities, such an instruction, even if it is issued immediately after the request was generated, may be executed only after the request handling is actually finished; that is, it does not cancel anything.

Priority handling of inputs. If several requests simultaneously arrive in the system, it is often required that they be handled according to certain priorities. This requirement is often implemented using a priority queue or several queues with different priorities assigned to them. For example, hardware interruption handling in operating systems uses this type of priorities.

The absence of priorities in models of software and hardware systems does not allow one to test the requirements that can be stated only in the form of priorities. In this paper, we propose a technique for introducing priorities in the conformance theory. Namely, priorities are included in the semantics of the interaction and in the system model, in the conformance relation, in test generation methods, and in the composition operator (which assembles the system from interacting components). The conformance theory

without priorities is briefly described in [2]; a detailed presentation including all the proofs is given in the dissertation of one of the authors of this paper (see [3]). The conformance theory for the class of the so-called $\beta\gamma\delta$ semantics is presented in [4]. Here, we first recall the fundamental concepts of this theory, and then modify them for the case of priorities.

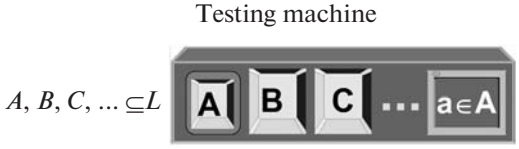


Fig. 1.

1. CONFORMANCE THEORY WITHOUT PRIORITIES

1.1. Semantics of the Interaction and Safe Testing

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

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

We stress that the operator may allow the machine to perform a set of actions (but not necessarily only a single action). For example, when testing reactive systems based on the exchange of inputs and responses, sending one input from the test to the implementation can be interpreted as the permission for the implementation to execute only one action, namely, to receive this input. However, the reception of the response by the test means that the implementation is allowed to produce an arbitrary response precisely to check whether or not this response is correct. We will assume that the operator presses a single button, but this allows the implementation to execute a set of actions. After an observation (of an action or a refusal), the button is automatically released, and all the external actions are disabled. Then, the operator may press another (or the same) button.

The "button" set is, generally, not an arbitrary subset of the set of all actions. There is a great variety of opinions among the researches concerning the sets of actions that can be enabled by a test and which sets are prohibited. For example, it is usually assumed that, for the reactive systems, it is not allowed (senseless) to mix sending inputs with receiving the responses (Tretmans). However, there is also the opposite approach stating that the implementation's responses must not be hampered; therefore, even when sending an input, the test must be ready to receive a response (Petrenko in [7]).

We also stress that the observation of a refusal is possible not for each button that is pressed. Concerning this issue, different researches also lean on different assumptions. For reactive systems, it has long been assumed that a test can observe the absence of response (*quiescence*) when a timeout has expired; however, it cannot detect whether or not the implementation receives an input (this is called *input refusal*). On the other hand, an increasing number of studies that have recently appeared admit or partially admit such input refusals [2, 4, 8–14]. Moreover, the responses, if they are received via different "output channels," can be received selectively; more precisely, the test can receive only the responses that go via one or several selected channels [12, 13].

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

In addition to the external observable actions, the implementation can execute internal (unobservable and, therefore indistinguishable for the operator) actions, which are denoted by τ. The execution of such actions is not controlled by the operator—they are

always enabled. It is assumed that any finite sequence of such actions terminates in a finite amount of time and an infinite sequence of actions terminates in an infinite amount of time. The infinite sequence of τ-actions (an infinite loop) is called *divergence*; it is denoted by Δ. We also define a special action called *destruction* that is not controlled by the buttons; this action is denoted by γ. It models any prohibited or unspecified behavior of the implementation. For example, in terms of the preconditions and postconditions, the behavior of a program is defined (by a postcondition) only if the corresponding precondition is fulfilled. Otherwise, if the precondition is not fulfilled, the program behavior is completely undefined. The destruction semantics assumes, in particular, that the system can be destructed as a result of such a behavior.

In testing, we should avoid unobserved refusals ($\mathfrak{Q}$-refusals), attempts to exit from the divergence, and destruction. Such a testing is said to be safe. The harmfulness of destruction is implied by its semantics. Let us explain the other cases. On the whole, their harmfulness is in that, after pressing a button, the operator does not always get an observation; hence, he can neither continue the testing nor finish it. If, after pressing the button P, there is the $\mathfrak{Q}$-refusal of P, the operator does not know whether to continue waiting for the observation of an external action enabled by the pressed button or it is senseless to wait because the machine stopped. However, the operator cannot find out whether the machine is stopped because this would mean that the refusal of P is observed. The divergence per se is not harmful, but, pressing any button after its occurrence and observing no external actions or an $\mathfrak{R}$-refusal, the operator does not know whether such an occurrence will happen later or the implementation will endlessly continue its internal activity.

Note that, due to the internal activity, pressing an empty $\mathfrak{R}$-button (the button with the empty set of enabled actions) is not equivalent to the absence of a pressed button. In both cases, all the external actions are not allowed; however, the observation of a refusal implies that the operator learns that the machine is stopped (when it cannot perform even the internal actions). An empty $\mathfrak{R}$-button cannot provoke a destruction after an action (because no actions can be performed), but it can be not safe (as any other button) if there is the divergence. An empty $\mathfrak{Q}$-button must never be pressed because no observations can be made; indeed, all the external actions are disabled and the refusals are not observed. Therefore, we may assume that $\emptyset \notin \mathfrak{Q}$-button.

1.2. LTS-Model and Trace Model

We will use a *labeled transition system* (LTS) as a model of the implementation and the specification. LTS is a directed graph with a distinguished initial vertex in which the arcs are labeled by certain symbols. Formally, LTS is the collection $\mathbf{S} = LTS(V_S, L, E_S, s_0)$, where V_S is a nonempty set of states (graph vertices), L is the alphabet of external actions, τ is the symbol of the internal action, γ is the symbol for destruction, $E_S \subseteq V_S \times (L \cup \{\tau, \gamma\}) \times V_S$ is the set of transitions (labeled arcs), and $s_0 \in V_S$ is the initial state (the initial vertex of the graph). The transition from the state s to the state s' by the action z is denoted by $s \xrightarrow{z} s'$. Define $s \overset{z}{\not\longrightarrow} =_{def} \nexists s'\ s \xrightarrow{z} s'$. The execution of an LTS placed in the black box of a testing machine is reduced to performing a transition defined in the current state and enabled by the pressed button (τ- and γ-transitions are enabled when any and none of the buttons is pressed). A state is called *stable* if no τ- and γ-transitions are defined in it; a state is said to be *divergent* if it begins an infinite chain of τ-transitions (in particular, a τ-cycle including a τ-loop). The refusal of P is induced by a stable state that contains no transitions initiated by the actions from P.

In order to obtain LTS traces, it suffices to add, in each stable state, virtual loops labeled by the refusals induced by this state and add Δ-transitions in all the divergent states. Then, all the finite routes in the LTS that begin in the initial state and do not continue beyond a Δ- or γ-transition are considered. The trace of a route is defined as the sequence of labels of its transitions in which the τ-transitions are omitted. Such traces are called *complete* or *F-traces*, and the set S of the F-traces of the LTS is called the *complete trace model* or the *F-model*; it is denoted by $F(\mathbf{S})$. An F-trace in which all the refusals belong to the family $\mathfrak{R}$ is called an $\mathfrak{R}$-trace. These are the traces that can be observed on the testing machine in the $\mathfrak{R}/\mathfrak{Q}$-semantics. The set of all $\mathfrak{R}$-traces of an LTS, that is the projection of its F-model on the alphabet consisting of all the external actions, $\mathfrak{R}$-refusals, and the symbols Δ and γ, is called the $\mathfrak{R}$-model corresponding to the "view" of the implementation in the $\mathfrak{R}/\mathfrak{Q}$-semantics.

1.3. Safety Hypothesis and Safe Conformance

Safe testing requires that a formal definition of the safety relation a button is safe in the model after an $\mathfrak{R}$-trace be given at the model level. Under safe testing, only safe buttons are pressed. This relation is different for the implementation and the specification models. In the LTS implementation $\mathbf{I}$, the safety relation *safe in* implies that pressing a button P after an $\mathfrak{R}$-trace σ cannot entail an attempt of exiting from the divergence (there can be no divergence after a trace), cannot entail destruction (after an action enabled by the button), and cannot entail a unobservable refusal (if this is a $\mathfrak{Q}$-button):

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

In the LTS specification **S**, the safety relation *safe by* is different only for the $\mathfrak{Q}$-buttons: we do not require that the trace σ be continued by a safe $\mathfrak{Q}$-refusal $\mathfrak{Q}$; rather, we require that it be continued by at least one action $z \in \mathfrak{Q}$. Moreover, if an action is enabled by at least one nondestructive $\mathfrak{Q}$-button, it must also be enabled by a safe button. If this is a nondestructive $\mathfrak{R}$-button, it is also safe. However, if all the nondestructive buttons that enable this action are $\mathfrak{Q}$-buttons, at least one of them must be declared safe. Such a safety relation always exists; indeed, it is sufficient to declare safe every nondestructive button that enables an action continuing a trace. Nevertheless, these requirements do not uniquely define the relation *safe by*, and a specific relation must be chosen when the specification is defined. The requirements for the relation *safe by* are written as follows: $\forall R \in \mathfrak{R}\ \forall z \in L\ \forall Q \in \mathfrak{Q}$,

(1) R *safe by* **S** *after* $\sigma \Leftrightarrow R\ safe_{\gamma\Delta}$ *in* **S** *after* σ,

(2) $\exists T \in \mathfrak{Q}\ T\ safe_{\gamma\Delta}$ *in* **S** *after* $\sigma\ \&\ z \in T\ \&\ \sigma \cdot \langle z\rangle \in F(\mathbf{S}) \Rightarrow \exists P \in \mathfrak{R} \cup \mathfrak{Q}\ z \in P\ \&\ P$ *safe by* **S** *after* σ,

(3) Q *safe by* **S** *after* $\sigma \Rightarrow Q\ safe_{\gamma\Delta}$ *in* **S** *after* $\sigma\ \&\ \exists v \in Q\ \sigma \cdot \langle v\rangle \in F(\mathbf{S})$.

The safety of buttons determines the safety of the actions and $\mathfrak{R}$-refusals after $\mathfrak{R}$-traces. The $\mathfrak{R}$-refusal R is safe if the button R is safe after the trace. An action is safe if it is enabled by a button that is safe after the trace. Now, we can define *safe traces*. An $\mathfrak{R}$-trace is said to be safe if the following holds. (1) The model is not destructed at the very beginning (immediately after the machine is switched on even before any button is pressed); that is, the model does not contain the trace $\langle\gamma\rangle$. (2) Every symbol of the trace is safe after the trace prefix that immediately precedes this symbol. The sets of safe traces in the implementation **I** and the specification **S** are denoted by *SafeIn*(**I**) and *SafeBy*(**S**), respectively.

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

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

Then, we can define the (safe) conformance relation: The implementation **I** *safely conforms* (or just *conforms*) with the specification **S** if it is safe and the following *condition under test* is fulfilled: any observation that is possible in the implementation as a response to pressing a safe (in the specification) button is allowed by the specification:

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

Here, $obs(\sigma, P, \mathbf{T}) =_{def} \{u | \sigma \cdot \langle u\rangle \in F(\mathbf{T})\ \&\ (u \in P \vee u = P\ \&\ P \in \mathfrak{R})\}$ is the set of observations of the model **T** that can be obtained by pressing the button P after the trace σ. Note that the safety hypothesis cannot be checked in the course of testing; this hypothesis is the testing precondition. In the test, the remaining part of the conformance condition is checked.

1.4. Parallel Composition and Test Generation

In the LTS theory, the interaction of two systems is modeled by the parallel composition operator. We use the composition operator similar to that used in the CCS (Calculus of Communicating Systems) [15, 16]. We assume that, for each external action z, an opposite action $\underline{z}$ is defined such that $\underline{\underline{z}} = z$. For example, for sending an input from the test, the opposite action is receiving the input in the implementation; and, for producing a response by the implementation, the opposite action is the reception of this response in the test. The parallel execution of two LTSs over the alphabets A and B is interpreted in such a way that the transitions corresponding to mutually opposite actions z and $\underline{z}$, where $z \in A$ and $\underline{z} \in B$, are executed simultaneously in both LTSs; in the composition, this pair of transitions becomes a τ-transition. Such transitions are called synchronous. The other external actions $z \in A\backslash\underline{B}$, $z \in B\backslash\underline{A}$, and τ and γ are called asynchronous. The transition corresponding to such an action is executed in one of the LTSs, while the state of the other LTS remains unchanged. The result of the composition of two LTSs **I** and **T** is the LTS $\mathbf{I} \uparrow\downarrow \mathbf{T}$ over the alphabet $A \uparrow\downarrow B =_{def} (A\backslash\underline{B}) \cup (B\backslash\underline{A})$. Its sates are the pairs of states *it* of the LTS operands, the initial state is the pair of the initial states of the original LTSs, and the transitions are induced by the following inference rules:

(1) $z \in \{\tau, \gamma\} \cup A\backslash\underline{B}\ \&\ i \xrightarrow{z} i' \vdash it \xrightarrow{z} i't$,

(2) $z \in \{\tau, \gamma\} \cup B\backslash\underline{A}\ \&\ t \xrightarrow{z} t' \vdash it \xrightarrow{z} it'$,

(3) $z \in A \cup \underline{B}\ i \xrightarrow{z} i'\ \&\ t \xrightarrow{\underline{z}} t' \vdash it \xrightarrow{\tau} i't'$.

Testing is regarded as the closed composition of the LTS implementation **I** over the alphabet A and the LTS test **T** over the opposite alphabet $B = \underline{A}$. To detect refusals in the test (but not in the implementation!), special θ-transitions are allowed that fire if and only if no other transitions can be performed:

(4) $t \xrightarrow{\theta} t'\ \&\ Deadlock(i, t) \vdash it \xrightarrow{\tau} it'$.

Here, $Deadlock(i, t) = i \overset{\tau}{\not\longrightarrow}\ \&\ i \overset{\gamma}{\not\longrightarrow}\ \&\ t \overset{\tau}{\not\longrightarrow}\ \&\ t \overset{\gamma}{\not\longrightarrow}\ \&\ (\forall z \in A \cap \underline{B}\ i \overset{z}{\not\longrightarrow} \vee t \overset{\underline{z}}{\not\longrightarrow})$. We assume that the test does not contain destruction (the requirement $t \overset{\gamma}{\not\longrightarrow}$ is always satisfied).

Since the alphabets of the implementation and of the test are mutually opposite, the composition alphabet is empty, and the composition LTS contains only τ- and γ-transitions. Under safe testing, τ-transitions are unreachable. The execution of a test is preceded by passing a τ-route that begins at the initial state of the composition $\mathbf{I} \uparrow\downarrow \mathbf{T}$. The test ends when a terminal state of the test is attained. Each terminal state is assigned the verdict *pass* or *fail*. An implementation *passes* a test if the test states with the verdict *fail* are unreachable. An implementation passes a suite of tests if it passes every test in this suite. A suite of tests is called *significant* if it is passed by every conformal implementation; a suite of tests is called *exhaustive* if no nonconformal implementations pass it. A suite of tests is *complete* if it is both significant and exhaustive. The task is to generate a complete suite of tests given a specification.

Usually, only the so-called *controllable* tests are considered, that is, the tests without excessive nondeterminism. To be controllable, the set of external actions for which transitions in the given test state are defined must be one of the button sets of the $\Re/\mathfrak{Q}$-semantics (more precisely, it must be a set of opposite actions because the tests are defined over the opposite alphabet in the CSS composition). When performing a test, the operator uniquely determines which button must be pressed in every given state of the test. If this as an $\Re$-button, a θ-transition must be defined in the test state.

The set of *primitive* tests is always complete. A primitive test is constructed on the basis of a distinguished safe $\Re$-trace of the specification. To this end, before every $\Re$-refusal R, the button R is included, and before every action z, any one safe (after the trace prefix) button P is included that enables the action z. The safety of the trace guarantees that the button R is safe and that an appropriate safe button P exists. The button P is generally, not unique; therefore, given the same safe trace, many different primitive tests can be generated. To differentiate between buttons and refusals (both are subsets of the external actions), we will enclose buttons in double quotes and write "P" instead of P. After all the buttons have been specified, we obtain a sequence, which is called the $\Re/\mathfrak{Q}$-log (or history) in the next section. This sequence provides the basis for constructing the LTS test (Fig. 2). The states of this test are the specified buttons, the initial state is the first button in the sequence, the symbols of transitions from the button-state are the actions that are opposite to those that can be observed after pressing this button or the symbol θ if this is an $\Re$-button. If this button is not the last one, one of the transitions leads to the state corresponding to the next button. The other transitions lead to terminal states. The verdict *pass* is assigned if the specification contains the corresponding trace; otherwise, the verdict *fail* is returned. Such a verdict corresponds to *strict* tests; these are the tests that, first, are significant (do not detect false bugs) and, second, do not miss detected bugs. Any strict test can be replaced by a union of primitive tests that detects the same bugs.

2. CONFORMANCE THEORY WITH PRIORITIES

2.1. Predicates on the Transitions of the LTS Model

Independently of the presence or absence of priorities, the interaction semantics assumes that only the transitions that are specified in the implementation and are enabled by the operator of the testing machine can be performed. In the absence of priorities, any defined and enabled action can be executed, and the choice of this action is nondeterministic. The presence of priorities implies that not all the defined and enabled actions can be executed; in other words, the executability of an action depends on the other defined and (or) enabled actions. This dependence can be illustrated by a predicate of the set of enabled actions that is assigned to the corresponding transition in the LTS model. Since the preceding state s is known for the transition $s \xrightarrow{z} s'$ and the other transitions that begin at this state are known, the predicate assigned to a transition can be assumed to be independent of the set of defined (in the state s) actions. In other words, transitions corresponding to the same action beginning at different states can have different predicates.

An LTS with priorities is an LTS with the alphabet defined as the Cartesian product of the alphabet of the external actions and the set of predicates over the alphabet of the external actions $\Pi = \{\pi : \mathcal{P}(L) \longrightarrow Bool\} : \mathbf{S} = LTS(V_S, L \times \Pi, E_S, s_0)$. The transition $s \xrightarrow{z, \pi} s'$ can be performed only if the operator enables a set of external actions $R \subseteq L$ such that $z \in R \cup \{\tau, \gamma\}$ and $\pi(R) = true$. If there are several executable actions, one of them is actually executed (as before, the choice of this action is nondeterministic).

The predicate may be regarded as a Boolean function of Boolean variables z_1, z_2, ... that are in one-to-one correspondence with the corresponding external actions over the alphabet L. For example, for the predicate $\pi = a\&\neg b \vee c$, the transition $s \xrightarrow{z, \pi} s'$ can be performed only if the operator enabled a set of external actions R such that $z \in R \cup \{\tau, \gamma\}$ & ($a \in R$ & $b \notin R \vee c \in R$). This implies that z is an internal action, the destruction, or an external action enabled by the operator and that at least one of the two conditions is fulfilled: (1) the operator enabled the action a and disabled the action b; (2) the operator enabled the action c.

Therefore, a predicate is a Boolean function of the set of enabled actions. An important particular case is when the predicate depends only on the enabled and *defined* external actions. In other words, the predicate on the transition $s \xrightarrow{z, \pi} s'$ is independent of the Boolean variables that correspond to the external actions that do not have corresponding transitions from the state s. This implies that the executability of a transition depends only on whether the action z is enabled and what other actions are defined in the state s and are enabled by the operator. In this case, the actions that were enabled by the operator but cannot be executed because they are not defined in the current state of the implementation are of no importance for the implementation (it "does not see" them). By pressing a button, the operator sort of "highlights" certain actions of the implementation that are defined in its current state, and the executability of a transition corresponding to the highlighted action is determined by the corresponding predicate of the set of the highlighted actions.

Being a Boolean function of actions, the predicate can be represented by a principal disjunctive normal form $\pi = \eta_1 \vee \eta_2 \vee ...$, where $\eta_i = x_{i1}$ & x_{i2} & ... or $x_{ij} = \neg z_j$ and z_j runs over the set of all external actions. Then, the transition $s \xrightarrow{z, \pi} s'$ can be replaced by the set of multiple transitions of which the predicates are the clauses of $s \xrightarrow{z, \eta_i} s'$. In turn, the clause η_i is in one-to-one correspondence with the set of actions P_i for which $x_{ij} = z_j$. For the LTS composition, this set is the set of actions enabled by the other operand of the cpmposition. In the case when this operand is a test for the given $\mathfrak{R}/\mathfrak{Q}$-semantics, these sets of enabled actions correspond to the buttons contained in $\mathfrak{R} \cup \mathfrak{Q}$. Therefore, we may assume that the buttons (i.e., transitions) $s \xrightarrow{z, P_i} s'$ are assigned to the transitions rather then the arbitrary predicates. When a button P_i is pressed, only the transitions $s \xrightarrow{z, P_i} s'$, where $z \in P_i \cup \{\tau, \gamma\}$ can be performed. Such a transition from an LTS with predicates to an LTS with buttons is similar to the transition from the Extended Finite State Machines (EFSMs) to the conventional FSMs).

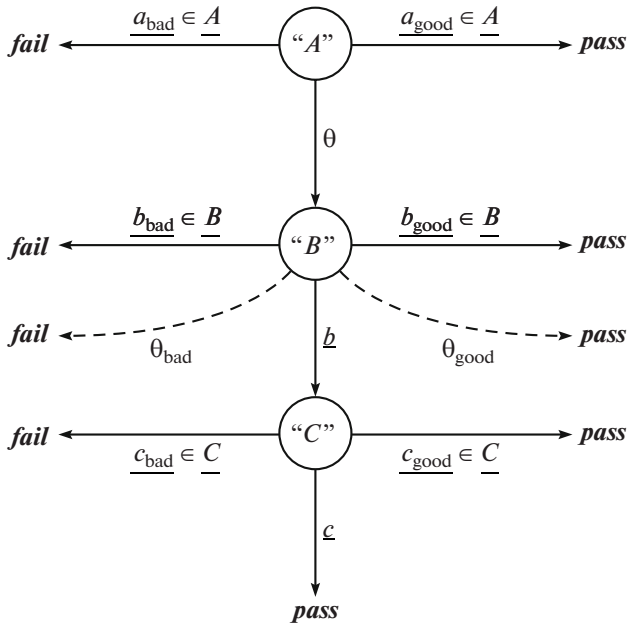
2.2. Stop and Divergence

A machine without priorities stops at at a stable (quiescent) state in which there are no transitions corresponding to the enabled external action. In the presence of priorities, the very concept of stability is changed. It becomes conditional; namely, a state is stable if, for all the transitions from this state, their predicates of the empty set of the enabled external actions are false, that is, if $\pi(\emptyset) = \mathit{false}$. Respectively, the stopping criterion is changed: the machine stops if, for the enabled set of external actions P, for all the transitions from the current state, their predicates are false, that is, if $\pi(P) = \mathit{false}$.

Here, we must define more precisely what happens when a button is released. For a machine without priorities, any button is automatically released when any action or refusal is observed. After an action, the machine can execute arbitrary τ-transitions (and also the γ-transition); however, the machine stops after the refusal because refusals occur in a quiescent state in which there are no τ- and γ-transitions. For a machine with priorities, releasing a button changes the set of enabled actions (in an empty $\mathfrak{R}$-button, which invokes the isolated observation of the empty refusal, was not pressed). After an observation, the implementation starts performing τ-transitions having the priority $\pi(\emptyset) = \mathit{true}$. Note that this observation may be not only an action but a refusal as well. The cause is that the refusal P means the impossibility to perform the enabled external actions $z \in P$ and τ- and γ-transitions because their predicates became false: $\pi(P) = \mathit{false}$. After the button P is released, the set of enabled external actions is empty, and now τ- and γ-transitions with the predicates $\pi(\emptyset) = \mathit{true}$ may be performed. Further, the operator can press the same button (but the repeated observation of the same refusal is not guaranteed if the implementation changed its state due to τ-transitions) or another button.

If button switching is allowed (that is, if the next button may be pressed before an observation after pressing the preceding button is made), such a switching is interpreted as releasing the preceding button and then pressing the next one. We assume that, in between pressing two buttons, there is a situation in which no buttons are pressed, and the implementation may perform τ- and γ-transitions with the predicates $\pi(\emptyset) = \mathit{true}$. The general rule is that the situation when there is no test input occurs every time when the machine is switched on (before the first button is pressed), after any observation, and between two test inputs. In the next subsection, the button switching is discussed in more detail.

We have already noted that, even for the machine without priorities, the divergence is inconvenient not per se but because it is difficult to quit it. In the presence of priorities, if an external action has a higher priority than the internal activity, the divergence stops.



A primitive test for the $\mathfrak{R}$ trace σ

$\sigma = \langle A, b, c\rangle \Rightarrow$ insert safe buttons $\Rightarrow$ $\langle$"A", A, "B", b, "C", $c\rangle$ such that $A, B \in \mathfrak{R}$, $C \in \mathfrak{Q}$, $b \in B$, $c \in C$.

The external action has the subscript "good" if it is included in the specification after the trace prefix; otherwise, it has the subscript "bad". Dashed lines show mutually exclusive θ-transitions beginning in the same state. The subscripts "good" and "bad" denote the presence and, respectively, absence of the continuation of the trace prefix by the corresponding $\mathfrak{R}$-refusal.

Fig. 2.

Now, the executability of τ-actions depends on the pressed button, and we can indirectly control those actions and, therefore, control the divergence. Consequently, we may speak of the *executable* divergence; namely, for some pressed buttons (or when no buttons are pressed), all the τ-actions of the infinite chain are executable; for other pressed buttons, they are not executable; therefore, there is no endless loop. One can quit the divergence that occurs when a button A is pressed by pressing a button B for which the divergence is not executable. Note that, for that purpose, some buttons must be switched; that is, a button must be pressed without observing the result of pressing the preceding button. The only situation in which the divergence cannot be quit for sure is the case when the divergence is executable after pressing any button.

2.3. Button Switching

In the machine without priorities, a button may be pressed either after the machine is turned on or after an observation after pressing the preceding button is made. In other words, buttons may not be switched without observation (that is, one cannot release one button and press another). This inhibition is explained by the fact that, in the absence of priorities, the possibility of switching buttons does not enhance the capacity of testing. Indeed, if a button P was pressed, and then another button Q was pressed (and P was released) without making an observation, the implementation could only perform τ-actions between pressing these buttons. However, the τ-actions are always enabled; therefore, the implementation could also perform these actions when the button Q was pressed immediately without pressing P. Consequently, any behavior that can be observed in the first case can also be observed in the second case.

In the presence of priorities, switching without observation is necessary for the completeness of testing because different sets of enabled actions have a different effect on the execution of τ-actions (τ-transitions can have predicates as well), which results in externally distinct behavior. For example, suppose that the reception of an input in a reactive system has a higher priority than producing responses and performing τ-actions. Then, the τ-actions are executed if and only if the implementation cannot receive an input sent by the test. In the example shown in Fig. 3, in order to obtain the response $!y$ after the input $?a$, this input must not be sent directly (in this case, the response will be $!x$); rather, the input $?b$ must be sent first, and then the button $\{?b\}$ must be switched over to $\{?a\}$ thus sending the input $?a$. If the implementation receives the input $?b$, the buttons must be switched before the input $?b$ is received. If the implementation refuses $?b$ (there is not dotted transition), one does not need to hurry; if the refusal of $?b$ is observable, one may wait for it, and then send the input $?a$.

However, there are strong reasons for prohibiting button switching in the presence of priorities. The point is the switching of buttons makes it possible to evade unobservable refusals. Indeed, if the operator switches the $\mathfrak{Q}$-button P over to any other button Q, the occurrence of an unobservable refusal of P does not prevent such a switching (as illustrated in Fig. 3, when there is no dotted transition corresponding to the input $?b$ and the refusal of $\{?b\}$ is not observable). From a different point of view, if the refusal of P is pos-

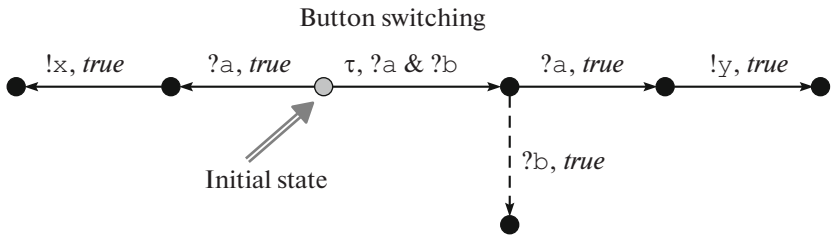


Fig. 3.

sible and the testing is safe, we must not press the button P without the subsequent switching to another button (that is, with waiting for an observation) (in Fig. 3, the button $\{?b\}$ must be always switched to the button $\{?a\}$ or some other button). Therefore, if the switching of buttons is allowed, the condition for the safety of Ω-buttons is more complicated (it will be thoroughly considered below). If the switching of buttons is allowed, testing in the presence of priorities looks more conventional, namely, as an alternating sequence of test inputs (a button is pressed) and observations. Furthermore, less temporal requirements are placed upon the operator in this case.

2.4. Temporal Constraints on the Operator (Test) Actions

The introduction of priorities complicates the operator's work by imposing stricter temporal constraints. In the absence of priorities, the operator must know how to quickly press a button after the machine is turned on or after the preceding observation. Note that, if the operator has not managed to press the button quickly enough, there is nothing to worry of because the machine has time to perform only one or several τ-actions that (in a machine without priorities) it can also perform in the case when the button was pressed immediately. In other words, we require that the operator could work quickly but do not make him always work quickly.

In the presence of priorities, the observation of various behaviors of the implementation requires from the operator not only swift, but also slow, moderate, and so on work. In Fig. 4, the input $?a$ can be received in three states 1, 2, and 3; however, the responses are

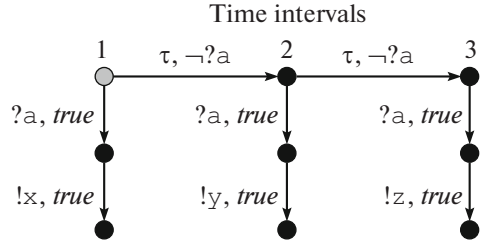


Fig. 4.

different: $!x$, $!y$, or $!z$. These states are connected by τ-transitions that are executable only if the test does not send the input $?a$. Therefore, the response $!x$ is observed only if the operator quickly presses the button $\{?a\}$, the response $!y$ is observed only if the operator does not hurry, and the response $!z$ is observed only if the operation speed is moderate.

If the button switching is allowed, one must know how to perform switching with different time intervals so as to "make" the implementation perform the desired number of τ-transitions between pressing two buttons.

Therefore, for a machine with priorities, one must take into account the time delays that are made by the operator between the observation and the subsequent button pressing or between button pressings when they are switched without observation. We may assume that the "weather conditions" that determine the nondeterministic choice also include the factors that affect the "free will" of the operator thus determining the time delays between button pressings. This is in agreement with the requirement that the operator must simulate arbitrary rate of the environment operation. The operator models the execution of a test program on a computer. Such a program is nondeterministic only at a certain level of abstraction, when we neglect other programs or the hardware affecting its behavior.

2.5. Logs

In the absence of priorities, the possibility of observing an action after a certain history of interactions is independent of which button enabling this action is pressed. In the presence of priorities, this is important because different buttons are associated with different sets of enabled actions; therefore, the corresponding predicate can be true when a certain button is pressed (then, the action can be observed), and it can be false when another button is pressed (then, the action cannot be observed). Therefore, we must keep in memory not only the observations but also the buttons that were pressed. Hence, the result of a test experiment is a sequence of actions, refusals, and buttons. Such a sequence will be called *history* or *log*. If we do not restrict ourselves to safe testing, we also must include in the log the destruction and divergence; however, the log cannot be continued after

them (much like traces). It is clear that every external action z in the log is immediately preceded by the button "P" that enabled this action $z \in P$; every refusal R is preceded by an $\mathfrak{R}$-button "R." Whether two buttons may follow one the other in succession depends on whether or not button switching is allowed.

For the given $\mathfrak{R}/\mathfrak{Q}$-semantics, the logs are called $\mathfrak{R}/\mathfrak{Q}$-logs. Let us define them more formally.

Consider an LTS with the predicates **S**. For the set of enabled actions P, the transition $s \xrightarrow{z, \pi} s'$ is said to be *P-executable* if its predicate is true, that is, if $\pi(P) =$ *true*. A τ-route is said to be *P-executable* for the set of enabled actions P if all its transitions are *P-executable*.

The empty $\mathfrak{R}/\mathfrak{Q}$-log ends at the states that can be attained from the initial state using $\emptyset$-executable τ-routes, that is, if no buttons are pressed after starting the machine. Let the $\mathfrak{R}/\mathfrak{Q}$-log end in the set of states **S** *after* σ. Consider various variants of continuing such an $\mathfrak{R}/\mathfrak{Q}$-log. We assume that the $\mathfrak{R}/\mathfrak{Q}$-log does not end by the destruction or divergence because there can be no continuation after them.

Continuation by the button P, where $P \in \mathfrak{R} \cup \mathfrak{Q}$. If button switching is allowed, such a continuation is always possible. Otherwise, the $\mathfrak{R}/\mathfrak{Q}$-log must not end in a button. Switching is interpreted as releasing one button and then pressing another one. Therefore, the implementation can first execute any $\emptyset$-executable τ-route beginning at a state in **S** *after* σ and then continue the execution by any P-executable τ-route. The set of endpoints of such routes forms the set of states **S** *after* $\sigma \cdot \langle$"P"$\rangle$. Note that, if the log was not ended by a button, the endpoints of all the $\emptyset$-executable τ-routes already belong to **S** *after* σ.

Continuation by an external action z. Such a continuation is possible only if the $\mathfrak{R}/\mathfrak{Q}$-log has the form $\sigma \cdot \langle$"P"$\rangle$, that is, if it ends by a button P enabling this action ($z \in P$). The action z is observed when a P-executable transition by z from the state after the preceding $\mathfrak{R}/\mathfrak{Q}$-log is performed; this is a transition $s \xrightarrow{z, \pi} s'$, where $s \in$ (**S** *after* $\sigma \cdot \langle$"P"$\rangle$) and $\pi(P) =$ *true*. As a result of such a transition, the button is automatically released, and $\emptyset$-executable τ-routes can be executed until there is destruction, or a button (the same or another) is pressed, or the operator turns the machine off thus terminating the test session. The set of endpoints of these τ-routes forms the set **S** *after* $\sigma \cdot \langle$"P", $z\rangle$. Note that the states at which a refusal was observed are also included in this set (for the empty τ-route).

Continuation by the $\mathfrak{R}$-refusal P. Such a continuation is possible only if the $\mathfrak{R}/\mathfrak{Q}$-log has the form $\sigma \cdot \langle$"P"$\rangle$. The refusal of P occurs in the state $s \in$ (**S** *after* $\sigma \cdot \langle$"P"$\rangle$) in which the machine stopping condition is fulfilled: for every transition (by any action including τ and γ) $s \xrightarrow{z, \pi} s'$, it must be $\pi(P) =$ *false*. After the refusal, the button is released, and the implementation can execute an $\emptyset$-executable τ-route beginning in one of the states in which the refusal was observed. The set of endpoints of these τ-routes forms the set **S** *after* $\sigma \cdot \langle$"P", $P\rangle$. Note that the states in which the refusal was observed also belong to this set.

Continuation by the destruction γ. Such a continuation is possible only if the transition $s \xrightarrow{z, \pi} s'$ is P-executable in a state $s \in$ (**S** *after* σ) if the $\mathfrak{R}/\mathfrak{Q}$-log ends by the button P (no observation has been made yet and the button P continues to be active) or if this transition is $\emptyset$-executable (no buttons are active after the observation). Since there is no continuation after the destruction, we are not interested in the set of states after this kind of continuation.

Continuation by the divergence Δ. Since the divergence is not harmful per se but an attempt to quit it is harmful, we are interested only in such kind of divergences that are executable when a button P is pressed. Such kind of divergence occurs after an $\mathfrak{R}/\mathfrak{Q}$-log of the form $\sigma \cdot \langle$"P"$\rangle$ if there is an endless P-executable τ-route starting at a state in **S** *after* $\sigma \cdot \langle$"P"$\rangle$ (it is clear that it is sufficient to assume that this route begins in **S** *after* σ). In this case, the symbol Δ continues the $\mathfrak{R}/\mathfrak{Q}$-log after the button P. Since there is no continuation after the divergence, we are not interested in the set of states after this kind of continuation.

Much like traces, we define *complete logs* or *F-logs* as $\mathfrak{R}/\mathfrak{Q}$-logs for $\mathfrak{R} = \mathcal{P}(L)$ and, respectively, for $\mathfrak{Q} = \emptyset$ when any subset of external actions is an $\mathfrak{R}$-button. The set of *F-logs* LTS **S** will be denoted by the same symbol as the set of F-traces F(**S**) because we will only consider logs rather than traces. An $\mathfrak{R}/\mathfrak{Q}$-log of an LTS is an F-log of this system in which there are only buttons from $\mathfrak{R}$ and $\mathfrak{Q}$ and only $\mathfrak{R}$-refusals.

2.6. Safety and Conformance without Button Switching

Since the executability of transitions of the LTS model with priorities depends on predicates defined on these transitions, the button safety relations in the implementation (*safe in*) and in the specification (*safe by*) are changed.

If the button switching is not allowed, the relations *safe in* and *safe by* are defined almost in the same manner as for the machines without priorities except for the following points: instead of $\mathfrak{R}$-traces, $\mathfrak{R}/\mathfrak{Q}$-logs are considered, the safety or harmfulness of a button is defined only after the $\mathfrak{R}/\mathfrak{Q}$-logs that do not end by a button, continuation by an external action depends on the button, divergence is possible only after a button, and no destruction is possible not only after an action but also after a refusal (for an $\mathfrak{R}$-button) and immediately after a button is pressed.

Here is the definition of safety in the implementation without button switching:

$$P\ safe_\gamma\ in\ \mathbf{I}\ after\ \sigma =_{def} \sigma \cdot \langle \text{“}P\text{”}, \gamma\rangle \notin F(\mathbf{I})$$
$$\& \ \forall u \in P\ \sigma \cdot \langle \text{“}P\text{”}, u, \gamma\rangle \notin F(\mathbf{I})$$
$$\& \ (P \in \Re \Rightarrow \sigma \cdot \langle \text{“}P\text{”}, P, \gamma\rangle \notin F(\mathbf{I})).$$
$$P\ safe_{\gamma\Delta}\ in\ \mathbf{I}\ after\ \sigma =_{def} P\ safe_\gamma\ in\ \mathbf{I}$$
$$after\ \sigma\ \underline{\underline{\& \ \sigma \cdot \langle \text{“}P\text{”}, \Delta\rangle \notin F(\mathbf{I}).}}$$
$$P\ safe_1\ in\ \mathbf{I}\ after\ \sigma =_{def} P\ safe_{\gamma\Delta}\ in\ \mathbf{I}$$
$$after\ \sigma\ \underline{\underline{\& \ (P \in \mathfrak{Q} \Rightarrow \sigma \cdot \langle \text{“}P\text{”}, P\rangle \notin F(\mathbf{I})).}}$$

The requirements for the safety relation in the specification without button switching are as follows: $\forall R \in \Re\ \forall z \in L\ \forall Q \in \mathfrak{Q}$,

(1) $R\ safe_1\ by\ \mathbf{S}\ after\ \sigma \Leftrightarrow R\ safe_{\gamma\Delta}\ in\ \mathbf{S}\ after\ \sigma$,

(2) $\exists T \in \mathfrak{Q}\ T\ safe_{\gamma\Delta}\ in\ \mathbf{S}\ after\ \sigma\ \&\ \sigma \cdot \langle \text{“}T\text{”}, z\rangle \in F(\mathbf{S}) \Rightarrow \exists P \in \Re \cup \mathfrak{Q}\ P safe_1 by\ \mathbf{S}\ after \sigma\ \&\ \sigma \cdot \langle \text{“}P\text{”}, z\rangle \in F(\mathbf{S})$,

(3) $Q\ safe_1 by\ \mathbf{S}\ after\ \sigma \Rightarrow Q\ safe_{\gamma\Delta}\ in\ \mathbf{S}\ after\ \sigma$ $\underline{\underline{\& \ \exists v \in Q\ \sigma \cdot \langle \text{“}Q\text{”}, v\rangle \in F(\mathbf{S})}}$.

The button safety relations in the implementation and in the specification provide a basis for defining safe actions, safe $\Re/\mathfrak{Q}$-logs $Safe_1In(\mathbf{I})$ and $Safe_1By(\mathbf{S})$, the safety hypothesis, and the safety conformance much like it was done for the traces in the case of the machines without priorities. The differences are as follows:

(1) In the definition of safety of $\Re/\mathfrak{Q}$-logs and in the safety hypothesis, one must use the log $\langle\gamma\rangle$ rather than the trace $\langle\gamma\rangle$; that is, this definition involves the destruction that is executable without pressing any buttons ($\emptyset$-executable):

$$\mathbf{I}\ safe\ for\ \mathbf{S} =_{def} (\langle \text{“}\emptyset\text{”}, \gamma\rangle \notin F(\mathbf{S})$$
$$\Rightarrow \langle \text{“}\emptyset\text{”}, \gamma\rangle \notin F(\mathbf{I}))$$
$$\& \ \forall \sigma\ Safe_1By(\mathbf{S}) \cap Safe_1In(\mathbf{I})$$
$$\forall P \in \Re \cup \mathfrak{Q}(P\ safe_1 by\ \mathbf{S}\ after\ \sigma$$
$$\Rightarrow P\ safe_1\ in\ \mathbf{I}\ after\ \sigma).$$

(2) In the definition of the set of observations that can be obtained over the model $\mathbf{T}$ by pressing the button P after the trace σ, one must use the continuation of the $\Re/\mathfrak{Q}$-log by a button and by an observation rather than the continuation of the trace by an observation:

$$obs(\sigma, P, \mathbf{T}) =_{def} \{u \,|\, \sigma \cdot \langle \text{“}P\text{”}, \gamma\rangle \in F(\mathbf{T})$$
$$\& \ (u \in P \vee u = P \ \&\ P \in \Re)\}.$$
$$\mathbf{I}\ saco\ \mathbf{S} =_{def} \mathbf{I}\ safe\ for\ \mathbf{S}$$
$$\& \ \forall \sigma \in Safe_1By(\mathbf{S}) \cap Safe_1In(\mathbf{I})$$
$$\forall P\ safe_1 by\ \mathbf{S}\ after\ \sigma$$
$$obs(\sigma, P, \mathbf{I}) \subseteq obc(\sigma, P, \mathbf{S}).$$

2.7. Safety and Conformance with Button Switching

If button switching is allowed, we can evade the prohibition of unobservable refusals after pressing a $\mathfrak{Q}$-button Q and the prohibition of divergence simply by switching the button Q to another button P. The safety relations are modified accordingly: the doubly underscored clauses in the definitions of $safe_1 in$ and $safe_1 by$ are removed, and only the conditions concerning destruction remain:

$$P\ safe_2 in\ \mathbf{I}\ after\ \sigma$$
$$=_{def} P\ safe_\gamma in\ \mathbf{I}\ after\ \sigma.$$

Here are the requirements for the safety relation in the specification with button switching: $\forall R \in \Re\ \forall z \in L\ \forall Q \in \mathfrak{Q}$,

(1) $R\ safe_2\ by\ \mathbf{S}\ after\ \sigma \Leftrightarrow R\ safe_\gamma\ in\ \mathbf{S}\ after\ \sigma$,

(2) $\exists T \in \mathfrak{Q}\ T safe_\gamma\ in\ \mathbf{S}\ after \sigma\ \&\ \sigma \cdot \langle \text{“}T\text{”}, z\rangle \in F(\mathbf{S}) \Rightarrow \exists P \in \Re \cup \mathfrak{Q}\ P safe_2\ by\ \mathbf{S}\ after \sigma\ \&\ \sigma \cdot \langle \text{“}P\text{”}, z\rangle \in F(\mathbf{S})$,

(3) $Q\ safe_2\ by\ \mathbf{S}\ after\ \sigma \Rightarrow Q\ safe_\gamma\ in\ \mathbf{S}\ after\ \sigma$.

There is a question: how many times can the operator switch buttons? We must take into account that the ultimate aim of pressing buttons is making an observation. Since we consider finite (in terms of the execution time) tests, the chain of button switching is finite; that is, it must end by pressing a button after which the operator expects a guaranteed observation, which, in particular, makes it possible to end the testing session. This means that all the buttons in the chain except for the last one are safe with respect to the relation $safe_2 in/by$ after the log prefix that immediately precedes them, and the last button is safe with respect to $safe_2\ in/by$:

$$P\ safe\ in\ \mathbf{I}\ after\ \sigma$$
$$=_{def} \exists P_0 = P, P_1, \ldots, P_n \in \Re \cup \mathfrak{Q}$$
$$\& \ \forall i = 0 \ldots n-1 P_i\ safe_2\ in\ \mathbf{I}$$
$$after\ \sigma \cdot \langle \text{“}P_0\text{”}, \text{“}P_1\text{”}, \ldots, \text{“}P_{i-1}\text{”}\rangle$$
$$\& \ P_n\ safe_1\ in\ \mathbf{I}\ after\ \sigma \cdot \langle \text{“}P_0\text{”}, \text{“}P_1\text{”}, \ldots, \text{“}P_{n-1}\text{”}\rangle,$$
$$P\ safe\ by\ \mathbf{S}\ after\ \sigma =_{def} \exists P_0 = P, P_1, \ldots, P_n$$
$$\in \Re \cup \mathfrak{Q}\ \&\ \forall i = 0 \ldots n-1\ P_i\ safe_2\ by\ \mathbf{S}$$
$$after\ \sigma \cdot \langle \text{“}P_0\text{”}, \text{“}P_1\text{”}, \ldots, \text{“}P_{i-1}\text{”}\rangle$$
$$\& \ P_n\ safe_1\ by\ \mathbf{S}\ after\ \sigma \cdot \langle \text{“}P_0\text{”}, \text{“}P_1\text{”}, \ldots, \text{“}P_{n-1}\text{”}\rangle.$$

Therefore, the safety relation with the subscript 2 defines the continuation of the log by a button that does not cause destruction; the safety relation with the subscript 1 additionally prohibits the unobservable refusal and divergence. It is clear that any 2-safe button is also 1-safe; however, the converse is generally not true. For a button to be completely safe, it must be 1-safe, and it must be possible to place a finite chain of 1-safe buttons after it and then a 2-safe button that guarantees an observation.

On the basis of the button safety relations in the implementation and in the specification, safe actions, safe logs, the safety hypothesis, and the safe conformance are defined as it was done in the case when button switching was not allowed. The differences are as follows.

(1) In the safety hypothesis, the *i*-safety of a button in the specification must imply the *i*-safety of this button in the implementation (here, $i = 1, 2$):

$$\mathbf{I}\ \textit{safe for}\ \mathbf{S} =_{def} (\langle\text{“}\emptyset\text{”}, \gamma\rangle \notin F(\mathbf{S})$$
$$\Rightarrow \langle\text{“}\emptyset\text{”}, \gamma\rangle \notin F(\mathbf{I}))$$
$$\&\ \forall\sigma \in SafeBy(\mathbf{S}) \cap SafeIn(\mathbf{I})$$
$$\forall P \in \Re \cup \Omega(P\ safe_i\ by\ \mathbf{S}\ after\ \sigma$$
$$\Rightarrow P\ safe_i\ in\ \mathbf{I}\ after\ \sigma).$$

(2) In the definition of conformance, the nesting of sets is required only after 1-safe logs, that is, after the logs ending in a 1-safe button:

$$\mathbf{I}\ saco\ \mathbf{S} =_{def} \mathbf{I}\ safe\ for\ \mathbf{S}$$
$$\&\ \forall\sigma \in SafeBy(\mathbf{S}) \cap SafeIn(\mathbf{I})$$
$$\forall P\ safe_1 by\ \mathbf{S}\ after\ \sigma$$
$$obs(\sigma, P, \mathbf{I}) \subseteq obs(\sigma, P, \mathbf{S}).$$

2.8. Parallel Composition and Test Generation

Consider the composition of two LTSs with priorities **I** and **T** over the alphabets A and B, respectively. Consider any composite state *it*. In the composition, the set of enabled external actions for the LTS **I** in the state *i* is the set of opposite external actions for which there are transitions from the state *t* of the other LTS **T** and conversely. Therefore, we first need to recalculate the predicates of transitions from these states. Since the composition operator is commutative (up to an isomorphism, that is, up to the names of states *it* and *ti*), it is sufficient to consider only the recalculation of the predicates in one of the LTSs, for example, in **I**. For the transition $i \xrightarrow{z, \pi_i} i'$, we must substitute a constant expression of each variable corresponding to the synchronous action $z \in A \cap \underline{B}$ into the predicate π_i interpreted as a Boolean function of Boolean action variables. If there is the transition $t \xrightarrow{z, \pi_t} t'$, then *true* is substituted; otherwise, the substituted value is *false*. Thus, we obtain a new predicate π_{it}. Note that the evaluation of the new predicate on the transition from the state *i* depends on the state *t* with which it is composed; that is, the predicates π_{it} for different states *t* are different.

The new predicate π_{it} can be not a constant because it can include variables corresponding to asynchronous external actions belonging to $A\backslash\underline{B}$. Furthermore, now this predicate must be considered over the composite alphabet $A \uparrow\downarrow B = (A\backslash\underline{B}) \cup (B/\underline{A})$, although it actually depends only on the variables corresponding to the actions in $(A\backslash\underline{B})$.

Any asynchronous transition corresponds to a certain (unique) transition in one of the LTS operands. It can be executed if the inherited transition is executable. Therefore, the predicate of an asynchronous composite transition is identical to the predicate of the inherited transition after the recalculation; that is, it is identical to the predicate π_{it} rather than to the original predicate π_i. A synchronous (or simultaneous) transition is the simultaneous execution of the transitions in each LTS operand. It can be performed if both operands (transitions) are executable. Therefore, the predicate of the synchronous composite transition is equal to the composition of the recalculated transition operands π_{it} & π_{ti}. By and large, the composite transitions are generated by the following inference rules:

(1*) $z \in \{\tau, \gamma\} \cup A\backslash\underline{B}$ & $i \xrightarrow{z, \pi_i} i' \vdash it \xrightarrow{z, \pi_{it}} i't$,

(2*) $z \in \{\tau, \gamma\} \cup B\backslash\underline{A}$ & $t \xrightarrow{z, \pi_t} t' \vdash it \xrightarrow{z, \pi_{ti}} it'$,

(3*) $z \in A \cap \underline{B}$ & $i \xrightarrow{z, \pi_i} i'$ & $t \xrightarrow{z, \pi_t} t' \vdash it \xrightarrow{\tau, \pi_{it} \& \pi_{ti}} i't'$.

As in the case of a machine without priorities, testing is interpreted as the composition of the LTS implementation **I** over the alphabet A and the LTS test **T** over the opposite alphabet $B = \underline{A}$. We also assume that the test does not include destruction. The transitions corresponding to the external actions do not have predicates in the test; more precisely, their predicates are identically true. Therefore, all the transitions in the composite LTS (they are only τ- or γ-transitions) are the recalculated predicates of the implementation transitions. Since the composite alphabet is empty, these predicates are constants (*true* of *false*).

To detect refusals in the test (but not in the implementation!), θ-transitions with identically true predicates are also used. If button switching is not allowed, such a transition is performed if and only if no other transitions are executable:

(4*) $t \xrightarrow{\theta} t'$ & $Deadlock(i, t) \vdash it \xrightarrow{\tau} it'$;

here, $Deadlock(i, t) = i \xrightarrow{\tau, \pi_{it}}\!\!\!\!\!\!/\ \ \&\ i \xrightarrow{\gamma, \pi_{it}}\!\!\!\!\!\!/\ \ \&\ t \xrightarrow{\tau}\!\!\!\!\!\!/\ \ \&$ $(\forall z \in A \cap \underline{B}\ i \xrightarrow{z, \pi_{it}}\!\!\!\!\!\!/\ \vee\ t \xrightarrow{z}\!\!\!\!\!\!/\ )$.

If button switching is allowed, it is represented in the test as a τ-transition “P” $\xrightarrow{\tau}$ “Q” from the state corresponding to the button P to the state corresponding to the other button Q. A θ-transition is defined in the state “P” if P is an $\Re$-button. It is required that the θ-transition could be performed independently of the τ-transition of the button switching “P” $\xrightarrow{\tau}$ “Q”; for that purpose, the doubly underscored condition $t \xrightarrow{\tau}\!\!\!\!\!\!/$ is removed.

We will consider only safe implementations and safe tests. Here, by a safe test, we mean a test such that its interaction with any safe implementation does not

cause destruction, divergence that is executable after pressing a button, and deadlocks. Such tests are constructed on the basis of the safe specification logs. A test log is either a safe specification log or a safe specification log ending by a button $\sigma \cdot \langle \text{“}P\text{”} \rangle$ that is continued by an observation (an action $z \in P$ or an $\Re$-refusal of P) that, in turn, is not included in the specification after the log $\sigma \cdot \langle \text{“}P\text{”} \rangle$. We consider only the tests that terminate in a finite amount of time; they are called finite tests. For a safe LTS test, this implies that it does not contain infinite routes.

In the composition of the test with the implementation, all the predicates are constants; therefore, we can remove all the transitions having false predicates. If the implementation is safe and the test is finite and safe, the remaining γ-transitions are unreachable. As for the machines without priorities, the test execution corresponds to the passage of the τ-route beginning at the initial state of the composition and ending in the composite state it, where t is the terminal state of the test assigned a verdict *pass* or *fail*. Note that the composition can contain infinite routes; however, they cannot be passed when testing. Indeed, since the test is finite, the state of the test does not change in such a route, and only asynchronous τ-transitions of the implementation can follow. In the course of testing, the operator always gets an observation in a finite amount of time, presses or switches a button, which means the change of the test state and, therefore, the termination of the execution of an infinite chain of τ-transitions of the implementation. Here, we use the fact that only a finite chain of transitions can be executed in finite time. The operator may also turn the machine off (stop the testing), which happens in a finite amount of time after observation.

The test must interact with the implementation according to the same $\Re/\Omega$-semantics in which the corresponding specification was considered. For that reason, in each state of the test, the set of actions for which transitions from it are defined must correspond to an $\Re$- or a Ω-button and a θ-transition must be additionally defined for the $\Re$-button. We assumed that, immediately after turning the machine on before the first button is pressed, after any observation, and when buttons are switched, $\emptyset$-executable τ- and γ-transitions can be executed in the implementation. This assumption is a part of the interaction semantics that must be adhered to by the test. Therefore, we must include in the implementation additional *empty states* corresponding to the situation when no buttons are pressed. These test states must enable the implementation to execute $\emptyset$-executable τ-transitions. When all the buttons are released, the set of enabled actions is empty; therefore, only τ-transitions can be defined in the empty state. Ultimately, these τ-transitions must lead to nonempty states corresponding to a certain button (the empty $\Re$-button corresponds to the state in which a θ-transition is defined, and the empty Ω-button was disabled). Such an empty button has to be the initial state and the poststate of each transition with respect to the observation if this is not a terminal state.

A primitive test is constructed in the same way as in the case of the machine without priorities. However, there are three differences. (1) In the absence of priorities, we constructed a test on the basis of a satrace by converting it into an $\Re/\Omega$-log; now, we immediately begin with a safe $\Re/\Omega$-log. (2) If the log includes switching from the button P to the button Q, then the τ-transition “P” $\xrightarrow{\tau}$ “Q” is performed in the test. (3) Empty states are added. As before, the set of all primitive tests is complete, and any strict test can be replaced by a union of primitive tests that detects the same bugs.

2.9. Examples of Setting Priorities

Let us show how the priorities can be set using predicates on the transitions of an LTS model for the examples discussed in the Introduction.

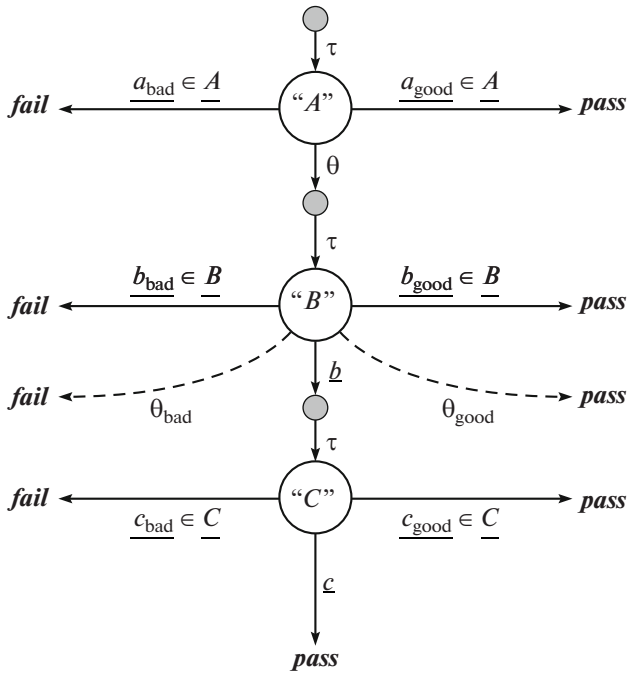
Divergence exit. The transition corresponding to an external action has an identically true predicate, and the τ-transition's predicate π is true only on the empty subset of the alphabet of external actions: $\pi(U) = (U = \emptyset)$.

Oscillation exit (the priority of the input over the output). The transition corresponding to an input has an identically true predicate, and the transition corresponding to a response has a predicate π that is true on any subset of actions that does not include inputs: $\pi(U) = (\forall ?x\ ?x \notin U)$. Usually, it is also assumed that the internal activity has a lower priority than the input reception; that is the τ-transition's predicate is the same as that of the transition corresponding to the response.

Priority of the output over the input in infinite queues. The transition corresponding to a response has an identically true predicate, and the transition corresponding to an input has a predicate π that is true on any subset of actions that does not include responses: $\pi(U) = (\forall !y\ !y \notin U)$. As in the preceding case, the τ-transition's predicate is the same as that of the transition corresponding to the input.

Interrupting a chain of actions. The transition corresponding to the command cancel has an identically true predicate, and all the other transitions have the predicate π that is true on any subset of actions that does not contain *cancel*: $\pi(U) = (\mathit{cancel} \notin U)$.

Priority handling of inputs. The set of inputs is decomposed into nonoverlapping subsets X_1, X_2, ... so that the inputs in the sets with a greater index have a higher priority. On the transition corresponding to the input from X_i, the predicate π is true on any subset of actions that does not contain inputs from the sets with a greater index: $\pi_i(U) = (\forall j > i\ U \cap X_j = \emptyset)$. It is also



A primitive test for the safe $\mathfrak{R}/\mathfrak{Q}$ history σ

$\sigma = \langle$"A", A, "B", b, "C", $c\rangle$ where $A, B \in \mathfrak{R}$, $C \in \mathfrak{Q}$.

The external action has the subscript "good" if it is included in the specification after the log prefix; otherwise, it has the subscript "bad." Dashed lines show mutually exclusive θ-transitions beginning in the same state. The subscripts "good" and "bad" denote the presence and, respectively, absence of the continuation of the log prefix by the corresponding $\mathfrak{R}$-refusal. Small shaded circles denote empty states

Fig. 5.

possible to differentiate the transitions from a certain state corresponding to the same input depending on the presence or absence of less priority inputs. For example, a transition corresponding to an input from X_i is performed if the environment offers less priority inputs: $\pi_{i1}(U) = \pi_i(U)$ & $(\exists j < i\ U \cap X_j \neq \emptyset)$; such a behavior is based on the assumption that this offer remains and it will be possible to handle these inputs later. Another transition is performed if there are no less priority inputs: $\pi_{i2}(U) = \pi_i(U)$ & $(\forall j < i\ U \cap X_j = \emptyset)$. If a state has no transitions corresponding to inputs, such kind of differentiation is also possible between the transitions corresponding to responses and (or) τ-transitions.

More exotic priorities can also be implemented. An example is a cyclic priority of moving in different directions: one goes to the North if it is impossible to go to the East; one goes to the East if it is impossible to go to the South; one goes to the South if it is impossible to go to the West; and one goes to the West if it is impossible to go to the North. If all the four directions are allowed, any of them may be chosen. Except for this case, only the opposite directions have the same priority when the two other directions are not allowed. For example, the predicate of moving to the North is as follows: $\pi_{North}(U) = (\text{East} \notin U \vee U = \{\text{North, East, South, West}\})$. Similarly, the predicates of moving to the East, South, and West are constructed.

3. CONCLUSIONS

One can consider semantics in which, after the machine is turned on, after observations, and after button switching, the execution of τ- and γ-actions can be disabled by the implementation even when they are $\emptyset$-executable. One can assume that, immediately after turning on and after making an observation, the machine stops and can execute actions only after a button is pressed. Furthermore, button switching is not interpreted as releasing the first button (with enabling $\emptyset$-executable τ- and γ-actions) and then pressing the second button. In other words, after the machine is turned on, after an observation, and between two buttons when they are switched, there is no "empty" interval. Such kind of semantics obviously assumes enhanced testing possibilities than the weak semantics considered in this paper. These semantics have different requirements for safety and conformance.

Any behavior that can be observed under the strong semantics can also be observed under the weak one—it is sufficient to find appropriate "weather conditions" in which the operator has time to press or switch a button sufficiently quickly. The converse is also true: the behavior observed under the weak semantics can also be observed under the strong semantics if an empty button is added and pressed. However, the safety conditions for these semantics are different. Under the weak semantics, we must take into account that τ- and γ-actions can be performed (in the presence of priorities, they must be $\emptyset$-executable) after an

observation by the button P, and these actions can lead to the divergence or destruction; therefore, this button is not safe. Under the strong semantics, we can simply avoid pressing the empty button in this situation after such an observation because it is not safe, and the button P will be safe. This also implies the corresponding differences in conformance: an implementation can be safe under the weak semantics and, therefore, nonconformal; under the strong semantics it can be safe and conformal. Under the same safety conditions (for example, when the specification does not stipulate the divergence, destruction, and unobservable refusals) and in the presence of priorities, the strong semantics imposes higher requirements for conformance. This is explained by the fact that we obtain the possibility to distinguish between the implementations in which an action b enabled by the button B is executed immediately after the action a and the implementations in which the action b is executed after the intermediate $\emptyset$-executable but not B-executable τ-activity.

REFERENCES

1. van Glabbeek, R.J., The Linear Time—Branching Time Spectrum, *Proc. of CONCUR'93*, Baeten, J.C.M. and Klop, J.W., Eds., *Lect. Notes Comput. Sci.*, 1990, vol. 458, pp. 278–297.
2. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., Formalization of Test Experiments, *Programmirovanie*, 2007, no. 5 [*Programming Comput. Software* (Engl. Transl.), 2007, vol. 33, no. 5, pp. 239–260].
3. Bourdonov, I.B., Conformance Theory for the Functional Testing of Software Systems Based on Formal Models, *Doctoral (Math.) Dissertation*, Moscow: Institute for System Programming, Russian Academy of Sciences, 2008; http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf.
4. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., *Teoriya sootvetstviya dlya system s blokirovkami i razrusheniem (Conformance Theory for Systems with Refusals and Destruction,* Moscow: Nauka, 2008.
5. 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.
6. Milner, R., Modal Characterization of Observable Machine Behavior, *Proc. CAAP,* 1981, Astesiano, G. and Bohm, C., Eds., *Lect. Notes Comput. Sci.*, 1981, vol. 112, pp. 25–34.
7. Petrenko, A., Yevstushenko, 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.
8. 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.
9. 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.
10. 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.
11. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V., "Security, Verification, and Conformance Theory," in *Materialy vtoroi mezhdunarodnoi nauchnoi konferentsii po problemam bezopasnosti I protivideistviya terrorizmu* (Proc. of the Second Int. Conf. on Security Problems and Terrorism Counteractions), Moscow: MNTsMO, 2007.
12. Heerink, L. and Tretmans, J., Refusal Testing for Classes of Transition Systems with Inputs and Outputs, in *Formal Description Techniques and Protocol Specification, Testing and Verification*, Chapman & Hill, 1997.
13. Heerink, L., Ins and Outs in Refusal Testing, *PhD Thesis*, Enschede, Netherlands: Univ. of Twente, 1998.
14. 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.
15. Milner, R., A Calculus of Communicating Systems, *Lect. Notes Comput. Sci.*, 1980, vol. 92.
16. Milner, R., *Communication and Concurrency,* Prentice-Hall, 1989.