Formal Conformance Testing of Systems with Refused Inputs and Forbidden Actions

Igor B. Bourdonov [1,2],
Alexander S. Kossatchev [1,3], and Victor V. Kuliamin [1,4]

Institute for System Programming of Russian Academy of Sciences
1009004, B. Kommunisticheskaya, 25, Moscow, Russia

Abstract

The article introduces an extension of the well-known conformance relation ***ioco*** on labeled transition systems (LTS) with refused inputs and forbidden actions. This extension helps to apply the usual formal testing theory based on LTS models to incompletely specified systems, which are often met in practice. Another topic concerned in the article is compositional conformance. More precisely, we try to define a completion operation that turns any LTS into input-enabled one having the same set of ***ioco***-conforming implementations. Such a completion enforces preservation of ***ioco*** conformance by parallel composition operation on LTSes.

Key words: Formal testing, conformance testing, LTS, implementation relation, refusals, ***ioco***.

1 Introduction

In the modern world a large part of human activities is controlled by various computer-based systems. Reliability and quality of such systems become urgent for dependable evolution of our society. One of the tools that help us to ensure system quality is *conformance testing.* Conformance testing in general is an activity that checks conformance between the real behavior of software or hardware system and the requirements to this behavior. To make results of conformance testing more sound and convincing the testing process needs in

[1] This work is partially supported by RFBR grants 05-01-00999-a and 04-07-90386-b, by grant of Russian Science Support Foundation, and by Program 4 of Mathematics Branch of RAS.
[2] Email: igor@ispras.ru
[3] Email: kos@ispras.ru
[4] Email: kuliamin@ispras.ru

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

a formal framework, including formalism for description of requirements and formal definition of conformance relation.

To make the reasoning about conformance rigorous one models both the actual behavior of the system under test (SUT) and the requirements to it in some formalism. The choice of such formalism is directed by a class of systems we need to describe with it. It is preferable to use a theory that allows reasoning about a wide range of software and hardware systems of practical significance. *Labeled transition systems* (LTS) formalism is a good candidate, and it is used successfully for a long time to model rather complex behavior of distributed software and hardware units, including concurrency aspects. LTSes also serve as semantic metamodel for various process calculi, such as CSP [1] and CCS [2], and for formal languages actively applied in distributed software and hardware verification, e.g. SDL, LOTOS, and Estelle.

During testing one usually distinguishes between inputs and outputs of the SUT. A tester provides the former to it, it provides the latter to the tester. So, LTS model should be regarded as *IOLTS,* i.e. input-output labeled transition system, where labels on transitions are partitioned into input and output symbols.

By *a specification* one means a description of requirements to SUT's behavior in terms of the formalism chosen, e.g. an LTS modeling the required behavior. Since the requirements are represented formally, one can speak about formal conformance between them and the actual behavior of the SUT, but only if this actual behavior also has some formal representation. Usually the basic *test hypothesis* states that the actual behavior of the SUT can be adequately described by a model of the same kind [3,4]. In our case this means that there exists an LTS, which is called *an implementation,* adequately representing the real behavior of the SUT. One does not know it exactly, but can reason on its properties on the base of observations of the SUT's behavior.

Many relations between LTSes can be chosen as conformance relations checked in testing. [6] gives an extensive review of them. The choice of conformance relation depends on the *testing abilities* – abilities to control the SUT and to observe various aspects of its behavior during testing. On the other hand, the testing abilities determine properties of the system under test that can be checked. One of the most useful and natural conformance relations used in testing is ***ioco***, introduced in works of Jan Tretmans [8,7]. He also developed the theory that helps to construct test suites necessary and sufficient to check conformance between model and implementation according to ***ioco***.

1.1 **ioco** *relation and its problems*

ioco uses three rather natural and basic testing abilities – ability to provide inputs, ability to observe outputs, and less obvious ability to observe *a quiescence,* a situation, in which the SUT will not provide any more output.

Further observation of a quiescence in traces is denoted as δ. In practice one usually supposes that there exists some finite time T that in any state if no inputs are provided and the SUT is going to provide an output, it always does this in a time less that T. This hypothesis allows us to detect quiescence as the observation of no outputs during some timeout.

More attentive analysis of test abilities used by ***ioco*** gives two subtle issues.

- We suppose that the implementation is *input-enabled,* i.e. in each stable state (where there are no internal transitions) it has a transition for each input symbol. Informally, an input-enabled system should always accept any input provided to it. This may be reasonable when we test large components and systems as a whole, because it is natural to them to process any possible inputs. But internal components are often developed in collaborative mode, not the protective one, and rely upon some restrictions on the input.
- We suppose that during testing we can prevent SUT from giving us an output, if we want it to accept our input first. This property follows from the semantics of LTS interaction based on rendezvous mechanism. If we model testing as interaction between an implementation LTS and a tester LTS by means of parallel composition, we need to have in practice the special ability to prevent the SUT from producing an output to the testing system if the testing system is not ready to accept it.

Both issues were already mentioned by several authors, including Tretmans himself [8]. These assumptions give tester very high level of control over the SUT. The second property is considered by some authors as particularly suspicious, since it is rarely can be met in practice. Only in special contexts, for example, during debugging, tester has enough control over the execution of the SUT to make this assumption valid.

However, in the framework of LTS models the lack of control over SUT is the consequence of the presence of some *testing context,* which represents the transport mechanism, delivering actions from the tester to the SUT and backward. The testing performed through some context is called *asynchronous testing,* while the one giving the tester full control over the SUT is called *synchronous.* The second issue can be interpreted that ***ioco*** is intended to be used in synchronous testing only. If we need to check conformance between an implementation and a specification by means of testing through some context, it is natural to use the composition of the LTS modeling the context with the original specification as the specification of the observable SUT's behavior and check its real behavior against the derived specification [4].

Here we face with a known problem of ***ioco*** – it is not preserved by the parallel composition of LTSes, i.e. composition $I\|Q$ of an implementation I conforming with a specification S and an LTS Q modeling the context may be not conforming with $S\|Q$. Examples of such implementation and specification can found in [9]. Another example is shown on Fig. 1. The specification S

and the implementation I presented there are ***ioco***-conforming, but are not ***ioco***-conforming if they are observed through input and output queues (that is, being composed with two endless queues or even queues of length 2). This problem seems to be a consequence of some bias of process calculi to consider bisimulation relation as the most natural conformance relation between processes. Parallel composition preserves bisimulation, which is thought to be the desired relation between specification and its implementation. However, bisimulation is not testable in natural settings. During black-box testing we cannot check it completely and often actually do not want to do it, because specification may describe more general behavior, only a part of which should be realized in any implementation.

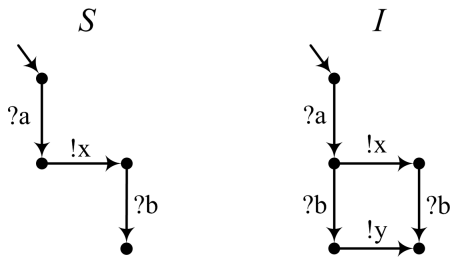

Fig. 1. Example of ***ioco***-conforming specification and implementation, which are not ***ioco***-conforming when observed through queues.

So, to propose more practical conformance relation for the testing in context, we can go in two ways.

- To consider some practical variants of contexts and develop testing framework for them, including specialized conformance relations. This approach for context modeled by infinite or bounded input and output queues is presented in works of Petrenko and Yevtushenko [10,11]. Another paper taking such an approach is [12] where the authors propose to augment events provided by the SUT with special stamps revealing the actual order of events in the SUT for tester. Such instrumentation makes another conformance relation, ***ioconf***, also used in synchronous testing, useful for the asynchronous one.
- To define more convenient composition operation that preserves conformance relation, in so far that we can check the SUT's behavior through any context against the specifications composed with LTS modeling this context. This way is chosen in recent works of Tretmans with co-authors [9]. It is shown there that input-enabled ***ioco***-conforming LTSes has no problems with composition – if both the specification and the implementation are completely specified and they are ***ioco***-conforming, then their compositions with any context LTS are also ***ioco***-conforming. So, the main problem to be overcome on the way to more convenient composition is unspecified inputs. *The demonic completion* of specification is proposed in [9]. It forces unspecified inputs to take the specification into special chaotic state, where

any behavior is possible. This is done to make any possible SUT's behavior in the unspecified area conforming to the completed specification.

1.2 The proposed approach

We also would like to go in the second way, since it makes possible testing through different contexts, which is useful in practice. For example, contexts not preserving the sequence of actions (as queues do) can be met in practical testing of Internet protocols, components of GRID networks, and Web services. Instrumentation of the SUT is not always possible, especially if it is distributed itself. On this way it is reasonable first to examine more thoroughly the meaning of unspecified inputs, which are the main source of the problems with definition of 'good' composition operation.

One can notice that this issue is related with the implementation input-enabledness hypothesis. Original definition of ***ioco*** is asymmetric in two ways – first, it assumes that an implementation should always accept inputs provided to it, while the tester can abstain from acceptance of an implementation's output, second, a specification can be partial and not input-enabled in contrast with an implementation. Both sources of asymmetry can be removed if we allow an implementation also to be partially defined, not input-enabled.

One can find the following ways of unspecified input understanding. Some of them were already mentioned in the literature [13].

- *Forbidden input.* Such an input is forbidden to be provided to the SUT, due to various reasons. It may cause serious destruction of the SUT, or move it into a situation, which we want to avoid during testing, for example, *divergence,* an infinite path through internal actions. In fact, when demonic completion is introduced, it means the same thing – we don't want to check the behavior of the SUT after accepting this input, but such a completion may cause us to perform these unwanted checks.

  We prefer to mark 'bad' situation we need to avoid with special *forbidden action* label γ. Any input that can lead in the state where a forbidden action can occur (maybe after a path through internal transitions) is considered as forbidden. The same holds for outputs that can lead us to the state with a forbidden action. But outputs in some state are under full control of the SUT – it is the SUT, which choose an output to produce. So, we need to ban the mere waiting for an output in states where some output can lead us to a forbidden action.
- *Refused input.* This input can be provided to the SUT and in response it *demonstrates* refusal to accept it. Here we need the new testing ability to observe input refusals. Refused inputs can model situations of practical significance. For example, tea-coffee machine having two buttons for requesting tea and coffee and a slot for coin insertion may also have a special shutter closing the slot until some button is pressed. When trying to insert a coin before pressing a button we may observe that the coin is not taken.

More practical example is given by Graphical User Interface controls – menu item and buttons, which can be enabled or disabled. In this case control's disability means that the system refuses to accept actions on this control.

Refused inputs are considered as particular case of *refusals* forming *refusal sets* in [5,6] and some papers on conformance testing, e.g. [14] and works on Multi Input-Output Transition Systems (MIOTS) [15,16,17]. In testing based on MIOTS testing concerning input refusals attract more attention, since blocking of one channel caused by a refused input can be resolved after accepting an input on another channel. Here we do not need in detailed consideration of refusal sets and pay more attention to refused inputs.

- *Erroneous input.* This is more subtle case. In some situations we can provide an input to the SUT, but the fact that the SUT has accepted it says that it is not conforming to the specification. Consider the example presented on Fig. 2. In the specification LTS presented there δ-trace $\delta?a\delta$ ends in the state where input a is not specified. And its subtraces $\delta?a$ and $?a\delta$ end in states where input a is defined, but is followed by different outputs. So, what if we observe the trace $\delta?a\delta$ in the implementation and then provide an input a? The conforming implementation should be input-enabled and it should accept a, by it cannot provide neither x, nor y, nor it can demonstrate quiescence in response. Otherwise, if it has the trace $\delta?a\delta?a\delta$, it should have $\delta?a?a\delta$, which is absent in the current specification, if it has the trace $\delta?a\delta?a!x$, it should have $?a\delta?a!x$, which is also absent, and if it has the trace $\delta?a\delta?a!y$, it should have $\delta?a?a!y$, which is absent in specification again. So, the only reasonable conclusion is that this implementation is not conforming to the specification presented, just after it demonstrated the trace $\delta?a\delta?a$. The last input a is erroneous in the sense that any possible behavior after it (any output or refusal) cannot be observed in the conforming implementation.

  We model such an input as leading to a separate state with the single outgoing transition marked with special *an error output.* This construction will be necessary in consideration of possible completion operations for LTSes.

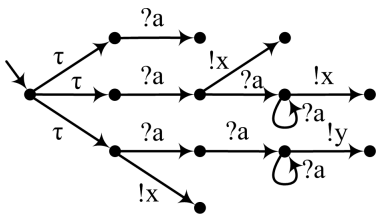

Fig. 2. Example of the specification having the trace $\delta?a\delta$ that should not exist in any ***ioco***-conforming implementation.

- Unspecified input can be considered as doing nothing and so corresponding to a self-loop transition (so called 'angelic' behavior). We think, however,

that such inputs should be specified in an accurate specification and it should be tested that they actually do nothing. To make an input unspecified there must be more serious reasons (see above).

Bearing in mind all the listed possibilities, we do the following.

(i) Define an extension of ***ioco*** relation for LTSes that can have forbidden actions and refused inputs. Error output is an auxiliary mark to check conformance. This relation is designated as $\boldsymbol{ioco}_{\beta\gamma\delta}$ in this paper.

(ii) Since parallel composition of LTSes breaks ***ioco*** only on partially specified LTSes, we need to define some completion of the original LTS before composition. This completion from one hand should give an input-enabled LTS, and from the other hand the original LTS and the completed one should have the same set of ***ioco***-conforming implementations.

Next sections of the article present the implementation of those steps. It seems that the main contribution of this paper is direct introduction of forbidden actions into the definition of conformance relation and construction of the corresponding completion operation disallowing processing of inputs unspecified in the original LTS.

2 Extended *ioco* Conformance Relation

Below we recall some part of LTS-based formalism and usual arrow notation.

Definition 2.1 *An LTS is a tuple* $L = (Q, C, T, q_0)$ *where*

- Q *is non-empty set of* states;
- $C = I \cup U$ *is a set of symbols,* I *consists of* input symbols*,* U *is disjoint from* I *and consists of* output symbols;
- $T \subseteq Q \times (C \cup \{\tau, \gamma\}) \times Q$ *is a set of* transitions. *A transition* (q, a, q') *starts in the state* q*, ends in the state* q'*, and is marked with the label* a*. We use labels with question mark (*$?a$*) to denote input symbols and labels with exclamation mark (*$!x$*) to denote output symbols.* $\tau \notin I \cup U$ *is considered as* empty symbol *marking* internal transitions. $\gamma \notin I \cup U, \gamma \neq \tau$ *is considered as* forbidden action symbol.
- $q_0 \in Q$ *is* the initial state.

We denote the fact that in an LTS L $(q, x, q') \in T_L$ as $q \xrightarrow{x} q'$. $q \xrightarrow{x}$ denotes $\exists q' \in Q\ q \xrightarrow{x} q'$. $q \overset{x}{\not\longrightarrow}$ denotes $\forall q' \in Q\ (q, x, q') \notin T_L$. By *a stable state* we mean a state q such that $q \overset{\tau}{\not\longrightarrow} \wedge q \overset{\gamma}{\not\longrightarrow}$.

LTS can be partially specified, i.e. it can have states where not all inputs are possible. However, we can consider it as completely specified due to the following interpretation. If for a stable state q $q \overset{?a}{\not\longrightarrow}$, we may mean that this $?a$ can be given in q and the LTS should demonstrate refusal to accept it in response to this. For an input symbol $?a$ we denote refusal of this input

as $\{?a\}$. In addition to input symbols, output symbols, empty symbol, and forbidden action we use symbol δ to denote quiescence, i.e. situation where LTS does not have any transitions marked with output symbols, γ, or τ. Input refusals and quiescence together are called refusals.

We call an LTS L *strongly convergent* if it does not have infinite paths through internal transitions. It is possible to convert any LTS into strongly convergent one by replacing the symbol τ on the transitions of every such path with γ. From testing viewpoint this means that we avoid actions that can lead us to such a path. Further we consider only strongly convergent LTSes.

We can transform an original LTS by converting convergence into forbidden actions, making all transitions marked with γ to lead into a special additional state, and adding refusal transitions as self-loops in stable states. An example of such a transformation is shown on Fig. 3.

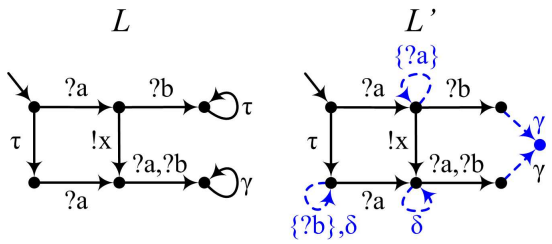

Fig. 3. Example of transformation making refusals explicit and adding a special state to go after forbidden actions.

By *$\beta\gamma\delta$-traces* in alphabet $I \cup U$ we mean sequences consisting of input and output symbols, γ and refusal symbols – δ and refusals of input symbols. We denote concatenation of traces σ and μ by $\sigma\mu$. $\mu \preceq \sigma$ denotes that μ is a beginning of σ. If s is input, output, input refusal symbol, γ or δ, then $\langle s \rangle$ means the sequence with the single element s.

A run of LTS L starting in a state q is a sequence of transitions of L, transformed according to the procedure described above, the first of which starts in q, and each next transition starts in the end state of the previous. *A $\beta\gamma\delta$-trace of a run p* is a sequence of labels of transitions of p, from which all symbols τ are skipped.

One can see that each time when p goes through a stable state without outgoing output transitions, δ may be inserted in the corresponding place several times. Similarly, each time when p goes through a stable state without outgoing transition marked with input symbol $?a$, the symbol $\{?a\}$ may be inserted in the corresponding place several times. According to the transformation rules the first γ met in $\beta\gamma\delta$-trace is always the last symbol – there is no need to extend a $\beta\gamma\delta$-trace after the first forbidden action in it. The set of all the $\beta\gamma\delta$-traces of runs of L starting in a state q is denoted as $Traces_{\beta\gamma\delta}(q, L)$. $Traces_{\beta\gamma\delta}(L)$ is $Traces_{\beta\gamma\delta}(q_0, L)$.

If σ is a $\beta\gamma\delta$-trace of LTS L then L **after** σ is a set of all states of L that can be reached by paths having σ as their $\beta\gamma\delta$-trace.

We need the notion of safe actions, which makes us safe from triggering a forbidden action. An input symbol $?x$ or its refusal are called *safe in LTS L after a $\beta\gamma\delta$-trace σ* if $\forall q \in (L \,\mathbf{after}\, \sigma\langle ?x\rangle)\ q \stackrel{\gamma}{\not\longrightarrow}$. An output symbol or δ is said to be *safe in L after its trace σ* if $\forall !x \in U \forall q \in (L \,\mathbf{after}\, \sigma\langle !x\rangle)\ q \stackrel{\gamma}{\not\longrightarrow}$. A $\beta\gamma\delta$-trace σ of L is *safe* if each its symbol is safe in L after the beginning of σ preceding this symbol. A set of all safe $\beta\gamma\delta$-traces of L is denoted as $Safe(L)$. We also call an extension of a safe trace σ of L with a safe symbol in L after σ *a test trace* of L. A set of all test $\beta\gamma\delta$-traces of L is denoted as $TT(L)$. It is easy to note that $TT(L) \cap Traces_{\beta\gamma\delta}(L) = Safe(L)$.

Consider again specification LTS S and implementation LTS I. We may perform testing according to S only if we are sure that I operates properly during this process. In ***ioco*** theory this is guaranteed by the input-enabledness of I. Although we ease this assumption, we still need some safety hypothesis about I. This leads us to the following definition.

Definition 2.2 *If I and S are LTSes, I is said to be* safe for *S if $TT(S) \cap Traces_{\beta\gamma\delta}(I) \subseteq Safe(I)$.*

This definition says that if we construct a test avoiding possibility of forbidden action occurrence in a specification, its application to any implementation safe for this specification cannot lead to forbidden action too. So, implementations safe for a specification can be safely tested according to it.

Now we are ready to give the definition of $\boldsymbol{ioco}_{\beta\gamma\delta}$ relation.

Definition 2.3 *Let I and S are LTSes. Then $I \,\mathbf{ioco}_{\beta\gamma\delta}\, S$ if and only if I is safe for S and for each $\beta\gamma\delta$-trace $\sigma \in Safe(S)$ and for each symbol s (including refusals) safe in S after σ $\quad \sigma\langle s\rangle \in Traces_{\beta\gamma\delta}(I) \Rightarrow \sigma\langle s\rangle \in Traces_{\beta\gamma\delta}(S)$.*

More fine (but less intuitive) expression of this fact can be given by the expression $TT(S) \cap Traces_{\beta\gamma\delta}(I) \subseteq Traces_{\beta\gamma\delta}(S) \cap TT(I)$. Informally, an implementation I safe for S is said to be $\boldsymbol{ioco}_{\beta\gamma\delta}$-conforming to S when after an S-safe trace I can accept an input symbol, give an output symbol, demonstrate a quiescence, or input refusal only if S can do just the same thing after the same trace.

It is easy to show that $\boldsymbol{ioco}_{\beta\gamma\delta}$ defines a preorder on LTSes. Note, that classic ***ioco*** is not a 'good' preorder, since it imposes asymmetric restrictions on implementation and specification. While the latter can be incompletely specified, the former should not. The fact that for specifications without forbidden actions and input refusals (usual completely specified LTSes) $\boldsymbol{ioco}_{\beta\gamma\delta}$ is equivalent to ***ioco*** is also rather obvious. In this case they both are equivalent to trace inclusion.

2.1 Test Derivation

During testing we should check SUT's behavior on every trace that is safe in the specification. Moreover, we should check it for all the symbols safe after

such a trace in the specification. As usual we model *test cases* by LTSes with inverted inputs and outputs – inputs of the specification become outputs of a test case, outputs of the specification (and implementation) are inputs of a test case. In addition, the symbol θ is used to mark deadlock resolution transitions in a test case. θ is considered as input symbol and means observation of quiescence in the test case states where any SUT's outputs can be accepted or observation of an input refusal in the test case states where a specification's input symbol is provided by the test.

A test case has two special states **fail** and **pass** without outgoing transitions. Other constraints on test case LTS are given below.

- Each maximal trace of a test case should be finite and should end either in the **fail** state or in the **pass** state.
- A test case should resolve all possible deadlocks in its interaction with an implementation. If in some state q of a test case there exists $a \in I$ $q \xrightarrow{!a}$, then $q \xrightarrow{\theta}$, which fires if the input a is refused by the SUT. If in some state q of a test case for all $x \in U$ $q \xrightarrow{?x}$, then $q \xrightarrow{\theta}$, which fires if no output is observed.
- A test case should be deterministic as much as it is possible. Each its state should be an input state or an output state. An input state should have outgoing transitions marked with all possible SUT's outputs and θ. An output state should have only one outgoing transition marked with some input of the specification and one outgoing transition marked with θ.

An implementation LTS I *passes* a test case T if their parallel composition (extended by correlating θ in the test case with δ and input refusals in the implementation) has no states with fail component achievable from the initial state. *A test suite for a specification* S is a set of test cases for S. An implementation *passes* a test suite if it passes each its test case. A test suite is called *sound for a specification* S if any $\boldsymbol{ioco}_{\beta\gamma\delta}$-conforming implementation passes it, and *exhaustive for* S if any implementation passing it is $\boldsymbol{ioco}_{\beta\gamma\delta}$-conforming to S. Sound and exhaustive test suite is called *complete.*

Theorem 2.4 *Let us denote a set of safe finite* $\beta\gamma\delta$*-traces of a specification* S *as* $Safe_f(S)$*. For each trace* $\sigma \in Safe_f(S)$ *construct a test case* $T(\sigma)$ *with the help of the following transformations.*

- *Take a sequence of symbols of* σ*, construct an inverted symbol for each* $(?a \mapsto !a,\ !x \mapsto ?x,\ \delta \mapsto \theta,\ \{?a\} \mapsto \theta)$*, and make the sequence of transitions marked with the resulting symbols. Let us denote a state of this LTS by* $\bar{\mu}$*, where* μ *is the corresponding prefix of* σ*.* $\bar{\sigma}$ *should be* **pass**.
- *For each prefix* μ *of* σ *and symbol* s *such that* $\mu\langle s\rangle \preceq \sigma$ *we add new transitions. There are several possibilities listed below. For each case we consider possible extensions of* μ *with alternatives to* s*. If* s *is an input, its alternative is the corresponding input refusal, and the alternative to an input refusal is*

the corresponding input. Alternatives to an output are all other output symbols from the alphabet and quiescence, and alternatives to δ are all output symbols. For each alternative to s we should add an additional transition to our test case. If this alternative is possible into the specification (the trace μ can have several different extensions in the specification), we add the corresponding transition leading to **pass**, *otherwise it should lead to* **fail**. *More precise rules are given in the following list.*

- $?s \in I$. *Then, add a transition* $\bar{\mu} \xrightarrow{\theta}$ **pass**, *if* $\mu\langle\{?s\}\rangle \in Traces_{\beta\gamma\delta}(S)$ *and* $\bar{\mu} \xrightarrow{\theta}$ **fail** *otherwise.*
- s *is* $\{?r\}$, *where* $?r \in I$. *Then, add a transition* $\bar{\mu} \xrightarrow{!r}$ **pass**, *if* $\mu\langle?r\rangle \in Traces_{\beta\gamma\delta}(S)$ *and* $\bar{\mu} \xrightarrow{!r}$ **fail** *otherwise.*
- $!s \in U$. *Then any* $!r \in U, !r \neq !s$ *and* δ *are safe in* S *after* μ. *Add a transition* $\bar{\mu} \xrightarrow{?r}$ **pass**, *if* $\mu\langle!r\rangle \in Traces_{\beta\gamma\delta}(S)$ *and* $\bar{\mu} \xrightarrow{?r}$ **fail** *otherwise. Also add a transition* $\bar{\mu} \xrightarrow{\theta}$ **pass**, *if* $\mu\langle\delta\rangle \in Traces_{\beta\gamma\delta}(S)$ *and* $\bar{\mu} \xrightarrow{\theta}$ **fail** *otherwise.*
- s *is* δ. *Then any* $!r \in U$ *is safe after* μ *in* S. *Add a transition* $\bar{\mu} \xrightarrow{?r}$ **pass**, *if* $\mu\langle!r\rangle \in Traces_{\beta\gamma\delta}(S)$ *and* $\bar{\mu} \xrightarrow{?r}$ **fail** *otherwise.*

Then $T(Safe_f(S))$ *is a complete test suite for* S.

Soundness of test cases from $T(Safe_f(S))$ is implied by their construction – if the composition of such a test and an implementation comes to a state with **fail** component, then the implementation has a trace that does not exist in the specification. To prove the exhaustiveness of the constructed test suite one should take an implementation that does not conform to the specification, found a safe trace σ in the specification that can be extended in the implementation by a safe symbol s, for which $\sigma\langle s\rangle \notin Traces_{\beta\gamma\delta}(S)$ holds. Then, it is sufficient to consider the test case constructed for σ extended with an alternative to s, which is a safe trace in S. The implementation chosen cannot pass this test case. More details of the proof can be found in [18].

3 Completion Operations

The next step is to define such a completion operation *Comp* for LTSes, that for each LTS S $Comp(S)$ is input-enabled and has the same set of $\boldsymbol{ioco}_{\beta\gamma\delta}$-conforming implementations. Results presented further are partial. Only a solution for classic ***ioco*** relation is given. The authors are working now on the full completion operation, but have no compact and proved construction for it.

In [9] the demonic completion Ξ is defined as a candidate of the needed completion for ***ioco***. However, as it is also noted there, this completion does not preserve full information on unspecified inputs. Moreover, demonic completion from [9] is state completion – it defines some additional behavior after an input in some state – and just this fact makes it slightly inadequate. State

completions can make non-conforming implementation conforming, as it is mentioned in [12]. Fig. 4 shows an example of specification S and implementation I such that I **ioco** S does not hold, but I **ioco** $\Xi(S)$.

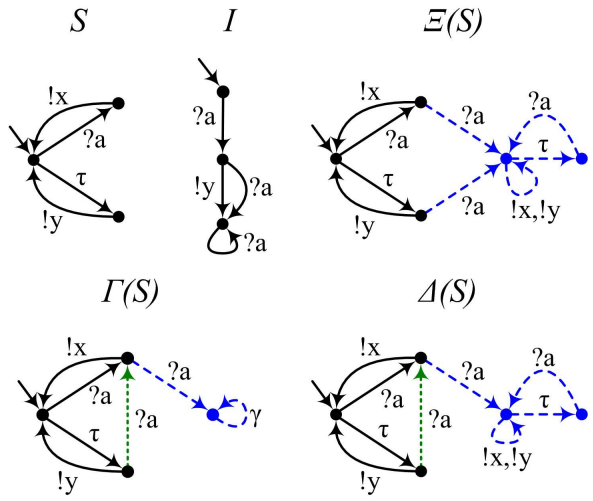

Fig. 4. Example of the specification, for which Ξ changes the ***ioco*** relation. The 'correct' completion variants Δ and Γ are also presented.

The same Fig. 4 also presents the examples of Δ and Γ completions defined below. They both are more suitable completion operations, not extending the set of ***ioco***-conforming implementations. On this figure additional transitions added by completion operations are shown as hatch lines.

We propose two completion operations, Δ and Γ, that differs in interpretation of unspecified inputs in the original LTS. Δ-completion treats them as leading into the states where any possible behavior can be observed, but they can be given to the completed LTS. Γ-completion treats them as forbidden inputs, which should not be provided during testing at all.

The main ideas are the following. At first we construct *a basic completion,* which augments an LTS with additional transitions and states that do not change the set of traces and protects the LTS from extending the set of ***ioco***-conforming implementations by the further state completion. On Fig. 4 transitions added by basic completion are shown as small-hatch lines. Then, we perform state completion according to the operation used – for Δ-completion we add all possible behaviors after all inputs that remained unspecified after the first step, for Γ-completion we add γ transitions after those inputs. On Fig. 4 transitions added by Δ- or *Gamma*-completions are shown as long-hatch lines.

Definition 3.1 Basic completion *operation Bc transforms an LTS with states Q, inputs I and outputs U in the following way. The resulting LTS $Bc(L)$ has the states corresponding to C^*_δ – all possible sequences of symbols from $I \cup U \cup \{\delta\}$. Inputs of $Bc(L)$ coincide with L, and outputs are $U \cup \{!error\}$. For each $\sigma \in C^*_\delta$ $R(\sigma)$ denotes the set of δ-traces of L obtained from σ by*

deletion some or all δ symbols. The set of transitions is the minimal set derived from the following rules.

- $\forall ?a \in I \, \exists \mu \in R(\sigma) \; \mu\langle ?a\rangle \in Traces_\delta(L) \Rightarrow \sigma \xrightarrow{?a} \sigma\langle ?a\rangle$ *in* $Bc(L)$.
- $\forall !x \in U \, \forall \mu \in R(\sigma) \; \mu\langle !x\rangle \in Traces_\delta(L) \Rightarrow \sigma \xrightarrow{!x} \sigma\langle !x\rangle$ *in* $Bc(L)$.
- $\forall \mu \in R(\sigma) \; \mu\langle\delta\rangle \in Traces_\delta(L) \wedge \sigma$ *does not end on* $\delta \;\Rightarrow\; \sigma \xrightarrow{\tau} \sigma\langle\delta\rangle$ *in* $Bc(L)$.
- $\forall \mu \in R(\sigma) \, \forall !x \in U \; \mu\langle !x\rangle \notin Traces_\delta(L) \wedge \mu\langle\delta\rangle \notin Traces_\delta(L) \Rightarrow \sigma \xrightarrow{!error} \sigma\langle !error\rangle$ *in* $Bc(L)$.

Δ-completion of an LTS L is constructed as completion of $Bc(L)$ with two states q_U and q_I demonstrating all possible behaviors, i.e. $q_U \xrightarrow{\tau} q_I$ and $\forall !x \in U \; q_U \xrightarrow{!x} q_U$ and $\forall ?a \in I \; q_I \xrightarrow{?a} q_U$. For each state q of $Bc(L)$ and each $?a \in I$ if $q \not\xrightarrow{?a}$ in $Bc(L)$ then $q \xrightarrow{?a} q_U$ in $\Delta(L)$.

Γ-completion of an LTS L is constructed as completion of $Bc(L)$ with one state q_γ having γ-self-loop. For each state q of $Bc(L)$ and each $?a \in I$ if $q \not\xrightarrow{?a}$ in $Bc(L)$ then $q \xrightarrow{?a} q_\gamma$ in $\Gamma(L)$.

Theorem 3.2 *Δ and Γ turn any LTS S into input-enabled one and preserve the set of* **ioco***-conforming implementations, i.e.*

$$\forall I \;\; I \,\mathbf{ioco}\, S \Leftrightarrow I \,\mathbf{ioco}\, \Delta(S) \Leftrightarrow I \,\mathbf{ioco}\, \Gamma(S).$$

We need to skip the proof (see its details in [18]) due to restrictions on the size of the paper.

The two completions defined can be used to describe relation between classic ***ioco*** and ***ioco***$_{\beta\gamma\delta}$ introduced above. To formulate this relation we first note that ***ioco*** conformance to a specification S can be naturally extended on the set $I_\gamma(S)$ of LTSes that may have refused inputs and forbidden actions, but satisfy the following conditions.

- Empty trace is safe in any $I \in I_\gamma(S)$.
- Let us call $\beta\gamma\delta$-trace without input refusals and γ *δ-traces* and denote the set of all δ-traces of an LTS L as $Traces_\delta(L)$. For each σ, which is δ-trace of both S and $I \in I_\gamma(S)$, any output should be safe in $I \in I_\gamma(S)$ after σ.
- For each σ, which is δ-trace of both S and $I \in I_\gamma(S)$, and each input $?a$, which can extend σ in the specification (that is, $\sigma\langle ?a\rangle \in Traces_\delta(S)$), $?a$ should be safe in $I \in I_\gamma(S)$ after σ and $\sigma\langle ?a\rangle$ should also be a δ-trace of I.

For $I \in I_\gamma(S)$ we can say that $I \,\mathbf{ioco}\, S$ if and only if for each $\sigma \in Traces_\delta(S)$ and for each $s \in U \cup \{\delta\}$ $\quad \sigma\langle s\rangle \in Traces_\delta(I) \Rightarrow \sigma\langle s\rangle \in Traces_\delta(S)$.

Theorem 3.3 • *For each specification S without forbidden actions and completely defined implementation I without forbidden actions (the domain of*

the classic **ioco***)*

$$I \,\mathbf{ioco}\, S \Leftrightarrow I \,\mathbf{ioco}_{\beta\gamma\delta}\, \Delta(S) \Leftrightarrow I \,\mathbf{ioco}_{\beta\gamma\delta}\, \Gamma(S).$$

- *For each specification S without forbidden actions and $I \in I_\gamma(S)$*

$$I \,\mathbf{ioco}\, S \Leftrightarrow I \,\mathbf{ioco}_{\beta\gamma\delta}\, \Gamma(S).$$

The proof of this statement can also be found in [18]. Note, that in the second case $\Gamma(S)$ cannot be substituted by $\Delta(S)$, since implementations from $I_\gamma(S)$ nonconforming to S may conform to $\Delta(S)$.

4 Conclusion

The main results of this paper are definition of a conformance relation $\boldsymbol{ioco}_{\beta\gamma\delta}$ introducing semantics of forbidden actions and refused inputs into conformance testing theory based on LTS models and construction of two completion operation that transform any LTS into the input-enabled ones having the same sets of ***ioco***-conforming implementations. The second result makes possible definition of 'proper' LTS composition preserving ***ioco***-conformance.

Nevertheless, the problems stated in the end of Introduction are not solved completely. We have no compact construction of the analogous completion preserving the set of $\boldsymbol{ioco}_{\beta\gamma\delta}$-conforming implementations for an LTS with refused inputs. This construction is under development now.

Acknowledgements. We thank A. Petrenko from CRIM for helpful discussions.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[2] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[3] G. Bernot. *Testing against Formal Specifications: A Theoretical View.* In Proc. of TAPSOFT'91, Vol. 2. S. Abramsky and T. S. E. Maibaum, eds. LNCS 494, pp. 99–119, Springer-Verlag, 1991.

[4] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Information Retreival, Transfer, and Management for OSI. Framework: Formal Methods in Conformance Testing.* Committee Draft CD 13245-1, ITU-T Proposed Recommendation Z.500. ISO–ITU-T, Geneve, 1996. See also ITU-T. Recommendation Z.500. *Framework on formal methods in conformance testing.* International Telecommunications Union, Geneve, Switzerland, 1997.

[5] I. C. C. Phillips. *Refusal Testing.* Theoretical Computer Science 50, pp. 241–284, 1987.

[6] R. J. van Glaabek. *The Linear Time-Branching Time Spectrum II; the Semantics of Sequential Processes with Silent Moves.* Proc. of CONCUR'93, Hildesheim, Germany, August 1993. E. Best, ed. LNCS 715, pp. 66–81, Springer-Verlag, 1993.

[7] J. Tretmans. *Test Generation with Inputs, Outputs, and Repetitive Quiescence.* Software – Concepts and Tools, 17(3):103–120, 1996.

[8] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

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

[10] A. Petrenko, N. Yevtushenko, J. L. Huo. *Testing Transition Systems with Input and Output Testers.* Proc. of TestCom 2003, LNCS 2644, pp. 129–145, Springer-Verlag, 2003.

[11] J. L. Huo, A. Petrenko. *On Testing Partially Specified IOTS through Lossless Queues.* Proc. of TestCom 2004, LNCS 2978, pp. 76–94, Springer 2004.

[12] C. Jard , T. Jéron , L. Tanguy , C. Viho. *Remote testing can be as powerful as local testing.* In Proc. of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), October 1999, p.25–40.

[13] G. V. Bochmann, A. Petrenko. *Protocol Testing: Review of Methods and Relevance for Software Testing.* Proc. of ACM SIGSOFT ISSTA'1994, Software Engineering Notes, Special Issue, pp. 109–124.

[14] J. Helovuo, S. Leppanen. *Exploration Testing.* Proc of. 2-nd International Conference on Application of Concurrency to System Design, Newcastle upon Tyne, U.K., June 2001, pp. 201–210.

[15] L. Heerink. *Ins and Outs in Refusal Testing.* PhD thesis, IPA-CTIT, 1998.

[16] L. Heerink, J. Tretmans. *Refusal Testing for Classes of Transition Systems with inputs and Outputs.* In T. Mizuno, N. Shiratori, T. Higashino, A. Togashi, eds. Formal Description Techniques and Protocol Specification, Testing and Verification. Chapman & Hill, 1997.

[17] Z. Li, J. Wu, and X. Yin. *Refusal Testing for MIOTS with Nonlockable Output Channels.* In International Conference on Computer Networks and Mobile Computing, Beijing, China, October 2003, pp. 517–522.

[18] I. B. Bourdonov, A. S. Kossatchev, V. V. Kuliamin. *Theory of conformance testing for systems with refused inputs and forbidden actions. Synchronous case.* ISP RAS Technical Report 2005, in Russian. http://www.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2005.pdf