Model-Based Testing of Environmental Conformance of Components

Lars Frantzen[1,2] and Jan Tretmans[2,3]

[1] Instituto di Scienza e Tecnologie della Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche, Pisa – Italy
lars.frantzen@isti.cnr.it
[2] Institute for Computing and Information Sciences
Radboud University Nijmegen – The Netherlands
{lf,tretmans}@cs.ru.nl
[3] Embedded Systems Institute
Eindhoven – The Netherlands
jan.tretmans@esi.nl

Abstract. In component-based development, the correctness of a system depends on the correctness of the individual components and on their interactions. Model-based testing is a way of checking the correctness of a component by means of executing test cases that are systematically generated from a model of the component. This model should include the behaviour of how the component can be invoked, as well as how the component itself invokes other components. In many situations, however, only a model that specifies how others can use the component, is available. In this paper we present an approach for model-based testing of components where only these available models are used. Test cases for testing whether a component correctly reacts to invocations are generated from this model, whereas the test cases for testing whether a component correctly invokes other components, are generated from the models of these other components. A formal elaboration is given in the realm of labelled transition systems. This includes an implementation relation, called **eco**, which formally defines when a component is correct with respect to the components it uses, and a sound and exhaustive test generation algorithm for **eco**.

1 Introduction

Software testing involves checking of desired properties of a software product by systematically executing the software, while stimulating it with test inputs, and observing and checking the execution results. Testing is a widely used technique to assess the quality of software, but it is also a difficult, error-prone, and labor-intensive technique. Consequently, test automation is an important area of research and development: without automation it will not be feasible to test future generations of software products in an effective and efficient manner. Automation of the testing process involves automation of the execution of test cases, automation of the analysis of test results, as well as automation of the generation of sufficiently many and valid test cases.

F.S. de Boer et al. (Eds.): FMCO 2006, LNCS 4709, pp. 1–25, 2007.

Model-Based Testing. One of the emerging and promising techniques for test automation is model-based testing. In model based testing, a *model* of the desired behavior of the *implementation under test* (IUT) is the starting point for test generation and serves as the oracle for test result analysis. Large amounts of test cases can, in principle, be algorithmically and completely automatically generated from the model. If this model is valid, i.e., expresses precisely what the implementation under test should do, all these tests are valid, too. Model-based testing has recently gained increased attention with the popularization of modeling itself.

Most model-based testing methods deal with black-box testing of functionality. This implies that the kind of properties being tested concern the functionality of the system. Functionality properties express whether the system correctly does what it should do in terms of correct responses to given stimuli, as opposed to, e.g., performance, usability, or reliability properties. In black-box testing, the specification is the starting point for testing. The specification prescribes what the IUT should do, and what it should not do, in terms of the behavior observable at its external interfaces. The IUT is seen as a black box without internal detail, as opposed to white-box testing, where the internal structure of the IUT, i.e., the program code, is the basis for testing. Also in this paper we will restrict ourselves to black-box testing of functionality properties.

Model-based testing with labelled transition systems. One of the formal theories for model-based testing uses labelled transition systems as models, and a formal implementation relation called **ioco** for defining conformance between an IUT and a specification [10,11]. A labelled transition system is a structure with states representing the states of the system, and with transitions between states representing the actions that the system may perform. The implementation relation **ioco** expresses that an IUT conforms to its specification if the IUT never produces an output that cannot be produced by the specification. In this theory, an algorithm for the generation of test cases exists, which is provably sound for **ioco**-conformance, i.e., generated test cases only detect **ioco** errors, and exhaustive, i.e., all potential **ioco** errors can be detected.

Testing of Components. In component-based development, systems are built by gluing components together. Components are developed separately, often by different manufacturers, and they can be reused in different environments. A component is responsible for performing a specific task, or for delivering a specified service. A user requesting this service will invoke the component to provide its service. In doing so, the component may, in turn, invoke other components for providing their services, and these invoked components may again use other components. A component may at the same time act as a service provider and as a service requester.

A developer who composes a system from separate components, will only know about the services that the components perform, and not about their internal details. Consequently, clear and well-specified interfaces play a crucial role in

component technology, and components shall correctly implement these interface specifications. Correctness involves both the component's role as a service provider and its role as a service requester: a component must correctly provide its specified service, as well as correctly use other components.

Component-based testing. In our black-box setting, component-based testing concerns testing of behavior as it is observed at the component's interfaces. This applies to testing of individual components as well as to testing of aggregate systems built from components, and it applies to testing of provided services, as well as to testing of how other services are invoked.

When testing aggregated systems this can be done "bottom-up", i.e., starting with testing the components that do not invoke other components, and then adding components to the system that use the components already tested, and so forth, until the highest level has been reached. Another approach is to use *stubs* to simulate components that are invoked, so that a component can be tested without having the components available that are invoked by the component under test.

Model-based testing of components. For model-based testing of an individual component, we, in principle, need a complete model of the component. Such a model should specify the behavior at the service providing interface, the behavior at the service requesting interface, and the mutual dependencies between actions at both interfaces. Such a complete model, however, is often not available. Specifications of components are usually restricted to the behavior of the provided services. The specification of how other components are invoked is considered an internal implementation detail, and, from the point of view of a user of an aggregate system, it is.

Goal. The aim of this paper is to present an approach for model-based testing of a component at both the service providing interface and the requesting interface in a situation where a complete behavior model is not available. The approach assumes that a specification of the provided service is available for both the component under test, and for the components being invoked by the component under test. Test cases for the provided service are derived from the corresponding service specification. Test cases for checking how the component requests services from other components are derived from the provided service specifications of these other components.

The paper builds on the **ioco**-test theory for labelled transition systems, it discusses where this theory is applicable for testing components, and where it is not. A new implementation relation is introduced called *environmental conformance* – **eco**. This relation expresses that a component correctly invokes another component according to the provided service specification of that other component. A complete (sound and exhaustive) test generation algorithm for **eco** is given.

Overview. Section 2 starts with recalling the most important concepts of the **ioco**-test theory for labelled transition systems, after which Section 3 sets the

scene for formally testing components. The implementation relation **eco** is introduced in Section 4, followed by the test generation algorithm in Section 5. The combination of testing at different interfaces is briefly discussed in Section 6. Concluding remarks are presented in Section 7.

2 Testing for Labelled Transition Systems

Model-based testing deals with models, correctness (or conformance-) relations, test cases, test generation algorithms, and soundness and exhaustiveness of the generated test cases with respect to the conformance relations. This section presents the formal test theory for labelled transition systems using the **ioco**-conformance relation; see [10,11]. This theory will be our starting point for the discussion of model-based testing of components in the next sections.

Models. In the **ioco**-test theory, formal specifications, implementations, and test cases are all expressed as labelled transition systems.

Definition 1. *A* labelled transition system with inputs and outputs *is a 5-tuple* $\langle Q, L_I, L_U, T, q_0 \rangle$ *where* Q *is a countable, non-empty set of* states*;* L_I *is a countable set of* input labels*;* L_U *is a countable set of* output labels*, such that* $L_I \cap L_U = \emptyset$*;* $T \subseteq Q \times (L_I \cup L_U \cup \{\tau\}) \times Q$*, with* $\tau \notin L_I \cup L_U$*, is the* transition relation*; and* $q_0 \in Q$ *is the* initial state*.*

The labels in L_I and L_U represent the inputs and outputs, respectively, of a system, i.e., the system's possible interactions with its environment[1]. Inputs are usually decorated with '?' and outputs with '!'. We use $L = L_I \cup L_U$ when we abstract from the distinction between inputs and outputs.

The execution of an action is modeled as a transition: $(q, \mu, q') \in T$ expresses that the system, when in state q, may perform action μ, and go to state q' . This is more elegantly denoted as $q \xrightarrow{\mu} q'$. Transitions can be composed: $q \xrightarrow{\mu} q' \xrightarrow{\mu'} q''$, which is written as $q \xrightarrow{\mu \cdot \mu'} q''$.

Internal transitions are labelled by the special action τ ($\tau \notin L$), which is assumed to be unobservable for the system's environment. Consequently, the observable behavior of a system is captured by the system's ability to perform sequences of observable actions. Such a sequence of observable actions, say σ, is obtained from a sequence of actions under abstraction from the internal action τ, and it is denoted by $\stackrel{\sigma}{\Longrightarrow}$. If, for example, $q \xrightarrow{a \cdot \tau \cdot \tau \cdot b \cdot c \cdot \tau} q'$ $(a, b, c \in L)$, then we write $q \stackrel{a \cdot b \cdot c}{\Longrightarrow} q'$ for the τ-abstracted sequence of observable actions. We say that q is able to perform the *trace* $a \cdot b \cdot c \in L^*$. Here, the set of all finite sequences over L is denoted by L^*, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$ are finite sequences, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. Some more, standard notations and definitions are given in Definitions 2 and 3.

[1] The 'U' refers to 'uitvoer', the Dutch word for 'output', which is preferred for historical reasons, and to avoid confusion between L_O (letter 'O') and L_0 (digit zero).

Definition 2. *Let $p = \langle Q, L_I, L_U, T, q_0\rangle$ be a labelled transition system with $q, q' \in Q$, $\mu, \mu_i \in L \cup \{\tau\}$, $a, a_i \in L$, and $\sigma \in L^*$.*

$$
\begin{array}{lll}
q \xrightarrow{\mu} q' & \Leftrightarrow_{\text{def}} & (q, \mu, q') \in T \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' & \Leftrightarrow_{\text{def}} & \exists q_0, \ldots, q_n : \; q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q' \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & \Leftrightarrow_{\text{def}} & \exists q' : \; q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} \!\!\!\!/ & \Leftrightarrow_{\text{def}} & \textit{not } \exists q' : \; q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' \\
q \stackrel{\epsilon}{\Longrightarrow} q' & \Leftrightarrow_{\text{def}} & q = q' \;\; \text{or} \;\; q \xrightarrow{\tau \cdot \ldots \cdot \tau} q' \\
q \stackrel{a}{\Longrightarrow} q' & \Leftrightarrow_{\text{def}} & \exists q_1, q_2 : \; q \stackrel{\epsilon}{\Longrightarrow} q_1 \xrightarrow{a} q_2 \stackrel{\epsilon}{\Longrightarrow} q' \\
q \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} q' & \Leftrightarrow_{\text{def}} & \exists q_0 \ldots q_n : \; q = q_0 \stackrel{a_1}{\Longrightarrow} q_1 \stackrel{a_2}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} q_n = q' \\
q \stackrel{\sigma}{\Longrightarrow} & \Leftrightarrow_{\text{def}} & \exists q' : \; q \stackrel{\sigma}{\Longrightarrow} q' \\
q \stackrel{\sigma}{\Longrightarrow\!\!\!\!/} & \Leftrightarrow_{\text{def}} & \textit{not } \exists q' : q \stackrel{\sigma}{\Longrightarrow} q'
\end{array}
$$

In our reasoning about labelled transition systems we will not always distinguish between a transition system and its initial state. If $p = \langle Q, L_I, L_U, T, q_0\rangle$, we will identify the process p with its initial state q_0, and, e.g., we write $p \stackrel{\sigma}{\Longrightarrow}$ instead of $q_0 \stackrel{\sigma}{\Longrightarrow}$.

Definition 3. *Let p be a (state of a) labelled transition system, P a set of states, $A \subseteq L$ a set of labels, and $\sigma \in L^*$.*

1. $\textit{traces}(p) \;=_{\text{def}}\; \{\; \sigma \in L^* \mid p \stackrel{\sigma}{\Longrightarrow} \;\}$
2. $p \,\textbf{after}\, \sigma \;=_{\text{def}}\; \{\; p' \mid p \stackrel{\sigma}{\Longrightarrow} p' \;\}$
3. $P \,\textbf{after}\, \sigma \;=_{\text{def}}\; \bigcup \{\; p \,\textbf{after}\, \sigma \mid p \in P \;\}$
4. $P \,\textbf{refuses}\, A \;=_{\text{def}}\; \exists p \in P,\; \forall \mu \in A \cup \{\tau\} :\; p \xrightarrow{\mu}\!\!\!\!/$

The class of labelled transition systems with inputs in L_I and outputs in L_U is denoted as $\mathcal{LTS}(L_I, L_U)$. For technical reasons we restrict this class to *strongly converging* and *image finite* systems. Strong convergence means that infinite sequences of τ-actions are not allowed to occur. Image finiteness means that the number of non-deterministically reachable states shall be finite, i.e., for any σ, $p \,\textbf{after}\, \sigma$ shall be finite.

Representing labelled transition systems. To represent labelled transition systems we use either graphs (as in Fig. 1), or expressions in a process-algebraic-like language with the following syntax:

$$B \;::=\; a \,;\, B \;\;|\;\; \mathbf{i} \,;\, B \;\;|\;\; \mathbf{\Sigma}\, \mathcal{B} \;\;|\;\; B \,|[\, G \,]|\, B \;\;|\;\; P$$

Expressions in this language are called behavior expressions, and they define labelled transition systems following the axioms and rules given in Table 1.

In that table, $a \in L$ is a label, B is a behavior expression, $\mathcal{B}$ is a countable set of behavior expressions, $G \subseteq L$ is a set of labels, and P is a *process name*, which must be linked to a named behavior expression by a process definition of the form $P \;:=\; B_P$. In addition, we use $B_1 \,\Box\, B_2$ as an abbreviation for $\mathbf{\Sigma}\{B_1, B_2\}$, **stop** to denote $\mathbf{\Sigma}\,\emptyset$, $\|$ as an abbreviation for $|[\, L \,]|$, i.e., synchronization on all observable actions, and $|||$ as an abbreviation for $|[\, \emptyset \,]|$, i.e., full interleaving without synchronization.

Table 1. Structural operational semantics

$$\frac{}{a\,;B \xrightarrow{a} B} \qquad \frac{}{\mathbf{i}\,;B \xrightarrow{\tau} B} \qquad \frac{B \xrightarrow{\mu} B'}{\Sigma\,\mathcal{B} \xrightarrow{\mu} B'}\ B{\in}\mathcal{B},\ \mu{\in}L\cup\{\tau\}$$

$$\frac{B_1 \xrightarrow{\mu} B_1'}{B_1\,|[\,G\,]|\,B_2 \xrightarrow{\mu} B_1'\,|[\,G\,]|\,B_2} \qquad \frac{B_2 \xrightarrow{\mu} B_2'}{B_1\,|[\,G\,]|\,B_2 \xrightarrow{\mu} B_1\,|[\,G\,]|\,B_2'} \qquad \mu{\in}(L\cup\{\tau\})\backslash G$$

$$\frac{B_1 \xrightarrow{a} B_1',\ B_2 \xrightarrow{a} B_2'}{B_1\,|[\,G\,]|\,B_2 \xrightarrow{a} B_1'\,|[\,G\,]|\,B_2'}\ a{\in}G \qquad \frac{B_P \xrightarrow{\mu} B'}{P \xrightarrow{\mu} B'}\ P := B_P,\ \mu{\in}L\cup\{\tau\}$$

Input-output transition systems. In model-based testing there is a specification, which prescribes what an IUT shall do, and there is the IUT itself which is a black-box performing some behavior. In order to formally reason about the IUT's behavior the assumption is made that the IUT behaves as if it were some kind of formal model. This assumption is sometimes referred to as the test assumption or test hypothesis.

In the **ioco**-test theory a specification is a labelled transition system in $\mathcal{LTS}(L_I, L_U)$. An implementation is assumed to behave as if it were a labelled transition system that is always able to perform any input action, i.e., all inputs are enabled in all states. Such a system is defined as an *input-output transition system*. The class of such input-output transition systems is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$.

Definition 4. *An* input-output transition system *is a labelled transition system with inputs and outputs $\langle Q, L_I, L_U, T, q_0\rangle$ where all input actions are enabled in any reachable state:*

$$\forall \sigma, q:\ q_0 \stackrel{\sigma}{\Longrightarrow} q \ \text{ implies } \ \forall a{\in}L_I:\ q \stackrel{a}{\Longrightarrow}$$

A state of a system where no outputs are enabled, and consequently the system is forced to wait until its environment provides an input, is called *suspended*, or *quiescent*. An observer looking at a quiescent system does not see any outputs. This particular observation of seeing nothing can itself be considered as an event, which is denoted by δ ($\delta \notin L \cup \{\tau\}$); $p \xrightarrow{\delta} p$ expresses that p allows the observation of quiescence. Also these transitions can be composed, e.g., $p \stackrel{\delta\cdot ?a\cdot\delta\cdot ?b\cdot !x}{\Longrightarrow}$ expresses that initially p is quiescent, i.e., does not produce outputs, but p does accept input action $?a$, after which there are again no outputs; when then input $?b$ is performed, the output $!x$ is produced. We use L_δ for $L \cup \{\delta\}$, and traces that may contain the quiescence action δ are called *suspension traces*.

Definition 5. *Let $p = \langle Q, L_I, L_U, T, q_0\rangle{\in}\mathcal{LTS}(L_I, L_U)$.*

1. *A state q of p is* quiescent*, denoted by $\delta(q)$, if* $\forall \mu{\in}L_U \cup \{\tau\}:\ q \xrightarrow{\mu}\!\!\!\!\!/\ \ $

2. $p_\delta =_{\text{def}} \langle\, Q,\ L_I,\ L_U \cup \{\delta\},\ T \cup T_\delta,\ q_0 \,\rangle$,
   with $T_\delta =_{\text{def}} \{\ q \xrightarrow{\delta} q \mid q{\in}Q,\ \delta(q)\ \}$
3. *The* suspension traces *of* p *are* $Straces(p) =_{\text{def}} \{\ \sigma{\in}L_\delta^* \mid p_\delta \stackrel{\sigma}{\Longrightarrow}\ \}$

From now on we will usually include δ-transitions in the transition relations, i.e., we consider p_δ instead of p, unless otherwise indicated. Definitions 2 and 3 also apply to transition systems with label set L_δ.

The implementation relation **ioco**. An implementation relation is intended to precisely define when an implementation is correct with respect to a specification. The first implementation relation that we consider is **ioco**, which is abbreviated from input-output conformance. Informally, an implementation $i{\in}\mathcal{IOTS}(L_I, L_U)$ is **ioco**-conforming to specification $s{\in}\mathcal{LTS}(L_I, L_U)$ if any experiment derived from s and executed on i leads to an output (including quiescence) from i that is foreseen by s. We define **ioco** as a special case of the more general class of relations $\mathbf{ioco}_\mathcal{F}$, where $\mathcal{F} \subseteq L_\delta^*$ is a set of suspension traces, which typically depends on the specification s.

Definition 6. *Let q be a state in a transition system, Q be a set of states, $i{\in}\mathcal{IOTS}(L_I, L_U)$, $s{\in}\mathcal{LTS}(L_I, L_U)$, and $\mathcal{F} \subseteq (L_I \cup L_U \cup \{\delta\})^*$, then*

1. $out(q) =_{\text{def}} \{\ x{\in}L_U \mid q \xrightarrow{x}\ \} \cup \{\ \delta \mid \delta(q)\ \}$
2. $out(Q) =_{\text{def}} \bigcup \{\ out(q) \mid q{\in}Q\ \}$
3. $i\ \mathbf{ioco}_\mathcal{F}\ s \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma{\in}\mathcal{F}: \quad out(\, i\ \mathbf{after}\ \sigma\,) \subseteq out(\, s\ \mathbf{after}\ \sigma\,)$
4. $i\ \mathbf{ioco}\ s \quad \Leftrightarrow_{\text{def}} \quad i\ \mathbf{ioco}_{Straces(s)}\ s$

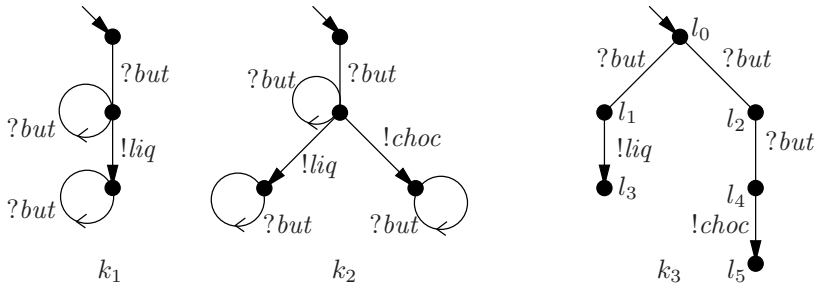


Fig. 1. Example labelled transition systems

Example 1. Figure 1 presents three examples of labelled transition systems modeling candy machines. There is an input action for pushing a button ?*but*, and there are outputs for obtaining chocolate !*choc* and liquorice !*liq*: $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$.

Since $k_1, k_2{\in}\mathcal{IOTS}(L_I, L_U)$ they can be both specifications and implementations; k_3 is not input-enabled, and can only be a specification. We have that $out(\, k_1\ \mathbf{after}\ ?but\,) = \{!liq\} \subseteq \{!liq, !choc\} = out(\, k_2\ \mathbf{after}\ ?but\,)$; so we get now

k_1 **ioco** k_2, but k_2 **io$\not$co** k_1. For k_3 we have $out(\, k_3$ **after** $?but\,) = \{!liq, \delta\}$ and $out(\, k_3$ **after** $?but{\cdot}?but\,) = \{!choc\}$, so both k_1, k_2 **io$\not$co** k_3.

The importance of having suspension actions δ in the set $\mathcal{F}$ over which **ioco** quantifies is also illustrated in Fig. 2. It holds that $out(\, r_1$ **after** $?but{\cdot}?but\,) = out(\, r_2$ **after** $?but{\cdot}?but\,) = \{!liq, !choc\}$, but we have $out(\, r_1$ **after** $?but{\cdot}\delta{\cdot}?but\,) = \{!liq, !choc\} \supset \{!choc\} = out(\, r_2$ **after** $?but{\cdot}\delta{\cdot}?but\,)$. So, without δ in these traces r_1 and r_2 would be considered implementations of each other in both directions, whereas with δ, r_2 **ioco** r_1 but r_1 **io$\not$co** r_2.

Underspecification and the implementation relation **uioco**. The implementation relation **ioco** allows to have partial specifications. A partial specification does not specify the required behavior of the implementation after all possible traces. This corresponds to the fact that specifications may be non-input enabled, and inclusion of *out*-sets is only required for suspension traces that explicitly occur in the specification. Traces that do not explicitly occur are called underspecified. There are different ways of dealing with underspecified traces. The relation **uioco** does it in a slightly different manner than **ioco**. For the rationale consider Example 2.

Example 2. Consider k_3 of Fig. 1 as a specification. Since k_3 is not input-enabled, it is a partial specification. For example, $?but{\cdot}?but{\cdot}?but$ is an underspecified trace, and any implementation behavior is allowed after it. On the other hand, $?but$ is clearly specified; the allowed outputs after it are $!liq$ and δ. For the trace $?but{\cdot}?but$ the situation is less clear. According to **ioco** the expected output after $?but{\cdot}?but$ is $out(\, k_3$ **after** $?but{\cdot}?but\,) = \{!choc\}$. But suppose that in the first $?but$-transition k_3 moves nondeterministically to state l_1 (the left branch) then one might argue that the second $?but$-transition is underspecified, and that, consequently, any possible behavior is allowed in an implementation. This is exactly where **ioco** and **uioco** differ: **ioco** postulates that $?but{\cdot}?but$ is not an underspecified trace, because there exists a state where it is specified, whereas **uioco** states that $?but{\cdot}?but$ is underspecified, because there exists a state where it is underspecified.

Formally, **ioco** quantifies over $\mathcal{F} = Straces(s)$, which are all possible suspension traces of the specification s. The relation **uioco** quantifies over $\mathcal{F} = Utraces(s) \subseteq Straces(s)$, which are the suspension traces without the possibly underspecified traces, i.e., all suspension traces σ of s for which it is *not* possible that a prefix σ_1 of σ ($\sigma = \sigma_1{\cdot}a{\cdot}\sigma_2$) leads to a state of s where the remainder $a{\cdot}\sigma_2$ of σ is underspecified, that is, a is refused.

Definition 7. *Let* $i \in \mathcal{IOTS}(L_I, L_U)$*, and* $s \in \mathcal{LTS}(L_I, L_U)$*.*

1. $Utraces(s) =_{\text{def}} \{\ \sigma \in Straces(s) \mid \forall \sigma_1, \sigma_2 \in L_\delta^*,\ a \in L_I :$
   $\sigma = \sigma_1{\cdot}a{\cdot}\sigma_2$ implies not s **after** σ_1 **refuses** $\{a\}\ \}$
2. i **uioco** $s \Leftrightarrow_{\text{def}} i$ **ioco**$_{Utraces(s)}$ s

Example 3. Because $Utraces(s) \subseteq Straces(s)$ it is evident that **uioco** is not stronger than **ioco**. That it is strictly weaker follows from the following example. Take k_3 in Fig. 1 as a (partial) specification, and consider r_1 and r_2 from Fig. 2 as potential implementations. Then r_2 **io$\not$co** k_3 because $!liq \in out(r_2$ **after** $?but{?}but\,)$

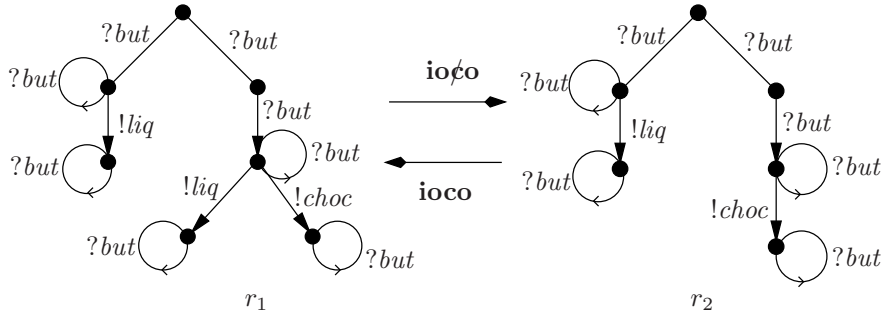


Fig. 2. More labelled transition systems

and $!liq \notin out(\, k_3 \textbf{ after } ?but{\cdot}?but\,)$. But r_2 **uioco** k_3 because we have $?but{\cdot}?but \notin Utraces(k_3)$. Also r_1 **io⊄co** k_3, but in this case also r_1 **ui⊄co** k_3. The reason for this is that we have $?but{\cdot}\delta{\cdot}?but \in Utraces(k_3)$, $!liq \in out(\, r_1 \textbf{ after } ?but{\cdot}\delta{\cdot}?but\,)$ and $!liq \notin out(\, k_3 \textbf{ after } ?but{\cdot}\delta{\cdot}?but\,)$.

Test Cases. For the generation of test cases from labelled transition system specifications, which can test implementations that behave as input-output transition systems, we must first define what test cases are. Then we discuss what test execution is, what it means to pass a test, and which correctness properties should hold for generated test cases so that they will detect all and only non-conforming implementations. A test generation algorithm is not given in this section; for **ioco** and **uioco** test generation algorithms we will refer to other publications. In Sect. 5, this paper will give a test generation algorithm for the new implementation relation **eco** for component conformance, which will be defined in Sect. 4.

A test case is a specification of the behavior of a tester in an experiment carried out on an implementation under test. The behavior of such a tester is also modeled as a special kind of input-output transition system, but, naturally, with inputs and outputs exchanged. Consequently, input-enabledness of a test case means that all actions in L_U (i.e., the set of outputs of the implementation) are enabled. For observing quiescence we add a special label θ to the transition systems modeling tests ($\theta \notin L$).

Definition 8. *A* test case t *for an implementation with inputs* L_I *and outputs* L_U *is an input-output transition system* $\langle Q, L_U, L_I \cup \{\theta\}, T, q_0 \rangle \in \mathcal{IOTS}(L_U, L_I \cup \{\theta\})$ *generated following the next fragment of the syntax for behavior expressions, where* **pass** *and* **fail** *are process names:*

$$
\begin{array}{rl}
t ::= & \textbf{pass} \\
\mid & \textbf{fail} \\
\mid & \Sigma\ \{\ x\ ;\ t \mid x \in L_U \cup \{a\}\ \} \quad \textit{for some } a \in L_I \\
\mid & \Sigma\ \{\ x\ ;\ t \mid x \in L_U \cup \{\theta\}\ \} \\
& \textit{where} \quad \textbf{pass} := \Sigma\ \{\ x\ ;\ \textbf{pass}\ \mid\ x \in L_U \cup \{\theta\}\ \} \\
& \qquad\quad\ \ \textbf{fail}\ := \Sigma\ \{\ x\ ;\ \textbf{fail}\ \mid\ x \in L_U \cup \{\theta\}\ \}
\end{array}
$$

The class of test cases for implementations with inputs L_I and outputs L_U is denoted as $\mathcal{TTS}(L_U, L_I)$. For testing an implementation, normally a set of test cases is used. Such a set is called a *test suite* $T \subseteq \mathcal{TTS}(L_U, L_I)$.

Test Execution. Test cases are run by putting them in parallel with the implementation under test, where inputs of the test case synchronize with the outputs of the implementations, and vice versa. Basically, this can be modeled using the behavior-expression operator $\|$. Since, however, we added the special label θ to test cases to test for quiescence, this operator has to be extended a bit, and is then denoted as $\rceil\!|\!|$.

Because of nondeterminism in implementations, it may be the case that testing the same implementation with the same test case may lead to different test results. An implementation passes a test case if and only if all its test runs lead to a pass state of the test case. All this is reflected in the following definition.

Definition 9. *Let* $t \in \mathcal{TTS}(L_U, L_I)$ *and* $i \in \mathcal{IOTS}(L_I, L_U)$.

1. Running *a test case* t *with an implementation* i *is expressed by the parallel operator* $\rceil\!|\!| : \mathcal{TTS}(L_U, L_I) \times \mathcal{IOTS}(L_I, L_U) \to \mathcal{LTS}(L_I \cup L_U \cup \{\theta\})$ *which is defined by the following inference rules:*

$$\frac{i \xrightarrow{\tau} i'}{t \rceil\!|\!| i \xrightarrow{\tau} t \rceil\!|\!| i'} \qquad \frac{t \xrightarrow{a} t',\ i \xrightarrow{a} i'}{t \rceil\!|\!| i \xrightarrow{a} t' \rceil\!|\!| i'}\ a \in L_I \cup L_U \qquad \frac{t \xrightarrow{\theta} t',\ i \xrightarrow{\delta}}{t \rceil\!|\!| i \xrightarrow{\theta} t' \rceil\!|\!| i}$$

2. *A* test run *of* t *with* i *is a trace of* $t \rceil\!|\!| i$ *leading to one of the states* **pass** *or* **fail** *of* t*:*

   σ *is a test run of* t *and* $i \quad \Leftrightarrow_{\text{def}} \quad \exists i' :\ t \rceil\!|\!| i \stackrel{\sigma}{\Longrightarrow} \mathbf{pass} \rceil\!|\!| i' \ \text{ or } \ t \rceil\!|\!| i \stackrel{\sigma}{\Longrightarrow} \mathbf{fail} \rceil\!|\!| i'$

3. *Implementation* i passes *test case* t *if all test runs go to the* **pass***-state of* t*:*

$$i\ \mathbf{passes}\ t \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in L_\theta^*,\ \forall i' :\ t \rceil\!|\!| i \stackrel{\sigma}{\not\Longrightarrow} \mathbf{fail} \rceil\!|\!| i'$$

4. *An implementation* i passes *a test suite* T *if it passes all test cases in* T*:*

$$i\ \mathbf{passes}\ T \quad \Leftrightarrow_{\text{def}} \quad \forall t \in T :\ i\ \mathbf{passes}\ t$$

   If i *does not pass a test case or a test suite, it* **fails**.

Completeness of testing. For **ioco**-testing a couple of algorithms exist that can generate test cases from labelled transition system specifications [10,12,8]. These algorithms have been shown to be correct, in the sense that the test suites generated with these algorithms are able to detect all, and only all, non-**ioco** correct implementations. This is expressed by the properties of soundness and exhaustiveness. A test suite is sound if any test run leading to **fail** indicates an error, and a test suite is exhaustive if all possible errors in implementations can be detected. Of course, exhaustiveness is merely a theoretical property: for realistic systems exhaustive test suites would be infinite, both in number of test cases and in the size of test cases. But yet, exhaustiveness does express that there are no **ioco**-errors that are undetectable.

Definition 10. *Let s be a specification and T a test suite; then for* **ioco***:*

$$\begin{array}{lll} T \textit{ is sound} & \Leftrightarrow_{\text{def}} & \forall i \in \mathcal{IOTS}(L_I, L_U) : i \textbf{ ioco } s \text{ implies } i \textbf{ passes } T \\ T \textit{ is exhaustive} & \Leftrightarrow_{\text{def}} & \forall i \in \mathcal{IOTS}(L_I, L_U) : i \textbf{ ioco } s \text{ if } i \textbf{ passes } T \end{array}$$

3 Towards Formal Component-Based Testing

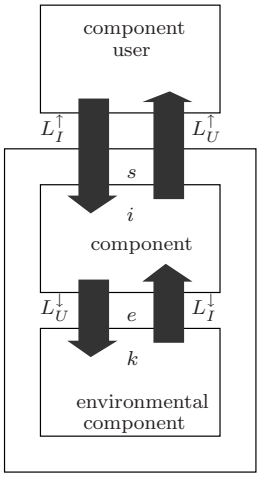
Correctness of components. In component-based testing we wish to test components. A component is a (software) entity that provides some *service* to a potential user. A user can invoke, or request this service. The service is provided via some *interface* of the component, referred to as the *service interface*, *providing interface*, *called interface*, or *upper interface*. A component, in turn, may use other components in its environment, i.e., the component acts as a user of, or requests a service from another component, which, in turn, provides that service. The services provided by these other components are requested via another interface, to which we refer as *required interface*, *calling interface*, or *lower interface*; see Fig. 3(a).

For a service requester it is transparent whether the component i invokes services of other environmental components, like k, at its lower interface, or not. The service requester is only interested whether the component i provides the requested service at the service interface in compliance with its specification s.

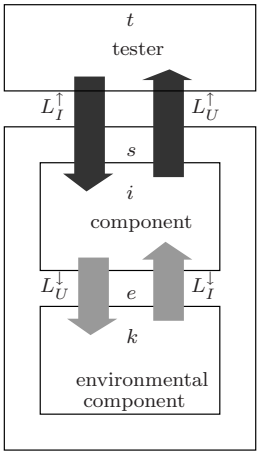
On the other hand, the environmental component k that is being invoked via the lower interface of i, does not care about the service being provided by the component i. It only cares whether the component i correctly requests for the services that the environmental component k provides, according to the rules laid down in k's service specification e.

Yet, although the correctness requirements on the behavior of a component can be clearly split into requirements on the upper interface and requirements on the lower interface, the correctness of the whole component, naturally, involves correct behavior on both interfaces. Moreover, the behavior of the component on both interfaces is in general not independent: a service request to an environmental component at the lower interface is typically triggered by a service request at the upper interface, and the result of the latter depends on the result of the first.

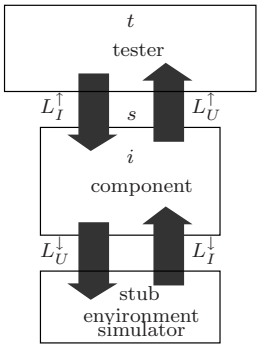
When specifying components, the emphasis is usually on the specification of the provided service, since this is what the component must fulfill and what a user of the component sees. The component's behavior at the lower interface is often not specified. It can only be indirectly derived from what the environmental component expects, i.e., from the provided service specification of that used component. In this paper we will formalize model-based testing of components at their lower interface using the upper interface specification of the environmental component that is invoked. By so doing, we strictly split the requirements on the lower interface from the requirements on the upper interface, since this is the only passable way to go when only specifications of the provided services are available.



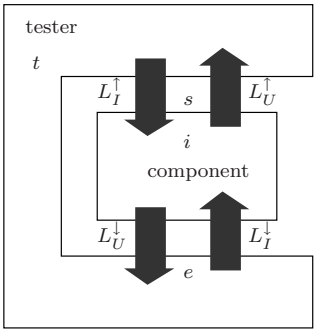
(a) A component i with its "lower-level" environment k.



(b) A composed system of i and k tested at the service level.



(c) A component i tested with a stub at its lower interface.



(d) A component i tested concurrently at the service- and lower interface in a "horse-shoe".

Fig. 3. Component-based testing

This is also the approach of recent, service-oriented testing frameworks like *audition* [2]. This framework assumes behavioral specifications of the provided service interfaces. Based on these specifications, a testing phase is introduced when services ask for being published at a service registry – the service undergoes a monitored trial before being put "on stage". During this testing phase, the service under test is actively tested at its upper interface, and it is additionally tested, whether the service correctly invokes other services via its lower interface.

If, instead, one wants to take requirements on the interdependency between the interfaces into account, more complete specifications are needed. This is not treated in this paper. For a survey of component-based testing see [9,6].

Formalizing components. We will formalize the behavior of component services in the realm of labelled transition systems. Fig. 3(a) gives a first step towards the formalization of these concepts. The component under consideration is a component implementation denoted by i; i is an input-output transition system, or, more precisely, the implementation i, which is seen as a black-box, is assumed to behave as an input-output transition system (cf. Section 2: test assumption). The actions that can occur at the upper interface are inputs $L_I^{\uparrow}$ and outputs $L_U^{\uparrow}$, whereas $L_I^{\downarrow}$ and $L_U^{\downarrow}$ represent the inputs and outputs, respectively, at the lower interface. Thus $i \in \mathcal{IOTS}(L_I^{\uparrow} \cup L_I^{\downarrow}, L_U^{\uparrow} \cup L_U^{\downarrow})$.

The service to be provided by the component at the upper interface is specified by s, which only involves the upper interface: $s \in \mathcal{LTS}(L_I^{\uparrow}, L_U^{\uparrow})$. The behavior of the provided service of the environmental component used by i is specified by $e \in \mathcal{LTS}(L_U^{\downarrow}, L_I^{\downarrow})$, and implemented by $k \in \mathcal{IOTS}(L_U^{\downarrow}, L_I^{\downarrow})$. Only the actions at the lower interface of i, which correspond to the actions of the upper interface of the invoked environmental component k, but with inputs and outputs exchanged, are involved here. Of course, the environmental component, in turn, may have a lower interface via which it will invoke yet other components, but for the component i, being just a service requester for e, this is transparent. In addition, in realistic situations i will usually request services from several different components, but we restrict our discussion to only one service being called. Considering several environmental components can in this setting, for instance, be expressed as their parallel (interleaved) composition, leading again to a single component.

Typically, input actions at the upper interface model the request for, i.e., the start of a service, whereas output actions model the result provided by that service. Conversely, at the lower interface the output actions model requests to an environmental component, whereas input actions model the results provided by the environmental component.

Testing components. A component can be tested in different ways. The simplest, and often used way is to test at the upper interface as in Fig. 3(b). This leads to a "bottom-up" test strategy, where the components that do not invoke other components, are tested first. After this, components are added that use these already tested components, so that these subsystems can be tested, to which then again components can be added, until all components have been added and tested. In principle, this way of testing is sufficient in the sense that all functionality that is observable from a service requester (user) point of view is tested. There are some disadvantages of this testing method, though. The first is that the behavior at the lower interface of the component is not thoroughly tested. This apparently did not lead to failures in the services provided (because these were tested), but it might cause problems when a component is replaced by a new or other (version of the) component, or if a component is reused in another environment. For instance, one environmental component may be robust

enough to deal with certain erroneous invocations, whereas another component providing the same service is not. If now the less forgiving one substitutes the original one, the system may not operate anymore. This would affect some of the basic ideas behind component-based development, viz., that of reusability and substitutability of components. A second disadvantage is that this test strategy leads to a strict order in testing of the components, and to a long critical test path. Higher level components cannot be tested before all lower level components have been finished and tested.

Fig. 3(c) shows an alternative test strategy where a lower level component is replaced by a stub or a simulator. Such a stub simulates the basic behavior of the lower level component, providing some functionality of e, typically with hard-coded responses for all requests which i might make on e. The advantage is that components need not to be tested in a strict bottom-up order, but still stubs are typically not powerful enough to guarantee thorough testing of the lower interface behavior of a component, in particular concerning testing of abnormal behavior or robustness. Moreover, stubs have to be developed separately.

The most desirable situation for testing components is depicted in Fig. 3(d): a test environment as a wrapper, or "horse-shoe", around the component with the possibility to fully control and observe all the interfaces of the component. This requires the development of such an environment, and, moreover, the availability of behavior specifications for all these interfaces. The aim of this paper is to work towards this way of testing in a formal context with model-based testing.

Model-based testing of components. For model-based testing of a component in a horse-shoe we need, in principle, a complete model of the behavior of the component specified at all its interfaces. But, as explained above, the specification of a component is usually restricted to the behavior at its upper interface. We indeed assume the availability of a specification of the upper interface of the component under test: $s \in \mathcal{LTS}(L_I^{\uparrow}, L_U^{\uparrow})$. Moreover, instead of having a specification of the lower interface itself, we use the specification of the upper interface of the environmental component that is invoked at the lower interface: $e \in \mathcal{LTS}(L_U^{\downarrow}, L_I^{\downarrow})$. This means that we are not directly testing what the component under test shall do, but what the environmental component expects it to do. Besides, what is missing in these two specifications, and what is consequently also missing in the model-based testing of the component, are the dependencies between the behaviors at the upper and the lower interfaces.

For testing the behavior at the upper interface the **ioco**- or **uioco**-test theory with the corresponding test generation algorithms can directly be used: there is a formal model $s \in \mathcal{LTS}(L_I^{\uparrow}, L_U^{\uparrow})$ from which test cases can be generated, and the implementation is assumed to behave as an input-enabled input-output transition system; see Sect. 2. Moreover, the implementation relations **ioco** and **uioco** seem to express what is intuitively required from a correct implementation at the upper interface: each possible output of the implementation must be included in the outputs of the specification, and also quiescence is only allowed if the specification allows that: a service requester would be disappointed if (s)he would not get an output result if an output is guaranteed in the specification.

For testing the behavior at the lower interface this testing theory is not directly applicable: there is no specification of the required behavior at the lower interface but only a specification of the environment of this lower interface: $e \in \mathcal{LTS}(L_U^{\downarrow}, L_I^{\downarrow})$. This means that we need an implementation relation and a test generation algorithm for such environmental specifications. An issue for such an implementation relation is the treatment of quiescence. Whereas a service requester expects a response when one is specified, a service provider will usually not care when no request is made when this is possible, i.e., the provider does not care about quiescence, but if a request is received it must be a correct request. In the next section we will formally elaborate these ideas, and define the implementation relation for *environmental conformance* **eco**. Subsequently, Sect. 5 will present a test generation algorithm for **eco** including soundness and exhaustiveness, and then Section 6 will briefly discuss the combined testing at the upper- and lower interfaces thus realizing a next step in the "horse-shoe" approach.

4 Environmental Conformance

In this section the implementation relation for environmental conformance **eco** is presented. Referring to Fig. 3(d) this concerns defining the correctness of the behavior of i at its lower interface with respect to what environment specification e expects. Here, we only consider the lower interface of i that communicates with the upper interface of e (or, more precisely, with an implementation k of specification e). Consequently, we use L_I to denote the inputs of i at its lower interface, which are the outputs of e, and L_U to denote the outputs if i at its lower interface, which correspond to the inputs of e. The implementation i is assumed to be input enabled: $i \in \mathcal{IOTS}(L_I, L_U)$; e is just a labelled transition system with inputs and outputs: $e \in \mathcal{LTS}(L_U, L_I)$.

An implementation i can be considered correct with respect to an environment e if the outputs that i produces can be accepted by e, and, conversely, if the outputs produced by e can be accepted by i. Since i is assumed to be input enabled, the latter requirement is trivially fulfilled in our setting. Considering the discussion in Sect. 3, quiescence of i is not an issue here, and consequently it is not considered as a possible output: if i requests a service from e it should do so in the correct way, but i is not forced to request a service just because e is ready to accept such a request. Conversely, quiescence of e does matter. The implementation i would be worried if the environment would not give a response, i.e., would be quiescent, if this were not specified. This, however, is an issue of the correctness of the environment implementation k with respect to the environment specification e, which is not of concern for **eco**.

For the formalization of **eco** we first have to define the sets of outputs (without quiescence), and inputs of a labelled transition system. Note that the set of outputs after a trace σ, $uit(\, p \,\mathbf{after}\, \sigma\,)$, collects all outputs that a system may nondeterministically execute, whereas for an input to be in $in(\, p \,\mathbf{after}\, \sigma\,)$ it must be executable in all nondeterministically reachable states (cf. the classical

may- and *must*-sets for transition systems [4]). This is justified by the fact that outputs are initiated by the system itself, whereas inputs are initiated by the system's environment, so that acceptance of an input requires that such an input is accepted in all possible states where a system can nondeterministically be. The thus defined set of inputs is strongly related to the set of *Utraces* (Def. 7 in Sect. 2), a fact that will turn out to be important for proving the correctness of test generation in Sect. 5.

Definition 11. *Let q be (a state of) an LTS, and let Q be a set of states.*

1. $in(q) =_{\text{def}} \{ a \in L_I \mid q \stackrel{a}{\Longrightarrow} \}$
2. $in(Q) =_{\text{def}} \bigcap \{ in(q) \mid q \in Q \}$
3. $uit(q) =_{\text{def}} \{ x \in L_U \mid q \stackrel{x}{\Longrightarrow} \}$
4. $uit(Q) =_{\text{def}} \bigcup \{ uit(q) \mid q \in Q \}$

Proposition 1

1. $in(\, q \,\textbf{after}\, \sigma \,) = \{ a \in L_I \mid \text{not } q \,\textbf{after}\, \sigma \,\textbf{refuses}\, \{a\} \}$
2. $uit(\, q \,\textbf{after}\, \sigma \,) = out(\, q \,\textbf{after}\, \sigma \,) \setminus \{ \delta \}$
3. $Utraces(p) = \{ \sigma \in Straces(s) \mid \forall \sigma_1, \sigma_2 \in L_\delta^*,\ a \in L_I : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \text{ implies } a \in in(\, p \,\textbf{after}\, \sigma_1 \,) \}$

Using these definitions we define **eco**: it expresses that after any possible $Utrace$ (without quiescence) of the environment e the outputs that implementation i may produce shall be specified inputs in all possible states that e may (nondeterministically) reach.

Definition 12. *Let $i \in \mathcal{IOTS}(L_I, L_U)$, $e \in \mathcal{LTS}(L_U, L_I)$.*

$$i \;\textbf{eco}\; e \quad \Leftrightarrow_{\text{def}} \quad \forall \sigma \in Utraces(e) \cap L^* : \quad uit(\, i \,\textbf{after}\, \sigma \,) \subseteq in(\, e \,\textbf{after}\, \sigma \,)$$

Now we have the desired property that after any common behavior of i and e, or of i and k, their outputs are mutually accepted as inputs. As mentioned above, some of these properties are trivial because our implementations are assumed to be input-enabled (we take the "pessimistic view on the environment", cf. [1]).

Definition 13. *$p \in \mathcal{LTS}(L_I, L_U)$ and $q \in \mathcal{LTS}(L_U, L_I)$ are* mutually receptive *iff $\forall \sigma \in L^*, \forall x \in L_U, \forall a \in L_I, \forall p', q'$ we have*

$$p \,\|\, q \stackrel{\sigma}{\Longrightarrow} p' \,\|\, q' \text{ implies } \Big(\big(p' \stackrel{!x}{\Longrightarrow} \text{ implies } q' \stackrel{?x}{\Longrightarrow} \big) \text{ and } \big(q' \stackrel{!a}{\Longrightarrow} \text{ implies } p' \stackrel{?a}{\Longrightarrow} \big) \Big)$$

Proposition 2. *Let $i \in \mathcal{IOTS}(L_I, L_U)$, $e \in \mathcal{LTS}(L_U, L_I)$, $k \in \mathcal{IOTS}(L_U, L_I)$.*

1. i **eco** e implies *i and e are mutually receptive*
2. i **eco** e and k **uioco** e implies *i and k are mutually receptive*

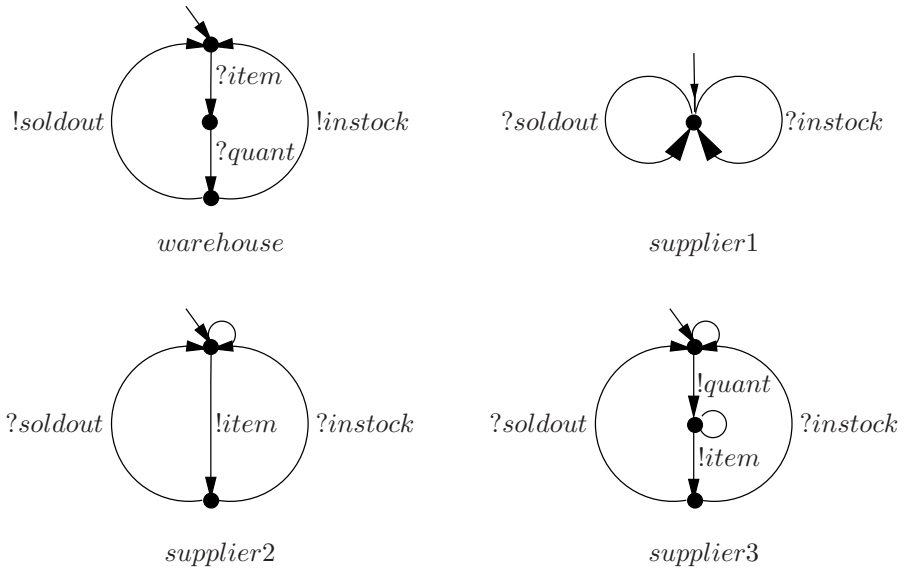


Fig. 4. Exemplifying **eco**

Example 4. To illustrate the **eco** implementation relation, a simple functionality of a *warehouse* component is given in Fig. 4 (top left). For a provided item and quantity the warehouse component reports either !*instock* or !*soldout*. A supplier component now may use this *warehouse* component to answer requests of customers. Such a supplier implementation must be input enabled for the outputs of the warehouse (!*instock* and !*soldout*). It communicates via its lower interface with the *warehouse*. The figure shows three supplier implementations; *supplier*1 never sends any message to the *warehouse*, it is just input enabled for the possible messages sent from the warehouse. This is fine, we have *supplier*1 **eco** *warehouse*, because **eco** does not demand a service requester to really interact with an environmental component. The only demand is that *if* there is communication with the warehouse, then this must be according to the *warehouse* specification.

Bottom left gives *supplier*2. To keep the figures clear, a non-labelled self-loop implicitly represents all input labels that are not explicitly specified, to make a system input-enabled. Here, the service implementer forgot to also inform the *warehouse* of the desired quantity, just the item is passed and then either an ?*instock* or ?*soldout* is expected. What will happen is that *supplier*2 will not get any answer from the *warehouse* after having sent the !*item* message since the *warehouse* waits for the ?*quant* message – both wait in vain. In other words, *supplier*2 observes quiescence of the *warehouse*. The *warehouse* does not observe anything since quiescence is not an observation in **eco**. Thus, also here we have *supplier*2 **eco** *warehouse*, since this supplier does never sent a wrong message to the warehouse. That the *intended transaction* (requesting the

warehouse for an item and quantity, and receiving an answer) is not completed does not matter here; see also Sect. 6: Limitations of **eco**.

Finally, in *supplier*3 the implementer confused the order of the messages to be sent to the *warehouse*: instead of sending the ?*item* first and then the ?*quant* it does it in the reverse order. Here the specification is violated since we have $uit(\, supplier3 \,\mathbf{after}\, \epsilon\,) = \{!quant\}$ and $\mathbf{in}(\, warehouse \,\mathbf{after}\, \epsilon\,) = \{?item\}$. Hence we get $!quant \notin \{?item\}$, and we have $supplier3 \;\mathbf{ec\not{o}}\; warehouse$.

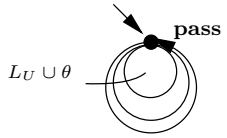
5 Test Generation

Having a notion of correctness with respect to an environmental component, as expressed by the environmental conformance relation **eco**, our next step is to generate test cases for testing implementations according to this relation. Whereas for **ioco** (and **uioco**) test cases are derived from a specification of the implementation under test, test cases for **eco** are not derived from a specification of the implementation but from a specification of the environment of the implementation. The test cases generated from this environment e should check for **eco**-conformance, i.e., they should check after all *Utraces* σ of the environment, whether all outputs produced by the implementation i – $uit(\, i \,\mathbf{after}\, \sigma\,)$ – are included in the set of inputs – $in(\, e \,\mathbf{after}\, \sigma\,)$ – of e.

Algorithm 1 (eco test generation). *Let $e \in \mathcal{LTS}(L_U, L_I)$ be an environmental specification, and let E be a subset of states of e, such that initially $E = e \,\mathbf{after}\, \epsilon$.*

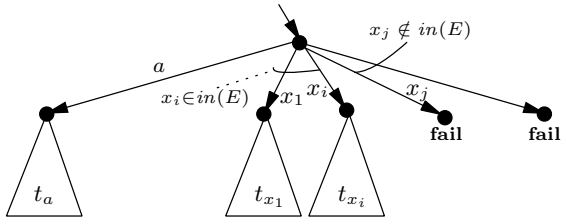
A test case $t \in \mathcal{TTS}(L_U, L_I)$ is obtained from a non-empty set of states E by a finite number of recursive applications of one of the following three nondeterministic choices:

1.



$t := \mathbf{pass}$

2.

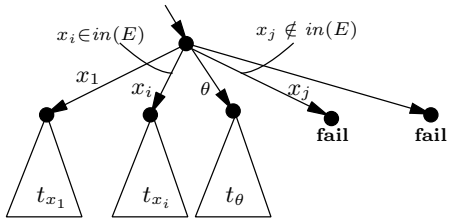


$t := \quad a \;;\; t_a$
$\quad \Box\; \Sigma\; \{\; x_j \;;\; \mathbf{fail} \mid x_j \in L_U,\; x_j \notin in(E)\;\}$
$\quad \Box\; \Sigma\; \{\; x_i \;;\; t_{x_i} \mid x_i \in L_U,\; x_i \in in(E)\;\}$

where $a \in L_I$ is an output of e, such that $E \,\mathbf{after}\, a \neq \emptyset$, t_a is obtained by recursively applying the algorithm for the set of states $E \,\mathbf{after}\, a$, and for

*each $x_i \in in(E)$, t_{x_i} is obtained by recursively applying the algorithm for the set of states E **after** x_i .*

3.



$$t := \quad \Sigma \{\ x_j\ ;\ \mathbf{fail} \mid x_j \in L_U,\ x_j \notin in(E)\ \}$$
$$\Box\ \Sigma \{\ x_i\ ;\ t_{x_i} \mid x_i \in L_U,\ x_i \in in(E)\ \}$$
$$\Box\ \theta\ ;\ t_\theta$$

*where for each $x_i \in in(E)$, t_{x_i} is obtained by recursively applying the algorithm for the set of states E **after** x_i , and t_θ is obtained by repeating the algorithm for E.*

Algorithm 1 generates a test case from a set of states E. This set represents the set of all possible states in which the environment can be at the given stage of the test case generation process. Initially, this is the set e **after** $\epsilon = e_0$ **after** ϵ, where e_0 is the initial state of e. Then the test case is built step by step. In each step there are three ways to make a test case:

1. The first choice is the single-state test case **pass**, which is always a sound test case. It stops the recursion in the algorithm, and thus terminates the test case.
2. In the second choice test case t attempts to supply input a to the implementation, which is an output of the environment. Subsequently, the test case behaves as t_a. Test case t_a is obtained by recursive application of the algorithm for the set E **after** a, which is the set of environment states that can be reached via an a-transition from some current state in E. Moreover, t is prepared to accept, as an input, any output x_i of the implementation, that might occur before a has been supplied. Analogous to t_a, each t_{x_i} is obtained from E **after** x_i, at least if x_i is allowed, i.e., $x_i \in in(E)$.
3. The third choice consists of checking the output of the implementation. Only outputs that are specified inputs in $in(E)$ of the environment are allowed; other outputs immediately lead to **fail**. In this case the test case does not attempt to supply an input; it waits until an output arrives, and if no output arrives it observes quiescence, which is always a correct response, since **eco** does not require to test for quiescence.

Now we can state one of our main results: Algorithm 1 is sound and exhaustive, i.e., the generated test cases only fail with non-**eco**-conforming implementations, and the test suite consisting of all test cases that can be generated detects all non-**eco**-conforming implementations.

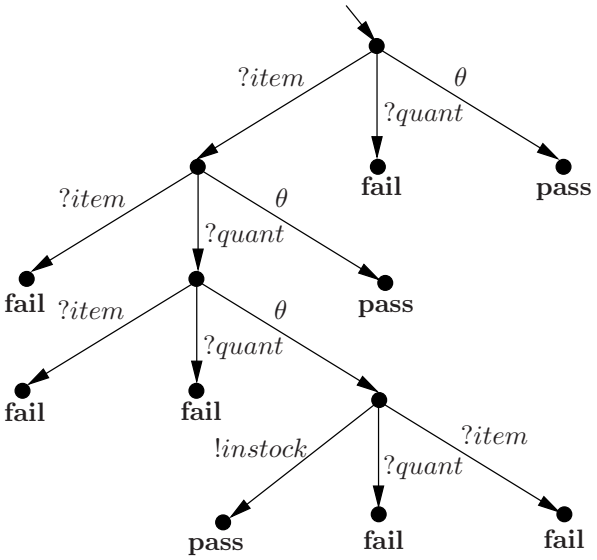


Fig. 5. An **eco** test case derived from the *warehouse* specification

Theorem 2. *Let* $i \in \mathcal{IOTS}(L_I, L_U)$, $e \in \mathcal{LTS}(L_U, L_I)$, *and let* $T_e \subseteq TTS(L_U, L_I)$ *be the set of all test cases that can be generated from* e *with Algorithm 1, then we have*

1. *Soundness:* i **eco** e implies $\forall t \in T_e :$ i **passes** t
2. *Exhaustiveness:* i **e$\not$co** e implies $\exists t \in T_e :$ i **fails** t

Example 5. We continue with Example 4 and Fig. 4, and give an **eco**-test case derived by Algorithm 1 from the *warehouse* specification; see Fig. 5. At the beginning, no input can be applied to the implementation since no outputs are specified in the initial state of the *warehouse*. Note that for the *warehouse* we have inputs $L_U = \{?item, ?quant\}$ and outputs $L_I = \{!instock, !soldout\}$, and that an output from the warehouse is an input to the implementation. First, the third option of the algorithm is chosen (checking the outputs of the implementation). Because ?*quant* is not initially allowed by the *warehouse* this leads to a **fail**. Observing quiescence (θ) is always allowed, and the test case is chosen to stop afterwards via a **pass** (first option). After observing ?*item* the test case continues by again observing the implementation outputs. Because ?*item* is not allowed anymore, since $?item \notin \mathbf{in}(\, warehouse \,\mathbf{after}\, ?item\,)$, this leads here to a **fail**. After ?*quant* is observed, again the third option is chosen to observe outputs. Now only quiescence is allowed since there is no implementation output specified in the set $\mathbf{in}(\, warehouse \,\mathbf{after}\, ?item \cdot ?quant\,)$. Finally, the second option is chosen (applying an input to the implementation). Both !*instock* and !*soldout* are possible here. The input !*instock* to i is chosen, and then the test case is chosen to end with **pass**.

6 Combining Upper and Lower Interface Testing

For testing the behavior at the lower interface of a component implementation i, we proposed the implementation relation **eco** in Sect. 4 with corresponding test generation algorithm in Sect. 5. For testing the behavior at the upper interface we proposed in Sect. 3 to use one of the existing implementation relations **uioco** or **ioco** with one of the corresponding test generation algorithms; see Sect. 2. Since **uioco** is based on *Utraces* like **eco**, whereas **ioco** uses *Straces*, it seems more natural and consistent to choose **uioco** here. Some further differences between **uioco** and **ioco** were discussed in Examples 2 and 3; more can be found in [3].

Now we continue towards testing the whole component implementation in the "horse-shoe" test architecture of Fig. 3(d). This involves concurrently testing for **uioco**-conformance to the provided service specification s at the upper interface, and for **eco**-conformance to the environment specification e at the lower interface. Here, we only indicate some principles and ideas by means of an example, and by discussing the limitations of **eco**. A more systematic treatment is left for further research.

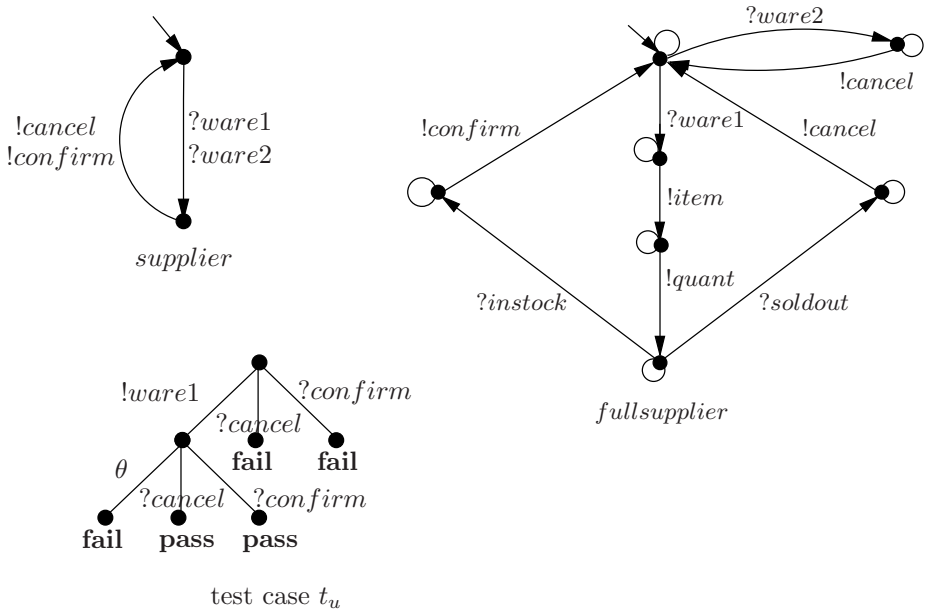


Fig. 6. An upper supplier specification, an implementation covering both interfaces, and an upper-interface test case

The specifications of the upper and lower interfaces are more or less independent, and can be considered as acting in a kind of interleaving manner (it is "a kind of" interleaving because s specifies i directly, and e specifies the environment of i, which implies that it does not make sense to just put $s \,|||\, e$, using the

parallel operator ||| of Sect. 2). This independence also holds for the derived test cases: we can generate the upper and lower test cases independently from s and e, respectively, after which they can be combined in a kind of interleaving manner. We will show this in Example 6.

Example 6. Fig. 6 shows in *supplier* a specification of the upper interface of a supplier component. This supplier can handle two different warehouses, and requests whether items are in stock. This is done by indicating the favored warehouse via a $?ware1$ or $?ware2$ message at the upper interface. The supplier is supposed to query the indicated warehouse and either return $!confirm$ if the item is in stock, or $!cancel$ otherwise. We abstract from modeling the specific items since this does not add here.

A supplier implementation called $fullsupplier$ is given at the right-hand side of the figure. This supplier is connected at its lower interface with a warehouse component as being specified in Fig. 4. Its label sets are: $L_I^{\uparrow} = \{?ware1, ?ware2\}$, $L_U^{\uparrow} = \{!confirm, !cancel\}$, $L_I^{\downarrow} = \{?instock, ?soldout\}$, $L_U^{\downarrow} = \{!item, !quant\}$.

Here we deal with both the upper and the lower interface, therefore the $fullsupplier$ must be input enabled for both input sets $L_I^{\uparrow}$ and $L_I^{\downarrow}$. For some reason this supplier cannot deal with a second warehouse, that is why it always reports $!cancel$ when being invoked via $?ware2$. For $?ware1$ it contacts the *warehouse* component, and behaves as assumed.

Fig. 6 also shows a **uioco**-test case t_u for the upper interface; Fig. 5 specified an **eco**-test case for the lower interface. Now we can test the $fullsupplier$ in the horse-shoe test architecture by executing both test cases concurrently, in an interleaved manner. Fig. 7 shows the initial part of such a test case. After $!ware1 \cdot ?item \cdot ?quant$ it can be continued with lower-interface input $!instock$ after which either $?confirm$ or $?cancel$ shall be observed by the test case. We deliberately did not complete this test case as a formal structure in Fig. 7, since there are still a couple of open questions, in particular, how to combine quiescence observations in an "interleaved" manner: is there one global quiescence for both interfaces, or does each interface have its own local quiescence? Analogous questions occur for **mioco**, which is a variant of **ioco** for multiple channels [7].

Limitations of eco. It is important to note that we are talking here only about local conformance at the upper and lower interfaces, and not about complete correctness of the component implementation i. The latter is not possible, simply, because we do not have a complete specification for i. In particular, as was already mentioned in the introduction, the dependencies between actions occurring at the upper and lower interfaces are not included in our partial specifications s and e. And where there is no (formal) specification of required behavior there will also be no test to check that behavior.

For instance, in Example 6 the $fullsupplier$ relates the $ware1$ input at its upper interface with a query at the warehouse component at its lower interface. This relation is invisible to **eco** and **uioco**/**ioco**. In other words, it is not possible to test requirements like *"the supplier must contact a specific warehouse component when, and only when being invoked with message ?ware1"*. In Fig. 7,

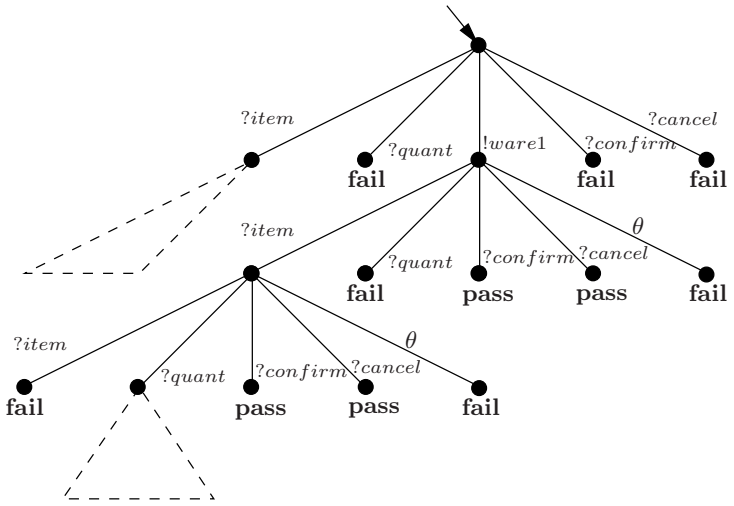


Fig. 7. Initial part of a combined test case

this is reflected in the sequence of actions $!ware1 \cdot ?confirm$, which necessarily leads to **pass**; there is no way to guarantee that the *warehouse* was really queried. In general, requirements like *"the supplier queries the right warehouse with the right product"* are not testable when only independent specifications of both interfaces are available.

Another noteworthy feature of **eco** is that quiescence cannot be observed by environmental components for several theoretical and practical reasons. For instance, it is not straightforward anymore to indirectly measure quiescence via timeouts here. This again means that a component can always choose to stop communicating with an environmental component. This is not always the desired behavior, since usually a chain of exchanged messages corresponds to a *transaction* that should be entirely performed. For instance, the *warehouse* from Example 4 only gives an answer ($!instock$ or $!soldout$) when being queried with first $?item$ and then $?quant$. Hence, the transaction that the *warehouse* offers, is *"send first an item followed by a quantity, and then the availability is returned"*. To enforce such transactions, the environmental component must be able to observe quiescence at certain steps within the transaction. For instance, after the reception of $?item$ the requesting service must not be quiescent, it must send $?quant$. Future research might allow to define a transaction-specific notion of quiescence which allows to test also for transactional behavior.

7 Conclusions

When testing a component, standard testing approaches only take the provided interface into account. This is due to the fact that usually only a specification of that interface is available. How the component interacts with environmental

components at its lower interface is not part of the test interest. By so doing, it is not possible to test if a component obeys the specifications of its environment. This is particularly problematic when this misbehavior at the lower interface does not imply an erroneous behavior at the upper interface.

We have introduced a new conformance relation called **eco** which allows to test the lower interface based on specifications of the provided interfaces of the environment. Together with the sound and exhaustive test generation algorithm, this allows to detect such malpractice.

Another important aspect is that a tester for **eco** can be automatically generated from the provided service specifications. In other words, it is possible to generate fully automatic replacements of components which behave according to their specification. This is very useful when implementations of such components are not yet available, or if for reasons like security or safety, a simulated replacement is preferred. The *audition* framework for testing Web Services [2] is currently instantiated with a test engine which combines symbolic versions of the **ioco** [5] and **eco** techniques to allow for sophisticated testing of Service Oriented Architectures.

Modeling and testing components which interact with their environment is not a trivial extension of the standard testing theories like **ioco** for reactive systems. In this paper we pursued the most simple and straightforward path to gain a testing theory which allows for basic testing of both the upper and the lower interface of a component. Though, there are still open questions on how to fully combine **eco** with, for instance, **uioco** or **ioco** on the level of combined specifications and test generation. This should lead to a notion of correctness at the upper interface which takes the lower interface into account. For instance, a deadlock at the lower interface (waiting for a message from an environmental component which never comes) does propagate to quiescence at the upper interface. Also, enriching the lower interface with the ability to observe quiescence of the environment is conceivable.

Finally, important concepts for components are reusability and substitutability. On a theoretical level these correspond to the notions of (pre-)congruences. It was already shown in [3] that without additional restrictions **ioco** is not a precongruence, yet for component based development it is desirable that such properties do hold. More investigations are necessary in this respect, e.g., inspired by the theory of interface automata [1] were such notions like congruence, replaceability, and refinement are the starting point.

Acknowledgments

The inspiration for this work came from two practical projects where model-based testing of components is investigated. In the first, the Tangram project (`www.esi.nl/tangram`), software components of *wafer scanners* developed by ASML (`www.asml.com`) are tested. It turned out to be difficult to make complete models of components with sufficient detail to perform model-based testing. In the second, the Plastic project (EU FP6 IST STREP grant number 26955;

www.ist-plastic.org), one of the research foci is the further elaboration of the *audition* framework with advanced testing methods for services in ubiquitous networking environments.

Lars Frantzen is supported by the EU Marie Curie Network TAROT (MRTN-CT-2004-505121), and by the Netherlands Organization for Scientific Research (NWO) under project STRESS – Systematic Testing of Realtime Embedded Software Systems.

Jan Tretmans carried out this work as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. TANGRAM is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

References

1. de Alfaro, L., Henzinger, T.A.: Interface Automata. SIGSOFT Softw. Eng. Notes 26(5), 109–120 (2001)
2. Bertolino, A., Frantzen, L., Polini, A., Tretmans, J.: Audition of Web Services for Testing Conformance to Open Specified Protocols. In: Reussner, R., Stafford, J.A., Szyperski, C.A. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 1–25. Springer, Heidelberg (2006)
3. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional Testing with IOCO. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)
4. De Nicola, R.: Extensional Equivalences for Transition Systems. Theoretical Computer Science 24, 211–237 (1987)
5. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) Formal Approaches to Software Testing and Runtime Verification. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
6. Gross, H.-G.: Component-Based Software Testing with UML. Springer, Heidelberg (2004)
7. Heerink, L.: Ins and Outs in Refusal Testing. PhD thesis, University of Twente, Enschede, The Netherlands (1998)
8. Jard, C., Jéron, T.: TGV: Theory, Principles and Algorithms. Software Tools for Technology Transfer 7(4), 297–315 (2005)
9. Rehman, M.J., Jabeen, F., Bertolino, A., Polini, A.: Testing Software Components for Integration: A Survey of Issues and Techniques. In: Software Testing Verification and Reliability, John Wiley & Sons, Chichester (2007)
10. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software—Concepts and Tools 17(3), 103–120 (1996)
11. Tretmans, J.: Model Based Testing with Labelled Transition Systems. Technical Report ICIS–R6037, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands (2006)
12. de Vries, R.G., Tretmans, J.: On-the-Fly Conformance Testing using SPIN. Software Tools for Technology Transfer 2(4), 382–393 (2000)