ELSEVIER Computer Networks and ISDN Systems 29 (1996) 49-79

COMPUTER NETWORKS and ISDN SYSTEMS

Conformance testing with labelled transition systems: Implementation relations and test generation

Jan Tretmans [1]

Tele-Informatics and Open Systems Group, Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

Abstract

This paper studies testing based on labelled transition systems, presenting two test generation algorithms with their corresponding implementation relations. The first algorithm assumes that implementations communicate with their environment via symmetric, synchronous interactions. It is based on the theory of testing equivalence and preorder, as is most of the testing theory for labelled transition systems, and it is found in the literature in some slightly different variations. The second algorithm is based on the assumption that implementations communicate with their environment via inputs and outputs. Such implementations are formalized by restricting the class of labelled transition systems to those systems that can always accept input actions. For these implementations a testing theory is developed, analogous to the theory of testing equivalence and preorder. It consists of implementation relations formalizing the notion of conformance of these implementations with respect to labelled transition system specifications, test cases and test suites, test execution, the notion of passing a test suite, and the test generation algorithm, which is proved to produce sound test suites for one of the implementation relations.

Keywords: Communication protocols; Formal description techniques; Transition systems; Conformance; Conformance testing; Test case generation

1. Introduction

Protocol conformance testing involves testing of a protocol implementation with respect to its specification. The aim is to increase the level of confidence in the correct functioning of the implementation as prescribed by the specification, and to contribute in this way to successful communication between computer systems.

With the increasing use of formal methods for specifying the required behaviour it is necessary to consider conformance testing of protocol implementations with respect to such specifications. Apart from this necessity, the use of formal methods in conformance testing has its advantages, such as the precise, formal definition of conformance and conformance testing concepts, the algorithmic, tool supported generation of test suites from formal specifications, and the possibility of formally verifying the correctness of a test case with respect to a specification. These possible advantages have led to a lot of research in the area of formal conformance testing, leading to several methods for the algorithmic generation of tests for different specification formalisms. To put

[1] Email: tretmans@cs.utwente.nl.

these different methods in a general context, and to define the basic concepts of conformance testing in an abstract way, also frameworks have been studied, among others within the standardization community in the project "Formal Methods in Conformance Testing" (ISO/IEC JTC 1/SC 21 Project 54, ITU T Q.8/10). The scope of this standardization activity is to define a general methodology on how to perform conformance testing of a protocol implementation given a formal specification of a protocol standard [31].

One of the specification formalisms studied in the realm of formal conformance testing is that of labelled transition systems. A labelled transition system is a structure consisting of states with transitions, labelled with actions, between them. The formalism of labelled transition systems can be used for modelling the behaviour of processes, and it serves as a semantic model for various formal specification languages, e.g., CCS [38,39], CSP [27], ACP [9], and LOTOS [6,30]. Also a large part of the semantics of languages like Estelle [29] and SDL [17] can be expressed in labelled transition systems. Testing theory and algorithms for the generation of tests from labelled transition system specifications have been developed during the last decade, e.g., [1,11,13,19,21,20,24,25,33,45,42,47,54,49]. All these methods, as most of the theory on labelled transition systems, are based on synchronous, symmetric communication between different processes: communication between two processes occurs if both processes offer to interact on a particular action, and if the interaction takes place it occurs synchronously in both participating processes, without a notion of distinction between input and output actions. For testing theories a particular case where such communication occurs, is the modelling of the interaction between a tester and an implementation under test during test execution. We will refer to above theories as testing with symmetric interactions.

This paper will present two algorithms for the generation of tests from a labelled transition system specification. The presentations will be in the vein of the framework in [31]: they involve the definition of a model to describe implementations, the definition of an implementation relation that formalizes the notion of conformance of an implementation with respect to a specification, the description of test cases, test suites, and how to pass a test suite, and finally the development of the test generation algorithm that produces provably correct test cases.

The first algorithm is based on symmetric interactions as explained above, and it can be found in the literature in some slightly different variations [10,11,19,42,54,49].

The second algorithm approaches communication in a different manner by distinguishing explicitly between the inputs and the outputs of a system. Outputs are actions that are initiated by, and under control of the system, while input actions are initiated by, and under control of the system's environment; a system can never refuse to perform its input actions. Communication takes place between inputs of the system and outputs of the environment, or the other way around. This implies that an interaction is not symmetric anymore with respect to the communicating processes. Many real-life implementations allow such a classification of their actions, communicating with their environment via inputs and outputs.

The next section introduces labelled transition systems as the formalism of our discourse. Section 3 gives some basic testing concepts for labelled transition systems, such as a test case, a test suite, a test run, and passing a test suite. The existing approaches to labelled transition system testing are presented in Section 4. A few implementation relations and a test generation algorithm are given, all based on symmetric interactions. Section 5 introduces input–output transition systems to model implementations that communicate via inputs and outputs. An input–output transition system is a special kind of labelled transition system with the restriction that inputs are always enabled. Implementation relations for input–output transition systems are studied in Section 6. Finally, a test generation algorithm that produces provably correct test cases to test input–output transition systems with respect to labelled transition system specifications for one of the implementation relations of Section 6 is developed in Section 7. In Section 8 some concluding remarks are given, among which a comparison of symmetric and input–output testing, and a brief comparison with other transition-system based models that distinguish between inputs and outputs, like input–output state machines [43], input/output automata [37], and queue contexts [51]. Complete proofs for some of the theorems are found in Appendix A.

2. Labelled transition systems

The formalism of labelled transition systems is used as the basis for describing the behaviour of processes, such as specifications, implementations, and tests.

Definition 2.1. A *labelled transition system* is a 4-tuple $\langle S, L, T, s_0 \rangle$ where
- S is a countable, non-empty set of *states*;
- L is a countable set of *labels*;
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the *transition relation*;
- $s_0 \in S$ is the *initial state*.

The labels in L represent the observable interactions of a system; the special label $\tau \notin L$ represents an unobservable, internal action. We denote the class of all labelled transition systems over L by $\mathcal{LTS}(L)$. For technical reasons we restrict $\mathcal{LTS}(L)$ to labelled transition systems that are strongly converging, i.e., ones that do not have infinite compositions of transitions with internal actions.

A *trace* is a finite sequence of observable actions. The set of all traces over L is denoted by L^*, with ε denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. With $|\sigma|$ the length of trace σ is denoted, i.e., the (finite) number of occurrences of actions in σ. Some additional notations and properties are introduced in Definitions 2.2 and 2.3.

Definition 2.2. Let $p = \langle S, L, T, s_0 \rangle$ be a labelled transition system with $s, s' \in S$, and let $\mu_{(i)} \in L \cup \{\tau\}$, $a_{(i)} \in L$, and $\sigma \in L^*$.

$$\begin{array}{lll}
s \xrightarrow{\mu} s' & =_{def} & (s, \mu, s') \in T \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' & =_{def} & \exists s_0, \ldots, s_n : \ s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{def} & \exists s' : \ s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \not\xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{def} & \text{not } \exists s' : \ s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \overset{\varepsilon}{\Longrightarrow} s' & =_{def} & s = s' \ \text{ or } \ s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \overset{a}{\Longrightarrow} s' & =_{def} & \exists s_1, s_2 : \ s \overset{\varepsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \overset{\varepsilon}{\Longrightarrow} s' \\
s \overset{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} s' & =_{def} & \exists s_0 \ldots s_n : \ s = s_0 \overset{a_1}{\Longrightarrow} s_1 \overset{a_2}{\Longrightarrow} \cdots \overset{a_n}{\Longrightarrow} s_n = s' \\
s \overset{\sigma}{\Longrightarrow} & =_{def} & \exists s' : \ s \overset{\sigma}{\Longrightarrow} s' \\
s \overset{\sigma}{\not\Longrightarrow} & =_{def} & \text{not } \exists s' : s \overset{\sigma}{\Longrightarrow} s'
\end{array}$$

We will not always distinguish between a labelled transition system and its initial state: if $p = \langle S, L, T, s_0 \rangle$, then we will identify the process p with its initial state s_0, and we write, for example, $p \overset{\sigma}{\Longrightarrow}$ instead of $s_0 \overset{\sigma}{\Longrightarrow}$.

Definition 2.3.
(1) $\mathit{traces}(p) \ =_{def} \ \{ \sigma \in L^* \mid p \overset{\sigma}{\Longrightarrow} \}$
(2) $\mathit{init}(p) \ =_{def} \ \{ a \in L \mid p \overset{a}{\Longrightarrow} \}$
(3) p **after** σ $=_{def} \ \{ p' \mid p \overset{\sigma}{\Longrightarrow} p' \}$
(4) $\mathit{der}(p) \ =_{def} \ \{ p' \mid \exists \sigma \in L^* : \ p \overset{\sigma}{\Longrightarrow} p' \}$
(5) p has *finite behaviour* if there is an $n \in \mathbf{N}$, such that $\forall \sigma \in \mathit{traces}(p) : |\sigma| < n$.
(6) p is *finite-state* if $\mathit{der}(p)$ is finite.
(7) p is *deterministic* if for all $\sigma \in L^*$, p **after** σ has at most one element. If $\sigma \in \mathit{traces}(p)$, then we overload p **after** σ to denote this element.

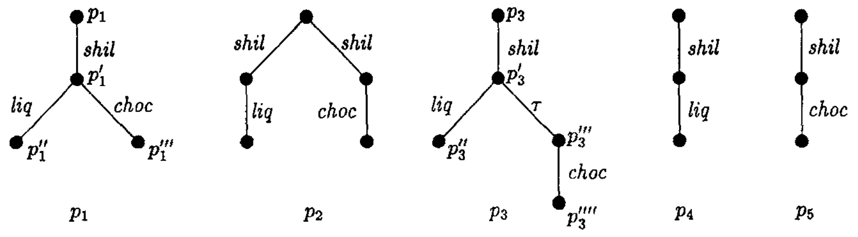

Fig. 1. Labelled transition systems.

We represent a labelled transition system by a tree or a graph, where nodes represent states and edges represent transitions, or in a process-algebraic manner by a *behaviour expression* (cf. LOTOS [6,30]).

Definition 2.4. A *behaviour expression* B is an expression with the following syntax:

$$B \;=_{def}\; \mathbf{stop} \mid a\,;B \mid \mathbf{i}\,;B \mid B \;\Box\; B \mid B \| B \mid \Sigma\, \mathcal{B}$$

where $a \in L$, and $\mathcal{B}$ is a countable set of behaviour expressions.

The operational semantics are given by the following axioms and inference rules, which define for each behaviour expression all its possible transitions (**stop** has no transitions):

$$\begin{array}{ll}
 & \vdash a;B \xrightarrow{a} B \\
 & \vdash \mathbf{i};B \xrightarrow{\tau} B \\
B_1 \xrightarrow{\mu} B_1',\ \mu \in L \cup \{\tau\} & \vdash B_1 \Box B_2 \xrightarrow{\mu} B_1' \\
B_2 \xrightarrow{\mu} B_2',\ \mu \in L \cup \{\tau\} & \vdash B_1 \Box B_2 \xrightarrow{\mu} B_2' \\
B_1 \xrightarrow{\tau} B_1' & \vdash B_1 \| B_2 \xrightarrow{\tau} B_1' \| B_2 \\
B_2 \xrightarrow{\tau} B_2' & \vdash B_1 \| B_2 \xrightarrow{\tau} B_1 \| B_2' \\
B_1 \xrightarrow{a} B_1',\ B_2 \xrightarrow{a} B_2',\ a \in L & \vdash B_1 \| B_2 \xrightarrow{a} B_1' \| B_2' \\
B \xrightarrow{\mu} B',\ B \in \mathcal{B},\ \mu \in L \cup \{\tau\} & \vdash \Sigma\, \mathcal{B} \xrightarrow{\mu} B'
\end{array}$$

Example 2.5. To illustrate the concepts of labelled transition systems we use very simple and intuitive candy machines. More complicate systems, e.g., communication protocols, can also be modelled as labelled transition systems, however, for the moment the complexity of such systems would divert the attention, while not being necessary to illustrate the main concepts of this paper.

Fig. 1 gives examples of candy machines over the labelset $L = \{shil, liq, choc\}$. The candy machines interact with their environment by insertions of *shillings*, and by supplying *liquorice* and *chocolate*. System p_3 models a machine that accepts a *shilling*, and then either it supplies *liquorice*, or it makes an internal transition to a state where it cannot supply *liquorice* anymore, but where it offers *chocolate*. A behaviour expression for p_3 is $shil; (liq; \mathbf{stop} \;\Box\; \mathbf{i}; choc; \mathbf{stop})$. For p_3 we have, for example,

$$p_3 \xRightarrow{shil} p_3''' \quad \text{and} \quad p_3''' \overset{liq}{\not\Rightarrow}.$$

3. Conformance testing for labelled transition systems

Starting point for conformance testing is a specification in some (formal) notation, and an implementation, that is, a device or program interacting with its environment, which is considered as a black box. Test cases are derived from the specification, and applied to the implementation, such that from the results of applying them it can be concluded whether the implementation conforms to the specification.

In this paper labelled transition systems, or any formal language with underlying semantics in terms of labelled transition systems, are considered as the formal notation for specifications. Implementations, being physical, real objects, are, in principle, not amenable to formal reasoning. We can only deal with implementations in a formal way, if we make the assumption that any real implementation has a formal model, with which we could reason formally. This formal model is only assumed to exist, but it is not known a priori. This assumption is referred to as the test hypothesis [8,31,50]. In Section 4 we will consider as the test hypothesis that also implementations could be described as labelled transition systems. In Sections 5, 6 and 7 a stronger test hypothesis will be put forward by assuming that implementations can be modelled by a subclass of labelled transition systems: the input-output transition systems. Thus the test hypothesis allows us to reason about implementations as if they were labelled transition systems, or input-output transition systems, respectively.

Having specifications and implementations the next important thing is to define what it means for an implementation to conform to a specification, otherwise no useful test can ever be generated. Conformance is defined by means of an implementation relation between the models of implementations and the specifications [5,31,50], in our case a relation $\mathbf{imp} \subseteq \mathcal{LTS}(L) \times \mathcal{LTS}(L)$: an implementation $i \in \mathcal{LTS}(L)$ conforms to specification $s \in \mathcal{LTS}(L)$ if and only if i $\mathbf{imp}$ s.

The next step is to consider test cases and test suites. A test case is a specification of the behaviour of a tester in an experiment to be carried out with an implementation under test. Such behaviour, like other behaviours, can be specified by a labelled transition system. An experiment should last for a finite time, so a test case should have finite behaviour. Moreover, a tester executing a test case would like to have as much control as possible over the testing process, so nondeterminism in a test case is undesirable. To be able to decide about the success of a test a verdict (**pass** or **fail**) is attached to each state of the test case.

Definition 3.1.

(1) A *test case* t is a 5-tuple $\langle S, L, T, \nu, s_0 \rangle$, such that $\langle S, L, T, s_0 \rangle$ is a deterministic labelled transition system with finite behaviour, and $\nu : S \rightarrow \{\mathbf{fail}, \mathbf{pass}\}$ is a *verdict function*.
The class of test cases over actions in L is denoted by $\mathcal{LTS}_t(L)$. Definitions applicable to $\mathcal{LTS}(L)$ are extended to $\mathcal{LTS}_t(L)$ by defining them over the underlying labelled transition system.

(2) A *test suite* T is a set of test cases: $T \in \mathcal{P}(\mathcal{LTS}_t(L))$, where $\mathcal{P}(\mathcal{LTS}_t(L))$ is the powerset of $\mathcal{LTS}_t(L)$, i.e., the set of all possible subsets of $\mathcal{LTS}_t(L)$.

Running a test case is modelled by the synchronous parallel execution of the test case with the implementation under test, which continues until no more interactions are possible, i.e., until a deadlock occurs. This deadlock may occur when the (finite) test case reaches a final state, or when the combination reaches a state where the actions proposed by the test case cannot be accepted by the implementation. An implementation passes a test run if and only if the verdict of the test case in the state where the deadlock is reached is **pass**. Since an implementation can behave nondeterministically different test runs of the same test case with the same implementation may lead to different final states, and hence to different verdicts. An implementation passes a test case if and only if all possible test runs lead to the verdict **pass**. This means that each test case must be executed several times in order to give a final verdict, theoretically even infinitely many times.

Definition 3.2.

(1) A *deadlock* of process $p \in \mathcal{LTS}(L)$ is a trace $\sigma \in L^*$, after which no more observable actions are possible:

$$p \ \mathbf{after}\ \sigma\ \mathbf{deadlocks} \quad =_{def} \quad \exists p' : \ p \overset{\sigma}{\Longrightarrow} p' \ \text{ and } \ init(p') = \emptyset$$

(2) A *test run* of a test case $t \in \mathcal{LTS}_t(L)$ with an implementation $i \in \mathcal{LTS}(L)$ is a trace of the synchronous

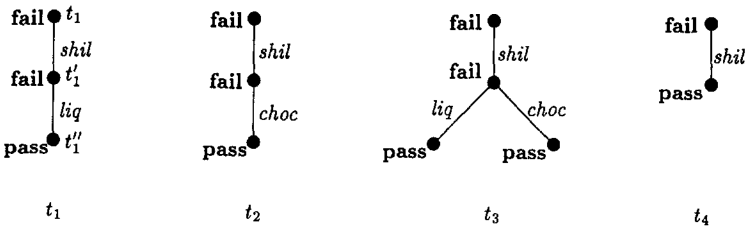

Fig. 2. Test cases.

parallel composition of t and i leading to deadlock:

$$\sigma \text{ is a test run of } t \text{ and } i \ =_{def}\ (t \,\|\, i) \textbf{ after } \sigma \textbf{ deadlocks}$$

(3) An implementation i *passes* a test case t, if all the test runs of t and i lead to a **pass**-state of t:

$$i \textbf{ passes } t \ =_{def}\ \forall \sigma \in L^* :\ t \,\|\, i \textbf{ after } \sigma \textbf{ deadlocks} \text{ implies } \nu(\, t \textbf{ after } \sigma\,) = \textbf{pass}$$

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

$$i \textbf{ passes } T \ =_{def}\ \forall t \in T :\ i \textbf{ passes } t$$

If an implementation does not pass a test suite, it fails: i **fails** T $=_{def}$ $\exists t \in T :\ i \not\textbf{passes}\ t$.

Example 3.3. Fig. 2 gives some test cases. The only test run of t_1 with p_1 is *shil·liq*:

$$t_1 \,\|\, p_1 \xRightarrow{shil\cdot liq} t_1'' \,\|\, p_1'' \quad \text{and} \quad \forall a \in L :\ t_1'' \,\|\, p_1'' \overset{a}{\not\Rightarrow}$$

Since $\nu(t_1'') = \textbf{pass}$, we have that p_1 **passes** t_1.

The test runs of t_1 with p_3 are $\{shil\cdot liq, shil\}$:

$$t_1 \,\|\, p_3 \xRightarrow{shil\cdot liq} t_1'' \,\|\, p_3'' \quad \text{and} \quad \forall a \in L :\ t_1'' \,\|\, p_3'' \overset{a}{\not\Rightarrow}$$
$$t_1 \,\|\, p_3 \xRightarrow{shil} t_1' \,\|\, p_3''' \quad \text{and} \quad \forall a \in L :\ t_1' \,\|\, p_3''' \overset{a}{\not\Rightarrow}$$

Since $\nu(t_1'') = \textbf{pass}$ and $\nu(t_1') = \textbf{fail}$, we have that p_3 **fails** t_1.

To obtain test suites test generation algorithms have to be developed, which, given a specification, generate a test suite. Formally, a test generation algorithm can be expressed as a function

$$gen_{\textbf{imp}} :\ \mathcal{LTS}(L) \longrightarrow \mathcal{P}(\mathcal{LTS}_t(L)) \tag{3.1}$$

A generated test suite $gen_{\textbf{imp}}(s)$ must test implementations for conformance with respect to s and **imp**. Ideally, an implementation should pass the test suite if and only if it is conforming. In this case the test suite is called *complete* [31]. Unfortunately, in almost all practical cases such a test suite would be infinitely large, hence for practical testing we have to restrict to test suites that can only detect non-conformance, but that cannot assure conformance. Such test suites are called *sound*. Test suites that can only assure conformance, but not non-conformance are called *exhaustive*.

Definition 3.4. Let s be a specification, **imp** an implementation relation, and T a test suite, then

$$\begin{array}{lllll} T \text{ is complete} & =_{def} & \forall i : i \ \mathbf{imp}\ s & \text{iff} & i \ \mathbf{passes}\ T \\ T \text{ is sound} & =_{def} & \forall i : i \ \mathbf{imp}\ s & \text{implies} & i \ \mathbf{passes}\ T \\ T \text{ is exhaustive} & =_{def} & \forall i : i \ \mathbf{imp}\ s & \text{if} & i \ \mathbf{passes}\ T \end{array}$$

4. Conformance testing based on symmetric interactions

This section presents some implementation relations for labelled transition systems, which can be found in the literature, together with a sound test generation method for one of these relations: the relation **conf**. These relations and the corresponding testing method use the test hypothesis that implementations can be modelled as labelled transition systems. Communication between a labelled transition system and its environment is modelled by the synchronized parallel composition $\|$ (Definition 2.4), where the communication is symmetric: if a system wishes to communicate with its environment it proposes some actions on which it is prepared to interact. The environment also proposes some actions, and then they interact on one of the actions that they both propose. The role of both communicating processes is the same and symmetric, and all actions (except for the internal action τ) are treated in the same way.

Implementation relations

Many different possibilities for implementation relations on $\mathcal{LTS}(L)$ have been studied, e.g., observation equivalence [38], strong bisimulation equivalence and weak bisimulation equivalence [41,39], failure equivalence and preorder [27], testing equivalence and preorder [21], failure trace equivalence and preorder [16], generalized failure equivalence and preorder [32], and many others [25,26]. A straightforward example, based on the definitions of Section 2, is *trace preorder* $\leq_{tr}$, which requires inclusion of trace sets. The intuition behind this relation is that an implementation $i \in \mathcal{LTS}(L)$ may show only behaviour, in terms of traces of observable actions, which is specified in the specification $s \in \mathcal{LTS}(L)$.

Definition 4.1. Let $i, s \in \mathcal{LTS}(L)$, then $i \leq_{tr} s \ =_{def} \ traces(i) \subseteq traces(s)$

Example 4.2. Consider Fig. 1: $p_1 \leq_{tr} p_2$ and $p_2 \leq_{tr} p_1$, since $traces(p_1) = traces(p_2) = \{\varepsilon, shil, shil{\cdot}liq, shil{\cdot}choc\}$. Also $p_4 \leq_{tr} p_1$, but $p_1 \not\leq_{tr} p_4$.

Considering Example 4.2 we have, for example, $p_2 \leq_{tr} p_1$, which is interpreted as 'implementation p_2 correctly implements specification p_1 with respect to trace preorder'. However, p_1 specifies that after inserting a *shilling* the user has a choice between *liquorice* and *chocolate*, while p_2 may refuse to supply one of these sweets: after inserting a *shilling* the machine makes the nondeterministic choice between offering *liquorice* or *chocolate*. Suppose the machine chooses to offer *liquorice*, and the user makes a choice for *chocolate*, then a *deadlock* occurs: no further interaction is possible between the machine offering the interaction *liquorice* and the user willing to interact on *chocolate*. A user faced with p_2 as an implementation of p_1 will certainly be disappointed.

The reason for the disappointment is that trace preorder $\leq_{tr}$ only considers sequences of observable actions; it does not care about who is going to resolve choices in the behaviour: the machine internally or the external environment. A more sophisticated, and stronger implementation relation is *testing preorder* [20]. In addition to requiring that the traces observed with the implementation are contained in those observed with the specification, testing preorder requires that any possible user encountering a deadlock with the implementation will experience the same deadlock when interacting with the specification. This idea for an implementation relation is formalized by modelling the observing users themselves as labelled transition systems, and by modelling the observation of deadlock as a trace leading to a combined state of the machine and the user from which no further interactions are possible (see Definition 3.2). One could say that testing preorder is the relation on labelled transition systems, where any discrepancy of the implementation from the specification can exactly be observed by another labelled transition system. As such, testing preorder is an important implementation relation in this paper from which other relations will be derived.

Definition 4.3.

(1) The sets of *observations*, *obs* and *obs'* respectively, that an observer $u \in \mathcal{LTS}(L)$ can make of process $p \in \mathcal{LTS}(L)$ are given by the deadlocks, respectively the traces of the synchronized parallel communication of u and p:

$$\begin{aligned} obs(u,p) &=_{def} \{ \sigma \in L^* \mid (u \| p) \textbf{ after } \sigma \textbf{ deadlocks} \} \\ obs'(u,p) &=_{def} \{ \sigma \in L^* \mid u \| p \overset{\sigma}{\Longrightarrow} \} \end{aligned}$$

(2) Implementation $i \in \mathcal{LTS}(L)$ is in *testing preorder* with specification $s \in \mathcal{LTS}(L)$, if for all possible observers the observations made with i are included in those of s:

$$i \leq_{te} s \; =_{def} \; \forall u \in \mathcal{LTS}(L) : \; obs(u,i) \subseteq obs(u,s) \text{ and } obs'(u,i) \subseteq obs'(u,s)$$

The definition of $\leq_{te}$ in Definition 4.3 is extensional, i.e., in terms of how the environment (i.c. the observer u) perceives a system. This definition can be rewritten into an intensional characterization, i.e., a characterization in terms of properties of the labelled transition systems themselves. This characterization, given in terms of failure pairs (Proposition 4.5) coincides with failure preorder on our class of strongly-converging transition systems (for a proof see, for example, [20,49]).

Definition 4.4. Let $p \in \mathcal{LTS}(L)$, $\sigma \in L^*$, and $A \subseteq L$, then

$$p \textbf{ after } \sigma \textbf{ refuses } A \; =_{def} \; \exists p' : \; p \overset{\sigma}{\Longrightarrow} p' \text{ and } \forall a \in A : \; p' \overset{a}{\not\Longrightarrow}$$

Proposition 4.5. $i \leq_{te} s$ *iff* ($\forall \sigma \in L^*, \forall A \subseteq L :$ i **after** σ **refuses** A *implies* s **after** σ **refuses** A)

Example 4.6. Consider again Fig. 1. We have $p_1 \leq_{te} p_2$: there is no σ, A, such that p_1 **after** σ **refuses** A and not (p_2 **after** σ **refuses** A).

But $p_2 \not\leq_{te} p_1$, since p_2 **after** *shil* **refuses** $\{liq\}$ and not (p_1 **after** *shil* **refuses** $\{liq\}$). Also $p_4 \not\leq_{te} p_3$, because p_4 **after** *shil* **refuses** $\{choc\}$, which does not hold for p_3, but $p_5 \leq_{te} p_3$, and also $p_1 \leq_{te} p_3$.

The relation $\leq_{te}$ does not allow extra traces in the implementation:
$p_1 \not\leq_{te} p_4$, since p_1 **after** *shil·choc* **refuses** $\emptyset$, and not (p_4 **after** *shil·choc* **refuses** $\emptyset$).

An implementation relation that is strongly related to $\leq_{te}$ is the relation **conf** [13,11]. It is a modification of $\leq_{te}$ by restricting all observations to only those traces that are contained in the specification s. This restriction makes testing a lot easier: only traces of the specification have to be considered, not the huge complement of this set, i.e., the traces not explicitly specified. Saying it in other words, **conf** requires that an implementation does what it should do, not that it does not do what it is not allowed to do.

Definition 4.7. i **conf** $s \; =_{def} \; \forall u \in \mathcal{LTS}(L) : (\; obs(u,i) \cap traces(s) \;) \subseteq obs(u,s)$
$\qquad\qquad\qquad\qquad\qquad\qquad$ and $(\; obs'(u,i) \cap traces(s) \;) \subseteq obs'(u,s)$

Proposition 4.8.

i **conf** s *iff* ($\forall \sigma \in traces(s), \forall A \subseteq L :$ i **after** σ **refuses** A *implies* s **after** σ **refuses** A)

We conclude this part with relating the different implementation relations.

Proposition 4.9.
(1) $\leq_{tr}$ *and* $\leq_{te}$ *are preorders;* **conf** *is reflexive, but not transitive.*
(2) $\leq_{te} = \leq_{tr} \cap$ **conf**

Example 4.10. Consider again Fig. 1. From Proposition 4.9(2) follows that $i \leq_{te} s$ implies i **conf** s, hence all systems that are related by $\leq_{te}$ (see Example 4.6) are also related by **conf**.

Consider the ones not related by $\leq_{te}$ in Example 4.6: p_2 $\not\text{\bf conf}$ p_1, since p_2 **after** *shil* **refuses** $\{liq\}$, not (p_1 **after** *shil* **refuses** $\{liq\}$) and $shil \in traces(p_1)$. Analogously p_4 $\not\text{\bf conf}$ p_3.

But the relation **conf** does allow extra traces in the implementation: p_1 **conf** p_4, although p_1 **after** *shil·choc* **refuses** $\emptyset$ and not (p_4 **after** *shil·choc* **refuses** $\emptyset$), but $shil{\cdot}choc \notin traces(p_4)$.

Test generation

Now that we have defined some implementation relations the next step is to develop test generation algorithms. We will give here an algorithm for the derivation of sound test cases for **conf** (Definition 4.7) [48,49]. Test derivation for the relation **conf** has been studied a lot, especially in the context of protocol testing [4,10,11,13,19,23,33,42,54]. Testing scenarios for other relations can, for example, be found in [1] (bisimulation equivalence), [18,35] (probabilistic testing), [20] (testing equivalence), [24] (testing preorder), [47] (trace and failure equivalence using techniques from Finite State Machine testing [15]), [25,26] (comparison of several testing scenarios), [32] (failure trace preorder), [45] (refusal testing), and [53] (queue preorder).

A sound test generation algorithm for **conf** is an algorithm that takes a specification $s \in \mathcal{LTS}(L)$, and returns a test suite $gen_{\text{conf}}(s) \subseteq \mathcal{LTS}_t(L)$, such that (Definitions 3.2 and 3.4):

$$i \textbf{ conf } s \text{ implies } \forall t \in gen_{\text{conf}}(s),\ \forall \sigma \in L^* :$$
$$t \,\|\, i \textbf{ after } \sigma \textbf{ deadlocks} \text{ implies } \nu(\, t \textbf{ after } \sigma\,) = \textbf{pass} \tag{4.1}$$

The following nondeterministic, recursive algorithm from [49] satisfies (4.1). The nondeterminism in the algorithm is a result of the freedom to choose any set of actions, $A \subseteq init(s)$ in Algorithm 1, having the given properties. Each possible choice for this set will result in another test case, but all test cases generated in this way are guaranteed to be sound (Theorem 4.11). Moreover, the test suite that consists of all test cases that can be generated in this way, is exhaustive, and thus complete, although not very efficient. For complete proofs and optimizations with respect to efficiency we refer to [49]. Analogous algorithms, following more or less the same ideas, can be found in [10,11,19,54].

Algorithm 1. Let $s \in \mathcal{LTS}(L)$, then a test case t for s is

$$t := \Sigma \{\, a\, ; t_a \mid a \in A \,\}$$

where, with $C_s := \{init(s') \mid s \overset{\varepsilon}{\Longrightarrow} s'\}$, the set $A \subseteq init(s)$ and the verdict $\nu(t)$ shall satisfy

$$\begin{array}{ll} & \forall C \in C_s :\ A \cap C \neq \emptyset \text{ and } \nu(t) = \textbf{fail} \\ \text{or} & \emptyset \in C_s \text{ and } A = init(s) \text{ and } \nu(t) = \textbf{pass} \\ \text{or} & A = \emptyset \text{ and } \nu(t) = \textbf{pass} \end{array}$$

and t_a is a test case for $\Sigma\ \{\, \mathbf{i}\, ; s' \mid s \overset{a}{\Longrightarrow} s' \,\}$, which is obtained by recursively applying the algorithm.

Theorem 4.11. *Any test case obtained from a specification s with Algorithm 1 is sound with respect to* **conf**, *and the set of all possible test cases which can be obtained using Algorithm 1, is exhaustive (and thus complete).*

Proof (*Sketch*). From the contraposition of the property in Proposition 4.8 it follows that i must be tested for all combinations of σ and A such that not (s **after** σ **refuses** A), i.e., s after σ can always continue with at least one action of A. These are exactly the sets A in Algorithm 1 satisfying $\forall C \in C_s : A \cap C \neq \emptyset$. To test such sets for all $\sigma \in traces(s)$ the algorithm does it for the trace ε, and then repeats it recursively for all traces $a \in init(s)$.

Example 4.12. We will derive some test cases from p_3 in Fig. 1. In the first step determine $C_{p_3} := \{init(p_3)\} = \{\{shil\}\}$, so we can choose $A := \emptyset$ and $\nu(t) =$ **pass**, or $A := \{shil\}$ and $\nu(t) =$ **fail**. Since the first choice does not lead to a very useful test case, we continue with the latter: we get the test case $\Sigma\{a; t_a \mid a \in \{shil\}\} = shil; t_{shil}$.

To obtain t_{shil} repeat the algorithm for $p_3^\circ :=$ **i**; (liq; **stop** $\Box$ **i**; $choc$; **stop**) $\Box$ **i**; $choc$; **stop**. This gives $C_{p_3^\circ} := \{init(p') \mid p_3^\circ \overset{\varepsilon}{\Longrightarrow} p'\} = \{\{liq, choc\}, \{choc\}\}$, so possibilities for A are $A = \{choc\}$, $A = \{liq, choc\}$ both with $\nu(t_{shil}) =$ **fail**, or $A = \emptyset$ with $\nu(t_{shil}) =$ **pass**.

With $A = \{choc\}$ we have to repeat the algorithm for $p_3^{\circ\circ} :=$ **i**; **stop**. This gives $A := \emptyset$ and $\nu(t) =$ **pass**. Combining all steps we obtain test case t_2 of Fig. 2. With $A = \{liq, choc\}$ test case t_3 is obtained, and with $A = \emptyset$ we get t_4.

Algorithm 1 allows to construct arbitrarily long, but finite test cases: the nondeterminism in choosing any set $A \subseteq init(s)$ satisfying one of the three constraints in the disjunction, makes that at any depth of the recursion in the algorithm the test case can be made longer by choosing the first or the second alternative, or the test case can be terminated by having the third alternative ($A = \emptyset$ implies that $t := \Sigma\{a; t_a \mid a \in A\} = \Sigma\emptyset =$ **stop**). To avoid infinite recursion the third alternative must be chosen some time. This means that the algorithm generates only finite test cases, thus complying with Definition 3.1. However, the algorithm may generate infinitely many test cases, e.g., in the case of a specification with infinite behaviour. Consider, for example, the specification $s := a; s$, then the test suite of all possible test cases (without their verdicts) generated by Algorithm 1 is $\{$**stop**$, a;$ **stop**$, a; a;$ **stop**$, a; a; a;$ **stop**$, a; a; a; a;$ **stop**$, \ldots\}$, i.e., a test suite with infinitely many test cases, but each of them with finite (but arbitrarily long) behaviour. The principle to describe, or test, a system with infinite behaviour as a (possibly infinite) set of finite approximations is referred to as the approximation induction principle [16].

For a practical test campaign a selection from the complete, infinite test suite should be made. This is referred to as test selection or test-suite size reduction [31]. It is easy to see that any such a selection will always result in a sound test suite (Definition 3.4), i.e., a test suite that only detects errors (according to **conf**), but not necessarily all errors. The importance of the second part of Theorem 4.11 is in the fact that for all possible errors there is a possible test case, i.e., there are no errors that are principally undetectable with test suites generated with Algorithm 1.

5. Input–output transition systems

The implementation relations $\leq_{te}$ and **conf** are defined (Definitions 4.3 and 4.7) based on symmetric interaction between an implementation and its environment: all actions are treated the same way, and the synchronous communication operator $\|$ is commutative and fully symmetric in its operands. An interaction can occur if both the implementation and its environment are able to perform that interaction. If they both offer more than one interaction then it is assumed that by some mysterious negotiation mechanism they will agree on a common interaction. There is no notion of input or output, nor of initiative or direction. All actions are treated in the same way for all communicating systems.

Many real implementations, however, communicate in a different manner. They do make a distinction between inputs and outputs, and one can clearly distinguish whether the initiative for a particular interaction is with the implementation or with its environment. There is a direction in the flow of information from the initiating

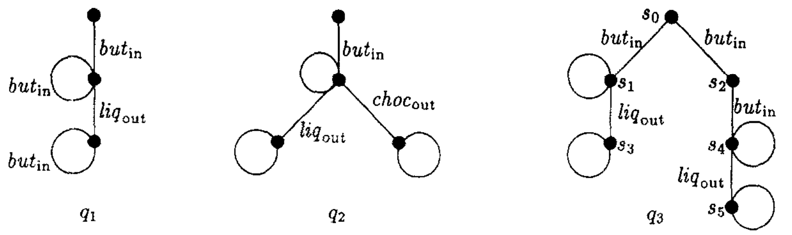

Fig. 3. Input-output transition systems.

communicating process to the other. The initiating process determines which interaction will take place, and the other one can just take it or leave it. Even if the other one decides not to accept the interaction, this is usually implemented by first accepting it, and then initiating a new interaction in the opposite direction explicitly signaling the non-acceptance. One could say that the mysterious negotiation mechanism is made explicit by exchanging two messages: one to propose an interaction and a next one to inform the initiating process about the (non-)acceptance of the proposed interaction.

We will now consider a class of (models of) implementations for which the set of actions can be partitioned into *output actions*, for which the initiative to perform them is with the implementation, and *input actions*, for which the initiative is with the environment. If an input action is initiated by the environment, the implementation is always prepared to participate in such an interaction: all inputs of an implementation are always enabled; they can never be refused. Naturally an input action of the implementation can only interact with an output of the environment, and vice versa. Although the initiative for any interaction is in exactly one of the communicating processes, the communication is still synchronous: if an interaction occurs it occurs at exactly the same time in both processes. The communication, however, is not symmetric: the communicating processes have different roles in an interaction.

Definition 5.1. An *input-output transition system* p is a labelled transition system in which the set of actions L is partitioned into input actions L_I and output actions L_U ($L_I \cup L_U = L,\ \ L_I \cap L_U = \emptyset$), and for which all input actions are always enabled in any state:

$$\forall p' \in der(p),\ \forall a \in L_I :\ p' \overset{a}{\Longrightarrow}$$

The class of input-output transition systems with input actions in L_I and output actions in L_U is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I \cup L_U)$.

Example 5.2. Fig. 3 gives some input-output transition systems with $L_I = \{but_{in}\}$ and $L_U = \{liq_{out}, choc_{out}\}$. In q_1 we can push the *button*, which is an input for the candy machine, and then the machine outputs *liquorice*. After the *button* has been pushed once, and also after having obtained *liquorice*, any more pushing of the *button* does not make anything happen: the machine makes a self-loop. In the sequel we use the convention that a self-loop of a state that is not explicitly labelled, is labelled with all inputs that cannot occur in that state (and also not via τ-transitions, cf. Definition 5.1).

Machine q_2 describes a candy machine that will output either *liquorice* or *chocolate* after the *button* has been pushed.

When studying input-output transition systems the notational convention will be that $a, b, c\ldots$ denote input actions, and $z, y, x, \ldots$ denote output actions. Since input-output transition systems are labelled transition systems all definitions for labelled transition systems apply. In particular, the synchronous parallel communication can be expressed by $\|$ (Definition 2.4), but now care should be taken that the outputs of one process interact with the inputs of the other.

6. Implementation relations for input–output transition systems

When we assume that implementations can be modelled by input-output transition systems in $\mathcal{IOTS}(L_I, L_U)$, then the next step for a testing theory is the study of implementation relations for such systems. Since specifications are not necessarily written in a style having the property that input actions cannot be refused, we still allow specifications to be labelled transition systems: we consider implementation relations **imp** $\subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U)$.

The implementation relations $\leq_{te}$ and **conf** were defined by relating the observations, made of the implementation by a symmetrically interacting observer $u \in \mathcal{LTS}(L)$, to the observations made of the specification (Definitions 4.3 and 4.7). Now that we consider implementations that communicate via inputs and outputs, it seems natural to restrict their observing environments in the same, complementary way: $u \in \mathcal{IOTS}(L_U, L_I)$. In a real observer inputs and outputs can be distinguished, of which input actions can never be refused, and communication takes place along the lines explained in Section 5: the input actions of the observer synchronize with the output actions of the implementation, and vice versa.

Analogous to the definition of testing preorder $\leq_{te}$ on $\mathcal{LTS}(L)$ the *input-output testing relation* $\leq_{iot}$ is defined between an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ and a specification $s \in \mathcal{LTS}(L_I \cup L_U)$ by requiring that any possible observation made of i by any "output-input" transition system is a possible observation of s by the same observer (cf. Definition 4.3).

Note that s can be any transition system, not necessarily an input-output transition system. This transition system is best interpreted as a not-completely specified input-output transition system, i.e., a transition system where a distinction is made between inputs and outputs, but where some inputs are not specified in some states. Since, technically speaking, the only distinction between inputs and outputs occurs in the transition systems themselves, and not in their communication or observations (The definitions of *obs*, *obs'*, $\|$, and . **after** . **deadlocks** are exactly the same as for the symmetric case), there is no problem in using an "output-input" observer u to observe such a not-completely specified input-output transition system. Below we will elaborate on this possibility to have $s \in \mathcal{LTS}$.

Definition 6.1. Let L be partitioned into L_I and L_U, and let $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$, then

$$i \leq_{iot} s \quad =_{def} \quad \forall u \in \mathcal{IOTS}(L_U, L_I) : \; obs(u,i) \subseteq obs(u,s) \;\; \text{and} \;\; obs'(u,i) \subseteq obs'(u,s)$$

The restriction to systems in which inputs and outputs can be distinguished, and in which inputs can never be refused, appears to simplify the corresponding intensional characterization of $\leq_{iot}$ (cf. Proposition 4.5): instead of sets of pairs consisting of a trace and a set of actions (failure pairs), it suffices to look at two sets of traces: the normal traces $traces(p)$ (Definition 2.3), and the output-suspension traces $\delta\text{-}traces(p)$.

Proposition 6.2. *Let* $\delta\text{-}traces(p) =_{def} \{\sigma \in L^* \mid p$ **after** σ **refuses** $L_U\}$ *be the set of* output-suspension traces *of* p, *then*

$$i \leq_{iot} s \;\; iff \;\; traces(i) \subseteq traces(s) \;\; and \;\; \delta\text{-}traces(i) \subseteq \delta\text{-}traces(s)$$

The notion of output suspension is analogous to the *null*-output that is sometimes used in Finite State Machine-based testing [15] to model the situation where an input does not produce any output. The *null*-output is then considered as a valid output, and it is an element of the output actions. In our approach the set L_U does not contain such a *null*-output; it only contains explicitly observable actions, and the absence of any outputs

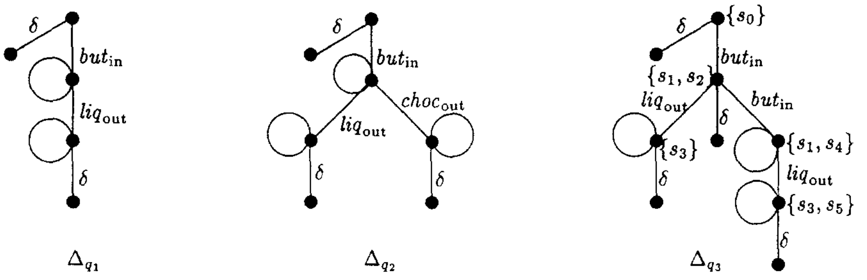

Fig. 4. δ-trace automata for Fig. 3.

after a certain trace is indicated by an output suspension trace. Below we will consider how output suspension can be considered as a special output action in our model.

The characterization of the input–output testing relation in Proposition 6.2 suggests to transform a labelled transition system into another one representing exactly these two sets of traces, so that the relation can be characterized by trace preorder $\leq_{tr}$ (Definition 4.1) on the results of this transformation. Such a transformation on a labelled transition system p can be defined, and the result is called the *δ-trace automaton* Δ_p. To obtain Δ_p a special transition is attached to each state where output suspension is possible. Then the resulting transition system is determinized (cf. the determinization of automata [28]). The special transition indicating output suspension has label δ, and goes to a state **stop**, from where no other transitions can be made. The label δ indicates the absence of output actions in a state, i.e., it makes the absence of output actions to an explicit observable action. It follows that, if $p \in \mathcal{LTS}(L)$, then $\Delta_p \in \mathcal{LTS}(L \cup \{\delta\})$.

Definition 6.3. Let L be partitioned into L_I and L_U, and let $p = \langle S, L_I \cup L_U, T, s_0\rangle \in \mathcal{LTS}(L_I \cup L_U)$ be a labelled transition system, then the *δ-trace automaton* of p, Δ_p, is the labelled transition system $\langle S_\delta, L_\delta, T_\delta, q_0\rangle \in \mathcal{LTS}(L_I \cup L_U \cup \{\delta\})$, where

- $S_\delta =_{def} \mathcal{P}(S) \cup \{\textbf{stop}\}$, with **stop** a distinguished state not occurring in S or $\mathcal{P}(S)$;
- $L_\delta =_{def} L_I \cup L_U \cup \{\delta\}$, with δ a distinguished label not occurring in $L_I \cup L_U$;
- $T_\delta =_{def} \{\, q \xrightarrow{a} q' \mid a \in L_I \cup L_U,\ q, q' \in S_\delta,\ q' = \{s' \in S \mid \exists s \in q : s \overset{a}{\Longrightarrow} s'\} \neq \emptyset \,\}$
  $\cup \{\, q \xrightarrow{\delta} \textbf{stop} \mid \exists s \in q,\ \forall x \in L_U : s \overset{x}{\not\Longrightarrow} \,\}$
- $q_0 =_{def} \{\, s' \in S \mid s_0 \overset{\varepsilon}{\Longrightarrow} s' \,\}$

Example 6.4. Fig. 4 gives the δ-trace automata for q_1, q_2, and q_3 of Fig. 3. For Δ_{q_3} the states, consisting of sets of states of q_3, have been added. Note that the nondeterminism of q_3 has been removed, and that state $\{s_1, s_2\}$ has a δ-transition, since there is a state in $\{s_1, s_2\}$, i.e. s_2, that can refuse all outputs.

From the δ-trace automaton of p the traces and the output-suspension traces of p are easily obtained, as is stated in Propositions 6.5(1) and 6.5(2): the traces of Δ_p that contain no δ are exactly the traces of p, and the traces of Δ_p that terminate with a δ-action point to an output-suspension trace. Moreover, Δ_p has the nice property that it is always deterministic (Definition 2.3), so that the transition relations $\xrightarrow{\sigma}$ and $\overset{\sigma}{\Longrightarrow}$ coincide, and each trace σ always goes to a unique state, denoted by Δ_p **after** σ. A state Δ_p **after** σ can always perform either an output transition with $x \in L_U$, or it can perform a δ-transition. Considering the special action δ as an output action of the δ-trace automaton, i.e., making the absence of any output action into a special, observable output action of the δ-trace automaton (cf. the *null*-output of an FSM, see above), we can say that any state of a δ-trace automaton (except **stop**) can always do at least one output transition (Proposition 6.5(4)).

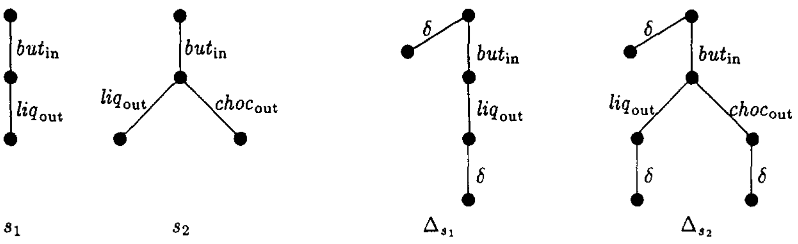

Fig. 5. Two specifications and their δ-trace automata.

Proposition 6.5.

(1) $traces(p) \;=\; traces(\Delta_p) \cap L^*$

(2) $\delta\text{-}traces(p) \;=\; \{\, \sigma \in L^* \mid \sigma\cdot\delta \in traces(\Delta_p) \,\}$

(3) Δ_p *is deterministic.*

(4) $\forall \sigma \in traces(\Delta_p) \cap L^*,\ \exists x \in L_U \cup \{\delta\} : (\,\Delta_p \textbf{ after } \sigma\,) \xrightarrow{x}$

An immediate corollary of Propositions 6.2 and 6.5 is that the input-output testing relation is completely characterized by trace preorder $\leq_{tr}$ on the corresponding δ-trace automata. The δ-trace automaton of a specification is sufficient and necessary to define the class of $\leq_{iot}$-conforming implementations. For our discussion concerning the implementation relation $\leq_{iot}$ we can now restrict to studying $\leq_{tr}$ on δ-trace automata.

Theorem 6.6. *Let* $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$, *then*

$$i \leq_{iot} s \ \ iff \ \ \Delta_i \leq_{tr} \Delta_s$$

Example 6.7. From Δ_{q_1}, Δ_{q_2}, and Δ_{q_3} (Figs. 3 and 4), using Theorem 6.6, it follows that $q_1 \leq_{iot} q_2$: an implementation capable of only producing *liquorice* conforms to a specification that prescribes to produce either *liquorice* or *chocolate*. Although q_2 looks deterministic, it in fact specifies that after *button* there is a nondeterministic choice between supplying *liquorice* or *chocolate*. It also implies that for this kind of testing q_2 is equivalent to $but_{in}; liq_{out};$ **stop** $\Box$ $but_{in}; choc_{out};$ **stop** (plus the input self-loops), an equivalence which does not hold for the symmetric case. If we want to specify a machine that produces both *liquorice* and *chocolate*, then two buttons are needed to select for the respective candies: *liq-button*; liq_{out}; **stop** $\Box$ *choc-button*; $choc_{out}$; **stop**.

On the other hand, $q_2 \not\leq_{iot} q_1, q_3$: if the specification prescribes to produce only *liquorice*, then an implementation should not have the possibility to produce *chocolate*: $but_{in}\cdot choc_{out} \in traces(\Delta_{q_2})$, while $but_{in}\cdot choc_{out} \notin traces(\Delta_{q_1}), traces(\Delta_{q_3})$.

We have $q_1 \leq_{iot} q_3$, but $q_3 \not\leq_{iot} q_1, q_2$, since q_3 may refuse to produce anything after the *button* has been pushed once, while both q_1 and q_2 will always output something. Formally: $but_{in}\cdot\delta \in traces(\Delta_{q_3})$, while $but_{in}\cdot\delta \notin traces(\Delta_{q_1}), traces(\Delta_{q_2})$.

Fig. 5 presents two non-input-output transition system specifications with their δ-trace automata, but none of q_1, q_2, q_3 correctly implements s_1 or s_2 with respect to $\leq_{iot}$; the problem occurs with non-specified input traces of the specification: $but_{in}\cdot but_{in} \in traces(\Delta_{q_1}), traces(\Delta_{q_2}), traces(\Delta_{q_3})$, while $but_{in}\cdot but_{in} \notin traces(\Delta_{s_1}), traces(\Delta_{s_2})$.

For the input-output testing relation it is allowed that the specification is not an input-output transition system. A specification may have states that can refuse input actions. The intention of such specifications often is that the specifyer does not care about the responses of an implementation on such non-specified inputs. If a candy machine is specified to deliver liquorice or chocolate after pushing a button (s_2 in Fig. 5), then it is left open what an implementation may do after pushing the button twice: perhaps ignoring it, supplying one of the candies, or responding with an error message. Many labelled transition system specifications contain such intended implementation freedom.

Looking at Theorem 6.6 and Fig. 5 in Example 6.7, however, we see that such implementation freedom cannot be expressed by the relation $\leq_{iot}$. Trace inclusion implies that for any state of the implementation all enabled actions, in particular all input actions, are also enabled in the corresponding state of the specification. Consequently, all input actions must always be enabled in any state of the specification, so the specification must be an input-output transition system, too, otherwise no implementation can exist. Labelled transition system specifications that are not input-output transition systems are not implementable with respect to $\leq_{iot}$.

To allow for non-input-output transition system specifications to express implementation freedom for non-enabled inputs, we introduce a weaker implementation relation. To define this relation, *i/o-conformance* **ioconf**, we first give an alternative characterization of $\leq_{iot}$ (Proposition 6.9) to see where the problem occurs, and how it might be solved. For this characterization the output actions $out(\Delta)$ of a δ-trace automaton are defined, where δ occurs as a special output action as explained above.

Definition 6.8. Let Δ be a δ-trace automaton, then $out(\Delta) =_{def} init(\Delta) \cap (L_U \cup \{\delta\})$

The set $out(\Delta)$ will be used in particular in expressions of the form $out(\Delta\,\mathbf{after}\,\sigma)$ to denote the set of outputs (possibly including δ) of the state reached after σ. If $\sigma \notin traces(\Delta)$, then we define $out(\Delta\,\mathbf{after}\,\sigma) =_{def} \emptyset$.

Proposition 6.9. $\Delta_i \leq_{tr} \Delta_s$ *iff* $\forall \sigma \in L^* : out(\Delta_i\,\mathbf{after}\,\sigma) \subseteq out(\Delta_s\,\mathbf{after}\,\sigma)$

In Proposition 6.9 we see that $\leq_{iot}$ requires that the outputs of the implementation are included in the outputs of the specification after any trace: traces of the specification, and traces that are not in the specification. A weaker implementation relation is obtained if this requirement is relaxed to inclusion for those traces that are explicitly specified in the specification (cf. the relation between $\leq_{te}$ and **conf**, Definitions 4.3 and 4.7, and Propositions 4.5 and 4.8).

Definition 6.10. Let $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$, then

$$i\ \mathbf{ioconf}\ s \ =_{def}\ \forall \sigma \in traces(\Delta_s) \cap L^* : out(\Delta_i\,\mathbf{after}\,\sigma) \subseteq out(\Delta_s\,\mathbf{after}\,\sigma)$$

Example 6.11. Consider again Figs. 3, 4, and 5. For q_1, q_2, q_3 we still only have q_1 **ioconf** q_2 and q_1 **ioconf** q_3 (cf. Proposition 6.12).

But our goal of defining **ioconf** is to allow for implementation freedom for unspecified behaviour as in s_1 and s_2. And indeed, we have q_1 **ioconf** s_1: following **ioconf**, s_1 specifies only that after one *button liquorice* must be produced; with **ioconf** s_1 does not care what happens if the *button* is pushed twice, as was the case with $\leq_{iot}$.

On the other hand, q_2 $\cancel{\mathbf{ioconf}}$ s_1, since q_2 can produce more than just *liquorice* after the *button* has been pushed once: $out(\Delta_{q_2}\,\mathbf{after}\,but_{\text{in}}) = \{liq, choc\} \nsubseteq \{liq\} = out(\Delta_{s_1}\,\mathbf{after}\,but_{\text{in}})$.

Moreover, q_1, q_2 **ioconf** s_2, but q_3 $\cancel{\mathbf{ioconf}}$ s_1, s_2, since $\delta \in out(\Delta_{q_3}\,\mathbf{after}\,but_{\text{in}})$,
while $\delta \notin out(\Delta_{s_1}\,\mathbf{after}\,but_{\text{in}}), out(\Delta_{s_2}\,\mathbf{after}\,but_{\text{in}})$.

The implementation relation **ioconf** will be the basis for the discussion of test generation in Section 7. We conclude this section with a brief comparison of the different implementation relations.

Relating implementation relations

The relation between the implementation relations for the symmetric case, $\leq_{tr}$, $\leq_{te}$, and **conf** (Section 4) is expressed in Proposition 4.9(2). To relate the implementation relations for input-output transition systems, first a generalization of **ioconf** is introduced. Let $\mathcal{F} \subseteq L^*$ be any set of traces, then

$$i\ \mathbf{ioconf}_{\mathcal{F}}\ s \ =_{def}\ \forall \sigma \in \mathcal{F} :\ out(\ \Delta_i\ \mathbf{after}\ \sigma\ ) \subseteq out(\ \Delta_s\ \mathbf{after}\ \sigma\ ) \tag{6.1}$$

The relations $\leq_{iot}$ and **ioconf** are special cases of $\mathbf{ioconf}_{\mathcal{F}}$, and different relations $\mathbf{ioconf}_{\mathcal{F}_1}$ and $\mathbf{ioconf}_{\mathcal{F}_2}$ are easily related if the sets $\mathcal{F}_1$ and $\mathcal{F}_2$ can be related: if $\mathcal{F}_1 \subseteq \mathcal{F}_2$ then $\mathbf{ioconf}_{\mathcal{F}_1} \supseteq \mathbf{ioconf}_{\mathcal{F}_2}$.

One might suspect that putting the relation **conf** (Definition 4.7) in an input–output context would result in **ioconf** i.e., that $\mathbf{conf}_{io}$ defined by

$$\begin{aligned} i\ \mathbf{conf}_{io}\ s \ =_{def}\ & \forall u \in \mathcal{IOTS}(L_U, L_I) :\ (\ obs(u,i) \cap traces(s)\ ) \subseteq obs(u,s) \\ & \text{and}\ \ (\ obs'(u,i) \cap traces(s)\ ) \subseteq obs'(u,s) \end{aligned} \tag{6.2}$$

would be equal to **ioconf**. This is not the case. The implication holds in only one direction:

$$\mathbf{ioconf} \subset \mathbf{conf}_{io} \tag{6.3}$$

A counter-example for the other direction is $i = x;\mathbf{stop}$ and $s = \mathbf{stop}$, with $L_I = \emptyset$ and $L_U = \{x\}$. One can even show that $\mathbf{conf}_{io}$ cannot be expressed in the form of (6.1); there is no $\mathcal{F}$ (depending only on s) such that $\mathbf{conf}_{io} = \mathbf{ioconf}_{\mathcal{F}}$.

We conclude this section with observing that on $\mathcal{IOTS}(L_I, L_U)$, i.e., the specification is an input–output transition system, too, the two relations $\leq_{iot}$ and **ioconf** coincide.

Proposition 6.12. *On* $\mathcal{IOTS}(L_I, L_U)$: $\leq_{iot}$ = **ioconf**

7. Testing input–output transition systems

The next point of discussion is the generation of sound test suites from labelled transition system specifications in order to test input–output transition system implementations with respect to the implementation relation **ioconf** (Section 6, Definition 6.10). This implies that, given a specification s, a test suite $gen_{\mathbf{ioconf}}(s) \subseteq \mathcal{LTS}_t$ must be generated, such that for any input–output transition system i:

$$i\ \mathbf{ioconf}\ s\ \text{ implies }\ i\ \mathbf{passes}\ gen_{\mathbf{ioconf}}(s) \tag{7.1}$$

or, using Definitions 3.2 and 3.4:

$$\begin{aligned} & i\ \mathbf{ioconf}\ s\ \text{ implies }\ \forall t \in gen_{\mathbf{ioconf}}(s),\ \forall \sigma \in L^* : \\ & \qquad (t \| i)\ \mathbf{after}\ \sigma\ \mathbf{deadlocks}\ \text{ implies }\ \nu(\ t\ \mathbf{after}\ \sigma\ ) = \mathbf{pass} \end{aligned} \tag{7.2}$$

But before we can develop such a test generation algorithm, a brief discussion on the nature of test cases is necessary. In Sections 5 and 6 it was stated that input-output transition systems are observed by "output-input" transition systems, while in Definition 3.1 test cases where defined as finite, deterministic labelled transition systems. The intersection of both is empty: an "output-input" transition system can never have finite behaviour (if $L_U \neq \emptyset$).

Since, as explained in Section 3, maximum control of the testing process by the tester is desirable, we will not allow test cases with a choice between an input action and an output action (input and output with respect to the implementation), nor a choice between multiple input actions. Both introduce nondeterminism in the test run: if a test case offers multiple input actions, or a choice between input and output, then the continuation of the test run is unnecessarily nondeterministic, since an implementation can always accept any input. This implies that in any state of a test case either one particular input is offered to the implementation, or all possible outputs are accepted. So such a test case is not an "output-input" transition system. Moreover, we still want test runs to be finite. This implies that at some instant the test case will stop: no actions are offered at all anymore. Combining these requirements we have the following definition of a test case for testing input-output transition systems.

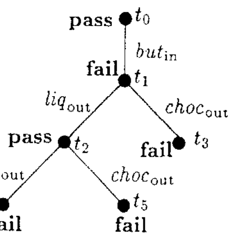

Fig. 6. An input-output test case.

Definition 7.1. An input–output test case t is a test case (Definition 3.1), which distinguishes between implementation inputs in L_I and implementation outputs in L_U, such that for any state t' of the test case, either $init(t') = \{a\}$ for some $a \in L_I$, or $init(t') = L_U$, or $init(t') = \emptyset$.
The class of input–output test cases over L_I and L_U is denoted as $\mathcal{IOTS}_t(L_U, L_I)$.

Example 7.2. For q_2 (Fig. 3) there are two test runs with t in Fig. 6:

$$t \,\|\, q_2 \xRightarrow{but_{in} \cdot liq_{out}} t_2 \,\|\, q_2' \quad\text{and}\quad \forall a \in L: \; t_2 \,\|\, q_2' \overset{a}{\not\Rightarrow}$$
$$t \,\|\, q_2 \xRightarrow{but_{in} \cdot choc_{out}} t_3 \,\|\, q_2'' \quad\text{and}\quad \forall a \in L: \; t_3 \,\|\, q_2'' \overset{a}{\not\Rightarrow}$$

where q_2' and q_2'' are the final states of q_2 after the liq_{out}- and $choc_{out}$-actions, respectively. Although $\nu(t_2) =$ **pass**, we have that q_2 **fails** t, since $\nu(t_3) =$ **fail**.
Similarly, q_1 **passes** t and q_3 **fails** t.

For the development of a test generation algorithm consider again the definition of **ioconf** (Definition 6.10):

$$i \;\textbf{ioconf}\; s \;\; =_{def} \;\; \forall \sigma \in traces(\Delta_s) \cap L^* : \; out(\, \Delta_i \,\textbf{after}\, \sigma\,) \subseteq out(\, \Delta_s \,\textbf{after}\, \sigma\,) \tag{7.3}$$

In (7.3) we see that to test for **ioconf** we have to check for each $\sigma \in traces(\Delta_s) \cap L^*$ whether $out(\, \Delta_i \textbf{after} \sigma\,) \subseteq out(\, \Delta_s \,\textbf{after}\, \sigma\,)$. Basically, this can be done by having a test case t that executes σ:

$$t \,\|\, i \overset{\sigma}{\Longrightarrow} t' \mid i'$$

and then checks $out(\, \Delta_i$ **after** $\sigma\,)$ by having transitions to **pass**-states for all allowed outputs (those in $out(\, \Delta_s$ **after** $\sigma\,)$), and transitions to **fail**-states for all erroneous outputs (those not in $out(\, \Delta_s$ **after** $\sigma\,)$). Special care should be taken for the special output δ: δ actually models the absence of any output, so no transition will be made at all if i "outputs" δ; the test run will deadlock in $t' \,\|\, i'$. This can be checked by having the verdict **pass** in the state t' if δ is allowed ($\delta \in out(\, \Delta_s$ **after** $\sigma\,)$), and by having the verdict **fail** in t', if the specification does not allow to have an output suspension at that point. All this is reflected in the following algorithm, which is proved to generate sound test cases with respect to **ioconf** (Theorem 7.3(1)), and which has the ability to detect all possible non-conforming implementations (Theorem 7.3(2)).

Algorithm 2. Let Δ be the δ-trace automaton of a specification, then a test case $t \in \mathcal{IOTS}_t(L_U, L_I)$ is obtained by a finite number of recursive applications of one of the following three nondeterministic choices:

```
1. (* terminate the test case *)
   $t$      := stop ;
   $\nu(t)$ := pass ;
```

2. (* give a next input to the implementation *)
   $t \quad := a \; ; \; t' \; ;$
   $\nu(t) := \textbf{pass} \; ;$
   where $a \in init(\Delta) \cap L_I$, and t' is obtained by recursively applying the algorithm for Δ', with $\Delta \xrightarrow{a} \Delta'$.
3. (* check the next output of the implementation *)
   $t \quad := \Sigma \; \{ \; x \; ; \; \textbf{stop} \mid x \in L_U, \; x \notin out(\Delta) \; \} \; \Box \; \Sigma \; \{ \; x \; ; \; t_x \mid x \in L_U, \; x \in out(\Delta) \; \} \; ;$
   $\nu(t) := \text{if } \; \delta \in out(\Delta) \; \text{ then } \textbf{pass} \text{ else } \textbf{fail} \; ;$
   where $\nu(\textbf{stop}) := \textbf{fail}$ for all x in the first operand, and t_x is obtained by recursively applying the algorithm for Δ', with $\Delta \xrightarrow{x} \Delta'$.

Theorem 7.3.
(1) *A test case obtained from Δ_s with Algorithm 2 is sound for s with respect to* **ioconf**.
(2) *The set containing all possible test cases that can be obtained with Algorithm 2 is exhaustive.*

Proof (*Sketch*). (For a complete proof see Appendix A.2.)
(1) The proof of soundness is based on defining a sufficient condition for soundness, and then using induction on the structure of t:
   (a) Any test case generated according to the first choice of Algorithm 2 is always sound.
   (b) Any test case generated according to the second choice is sound, if t' is sound for Δ'.
   (c) In the third choice a test case with $init(t) = L_U$ is generated:
   - If the implementation gives an output $x \notin out(\Delta_s)$, then the test case stops, and the verdict **fail** is assigned, which is sound.
   - If the implementation gives an output $x \in out(\Delta_s)$, then soundness follows by induction from soundness of t_x for Δ'.
   - If the implementation does not produce any output, i.e., $\delta \in out(\Delta_i)$, then the test run stops, and the verdict **pass** is assigned iff $\delta \in out(\Delta_s)$, which is sound.

(2) For proving exhaustiveness one defines a special test case $t_{[\Delta_s,\sigma]}$ that tests whether $out(\,\Delta_i\,\textbf{after}\,\sigma\,) \subseteq out(\,\Delta_s\,\textbf{after}\,\sigma\,)$. It is shown that for each $\sigma \in traces(\Delta_s) \cap L^*$ such a test case can be generated with the algorithm. Such a test case consists of the trace σ with transitions added at the end for each $x \in L_U$ exactly like in the third choice of the algorithm, and with a minimal number of transitions added in the other states of the test case to comply with Definition 7.1.

Example 7.4. We generate a test case for s_1 from Δ_{s_1} (Fig. 5).
We start with giving an input: $but_{\text{in}} \in init(\Delta_{s_1}) \cap L_I$, so $t := but_{\text{in}}; t'$ and $\nu(t) = \textbf{pass}$.
In the next step we generate the test case t' from $\Delta' = liq_{\text{out}}; \delta; \textbf{stop}$, where we check the outputs:
$t' := \Sigma\{x; \textbf{stop} \mid x \in L_U, x \notin \{liq_{\text{out}}\}\} \;\Box\; \Sigma\{x; t_x \mid x \in L_U, x \in \{liq_{\text{out}}\}\} \;=\; choc_{\text{out}}; \textbf{stop} \;\Box\; liq_{\text{out}}; t_{liq_{\text{out}}}$.
Since $\delta \notin out(\Delta')$, we have $\nu(t') = \textbf{fail}$. Moreover, $\nu(\textbf{stop}) = \textbf{fail}$.
Now generating $t_{liq_{\text{out}}}$ from $\Delta'' = \delta; \textbf{stop}$ we again check the outputs:
$t_{liq_{\text{out}}} := \Sigma\{x; \textbf{stop} \mid x \in L_U, x \notin \{\delta\}\} \;\Box\; \Sigma\{x; t_x \mid x \in L_U, x \in \{\delta\}\} \;=\; choc_{\text{out}}; \textbf{stop} \;\Box\; liq_{\text{out}}; \textbf{stop}$,
with for both $\nu(\textbf{stop}) = \textbf{fail}$, and $\nu(t_{liq_{\text{out}}}) = \textbf{pass}$.
Combining $t_{liq_{\text{out}}}$ and t' into t we get the test case t of Fig. 6 as a sound test case for s_1, which is consistent with the results that we found in Examples 6.11 and 7.2: q_1 **ioconf** s_1, q_2 **ioc$\not$onf** s_1, and q_3 **ioc$\not$onf** s_1, and indeed q_1 **passes** t, q_2 **fails** t, and q_3 **fails** t.

8. Concluding remarks

Two algorithms for test case generation from labelled transition system specifications have been presented, together with the implementation relations for which they test. The first relation, **conf**, and the corresponding algorithm have been published several times in literature in slightly different variations. The method is based on the assumptions that an implementation can be modelled as a labelled transition system, and that the interactions between an implementation and its environment are symmetric. An interaction can occur if both the implementation and the environment propose that interaction, which also means that they can both prevent an action from occurring.

The second implementation relation, **ioconf**, together with its test generation algorithm, is new. This method is based on the assumption that implementations communicate asymmetrically via inputs and outputs, where the outputs are under complete control of the implementation, whereas the implementation does not have any control over the inputs. Inputs are autonomously initiated by the environment, and the implementation can never refuse them. Such implementations were modelled by input–output transition systems, a subclass of labelled transition systems.

The theory of testing input–output transition systems can be applied to those domains where the implementations under test can be assumed to have the required property, which is the case for many realistic systems, and where the specification is expressed in a language for which the semantics can be expressed in labelled transition systems, which also holds for many formalisms. A special application area is the standardized formal description techniques Estelle [29] and SDL [17]. Estelle and SDL systems communicate with their environment via unbounded queues, which can never refuse their inputs, so any Estelle or SDL system can be modelled as an input–output transition system.

Symmetric testing versus testing with inputs and outputs

When comparing the testing theory for **conf** with that for **ioconf**, it can be noted that the additional assumption of always enabled input actions renders a testing theory which is simpler in some aspects. Whereas implementation relations based on symmetric interactions are characterized by a set of sets of actions for each trace (Propositions 4.5 and 4.8), the corresponding relations using inputs and outputs are characterized by just a set of actions for each trace (Proposition 6.9 and Definition 6.10). Moreover, using inputs and outputs each system is easily fully represented by a deterministic transition system: the δ-trace automaton (Definition 6.3 and Theorem 6.6).

The difference in simplicity between the respective test generation algorithms is also evident. For the symmetric case (Algorithm 1) the number of different possible sets $A \subseteq init(s)$ that have to be considered, is exponential in $|init(s)|$, whereas for the input-output case (Algorithm 2) the number of possibilities is restricted by $|(init(s) \cap L_I)|$.

Comparison with other models

Input-output state machines. The model of input–output transition systems is very much related to the model of input–output state machines (IOSM) [43]. The idea for the δ-trace automaton is inspired by the way output suspension is treated in [43]: a trace machine is made where a δ-transition is added to all states where no outputs are possible. However, the δ-transitions of an IOSM do not go a special state **stop**, but make a self-loop to the same state. This implies that the implementation relations of [43] $(R_1, \ldots, R_5)$, which are defined by trace inclusion on the resulting transformed machines, are different from ours: δ-actions can occur everywhere in a trace, not just at the end (cf. Proposition 6.5). The precise relation between these implementation relations and ours is a topic for further study. In particular, it would be interesting to relate them to implementation relations on $\mathcal{LTS}$ by means of a testing scenario, in the same way as $\leq_{te}$ and $\leq_{iot}$ are related.

Two additional minor differences between $\mathcal{IOTS}$ and IOSM can be noted. First, the sets of states and labels may be countably infinite in $\mathcal{IOTS}$, where finiteness is required for IOSM. Secondly, $\mathcal{IOTS}$, by defining them as a restriction on labelled transition systems, allows for a more easy embedding in, and relating to the more general theory of labelled transition systems, whereas [43] uses two rather complex mappings to map IOSM to $\mathcal{LTS}$ and vice versa.

Input/output automata. Another model that is closely related to both IOSM and $\mathcal{IOTS}$, is that of Input/Output Automata (IOA) [37]. An IOA is a transition system with the requirement that all inputs are directly enabled in all states, i.e., for all states s, for all $a \in L_I$: $s \xrightarrow{a}$. This stricter requirement on input enabling implies that some systems are more difficult to describe as IOA than as $\mathcal{IOTS}$. For example, a system consisting of an input/output automaton together with a bounded buffer with which it communicates with the environment, is not IOA, when the communication between the actual system and the buffer is hidden: if the buffer is full, no input actions are possible anymore without first performing an internal event. Such a system is $\mathcal{IOTS}$.

The implementation relation that is usually used for IOA is fair trace preorder [36]. This relation on IOA requires inclusion of so-called fair traces, which can be finite and infinite. An approximation using only finite traces is the quiescent trace preorder introduced in [52] and elaborated in [46]. This relation is characterized by inclusion of traces and quiescent traces, a quiescent trace being almost equal to an output-suspension trace: it is a trace to a state where only inputs are possible, i.e., no outputs *and* no internal transitions. Hence quiescent trace preorder is almost equal to $\leq_{iot}$ (Proposition 6.2). Our conjecture is that it is equal for strongly-converging processes, but for diverging processes quiescent trace preorder has some counter-intuitive properties. For example, let d be a divergent loop, $d := \tau; d$, then $i := a; d$ is in quiescent trace preorder with $s := a; x; \mathbf{stop}$ [46]. This looks counter-intuitive, and it does not hold for $\leq_{iot}$ if we apply Proposition 6.2 to diverging systems.

An effect analogous to that of **ioconf**, i.e., leaving implementation freedom for non-specified inputs, is obtained for IOA by having a so-called demonic semantics for process expressions. In this semantics a transition to a demonic process Ω is added for each non-specified input in the specification. From Ω any behaviour is possible. Thus, after such an input also any behaviour is allowed in the implementation [22].

Queue contexts. A special case of input–output transition systems are the queue systems of [51,53]. These queue systems are labelled transition systems in a queue context, i.e., to which two unbounded queues are attached to model asynchronous communication, one queue for input actions, and one for output actions. Communication between two processes is modelled by putting actions in the respective queues. An unbounded queue clearly has the property that input can never be refused, while the output queue makes that from the system's point of view output actions can never be refused by the environment.

Some of the results obtained for queue systems are indeed special cases of the results obtained in this paper, implying that only the distinction between inputs and outputs, and the permanent enabling of input actions are important, not the explicit form of communication by means of queues. In this way, the queue implementation relations of [51,53] are special cases of the relations $\leq_{iot}$ and $\mathbf{ioconf}_{\mathcal{F}}$, which is seen by noting that the observations $\mathcal{O}_s(\sigma)$ of a queue system are equal to our *out*-set (where Q_s is the queue context containing s):

$$\mathcal{O}_s(\sigma) \ =_{def} \ \{x \in L_U \mid Q_s \overset{\sigma \cdot x}{\Longrightarrow}\} \cup \{\delta \mid Q_s \ \mathbf{after}\ \sigma\ \mathbf{refuses}\ L_U\} \ = \ out(\Delta_{Q_s}\ \mathbf{after}\ \sigma)$$

It follows that queue preorder $\leq_{\mathcal{O}}$ is exactly the same as $\leq_{iot}$ applied to queue systems, and that $\leq_{tr(S)}$, **asco**, and **aconf** are instantiations of $\mathbf{ioconf}_{\mathcal{F}}$ (Eq. (6.1)) with appropriate trace sets $\mathcal{F}$. Moreover, Proposition 6.12 corresponds to the equality of $\leq_{\mathcal{O}}$ and $\leq_{tr(Q_S)}$, which was derived for queue systems in [51].

Open problems

Apart from establishing the precise relation with the other above mentioned theories based on inputs and outputs, some other open issues remain. First of all, there is the relation with the well-known Finite State Machine-based testing theories [15], which originate from hardware testing. These theories also distinguish between inputs and outputs, however, each transition is labelled with a pair of input and output, not with an input or an output. This implies that the notion of atomicity of actions differs, which makes comparison more difficult.

The problem of atomicity of actions also occurs when symmetric testing is considered as an abstraction of input–output testing. For example, the action *choc* of p_1 in Fig. 1 can be seen as an abstraction of first pushing a chocolate button and then obtaining chocolate. In this action refinement [2] the button-part can be seen as input to the candy machine, and obtaining chocolate as the output. Now test generation can be accomplished by first deriving a test from the abstract, symmetric specification in terms of the *choc* action, and then refining this test case into inputs and outputs, or the specification can be first refined after which an input–output based test generation algorithm can be used. The precise relation between testing, inputs and outputs, and action refinement needs further investigation.

A third open problem is the well-known test selection problem (test-suite size reduction [31]). Algorithms 1 and 2 can generate infinitely many sound test cases, but which ones shall be really executed? Solutions can be sought by defining coverage measures, fault models, test hypotheses, etc. [3,7,14,44].

Another aspect is the incorporation of data in the test generation procedure. The state explosion caused by the data in specifications needs to be handled in a symbolic way, otherwise automation of the test generation algorithm will probably not be feasible.

A more practical problem is the implementation of the observation of an output suspension. In practical testers timers will have to be used for this purpose, for which the time-out values need to be chosen carefully, in order not to observe a suspension where there is none.

A final remark concerns divergence. Divergence causes a lot of trouble and need for extra attention in the study of testing theories for labelled transition systems. That is why in this paper "for technical reasons" we assumed to deal with strongly converging systems (Section 2). However, divergence is not a problem that can always be neglected. As pointed out above, our conjecture is that also the relation between the IOA and $\mathcal{IOTS}$ preorders depends on divergence. The main question about divergence is whether fairness is assumed: if a system *can* perform infinitely many τ-transitions, while some observable action is constantly enabled, can we assume that this observable action will be finally executed? Different approaches to deal with divergence can be found in literature [20,34,40,12]. For the moment we leave the topic of divergence in the context of conformance testing for further study.

Acknowledgements

Stimulating discussions with and comments from Lex Heerink, Pedro D'Argenio, Jacob de Vries, Marc Phalippou (France Telecom CNET), Henk Schepers (Philips Research), and the anonymous referees helped me in writing this paper. The research for this paper was partly carried out while being an ERCIM Fellow (European Research Consortium for Informatics and Mathematics), supported by the Commission of the European Union.

Appendix A. Proofs

A.1. Proofs of Section 6 (Implementation relations for input–output transition systems)

Proposition 6.2. *Let* δ-*traces*(p) $=_{def}$ $\{\sigma \in L^* \mid p$ **after** σ **refuses** $L_U\}$ *be the set of* output-suspension traces *of p, then*

$$i \le_{iot} s \;\; iff \;\; traces(i) \subseteq traces(s) \; and \; \delta\text{-}traces(i) \subseteq \delta\text{-}traces(s)$$

Proof of Proposition 6.2. (*only if*) Let $\sigma \in traces(i)$, and define $u_\sigma \in \mathcal{IOTS}(L_U, L_I)$, such that $\exists u'$ $u_\sigma \xrightarrow{\sigma} u'$, then

$$\begin{array}{ll}
 & i \overset{\sigma}{\Longrightarrow} \text{ and } u_\sigma \xrightarrow{\sigma} u' \\
\text{implies} & u_\sigma \,\|\, i \overset{\sigma}{\Longrightarrow} \\
\text{implies} & \sigma \in obs'(u_\sigma, i) \\
\text{implies} & (*\text{ premiss, Definition 6.1 }*) \\
 & \sigma \in obs'(u_\sigma, s) \\
\text{implies} & u_\sigma \,\|\, s \overset{\sigma}{\Longrightarrow} \\
\text{implies} & s \overset{\sigma}{\Longrightarrow} \\
\text{implies} & \sigma \in traces(s)
\end{array}$$

Let $\sigma \in \delta\text{-}traces(i)$, and define u_σ as above, with additionally $init(u') = L_U$, then

$$\begin{array}{ll}
 & i \textbf{ after } \sigma \textbf{ refuses } L_U \text{ and } \exists u' : u_\sigma \xrightarrow{\sigma} u' \text{ and } init(u') = L_U \\
\text{implies} & (\ \exists i' : i \overset{\sigma}{\Longrightarrow} i' \text{ and } \forall x \in L_U : i' \overset{x}{\not\Longrightarrow}\ ) \text{ and } (\ \exists u' : u_\sigma \overset{\sigma}{\Longrightarrow} u' \text{ and } \forall a \in L_I : u' \overset{a}{\not\Longrightarrow}\ ) \\
\text{implies} & \exists i', u' :\ u_\sigma \,\|\, i \overset{\sigma}{\Longrightarrow} u' \,\|\, i' \text{ and } \forall a \in L :\ u' \,\|\, i' \overset{a}{\not\Longrightarrow} \\
\text{implies} & u_\sigma \,\|\, i \textbf{ after } \sigma \textbf{ deadlocks} \\
\text{implies} & \sigma \in obs(u_\sigma, i) \\
\text{implies} & (*\text{ premiss, Definition 6.1 }*) \\
 & \sigma \in obs(u_\sigma, s) \\
\text{implies} & u_\sigma \,\|\, s \textbf{ after } \sigma \textbf{ deadlocks} \\
\text{implies} & \exists u', s' :\ u_\sigma \,\|\, s \overset{\sigma}{\Longrightarrow} u' \,\|\, s' \text{ and } \forall a \in L :\ u' \,\|\, s' \overset{a}{\not\Longrightarrow} \\
\text{implies} & (*\ u \in \mathcal{IOTS}(L_U, L_I), \text{ so } u' \overset{x}{\Longrightarrow} \text{ for all } x \in L_U\ *) \\
 & \exists s' : s \overset{\sigma}{\Longrightarrow} s' \text{ and } \forall x \in L_U : s' \overset{x}{\not\Longrightarrow} \\
\text{implies} & s \textbf{ after } \sigma \textbf{ refuses } L_U \\
\text{implies} & \sigma \in \delta\text{-}traces(s)
\end{array}$$

(*if*) Let $u \in \mathcal{IOTS}(L_U, L_I)$, $\sigma \in obs(u, i)$, then

$$\begin{array}{ll}
 & u \,\|\, i \textbf{ after } \sigma \textbf{ deadlocks} \\
\text{implies} & \exists u', i' :\ u \,\|\, i \overset{\sigma}{\Longrightarrow} u' \,\|\, i' \text{ and } \forall a \in L :\ u' \,\|\, i' \overset{a}{\not\Longrightarrow} \\
\text{implies} & (*\ u' \text{ cannot refuse outputs; } i' \text{ cannot refuse inputs }*) \\
 & \exists u', i' :\ u \overset{\sigma}{\Longrightarrow} u' \text{ and } i \overset{\sigma}{\Longrightarrow} i' \text{ and } init(u') = L_U \text{ and } init(i') = L_I \\
\text{implies} & \exists u' : u \overset{\sigma}{\Longrightarrow} u' \text{ and } init(u') = L_U \text{ and } i \textbf{ after } \sigma \textbf{ refuses } L_U \\
\text{implies} & \exists u' : u \overset{\sigma}{\Longrightarrow} u' \text{ and } init(u') = L_U \text{ and } \sigma \in \delta\text{-}traces(i) \\
\text{implies} & (*\text{ premiss }*) \\
 & \exists u' : u \overset{\sigma}{\Longrightarrow} u' \text{ and } init(u') = L_U \text{ and } \sigma \in \delta\text{-}traces(s) \\
\text{implies} & \exists u' : u \overset{\sigma}{\Longrightarrow} u' \text{ and } init(u') = L_U \text{ and } s \textbf{ after } \sigma \textbf{ refuses } L_U \\
\text{implies} & \exists u' : u \overset{\sigma}{\Longrightarrow} u' \text{ and } init(u') = L_U \text{ and } \exists s' : s \overset{\sigma}{\Longrightarrow} s' \text{ and } \forall x \in L_U : s' \overset{x}{\not\Longrightarrow} \\
\text{implies} & \exists u', s' :\ u \,\|\, s \overset{\sigma}{\Longrightarrow} u' \,\|\, s' \text{ and } \forall a \in L :\ u' \,\|\, s' \overset{a}{\not\Longrightarrow} \\
\text{implies} & u \,\|\, s \textbf{ after } \sigma \textbf{ deadlocks} \\
\text{implies} & \sigma \in obs(u, s)
\end{array}$$

Let $u \in \mathcal{IOTS}(L_U, L_I)$, $\sigma \in obs'(u, i)$, then

$$\begin{array}{ll} & u \| i \overset{\sigma}{\Longrightarrow} \\ \text{implies} & u \overset{\sigma}{\Longrightarrow} \text{ and } i \overset{\sigma}{\Longrightarrow} \\ \text{implies} & u \overset{\sigma}{\Longrightarrow} \text{ and } \sigma \in \mathit{traces}(i) \\ \text{implies} & (* \text{ premiss } *) \\ & u \overset{\sigma}{\Longrightarrow} \text{ and } \sigma \in \mathit{traces}(s) \\ \text{implies} & u \overset{\sigma}{\Longrightarrow} \text{ and } s \overset{\sigma}{\Longrightarrow} \\ \text{implies} & u \| s \overset{\sigma}{\Longrightarrow} \\ \text{implies} & \sigma \in \mathit{obs}'(u,s) \quad \square \end{array}$$

Proposition 6.5.
(1) $\mathit{traces}(p) \;=\; \mathit{traces}(\Delta_p) \cap L^*$
(2) $\delta\text{-}\mathit{traces}(p) \;=\; \{\, \sigma \in L^* \mid \sigma\cdot\delta \in \mathit{traces}(\Delta_p) \,\}$
(3) *Δ_p is deterministic.*
(4) $\forall \sigma \in \mathit{traces}(\Delta_p) \cap L^*,\ \exists x \in L_U \cup \{\delta\} :\ (\,\Delta_p \textbf{ after } \sigma\,) \xrightarrow{x}$

Proof of Proposition 6.5.
(1) Without the additional δ-transitions (the second term of T_δ in Definition 6.3), p and the trace automaton of p accept the same language, i.e., have the same traces in L^*. This is easily proved by induction on the length of the traces [28].
(2) By adding the transitions $q \xrightarrow{\delta} \textbf{stop}$ the set of traces of Δ_p is extended with exactly the traces $\sigma\cdot\delta$, such that $\Delta_p \xrightarrow{\sigma} q$, where there is $s \in q$, such that $\forall x \in L_U :\ s \overset{x}{\not\Longrightarrow}$, which corresponds to $s_0 \overset{\sigma}{\Longrightarrow} s \overset{x}{\not\Longrightarrow}$, so σ is an output-suspension trace.
(3) Without the additional δ-transitions, the trace automaton of p is deterministic [28]. The addition of the δ-transitions cannot violate determinism, since $\delta \notin L_I \cup L_U$, and from any state of Δ_p there is at most one δ-transition.
(4) By construction of Δ_p (Definition 6.3): either $\exists x \in L_U :\ (\,\Delta_p \textbf{ after } \sigma\,) \xrightarrow{x}$, or a transition $(\,\Delta_p \textbf{ after } \sigma\,) \xrightarrow{\delta} \textbf{stop}$ is added. □

Theorem 6.6. *Let $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$, then $i \leq_{iot} s$ iff $\Delta_i \leq_{tr} \Delta_s$.*

Proof of Theorem 6.6. Directly from Propositions 6.2, 6.5(1), and 6.5(2). □

Proposition 6.9. $\Delta_i \leq_{tr} \Delta_s$ *iff* $\forall \sigma \in L^* :\ \mathit{out}(\,\Delta_i \textbf{ after } \sigma\,) \subseteq \mathit{out}(\,\Delta_s \textbf{ after } \sigma\,)$

Proof of Proposition 6.9. (*only if*) Let $\sigma \in L^*$, $x \in \mathit{out}(\,\Delta_i \textbf{ after } \sigma\,) \subseteq L_U \cup \{\delta\}$, then

$$\begin{array}{ll} & \Delta_i \xrightarrow{\sigma} (\,\Delta_i \textbf{ after } \sigma\,) \xrightarrow{x} \\ \text{implies} & \sigma\cdot x \in \mathit{traces}(\Delta_i) \\ \text{implies} & (* \text{ premiss } *) \\ & \sigma\cdot x \in \mathit{traces}(\Delta_s) \\ \text{implies} & x \in \mathit{out}(\,\Delta_s \textbf{ after } \sigma\,) \end{array}$$

(*if*) Let $\sigma \in \mathit{traces}(\Delta_i)$, and distinguish between $\sigma \in L^*$ and $\sigma = \sigma'\cdot\delta$ with $\sigma' \in L^*$, then
$\sigma \in L^*$:

$\Delta_i \xrightarrow{\sigma}$
implies (* Proposition 6.5(4) and Definition 6.8 *)
$out(\ \Delta_i\ \textbf{after}\ \sigma\ ) \neq \emptyset$
implies (* premiss *)
$out(\ \Delta_s\ \textbf{after}\ \sigma\ ) \neq \emptyset$
implies $\exists x \in L_U \cup \{\delta\} :\ \Delta_s \xrightarrow{\sigma} (\ \Delta_s\ \textbf{after}\ \sigma\ ) \xrightarrow{x}$
implies $\sigma \in traces(\Delta_s)$

$\sigma = \sigma' \cdot \delta$:

$\Delta_i \xrightarrow{\sigma' \cdot \delta}$
implies $\delta \in out(\ \Delta_i\ \textbf{after}\ \sigma'\ )$
implies (* premiss *)
$\delta \in out(\ \Delta_s\ \textbf{after}\ \sigma'\ )$
implies $\Delta_s \xrightarrow{\sigma'} (\ \Delta_s\ \textbf{after}\ \sigma'\ ) \xrightarrow{\delta}$
implies $\sigma' \cdot \delta \in traces(\Delta_s)$ □

Proposition 6.12. *On* $\mathcal{IOTS}(L_I, L_U)$: $\leq_{iot} = \textbf{ioconf}$

Proof of Proposition 6.12. Using Theorem 6.6, Proposition 6.9, and Definition 6.10, the $\subseteq$-part is trivial. For the $\supseteq$-part, let $\sigma \in L^*$ and $x \in out(\ \Delta_i\ \textbf{after}\ \sigma\ )$. If $\sigma \in traces(\Delta_s) \cap L^*$ also this part is trivial, so consider $\sigma \in L^* \backslash traces(\Delta_s)$ and prove by contradiction, i.e., prove:

$\exists \sigma \in L^* \backslash traces(\Delta_s) :\ out(\ \Delta_i\ \textbf{after}\ \sigma\ ) \not\subseteq out(\ \Delta_s\ \textbf{after}\ \sigma\ )$
implies $\exists \sigma \in traces(\Delta_s) \cap L^* :\ out(\ \Delta_i\ \textbf{after}\ \sigma\ ) \not\subseteq out(\ \Delta_s\ \textbf{after}\ \sigma\ )$

This is done as follows:

$\exists \sigma \in L^*,\ \sigma \notin traces(\Delta_s),\ \exists x \in out(\ \Delta_i\ \textbf{after}\ \sigma\ ) :\ x \notin out(\ \Delta_s\ \textbf{after}\ \sigma\ )$
implies (* $\sigma \notin traces(\Delta_s)$, so there is a longest prefix σ_1 of σ which is in $traces(\Delta_s)$;
$y \in L_U$, since $s \in \mathcal{IOTS}(L_I, L_U)$, so input actions are always possible *)
$\exists \sigma_1, \sigma_2 \in L^*, y \in L_U :\ \sigma = \sigma_1 \cdot y \cdot \sigma_2$ and $\sigma_1 \in traces(\Delta_s)$ and
$\sigma_1 \cdot y \notin traces(\Delta_s)$ and $\sigma_1 \cdot y \in traces(\Delta_i)$
implies $\exists \sigma_1 \in traces(\Delta_s) \cap L^*,\ y \in L_U :\ y \notin out(\ \Delta_s\ \textbf{after}\ \sigma_1\ )$ and $y \in out(\ \Delta_i\ \textbf{after}\ \sigma_1\ )$ □

A.2. Proofs of Section 7 (Testing input-output transition systems)

Algorithm 2 and Theorem 7.3 of Section 7 are generalized for the implementation relation $\textbf{ioconf}_{\mathcal{F}}$ (Eq. (6.1) in Section 6):

$$i\ \textbf{ioconf}_{\mathcal{F}}\ s \ =_{def}\ \forall \sigma \in \mathcal{F} :\ out(\ \Delta_i\ \textbf{after}\ \sigma\ ) \subseteq out(\ \Delta_s\ \textbf{after}\ \sigma\ )$$

so that the algorithm can also be used for other implementation relations that can be expressed as $\textbf{ioconf}_{\mathcal{F}}$ for some $\mathcal{F}$, such as $\leq_{iot}$, **asco, aconf**, ... Taking $\mathcal{F} = traces(s)$ Theorem 7.3 follows directly from Corollaries A.4 and A.7.

Definition A.1. Let $\mathcal{F} \subseteq L^*$ and $a \in L$, then $\mathcal{F}\ \textbf{after}\ a \ =_{def}\ \{\sigma \in L^* \mid a \cdot \sigma \in \mathcal{F}\}$.

Algorithm 3. Let Δ be the δ-trace automaton of a specification, and let $\mathcal{F} \subseteq L^*$, then a test case $t \in \mathcal{IOTS}_t(L_U, L_I)$ is obtained by a finite number of recursive applications of one of the following three nondeterministic choices:

1. (* terminate the test case *)
   t := **stop** ;
   $\nu(t)$:= **pass** ;
2. (* give a next input to the implementation *)
   t := a ; t' ;
   $\nu(t)$:= **pass** ;
   where $a \in L_I$, such that $\mathcal{F}$ **after** $a \neq \emptyset$, and t' is obtained by recursively applying the algorithm for $\mathcal{F}$ **after** a and Δ', with $\Delta \xrightarrow{a} \Delta'$.
3. (* check the next output of the implementation *)
   t := $\Sigma \, \{ \, x \, ; \, \textbf{stop} \mid x \in L_U, \; x \notin out(\Delta) \, \} \; \Box \; \Sigma \, \{ \, x \, ; \, t_x \mid x \in L_U, \; x \in out(\Delta) \, \}$;
   $\nu(t)$:= if $(\delta \in out(\Delta)$ or $\varepsilon \notin \mathcal{F})$ then **pass** else **fail** ;
   where $\nu(\textbf{stop})$:= if $\varepsilon \in \mathcal{F}$ then **fail** else **pass** for all x in the first operand, and t_x is obtained by recursively applying the algorithm for $\mathcal{F}$ **after** x and Δ', with $\Delta \xrightarrow{x} \Delta'$.

Lemma A.2. *Test suite $T \subseteq \mathcal{IOTS}_t(L_U, L_I)$ is sound for $s \in \mathcal{LTS}(L_I \cup L_U)$ with respect to* $\textbf{ioconf}_{\mathcal{F}}$, *if*

$$
\begin{array}{l}
\forall t \in T, \; \forall \sigma \in traces(t) : \nu(\, t \textbf{ after } \sigma \,) = \textbf{fail} \textit{ implies} \\
\qquad (\quad (\, \exists \sigma' \in \mathcal{F}, \; \exists x \in L_U : \; \sigma = \sigma' \cdot x \textit{ and } \\
\qquad\qquad\qquad x \notin out(\, \Delta_s \textbf{ after } \sigma' \,) \,) \\
\qquad \textit{or } (\, \sigma \in \mathcal{F} \textit{ and } \\
\qquad\qquad \delta \notin out(\, \Delta_s \textbf{ after } \sigma \,) \textit{ and } \\
\qquad\qquad out(\, \Delta_s \textbf{ after } \sigma \,) \subseteq init(\, t \textbf{ after } \sigma \,) \,) \quad)
\end{array}
$$

Proof of Lemma A.2. By contraposition:

$$
\begin{array}{ll}
 & T \text{ is not sound for } s \text{ with respect to } \textbf{ioconf}_{\mathcal{F}} \\
\text{implies} & (* \text{ Definition 3.4 } *) \\
 & \exists i \in \mathcal{IOTS}(L_I, L_U) : \; i \; \textbf{ioconf}_{\mathcal{F}} \; s \text{ and } i \textbf{ fails } T \\
\text{implies} & (* \text{ Definition 3.2 } *) \\
 & \exists i \in \mathcal{IOTS}(L_I, L_U) : \; i \; \textbf{ioconf}_{\mathcal{F}} \; s \text{ and} \\
 & \exists t \in T, \; \exists \sigma \in L^* : \; t \| i \textbf{ after } \sigma \textbf{ deadlocks} \text{ and } \nu(\, t \textbf{ after } \sigma \,) = \textbf{fail} \\
\text{implies} & (* \text{ Definition 4.4 } *) \\
 & \exists i \in \mathcal{IOTS}(L_I, L_U) : \; i \; \textbf{ioconf}_{\mathcal{F}} \; s \text{ and} \\
 & \exists t \in T, \exists \sigma \in L^*, \exists t', i' : \; t \| i \overset{\sigma}{\Longrightarrow} t' \| i' \text{ and } \forall a \in L : \; t' \| i' \overset{a}{\not\Longrightarrow} \text{ and} \\
 & \nu(\, t \textbf{ after } \sigma \,) = \textbf{fail} \\
\text{implies} & (* \; \| \text{ in Definition 2.4, Definitions 5.1 and 7.1 } *) \\
 & \exists i \in \mathcal{IOTS}(L_I, L_U) : \; i \; \textbf{ioconf}_{\mathcal{F}} \; s \text{ and} \\
 & \exists t \in T, \exists \sigma \in L^*, \exists t', i' : \; t \overset{\sigma}{\Longrightarrow} t' \text{ and } i \overset{\sigma}{\Longrightarrow} i' \text{ and} \\
 & (\; init(t') = \emptyset \text{ or } (\; init(t') = L_U \text{ and } init(i') = L_I \;) \;) \text{ and} \\
 & \nu(\, t \textbf{ after } \sigma \,) = \textbf{fail} \\
\text{implies} & (* \text{ rewrite and reorder } *) \\
 & \exists i \in \mathcal{IOTS}(L_I, L_U) : \; i \; \textbf{ioconf}_{\mathcal{F}} \; s \text{ and} \\
 & \exists t \in T, \exists \sigma \in traces(t) : \; \nu(\, t \textbf{ after } \sigma \,) = \textbf{fail} \text{ and} \\
 & (\; (\; \exists i' : i \overset{\sigma}{\Longrightarrow} i' \text{ and } init(\, t \textbf{ after } \sigma \,) = \emptyset \;) \text{ or } (\; \exists i' : i \overset{\sigma}{\Longrightarrow} i' \text{ and } init(i') = L_I \;) \;) \text{ and} \\
 & (\; \forall \sigma' \in \mathcal{F}, \forall x \in L_U : \; \sigma = \sigma' \cdot x \text{ implies } i \overset{\sigma' \cdot x}{\Longrightarrow} \;)
\end{array}
$$

implies (∗ definition $\mathbf{ioconf}_{\mathcal{F}}$ (6.1), Propositions 6.5(1), 6.5(2), and 6.5(4) ∗)
$\exists i \in \mathcal{IOTS}(L_I, L_U) : \forall \sigma \in \mathcal{F} : out(\ \Delta_i \ \mathbf{after}\ \sigma\ ) \subseteq out(\ \Delta_s\ \mathbf{after}\ \sigma\ )$ and
$\exists t \in T, \exists \sigma \in traces(t) :\ \nu(\ t\ \mathbf{after}\ \sigma\ ) = \mathbf{fail}$ and
$(\ (\ out(\ \Delta_i\ \mathbf{after}\ \sigma\ ) \neq \emptyset$ and $init(\ t\ \mathbf{after}\ \sigma\ ) = \emptyset\ )$ or $\delta \in out(\ \Delta_i\ \mathbf{after}\ \sigma\ )\ )$ and
$(\ \forall \sigma' \in \mathcal{F}, \forall x \in L_U :\ \sigma = \sigma' \cdot x$ implies $x \in out(\ \Delta_i\ \mathbf{after}\ \sigma'\ )\ )$
implies (∗ rewrite using the first line ∗)
$\exists t \in T, \exists \sigma \in traces(t) :\ \nu(\ t\ \mathbf{after}\ \sigma\ ) = \mathbf{fail}$ and
$(\ \sigma \in \mathcal{F}$ implies
$(\ (\ out(\ \Delta_s\ \mathbf{after}\ \sigma\ ) \neq \emptyset$ and $init(\ t\ \mathbf{after}\ \sigma\ ) = \emptyset\ )$ or $\delta \in out(\ \Delta_s\ \mathbf{after}\ \sigma\ )\ )\ )$ and
$(\ \forall \sigma' \in \mathcal{F}, \forall x \in L_U :\ \sigma = \sigma' \cdot x$ implies $x \in out(\ \Delta_s\ \mathbf{after}\ \sigma'\ )\ )$

implies (∗ reorder ∗)
$\exists t \in T,\ \exists \sigma \in traces(t) :\ \nu(\ t\ \mathbf{after}\ \sigma\ ) = \mathbf{fail}$ and
$(\quad (\ \forall \sigma' \in \mathcal{F},\ \forall x \in L_U :\ \sigma = \sigma' \cdot x$ implies
$x \in out(\ \Delta_s\ \mathbf{after}\ \sigma'\ )\ )$
and $(\ \sigma \notin \mathcal{F}$ or
$\delta \in out(\ \Delta_s\ \mathbf{after}\ \sigma\ )$ or
$out(\ \Delta_s\ \mathbf{after}\ \sigma\ ) \not\subseteq init(\ t\ \mathbf{after}\ \sigma\ )\ )\quad)$ □

Lemma A.3. *Let Δ be a δ-trace automaton, $\mathcal{F} \subseteq L^*$, and let t be a test case generated with Algorithm 3, then*
$\forall \sigma \in traces(t) :\ \nu(\ t\ \mathbf{after}\ \sigma\ ) = \mathbf{fail}$ *implies*
$(\quad (\ \exists \sigma' \in \mathcal{F},\ \exists x \in L_U :\ \sigma = \sigma' \cdot x$ *and*
$x \notin out(\ \Delta\ \mathbf{after}\ \sigma'\ )\ )$
or $(\ \sigma \in \mathcal{F}$ *and*
$\delta \notin out(\ \Delta\ \mathbf{after}\ \sigma\ )$ *and*
$out(\ \Delta\ \mathbf{after}\ \sigma\ ) \subseteq init(\ t\ \mathbf{after}\ \sigma\ )\ )\quad)$

Proof of Lemma A.3. By induction on the structure of t:

- Let $t = \mathbf{stop}$ and $\nu(t) = \mathbf{pass}$, then the lemma is trivially fulfilled.
- Let $t = a; t'$ and $\nu(t) = \mathbf{pass}$, with $a \in L_I$, and t' generated from $\mathcal{F}\ \mathbf{after}\ a$ and Δ', $\Delta \xrightarrow{a} \Delta'$, and let $\sigma \in traces(t)$ and $\nu(\ t\ \mathbf{after}\ \sigma\ ) = \mathbf{fail}$, then it follows that $\sigma = a \cdot \sigma'$ $(\sigma' \in L^*)$, $\sigma' \in traces(t')$, and $\nu(\ t'\ \mathbf{after}\ \sigma'\ ) = \mathbf{fail}$.
  According to the induction hypothesis the lemma can be assumed to hold for Δ', $\mathcal{F}\ \mathbf{after}\ a$, and t', hence

  $(\ \exists \sigma'' \in \mathcal{F}\ \mathbf{after}\ a,\ \exists x \in L_U : \sigma' = \sigma'' \cdot x$ and $x \notin out(\ \Delta'\ \mathbf{after}\ \sigma''\ )\ )$
  or $(\ \sigma' \in \mathcal{F}\ \mathbf{after}\ a$ and $\delta \notin out(\ \Delta'\ \mathbf{after}\ \sigma'\ )$ and $out(\ \Delta'\ \mathbf{after}\ \sigma'\ ) \subseteq init(\ t'\ \mathbf{after}\ \sigma'\ )\ )$

  If the first operand of the disjunction holds, then it follows directly from Definition A.1 that $a \cdot \sigma'' \in \mathcal{F}$ and $\sigma = a \cdot \sigma' = a \cdot \sigma'' \cdot x$ and $x \notin out(\ \Delta\ \mathbf{after}\ a \cdot \sigma''\ )$.
  If the second operand of the disjunction holds, then it follows directly from Definition A.1 that $a \cdot \sigma' \in \mathcal{F}$ and $\delta \notin out(\ \Delta\ \mathbf{after}\ a \cdot \sigma'\ )$, and moreover:
  $out(\ \Delta\ \mathbf{after}\ a \cdot \sigma'\ ) = out(\ \Delta'\ \mathbf{after}\ \sigma'\ ) \subseteq init(\ t'\ \mathbf{after}\ \sigma'\ ) = init(\ t\ \mathbf{after}\ a \cdot \sigma'\ )$.
- Let $t = \Sigma\ \{\ x\ ;\ \mathbf{stop} \mid x \in L_U,\ x \notin out(\Delta)\ \}\ \square\ \Sigma\ \{\ x\ ;\ t_x \mid x \in L_U,\ x \in out(\Delta)\ \}$ with $\nu(t)$ = if $(\delta \in out(\Delta)$ or $\varepsilon \notin \mathcal{F})$ then **pass** else **fail**, $\nu(\mathbf{stop})$ = if $\varepsilon \in \mathcal{F}$ then **fail** else **pass** for all x in the first operand, and t_x is generated from $\mathcal{F}\ \mathbf{after}\ x$ and Δ', with $\Delta \xrightarrow{x} \Delta'$.
  Let $\sigma \in traces(t)$ and $\nu(\ t\ \mathbf{after}\ \sigma\ ) = \mathbf{fail}$, then $\sigma = \varepsilon$ or $\sigma = y \cdot \sigma'$ $(\sigma' \in L^*, y \in L_U)$. Distinguish for the latter between $y \in out(\Delta)$ and $y \notin out(\Delta)$:
  $\sigma = \varepsilon$: From $\nu(\ t\ \mathbf{after}\ \sigma\ ) = \nu(t) = \mathbf{fail}$ we have $\delta \notin out(\Delta)$ and $\varepsilon \in \mathcal{F}$, and since $init(t) = L_U$ we have $out(\ \Delta\ \mathbf{after}\ \sigma\ ) = out(\Delta) \subseteq init(t) = init(\ t\ \mathbf{after}\ \sigma\ )$.

$\sigma = y\cdot\sigma'$, $y \in out(\Delta)$: According to the induction hypothesis the lemma can be assumed to hold for Δ', $\mathcal{F}$ **after** y, and t_y, and moreover $\sigma' \in traces(t_y)$ and $\nu(\, t_y \textbf{ after } \sigma'\,) = \textbf{fail}$, hence

$$\begin{aligned} &(\ \exists\sigma'' \in \mathcal{F} \textbf{ after } y\,,\ \exists x \in L_U : \sigma' = \sigma''\cdot x \ \text{ and } \ x \notin out(\,\Delta' \textbf{ after } \sigma''\,)\,) \\ \text{or}\,&(\sigma' \in \mathcal{F} \textbf{ after } y \ \text{ and } \ \delta \notin out(\,\Delta' \textbf{ after } \sigma'\,) \text{ and } out(\,\Delta' \textbf{ after } \sigma'\,) \subseteq init(\,t_y \textbf{ after } \sigma'\,)\,) \end{aligned}$$

If the first operand of the disjunction holds, then it follows directly from Definition A.1 that $y\cdot\sigma'' \in \mathcal{F}$ and $\sigma = y\cdot\sigma' = y\cdot\sigma''\cdot x$ and $x \notin out(\,\Delta \textbf{ after } y\cdot\sigma''\,)$.
If the second operand of the disjunction holds, then it follows directly from Definition A.1 that $y\cdot\sigma' \in \mathcal{F}$ and $\delta \notin out(\,\Delta \textbf{ after } y\cdot\sigma'\,)$, and moreover:
$out(\,\Delta \textbf{ after } y\cdot\sigma'\,) = out(\,\Delta' \textbf{ after } \sigma'\,) \subseteq init(\,t_y \textbf{ after } \sigma'\,) = init(\,t \textbf{ after } y\cdot\sigma'\,)$.
$\sigma = y\cdot\sigma'$, $y \notin out(\Delta)$: It follows that $t \xrightarrow{y} \textbf{stop}$ and $\sigma' = \varepsilon$, and hence from $\nu(\,t_y \textbf{ after } \sigma'\,) = \nu(t_y) = \textbf{fail}$ that $\varepsilon \in \mathcal{F}$, so $\sigma = y\cdot\sigma' = y = \varepsilon\cdot y$, and $y \notin out(\Delta) = out(\,\Delta \textbf{ after } \sigma'\,)$. □

Corollary A.4. *A test case obtained from Δ_s and $\mathcal{F}$ with Algorithm 3 is sound for s with respect to* $\textbf{ioconf}_{\mathcal{F}}$.

Proof of Corollary A.4. Directly from Lemmas A.2 and A.3. □

Definition A.5. Let Δ be a δ-trace automaton, and $\sigma \in L^*$, then the test case $t_{[\Delta,\sigma]} \in \mathcal{IOTS}_t(L_U, L_I)$ is defined by

$$\begin{array}{lll} t_{[\Delta,\varepsilon]} & =_{def} & \Sigma\ \{\ x\ ;\ \textbf{stop} \mid x \in L_U\ \} \\ & & \text{where } \nu(t_{[\Delta,\varepsilon]}) := \text{if } \ \delta \in out(\Delta) \ \text{ then } \ \textbf{pass} \ \text{ else } \ \textbf{fail} \\ & & \text{and for each branch } x; \textbf{stop}: \\ & & \nu(\textbf{stop}) := \text{if } \ x \in out(\Delta) \ \text{ then } \ \textbf{pass} \ \text{ else } \ \textbf{fail} \\ & & \\ t_{[\Delta,a\cdot\sigma]}\ (a \in L_I) & =_{def} & a\ ;\ t_{[\,\Delta \textbf{ after } a\,,\sigma]} \\ & & \text{where } \nu(t_{[\Delta,a\cdot\sigma]}) := \textbf{pass} \\ & & \\ t_{[\Delta,y\cdot\sigma]}\ (y \in L_U) & =_{def} & \Sigma\ \{\ x\ ;\ \textbf{stop} \mid x \in L_U,\ x \neq y\ \}\ \Box\ y\ ;\ t_{[\,\Delta \textbf{ after } y\,,\sigma]} \\ & & \text{where } \nu(t_{[\Delta,y\cdot\sigma]}) := \textbf{pass} \\ & & \text{and for each branch } x; \textbf{stop}:\ \ \nu(\textbf{stop}) := \textbf{pass} \end{array}$$

Lemma A.6.
(1) $t_{[\Delta,\sigma]} \xrightarrow{\sigma} t_{[\,\Delta \textbf{ after } \sigma\,,\varepsilon]}$.
(2) *Let $\mathcal{F} := \{\sigma\}$, then $t_{[\Delta,\sigma]}$ can be obtained with Algorithm 3.*
(3) *The test case $t_{[\Delta_s,\sigma]}$ is exhaustive for s with respect to* $\textbf{ioconf}_{\{\sigma\}}$.
(4) *The test suite $\{\ t_{[\Delta_s,\sigma]} \ \mid\ \sigma \in \mathcal{F}\ \}$ is exhaustive for s with respect to* $\textbf{ioconf}_{\mathcal{F}}$.

Proof of Lemma A.6.
(1) By induction on the length of σ:
$\sigma = \varepsilon$: Trivial.
$\sigma = a\cdot\sigma', a \in L_I$: $t_{[\Delta,a\cdot\sigma']} = a\ ;\ t_{[\,\Delta \textbf{ after } a\,,\sigma']} \xrightarrow{a}$
$t_{[\,\Delta \textbf{ after } a\,,\sigma']} \xrightarrow{\sigma'}$ (∗ induction ∗)
$t_{[\,(\,\Delta \textbf{ after } a\,) \textbf{ after } \sigma'\,,\varepsilon]} = t_{[\,\Delta \textbf{ after } \sigma\,,\varepsilon]}$
$\sigma = y\cdot\sigma', y \in L_U$: $t_{[\Delta,y\cdot\sigma']} = \Sigma\ \{\ x\ ;\ \textbf{stop} \mid x \in L_U, x \neq y\ \}\ \Box\ y\ ;\ t_{[\,\Delta \textbf{ after } y\,,\sigma']} \xrightarrow{y}$
$t_{[\,\Delta \textbf{ after } y\,,\sigma']} \xrightarrow{\sigma'}$ (∗ induction ∗)
$t_{[\,(\,\Delta \textbf{ after } y\,) \textbf{ after } \sigma'\,,\varepsilon]} = t_{[\,\Delta \textbf{ after } \sigma\,,\varepsilon]}$

(2) By induction on the structure of $t_{[\Delta,\sigma]}$:
$t_{[\Delta,\varepsilon]}$: Apply the third choice of Algorithm 3, and apply for each t_x the first choice.
$t_{[\Delta,a\cdot\sigma]}$: Apply the second choice of Algorithm 3, and repeat the algorithm for $t_{[\Delta\ \mathbf{after}\ a,\sigma]}$.
$t_{[\Delta,y\cdot\sigma]}$: Apply the third choice of Algorithm 3, apply for each t_x with $x \neq y$ the first choice, and repeat the algorithm for $t_{[\Delta\ \mathbf{after}\ y,\sigma]}$.
(3) To be proved (Definitions 3.4, 3.2, and (6.1)): i **passes** $t_{[\Delta_s,\sigma]}$ implies i $\mathbf{ioconf}_{\{\sigma\}}$ s, i.e.,

$$\forall \rho \in L^* : \ (t_{[\Delta_s,\sigma]} \| i)\ \mathbf{after}\ \rho\ \mathbf{deadlocks}\ \text{ implies }\ \nu(\, t_{[\Delta_s,\sigma]}\ \mathbf{after}\ \rho\,) = \mathbf{pass}$$
$$\text{implies}\ \ out(\,\Delta_i\ \mathbf{after}\ \sigma\,) \subseteq out(\,\Delta_s\ \mathbf{after}\ \sigma\,)$$

Let $x \in out(\,\Delta_i\ \mathbf{after}\ \sigma\,)$, and distinguish between $x \in L_U$ and $x = \delta$, then

$x \in L_U$:

$\Delta_i \xrightarrow{\sigma\cdot x}$
implies (* Proposition 6.5(1) *)
$\exists i', i'' :\ i \overset{\sigma}{\Longrightarrow} i' \overset{x}{\Longrightarrow} i''$
implies (* Lemma A.6(1) *)
$t_{[\Delta_s,\sigma]} \| i \overset{\sigma}{\Longrightarrow} t_{[\Delta_s\ \mathbf{after}\ \sigma,\varepsilon]} \| i' \overset{x}{\Longrightarrow} \mathbf{stop} \| i''$
implies (* Definition 4.4 *)
$(t_{[\Delta_s,\sigma]} \| i)\ \mathbf{after}\ \sigma\cdot x\ \mathbf{deadlocks}$
implies (* premiss *)
$\nu(\, t_{[\Delta_s,\sigma]}\ \mathbf{after}\ \sigma\cdot x\,) = \mathbf{pass}$
implies (* $t_{[\Delta_s,\sigma]}\ \mathbf{after}\ \sigma\cdot x\ =\ t_{[\Delta_s\ \mathbf{after}\ \sigma,\varepsilon]}\ \mathbf{after}\ x\ = \mathbf{stop}$,
verdict assignment for **stop** in Definition A.5 *)
$x \in out(\,\Delta_s\ \mathbf{after}\ \sigma\,)$

$x = \delta$:

$\Delta_i \xrightarrow{\sigma\cdot\delta}$
implies (* Proposition 6.5(2) *)
$\exists i' :\ i \overset{\sigma}{\Longrightarrow} i'$ and $\forall x \in L_U : i' \overset{x}{\not\Longrightarrow}$
implies (* Lemma A.6(1) and $init(t_{[\Delta_s\ \mathbf{after}\ \sigma,\varepsilon]}) = L_U$ *)
$\exists i' :\ t_{[\Delta_s,\sigma]} \| i \overset{\sigma}{\Longrightarrow} t_{[\Delta_s\ \mathbf{after}\ \sigma,\varepsilon]} \| i'$ and $\forall a \in L :\ t_{[\Delta_s\ \mathbf{after}\ \sigma,\varepsilon]} \| i' \overset{a}{\not\Longrightarrow}$
implies (* Definition 4.4 *)
$(t_{[\Delta_s,\sigma]} \| i)\ \mathbf{after}\ \sigma\ \mathbf{deadlocks}$
implies (* premiss *)
$\nu(\, t_{[\Delta_s,\sigma]}\ \mathbf{after}\ \sigma\,) = \mathbf{pass}$
implies (* verdict assignment for $(\, t_{[\Delta_s,\sigma]}\ \mathbf{after}\ \sigma\,) = t_{[\Delta\ \mathbf{after}\ \sigma,\varepsilon]}$ *)
$\delta \in out(\,\Delta_s\ \mathbf{after}\ \sigma\,)$

(4) Immediately from Lemma A.6(3). □

Corollary A.7. *The set containing all possible test cases that can be obtained with Algorithm 3 is exhaustive for s with respect to* $\mathbf{ioconf}_{\mathcal{F}}$.

Proof of Corollary A.7. Immediately from Lemma A.6(4), together with Lemma A.6(2), and the fact that for two test suites T_1 and T_2, if $T_1 \subseteq T_2$ and T_1 is exhaustive, then T_2 is exhaustive, which follows directly from Definitions 3.4 and 3.2. □

References

[1] S. Abramsky, Observational equivalence as a testing equivalence, *Theoret. Comput. Sci.* **53** (3) (1987) 225–241.

[2] L. Aceto, *Action Refinement in Process Algebras* (Cambridge University Press, Cambridge, MA, 1992).

[3] J. Alilovic-Curgus and S.T. Vuong, A metric based theory of test selection and coverage, in: A. Danthine, G. Leduc and P. Wolper, eds., *Protocol Specification, Testing and Verification XIII* (North-Holland, Amsterdam, 1993).

[4] R. Alderden, COOPER, the compositional construction of a canonical tester, in: S.T. Vuong, ed., *FORTE'89* (North-Holland, Amsterdam, 1990) 13–17.

[5] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat and J. Tretmans, A formal approach to conformance testing, in: J. de Meer, L. Mackert and W. Effelsberg, eds., *Proc. 2nd Internat. Workshop on Protocol Test Systems* (North-Holland, Amsterdam, 1990) 349–363; also: Memorandum INF-89-45, University of Twente, The Netherlands.

[6] T. Bolognesi and E. Brinksma, Introduction to the ISO specification language LOTOS, *Comput. Networks ISDN Systems* **14** (1987) 25–59.

[7] G. von Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi and G. Luo, Fault models in testing, in: J. Kroon, R. J. Heijink and E. Brinksma, eds., *Proc. 4th Internat. Workshop on Protocol Test Systems*, IFIP Transactions C-3 (North-Holland, Amsterdam, 1992) 17–30.

[8] G. Bernot, Testing against formal specifications: A theoretical view, in: S. Abramsky and T.S.E. Maibaum, eds., *TAPSOFT'91, Vol. 2*, Lecture Notes in Computer Science 494 (Springer, Berlin, 1991) 99–119.

[9] J.A. Bergstra and J.W. Klop, Algebra of communicating processes with abstraction, *Theoret. Comput. Sci.* **37** (1) (1985) 77–121.

[10] E. Brinksma, On the existence of canonical testers, Memorandum INF-87-5, University of Twente, Enschede, The Netherlands, 1987.

[11] E. Brinksma, A theory for the derivation of tests, in: S. Aggarwal and K. Sabnani, eds., *Protocol Specification, Testing and Verification VIII* (North-Holland, Amsterdam, 1988) 63–74; also: Memorandum INF-88-19, University of Twente, The Netherlands.

[12] E. Brinksma, A. Rensink and W. Vogler, Fair testing, in: *proc. CONCUR'95*, Lecture Notes in Computer Science (Springer, Berlin, 1995).

[13] E. Brinksma, G. Scollo and C. Steenbergen, LOTOS specifications, their implementations and their tests, in: G. von Bochmann and B. Sarikaya, eds., *Protocol Specification, Testing and Verification VI* (North-Holland, Amsterdam, 1987) 349-360.

[14] E. Brinksma, J. Tretmans and L. Verhaard, A framework for test selection, in: B. Jonsson, J. Parrow and B. Pehrson, eds., *Protocol Specification, Testing and Verification XI* (North-Holland, Amsterdam, 1991) 233–248; also: Memorandum INF-91-54, University of Twente, The Netherlands.

[15] B.S. Bosik and M.Ü. Uyar, Finite state machine based formal methods in protocol conformance testing: From theory to implementation, *Comput. Networks ISDN Systems* **22** (1) (1991) 7–33.

[16] J.C.M. Baeten and W.P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18 (Cambridge University Press, Cambridge, MA, 1990).

[17] CCITT, *Specification and Description Language (SDL)*, Recommendation Z.100. ITU-T General Secretariat, Geneve, Switzerland, 1992.

[18] I. Christoff, Testing equivalences and fully abstract models for probabilistic processes, in: J.C.M. Baeten and J.W. Klop, eds., *Proc. CONCUR'90*, Lecture Notes in Computer Science 458 (Springer, Berlin, 1990) 126–140.

[19] K. Drira, P. Azéma and F. Vernadat, Refusal graphs for conformance tester generation and simplification: A computational framework, in: A. Danthine, G. Leduc and P. Wolper, eds., *Protocol Specification, Testing and Verification XIII*, IFIP Transactions C-16 (North-Holland, Amsterdam, 1993).

[20] R. De Nicola, Extensional equivalences for transition systems, *Acta Inform.* **24** (1987) 211–237.

[21] R. De Nicola and M. Hennessy, Testing equivalences for processes, *Theoret. Comput. Sci.* **34** (1984) 83–133.

[22] R. De Nicola and R. Segala, A process algebraic view of input/output automata, *Theoret. Comput. Sci.* **138** (1995) 391–423.

[23] H. Eertink, The implementation of a test derivation algorithm, Memorandum INF-87-36, University of Twente, The Netherlands, 1987.

[24] S. Fujiwara and G. von Bochmann, Testing non-deterministic finite state machines, in: J. Kroon, R. J. Heijink and E. Brinksma, eds., *Proc. 4th Internat. Workshop on Protocol Test Systems*, IFIP Transactions C-3 (North-Holland, Amsterdam, 1992); extended abstract of Tech. Rept. 758, Université de Montréal, Canada, 1991.

[25] R.J. van Glabbeek, The linear time - Branching time spectrum, in: J.C.M. Baeten and J.W. Klop, eds., *Proc. CONCUR'90* Lecture Notes in Computer Science 458 (Springer, Berlin, 1990) 278–297.

[26] R.J. van Glabbeek, The linear time - Branching time spectrum II (The semantics of sequential systems with silent moves), in: E. Best and U. Goltz, eds., *proc. CONCUR'93*, Lecture Notes in Computer Science (Springer, Berlin, 1993).

[27] C.A.R. Hoare, *Communicating Sequential Processes* (Prentice-Hall, Englewood Cliffs, NJ, 1985).

[28] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, Reading, MA, 1979).

[29] ISO, *Information Processing Systems, Open Systems Interconnection, Estelle - A Formal Description Technique Based on an Extended State Transition Model*, International Standard IS-9074, ISO, 1989.

[30] ISO, *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, International Standard IS-8807, ISO, 1989.

[31] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8, *Information Retrieval, 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, Geneva, 1996.

[32] R. Langerak, A testing theory for LOTOS using deadlock detection, in: E. Brinksma, G. Scollo and C.A. Vissers, eds., *Protocol Specification, Testing and Verification IX* (North-Holland, Amsterdam, 1990) 87-98.

[33] G. Leduc, A framework based on implementation relations for implementing LOTOS specifications, *Comput. Networks ISDN Systems* **25** (1) (1992) 23-41.

[34] G. Leduc, Failure-based congruences, unfair divergences and new testing theory, in: S.T. Vuong and S.T. Chanson, eds., *Protocol Specification, Testing and Verification XIV* (Chapman and Hall, London, 1995) 252-267.

[35] K.G. Larsen and A. Skou, Bisimulation through probabilistic testing, in: *Proc. of Principles of Programming Languages 16* (ACM, New York, 1989).

[36] N. Lynch and M. Tuttle, Hierarchical correctness proofs for distributed algorithms, in: *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing* (1987) 137-151; also: Tech. Rept. MIT/LCS/TM-387, Massachusetts Institute of Technology, Cambridge, MA, 1987.

[37] N.A. Lynch and M.R. Tuttle, An introduction to input/output automata, *CWI Quarterly* 2 (3) (1989); also: Tech. Rept. MIT/LCS/TM-373 (TM-351 revised), Massachusetts Institute of Technology, Cambridge, MA, 1988.

[38] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92 (Springer, Berlin, 1980).

[39] R. Milner, *Communication and Concurrency* (Prentice-Hall, Englewood Cliffs, NJ, 1989).

[40] V. Natarajan and R. Cleaveland, Divergence and fair testing, in: *Proc. 22th Internat. Coll. on Automata, Languages and Programming (ICALP'95)*, Lecture Notes in Computer Science (Springer, Berlin, 1995).

[41] D. Park, Concurrency and automata on infinite sequences, in: P. Deussen, ed., *Proc. 5th GI Conf.* , Lecture Notes in Computer Science 104 (Springer, Berlin, 1981) 167-183.

[42] D.H. Pitt and D. Freestone, The derivation of conformance tests from LOTOS specifications, *IEEE Trans. Software Engineering* 16 (12) (1990) 1337-1343.

[43] M. Phalippou, *Relations d'Implantation et Hypothèses de Test sur des Automates à Entrées et Sorties*, Ph.D. Thesis, L'Université de Bordeaux I, France, 1994.

[44] M. Phalippou, Abstract testing and concrete testers, in: S.T. Vuong and S.T. Chanson, eds., *Protocol Specification, Testing and Verification XIV*, IFIP (Chapman and Hall, London, 1995) 221-236.

[45] I. Phillips, Refusal testing, *Theoret. Comput. Sci.* **50** (2) (1987) 241-284.

[46] R. Segala, Quiescence, fairness, testing and the notion of implementation (extended abstract), in: E. Best, ed., *Proc. CONCUR'93*, Lecture Notes in Computer Science 715 (Springer, Berlin, 1993) 324-338.

[47] Q.M. Tan, A. Petrenko and G. von Bochmann, Modeling Basic LOTOS by FSMs for conformance testing, in: P. Dembiński and M. Średniawa, eds., *Protocol Specification, Testing and Verification XV* (Chapman & Hall, London, 1996) 137-152.

[48] J. Tretmans, Test case derivation from LOTOS specifications, in: S.T. Vuong, ed., *Proc. FORTE'89* (North-Holland, Amsterdam, 1990) 345-359; also: Memorandum INF-90-21, University of Twente, The Netherlands.

[49] J. Tretmans, A formal approach to conformance testing, Ph.D. Thesis, University of Twente, The Netherlands, 1992.

[50] J. Tretmans, A formal approach to conformance testing, in: O. Rafiq, ed., *Proc. 6th Internat. Workshop on Protocol Test Systems*, IFIP Transactions C-19 (North-Holland, Amsterdam, 1994) 257-276.

[51] J. Tretmans and L. Verhaard, A queue model relating synchronous and asynchronous communication, in: R.J. Linn and M.Ü. Uyar, eds., *Protocol Specification, Testing and Verification XII*, IFIP Transactions C-8 (North-Holland, Amsterdam, 1992) 131-145; Extended abstract of Memorandum INF-92-04, University of Twente, Enschede, The Netherlands, 1992 and Internal Rept., TFL RR 1992-1, TFL, Hørsholm, Denmark.

[52] F. Vaandrager, On the relationship between process algebra and input/output automata, in: *Proc. 6th Ann. IEEE Symp. on Logic in Computer Science* (IEEE Computer Society Press, New York, 1991) 387-398.

[53] L. Verhaard, J. Tretmans, P. Kars and E. Brinksma, On asynchronous testing, in: G. von Bochmann, R. Dssouli and A. Das, eds., *Proc. 5th Internat. Workshop on Protocol Test Systems*, IFIP Transactions (North-Holland, Amsterdam, 1993); also: Memorandum INF-93-03, University of Twente, The Netherlands.

[54] C.D. Wezeman, The CO-OP method for compositional derivation of conformance testers, in: E. Brinksma, G. Scollo and C. A. Vissers, eds., *Protocol Specification, Testing and Verification IX* (North-Holland, Amsterdam, 1990) 145-158.

Jan Tretmans studied electrical engineering at the University of Twente, the Netherlands, where he graduated in 1986. He was a research assistant at that same university from 1986 till 1992, and he received his Ph.D. in computer science in 1992 with a dissertation on "A Formal Approach to Conformance Testing". During 1993 and 1994 he was an ERCIM fellow, visiting different ERCIM research laboratories (European Research Consortium for Informatics and Mathematics). In 1995 he returned to the University of Twente as a research associate. He participated in different European ESPRIT and RACE projects. He worked on tools for the formal description technique LOTOS, he contributed to the ISO standardization group on LOTOS, and he contributed to the ISO/ITU-T standardization group on "Formal Methods in Conformance Testing". His research interests include formal description techniques, verification, and conformance testing based on formal methods.