Test Generation with Inputs, Outputs, and Quiescence

Jan Tretmans[*]

ABSTRACT This paper studies testing based on labelled transition systems, using the assumption that implementations communicate with their environment via inputs and outputs. Such implementations are formalized by restricting the class of transition systems to those systems that can always accept input actions, as in input/output automata. Implementation relations, formalizing the notion of conformance of these implementations with respect to labelled transition system specifications, are defined analogous to the theory of testing equivalence and preorder. A test generation algorithm is given, which is proved to produce a sound and exhaustive test suite from a specification, i.e., a test suite that fully characterizes the set of correct implementations.

1 Introduction

Testing is an operational way to check the correctness of a system implementation by means of experimenting with it. Tests are applied to the implementation under test, and based on observations made during the execution of the tests a verdict about the correct functioning of the implementation is given. The correctness criterion that is to be tested is given in the system specification, preferably in some formal language. The specification is the basis for the derivation of test cases, when possible automatically, using a test generation algorithm.

Testing and verification are complementary techniques for analysis and checking of correctness of systems. While verification aims at proving properties about systems by formal manipulation on a mathematical model of the system, testing is performed by exercising the real, executing implementation (or an executable simulation model). Verification can give certainty about satisfaction of a required property, but this certainty only applies to the model of the system: any verification is only as good as the validity of the system model. Testing, being based on observing only a small subset of all possible instances of system behaviour, can never be complete: testing can only show the presence of errors, not their absence. But since testing can be applied to the real implementation, it is useful in those cases when a

[*] University of Twente, PO Box 217, NL-7500 AE Enschede, *tretmans@cs.utwente.nl*

valid and reliable model of the system is difficult to build due to complexity, when the complete system is a combination of formal parts and parts which cannot be formally modelled (e.g., physical devices), when the model is proprietary (e.g., third party testing), or when the validity of a constructed model is to be checked with respect to the physical implementation.

Many different aspects of a system can be tested: does the system do what it should do, i.e., does its behaviour comply with its functional specification (conformance testing), how fast can the system perform its tasks (performance testing), how does the system react if its environment does not behave as expected (robustness testing), and how long can we rely on the correct functioning of the system (reliability testing). This paper focuses on conformance testing based on formal specifications, in particular it aims at giving an algorithm for the generation of conformance test cases from transition system-based specifications.

The ingredients for defining such an algorithm comprise, apart from a formal specification, a class of implementations. An implementation under test, however, is a physical, real object, that is in principle not amenable to formal reasoning. It is treated as a black box, exhibiting behaviour, and interacting with its environment. 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 [1, 10, 15]. Thus the test hypothesis allows to reason about implementations as if they were formal objects, and to express the correctness of implementations with respect to specifications by a formal relation between such models of implementations and specifications. This relation is called the implementation relation [3, 10]. Conformance testing now consists of performing experiments to decide how the unknown model of the implementation relates to the specification. The experiments are specified in test cases. Given a specification, a test generation algorithm must produce a set of such test cases (a test suite), which must be sound, i.e., which give a negative verdict only if the implementation is not correct, and which, if the implementation is not correct, have a high probability to give a negative verdict.

One of the formalisms studied in the realm of 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, such as specifications, implementations, and tests, and it serves as a semantic model for various, well-known formal languages, e.g., ACP, CCS, and CSP. Also (most parts of) the semantics of standardized languages like LOTOS [9], SDL [4], and Estelle [8] can be expressed in labelled transition systems.

Traditionally, for labelled transition systems the term testing theory does not refer to conformance testing. Instead of starting with a specification to find a test suite to characterize the class of its conforming implementations, these testing theories aim at defining implementation relations, given a class of tests: a transition systems p is equivalent to a system q if any test case leads to the same observations with p as with q (or more generally, p relates to q if for all possible tests, the observations made of p are related in some sense to the observations made of q). Different relations are defined by variations of the class of tests, the way they are executed, and the required relation between observations, see e.g., [5, 7]. Conformance testing for labelled transition systems has been studied especially in the context of testing communication protocols with the language LOTOS, e.g., [2, 11, 15, 19]. This paper uses both kinds of testing theories: first an implementation relation is defined by using a class of tests, and, once defined, test generation from specifications for this particular relation is investigated.

Almost all of the testing theory mentioned above is 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. Both processes can propose and block the occurrence of an interaction; there is no distinction between input and output actions. For testing, a particular case where such communication occurs, is the modelling of the interaction between a tester and an implementation under test during the execution of a test. We will refer to above theories as testing with symmetric interactions.

This paper approaches communication in a different manner by distinguishing explicitly between the inputs and the outputs of a system. Such a distinction is made, for example, in Input/Output Automata [12], Input-Output State Machines [13], and Queue Contexts [17]. 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, while its output actions cannot be blocked by the environment. Communication takes place between inputs of the system and outputs of the environment, or the other way around. It 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, so it can be argued that such models have a closer link to reality. On the other hand, the input-output paradigm lacks some of the possibilities for abstraction, which can be a disadvantage when designing and specifying systems at a high level of abstraction. In an attempt to use the best of both worlds, this paper assumes that implementations communicate via inputs and outputs (as part of the test hypothesis), whereas specifications, although interpreting the same actions as inputs, respectively outputs, are allowed to refuse their inputs, which implies that technically specifications are just normal transition systems.

The aim of this paper is to study conformance testing and test generation algorithms for implementations that communicate via inputs and outputs, based on specifications that are labelled transition systems. The implementations are modelled by input-output transition systems, a special kind of labelled transition systems, where inputs are always enabled. These are introduced in section 2. Input-output transition systems differ only marginally from the input/output automata of [12]. Section 3 recalls some of the testing theory for symmetric interactions, in particular the definition of some often used implementation relations. Implementation relations with inputs and outputs are discussed in section 4. The first relation is defined following a testing scenario à la [5]. It is analogous to the scenario used in [14] to obtain a testing characterization of the relation quiescent trace preorder on input/output automata [18], and analogous results are obtained. However, it is shown that this relation does not make full use of the freedom to have specifications which are not input-enabled. A class of weaker implementation relations is defined, of which quiescent trace preorder is a special case. These relations allow to use the abstractness made possible by non-input-enabled specifications. A fully abstract model with respect to these relations is presented. Section 5 formalizes conformance testing by introducing test cases, test suites, and how to run, execute, and pass a test case. Finally, a test generation algorithm that produces provably correct test cases for any of the implementation relations of section 4 is developed in section 6. Some concluding remarks are given in section 7; for complete proofs we refer to [16].

2 Models

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

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

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.

Let $p = \langle S, L, T, s_0 \rangle$ be a labelled transition system with $s, s' \in S$, $\mu_{(i)} \in L \cup \{\tau\}$, $a_{(i)} \in L$, and $\sigma \in L^*$, then the following standard notations are used. Note that we identify the process p with its initial state s_0.

$$\begin{array}{lll}
s \xrightarrow{\mu} s' & =_{def} & (s, \mu, s') \in T \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' & =_{def} & \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{def} & \exists s' : s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \overset{\epsilon}{\Longrightarrow} s' & =_{def} & s = s' \text{ or } s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \overset{a}{\Longrightarrow} s' & =_{def} & \exists s_1, s_2 : s \overset{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Longrightarrow} s' \\
s \xRightarrow{a_1 \cdot \ldots \cdot a_n} s' & =_{def} & \exists s_0 \ldots s_n : s = s_0 \xRightarrow{a_1} s_1 \xRightarrow{a_2} \ldots \xRightarrow{a_n} s_n = s' \\
s \overset{\sigma}{\Longrightarrow} & =_{def} & \exists s' : s \overset{\sigma}{\Longrightarrow} s' \\
s \overset{\sigma}{\not\Longrightarrow} & =_{def} & \text{not } \exists s' : s \overset{\sigma}{\Longrightarrow} s' \\
\mathit{traces}(p) & =_{def} & \{ \sigma \in L^* \mid p \overset{\sigma}{\Longrightarrow} \} \\
\mathit{init}(p) & =_{def} & \{ a \in L \mid p \overset{a}{\Longrightarrow} \}
\end{array}$$

A process p has *finite behaviour* if there is a natural number n, such that all traces in $\mathit{traces}(p)$ have length smaller than n; p is *deterministic* if for all $\sigma \in L^*$, there is at most one p' such that $p \overset{\sigma}{\Longrightarrow} p'$. If $\sigma \in \mathit{traces}(p)$, then this p' is denoted by p **after** σ.

We represent a labelled transition system in the standard way, either by a tree or a graph, or by a process-algebraic behaviour expression, with a syntax inspired by LOTOS [9]:

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

Here $a \in L$, and $\mathcal{B}$ is a countable set of behaviour expressions. The operational semantics are given in the standard way by the following axioms and inference rules:

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

Communication between processes modelled as labelled transition systems is based on symmetric interaction, as expressed by the composition operator $\|$. An interaction can occur if both the process and its environment are able to perform that interaction, implying that both processes can also block the occurrence of an interaction. If both processes 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 both communicating partners.

Many real systems, 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 system or with its environment. There is a direction in the flow of information from the initiating communicating process to the other. The initiating process determines which interaction will take place. 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 signalling 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 use *input-output transition systems*, analogous to input/output automata [12], to model systems for which the set of actions can be partitioned into *output actions*, for which the initiative to perform them is with the system, and *input actions*, for which the initiative is with the environment. If an input action is initiated by the environment, the system is always prepared to participate in such an interaction: all the inputs of a system are always enabled; they can never be refused. Naturally an input action of the system can only interact with an output of the environment, and vice versa, implying that output actions can never be blocked by the environment. 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 2.2
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 inputs are always enabled in any state:

$$\text{whenever} \quad p \overset{\sigma}{\Longrightarrow} p' \quad \text{then} \quad \forall a \in L_I : \; p' \overset{a}{\Longrightarrow}$$

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 2.3
Figure 1 gives some input-output transition systems with $L_I = \{but_{\text{in}}\}$ and $L_U = \{liq_{\text{out}}, choc_{\text{out}}\}$. 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 2.2). □

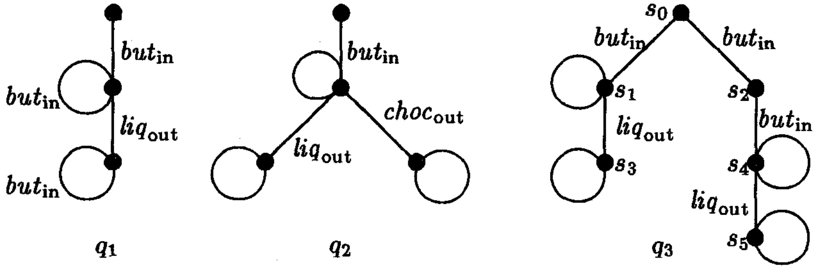

FIGURE 1. Input-output transition systems

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 $\|$, but now care should be taken that the outputs of one process interact with the inputs of the other.

Note that input-output transition systems differ marginally from input/output automata [12]: instead of requiring *strong input enabling* as in [12] ($\forall a \in L_I : p' \xrightarrow{a}$), input-output transition systems allow input enabling via internal transitions (*weak input enabling*, $\forall a \in L_I : p' \overset{a}{\Longrightarrow}$).

3 Testing with Symmetric Interactions

Before going to the test hypothesis that all implementations can be modelled by input-output transition systems in sections 4, 5, and 6, this section will briefly review the conformance testing theory that is based on the weaker hypothesis that implementations can be modelled as labelled transition systems. In this case correctness of an implementation with respect to a specification is expressed by an implementation relation on $\mathcal{LTS}(L)$. Many different relations have been studied, e.g., bisimulation equivalence, failure equivalence and preorder, testing equivalence and preorder, and many others [7]. A straightforward example 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 3.1
Let $i, s \in \mathcal{LTS}(L)$, then $\quad i \leq_{tr} s \quad =_{def} \quad traces(i) \subseteq traces(s)$ □

Another, more sophisticated relation is *testing preorder* $\leq_{te}$. 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 observer, or tester, encountering a deadlock with the implementation will experience the same deadlock when interacting with the specification. We formalize it using a testing scenario that is slightly different from the one in [5].

Definition 3.2
Let $p, i, s \in \mathcal{LTS}(L)$, $\sigma \in L^*$, and $A \subseteq L$, then

1. p **after** σ **ref** $A \quad =_{def} \quad \exists p' : p \overset{\sigma}{\Longrightarrow} p'$ and $\forall a \in A : p' \overset{a}{\not\Longrightarrow}$
2. p **after** σ **deadlocks** $\quad =_{def} \quad p$ **after** σ **ref** L
3. 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 their synchronization $u \| p$:

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

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

The definition of $\leq_{te}$ in definition 3.2 is extensional, i.e., in terms of how the environment (i.c. the observers u) perceives a system. It can be rewritten into an intensional characterization, i.e., a characterization in terms of properties of the transition systems themselves. This characterization, given in terms of failure pairs is known to coincide with failure preorder on our class of strongly converging transition systems [5].

Proposition 3.3
$i \leq_{te} s$ iff $\forall \sigma \in L^*, A \subseteq L :$ i **after** σ **ref** A implies s **after** σ **ref** A □

An implementation relation that is strongly related to $\leq_{te}$ is the relation **conf** [2]. 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. It is for the relation **conf** that several test generation algorithms have been developed and implemented, that generate provably correct test cases, e.g., [2, 15, 19].

Definition 3.4
i **conf** $s \quad =_{def} \quad \forall u \in \mathcal{LTS}(L) : \; (obs(u,i) \cap traces(s)) \subseteq obs(u,s)$ and $(obs'(u,i) \cap traces(s)) \subseteq obs'(u,s)$ □

Proposition 3.5
i **conf** s iff
$\forall \sigma \in traces(s), A \subseteq L :$ i **after** σ **ref** A implies s **after** σ **ref** A □

4 Relations with Inputs and Outputs

We now make the test assumption that implementations can be modelled by input-output transition systems: we consider implementation relations $\subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U)$.

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 3.2 and 3.4). An analogous testing scenario can be defined for input-output transition systems, using the fact that communication takes place along the lines explained in section 2: the input actions of the observer synchronize with the output actions of the implementation, and vice versa, so an input-output implementation in $\mathcal{IOTS}(L_I, L_U)$ communicates with an 'output-input' observer in $\mathcal{IOTS}(L_U, L_I)$. In this way the *input-output testing relation* $\leq_{iot}$ is defined between $i \in \mathcal{IOTS}(L_I, L_U)$ and $s \in \mathcal{LTS}(L_I \cup L_U)$ 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 3.2).

Definition 4.1
For $i \in \mathcal{IOTS}(L_I, L_U)$ and $s \in \mathcal{LTS}(L_I \cup L_U)$:

$$i \leq_{iot} s \quad =_{def} \quad \forall u \in \mathcal{IOTS}(L_U, L_I): \quad \begin{array}{rl} & obs(u,i) \subseteq obs(u,s) \\ \text{and} & obs'(u,i) \subseteq obs'(u,s) \end{array} \qquad \Box$$

Note that, despite what was said above about the communication between the implementation and the observer, the observations made of s are based on the communication between an input-output transition system and a standard labelled transition system, since s need not be an input-output system. Technically there is no problem in making such observations: the definitions of obs, obs', $\|$, and $\,.\,$ **after** $\,.\,$ **deadlocks** apply to labelled transition systems, not only to input-output transition systems. Below we will elaborate on this possibility to have $s \in \mathcal{LTS}$.

In [14] the testing scenario of testing preorder [5] was applied to define a relation on input/output automata, completely analogous to definition 4.1. It was shown to yield the implementation relation *quiescent trace preorder* introduced in [18]. Although we are more liberal with respect to the specification, $s \in \mathcal{LTS}(L_I \cup L_U)$, exactly the same intensional characterization is obtained: $\leq_{iot}$ is fully characterized by trace inclusion and inclusion of (weakly) quiescent traces. A weakly quiescent trace (output-suspension trace in [16]) is a trace after which no more outputs are possible. Note again the marginal difference with the original definition of quiescence on input/output automata [18]: there quiescence requires the absence of outputs and internal actions. We will refer to the latter as strong quiescence. It is easy to see that on our class of strongly converging transition systems both definitions coincide, but for diverging processes strong quiescence has some counter-intuitive properties. For example, let d be a divergent loop, $d := \tau; d$, then the trace a is not a strongly quiescent trace of $a; d$, which results in some counter-intuitive implementations following strongly quiescent trace preorder (cf. [14]).

Definition 4.2
Let $p \in \mathcal{LTS}(L)$. A trace $\sigma \in L^*$ is *weakly quiescent*, if p **after** σ **ref** L_U . The set of weakly quiescent traces of p is denoted by δ-*traces*(p). □

Proposition 4.3
$i \leq_{iot} s$ iff $\mathit{traces}(i) \subseteq \mathit{traces}(s)$ and $\delta\text{-}\mathit{traces}(i) \subseteq \delta\text{-}\mathit{traces}(s)$ □

Comparing the intensional characterization of $\leq_{iot}$ in proposition 4.3 with the one for $\leq_{te}$ (proposition 3.3), we see that the restriction to input-output systems simplifies the corresponding intensional characterization. Instead of sets of pairs consisting of a trace and a set of actions (failure pairs), it suffices to look at just two sets of traces. This relatively simple characterization 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 3.1) 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 quiescence is possible. Then the resulting transition system is determinized. The special transition indicating output quiescence 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.

Definition 4.4
Let $p = \langle S, L_I \cup L_U, T, s_0 \rangle \in \mathcal{LTS}(L_I \cup L_U)$, then the δ-*trace automaton* of p, Δ_p, is the transition system $\langle S_\delta, L_\delta, T_\delta, q_0 \rangle \in \mathcal{LTS}(L_I \cup L_U \cup \{\delta\})$, where

- $S_\delta \;=_{def}\; \mathcal{P}(S) \cup \{\mathbf{stop}\}$, with **stop** a unique state;
  ($\mathcal{P}(S)$ is the powerset of S)
- $L_\delta \;=_{def}\; L_I \cup L_U \cup \{\delta\}$, with $\delta \notin L_I \cup L_U$;
- $T_\delta \;=_{def}\; \{\, q \xrightarrow{a} q' \mid a \in L_I \cup L_U,\ q, q' \in S_\delta,$
  $q' = \{s' \in S \mid \exists s \in q : s \overset{a}{\Longrightarrow} s'\} \neq \emptyset \,\}$
  $\cup\, \{\, q \xrightarrow{\delta} \mathbf{stop} \mid \exists s \in q,\ \forall x \in L_U :\ s \overset{x}{\not\Longrightarrow} \,\}$
- $q_0 \;=_{def}\; \{\, s' \in S \mid s_0 \overset{\epsilon}{\Longrightarrow} s' \,\}$ □

Proposition 4.5

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

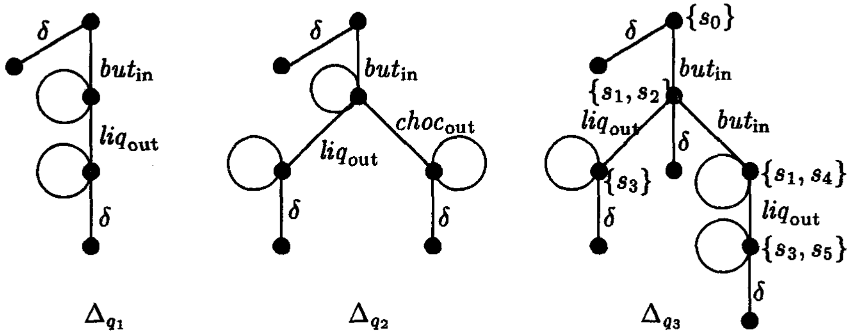

FIGURE 2. δ-trace automata for figure 1

Example 4.6
Figure 2 gives the δ-trace automata for q_1, q_2, and q_3 of figure 1. For Δ_{q_3} the states, subsets of states of q_3, have been added. Note that the nondeterminism of q_3 is removed, and that state $\{s_1, s_2\}$ has a δ-transition, since there is a state in $\{s_1, s_2\}$, i.c. s_2, that refuses all outputs. □

An immediate corollary of propositions 4.3 and 4.5 is that the input-output testing relation is completely characterized by trace preorder $\leq_{tr}$ on the corresponding δ-trace automata: they serve as a fully abstract model modulo $\leq_{iot}$. The δ-trace automaton of a specification is sufficient and necessary to define the class of $\leq_{iot}$-conforming implementations, and it will be the basis for the discussion of testing in section 6.

Theorem 4.7
$i \leq_{iot} s$ iff $\Delta_i \leq_{tr} \Delta_s$ □

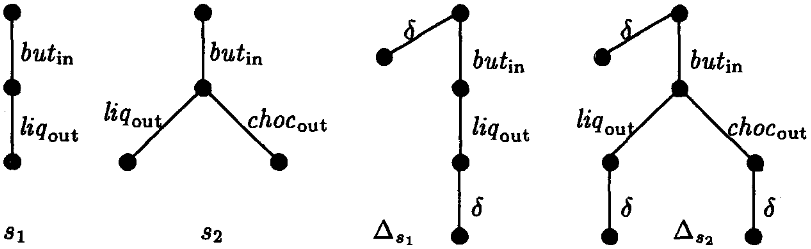

FIGURE 3. Two specifications and their δ-trace automata

Example 4.8
From Δ_{q_1}, Δ_{q_2}, and Δ_{q_3} (figures 1 and 2), using theorem 4.7, 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 $but_{\text{in}}; liq_{\text{out}};$ **stop** $\Box$ $but_{\text{in}}; choc_{\text{out}};$ **stop** (omitting the input self-loops), an equivalence which does not hold for $\leq_{te}$ in the symmetric case. If we want to specify a machine that produces both *liquorice* and *chocolate*, then two buttons are needed to select the respective candies:

$$liq\text{-}button;\ liq_{\text{out}};\ \textbf{stop}\ \Box\ choc\text{-}button;\ choc_{\text{out}};\ \textbf{stop}$$

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*. 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: $but_{\text{in}} \cdot \delta \in traces(\Delta_{q_3})$, while $but_{\text{in}} \cdot \delta \notin traces(\Delta_{q_1}), traces(\Delta_{q_2})$.

Figure 3 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; the problem occurs with non-specified input traces of the specification: $but_{\text{in}} \cdot but_{\text{in}} \in traces(\Delta_{q_1}), traces(\Delta_{q_2}), traces(\Delta_{q_3})$, while $but_{\text{in}} \cdot but_{\text{in}} \notin traces(\Delta_{s_1}), traces(\Delta_{s_2})$. □

For the relation $\leq_{iot}$ it is allowed that the specification is not an input-output transition system: a specification may have states that can refuse input actions. Such a specification is 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. 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 after pushing a button as in s_1 in figure 3, then it is intentionally 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. Intuitively, q_1 would conform to s_1, however, $q_1 \not\leq_{iot} s_1$, as was shown in example 4.8. The implementation freedom, intended with non-specified inputs, cannot be expressed with the relation $\leq_{iot}$. From theorem 4.7 the reason can be deduced: since the implementation can always perform input actions, all inputs must always be enabled in any state of the specification in order to satisfy trace inclusion, so the specification must be an input-output transition system, too, otherwise no implementation can exist.

For input/output automata a solution to this problem is given in [6], using the so-called demonic semantics for process expressions. In this semantics a transition to a demonic process Ω is added for each non-specified input. Since Ω exhibits any behaviour, the behaviour of the implementation is not prescribed after such a non-specified input. We choose another solution 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 4.10) 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 4.9
For Δ be a δ-trace automaton, $out(\Delta) =_{def} init(\Delta) \cap (L_U \cup \{\delta\})$ □

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

Proposition 4.10
$i \leq_{iot} s$ iff $\forall \sigma \in L^* : \; out(\, \Delta_i \textbf{ after } \sigma\,) \subseteq out(\, \Delta_s \textbf{ after } \sigma\,)$ □

In proposition 4.10 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 3.2 and 3.4, and propositions 3.3 and 3.5).

Definition 4.11
i **ioconf** $s =_{def} \forall \sigma \in traces(\Delta_s) \cap L^*$: $out(\Delta_i \textbf{ after } \sigma) \subseteq out(\Delta_s \textbf{ after } \sigma)$□

Example 4.12
Consider again figures 1, 2, and 3. Indeed we have q_1 **ioconf** s_1. On the other hand, q_2 **ioc$\not$onf** s_1, since q_2 can produce more than *liquorice* after the *button* has been pushed: $out(\Delta_{q_2} \textbf{ after } but_{\text{in}}) = \{liq, choc\} \not\subseteq \{liq\} = out(\Delta_{s_1} \textbf{ after } but_{\text{in}})$. Moreover, q_1, q_2 **ioconf** s_2, but q_3 **ioc$\not$onf** s_1, s_2, since $\delta \in out(\Delta_{q_3} \textbf{after}\, but_{\text{in}})$, while $\delta \notin out(\Delta_{s_1} \textbf{after}\, but_{\text{in}})$, $out(\Delta_{s_2} \textbf{after}\, but_{\text{in}})$.
□

The form of the characterizations of $\leq_{iot}$ in proposition 4.10 and of **ioconf** in definition 4.11 suggests to generalize them into a class of relations $\textbf{ioconf}_{\mathcal{F}}$ for any set of traces $\mathcal{F}$. Implementation relations of the form $\textbf{ioconf}_{\mathcal{F}}$ will be the basis for test generation in section 6.

Definition 4.13
Let $\mathcal{F} \subseteq L^*$, $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I \cup L_U)$, then

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

5 Testing Input-Output Transition Systems

Now that we have formal specifications, expressed as labelled transition systems, implementations, modelled by input-output transition systems, and a formal definition of conformance, expressed by one of the implementation relations **ioconf**$_{\mathcal{F}}$, the next point of discussion is how tests look like, and how tests are executed.

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 described by a labelled transition system. But to guarantee that the experiment lasts for a finite time, a test case should have finite behaviour. Moreover, a tester executing a test case would like to have control over the testing process as much as possible, so a test case should be specified in such a way that unnecessary nondeterminism is avoided. First of all, this implies that the test case itself must be deterministic. But also we will not allow test cases with a choice between an input action and an output action, nor a choice between multiple input actions (input and output with respect to the implementation). Both introduce unnecessary nondeterminism in the test run: if a test case can offer multiple input actions, or a choice between input and output, then the continuation of the test run is unnecessarily nondeterministic, since any input-output implementation can always accept any input action. 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. Finally, to be able to decide about the success of a test, a verdict (**pass** or **fail**) is attached to each state of the test. Altogether, we come to the following definition of a test case.

Definition 5.1

1. A *test case* t is a 6-tuple $\langle S, L_U, L_I, T, \nu, s_0\rangle$, such that:
   - ○ $\langle S, L_U \cup L_I, T, s_0\rangle$ is a deterministic labelled transition system with finite behaviour;
   - ○ for any state t' of the test case, either $\mathit{init}(t') = \{a\}$ for some $a \in L_I$, or $\mathit{init}(t') = L_U$, or $\mathit{init}(t') = \emptyset$;
   - ○ $\nu : S \to \{\mathbf{fail}, \mathbf{pass}\}$ is a *verdict function.*

   The class of test cases over L_U and L_I is denoted as $\mathcal{IOTS}_t(L_U, L_I)$.
2. A *test suite* T is a set of test cases: $T \subseteq \mathcal{IOTS}_t(L_U, L_I)$. □

Note that L_I and L_U refer the inputs and outputs from the point of view of the implementation under test, so L_I denotes the outputs, and L_U denotes the inputs of the test case. The definitions of $\mathcal{LTS}(L)$ are extended to $\mathcal{IOTS}_t(L_U, L_I)$ by defining them over the underlying transition system.

A test run of an implementation with 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 (definition 3.2). This deadlock may occur when the (finite) test case reaches a final state, or when the combination reaches a state where the test case expects an output from the implementation which is not produced. 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 5.2

1. A *test run* of a test case $t \in \mathcal{IOTS}_t(L_U, L_I)$ with an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ is a trace of the synchronous parallel composition of t and i, $t \,\|\, i$, leading to deadlock.
2. An implementation i *passes* a test case t, if all the test runs of t and i lead to a **pass**-state of t:
$$i \textbf{ passes } t \;=_{def}\; \forall \sigma \in L^* : \; (t \,\|\, i) \textbf{ after } \sigma \textbf{ deadlocks} \\ \text{implies} \;\; \nu(t \textbf{ after } \sigma) = \textbf{pass}$$
3. An implementation i *passes* a test suite T, if it passes all test cases in T: i **passes** $T \;=_{def}\; \forall t \in T : i$ **passes** t. If i does not pass the test suite, it fails: i **fails** $T \;=_{def}\; \exists t \in T : i$ **pas̸ses** t. □

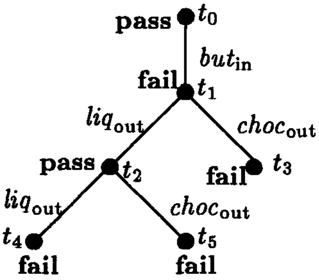

FIGURE 4. A test case

Example 5.3

For q_2 (figure 1) there are two test runs with t in figure 4:
$$t \,\|\, q_2 \xRightarrow{but_{\text{in}} \cdot liq_{\text{out}}} t_2 \,\|\, q_2' \;\text{ and }\; \forall a \in L : \; t_2 \,\|\, q_2' \overset{a}{\not\Rightarrow}$$
$$t \,\|\, q_2 \xRightarrow{but_{\text{in}} \cdot choc_{\text{out}}} t_3 \,\|\, q_2'' \;\text{ and }\; \forall a \in L : \; t_3 \,\|\, q_2'' \overset{a}{\not\Rightarrow}$$
where q_2' and q_2'' are the final states of q_2 after the liq_{out}– and $choc_{\text{out}}$–actions, respectively. Although $\nu(t_2) = \textbf{pass}$, we have that q_2 **fails** t, since $\nu(t_3) = \textbf{fail}$. Similarly, q_1 **passes** t and q_3 **fails** t . □

6 Test Generation for Inputs and Outputs

Now all ingredients are there to present an algorithm $gen_{\mathbf{ioconf}_\mathcal{F}}$ to generate test suites from labelled transition system specifications for any of the implementation relations $\mathbf{ioconf}_\mathcal{F}$. A generated test suite $gen_{\mathbf{ioconf}_\mathcal{F}}(s)$ must test implementations for conformance with respect to s and $\mathbf{ioconf}_\mathcal{F}$. Ideally, an implementation should pass the test suite if and only if it is conforming. In this case the test suite is called *complete* [10]. 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 6.1
Let s be a specification, and T a test suite, then for an implementation relation $\mathbf{ioconf}_\mathcal{F}$:

$$\begin{array}{lllccl} T \text{ is complete} & =_{def} & \forall i: & i\ \mathbf{ioconf}_\mathcal{F}\ s & \text{iff} & i\ \mathbf{passes}\ T \\ T \text{ is sound} & =_{def} & \forall i: & i\ \mathbf{ioconf}_\mathcal{F}\ s & \text{implies} & i\ \mathbf{passes}\ T \\ T \text{ is exhaustive} & =_{def} & \forall i: & i\ \mathbf{ioconf}_\mathcal{F}\ s & \text{if} & i\ \mathbf{passes}\ T \ \Box \end{array}$$

We aim at producing sound test suites. To get some idea how such test cases will look like we consider the definition of **ioconf**. In definition 4.11 we see that to test for **ioconf** we have to check for each $\sigma \in traces(\Delta_s) \cap L^*$ whether $out(\Delta_i\ \mathbf{after}\ \sigma) \subseteq out(\Delta_s\ \mathbf{after}\ \sigma)$. Basically, this can be done by having a test case t that executes σ:

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

and then checks $out(\Delta_i \mathbf{after} \sigma)$ by having transitions to **pass**-states for all allowed outputs (those in $out(\Delta_s \mathbf{after} \sigma)$), and transitions to **fail**-states for all erroneous outputs (those not in $out(\Delta_s\ \mathbf{after}\ \sigma)$). 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 ($\delta \in out(\Delta_s\ \mathbf{after}\ \sigma)$), and by having the verdict **fail** in t', if the specification does not allow to have quiescence at that point. All this is reflected in the following recursive algorithm. The algorithm is nondeterministic in the sense that in each recursive step it can be continued in many different ways: termination of the test case in choice 1, any input action satisfying the requirement of choice 2, or checking the allowed outputs in choice 3. Each continuation will result in another sound test case (theorem 6.4.1), and all possible test cases together form an exhaustive (and thus complete) test suite (theorem 6.4.2), so there are no errors in implementations that are principally undetectable with test suites generated with the algorithm. However, if the behaviour of the specification is infinite, the algorithm allows to construct infinitely many different test cases, which can be arbitrarily long, but which all have finite behaviour. To define the algorithm one additional definition is needed.

Definition 6.2
Let $\mathcal{F} \subseteq L^*$ and $a \in L$, then $\mathcal{F}\,\mathbf{after}\,a \;=_{def}\; \{\sigma \in L^* \mid a{\cdot}\sigma \in \mathcal{F}\}$. □

Algorithm 6.3
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 *)
$$\begin{array}{lll} t & := & \mathbf{stop}\ ; \\ \nu(t) & := & \mathbf{pass}\ ; \end{array}$$
2. (* give a next input to the implementation *)
$$\begin{array}{lll} t & := & a\ ;\ t'\ ; \\ \nu(t) & := & \mathbf{pass}\ ; \end{array}$$
where $a \in L_I$, such that $\mathcal{F}\,\mathbf{after}\,a \neq \emptyset$, and t' is obtained by recursively applying the algorithm for $\mathcal{F}\,\mathbf{after}\,a$ and Δ', with $\Delta \xrightarrow{a} \Delta'$.
3. (* check the next output of the implementation *)
$$\begin{array}{llll} t & := & & \Sigma\,\{\ x\ ;\ \mathbf{stop} \mid x \in L_U,\ x \notin out(\Delta)\ \} \\ & & \Box & \Sigma\,\{\ x\ ;\ t_x \mid x \in L_U,\ x \in out(\Delta)\ \}\ ; \\ \nu(t) & := & \multicolumn{2}{l}{\text{if } (\delta \in out(\Delta) \text{ or } \epsilon \notin \mathcal{F}) \text{ then } \mathbf{pass} \text{ else } \mathbf{fail}\ ;} \end{array}$$
where $\nu(\mathbf{stop}) :=$ if $\epsilon \in \mathcal{F}$ then **fail** else **pass** for all x in the first operand, and t_x is obtained by recursively applying the algorithm for $\mathcal{F}\,\mathbf{after}\,x$ and Δ', with $\Delta \xrightarrow{x} \Delta'$.

□

Theorem 6.4

1. A test case obtained with algorithm 6.3 from Δ_s and $\mathcal{F}$ is sound for s with respect to $\mathbf{ioconf}_{\mathcal{F}}$.
2. The set containing all possible test cases that can be obtained with algorithm 6.3 is exhaustive.

□

Example 6.5
We generate a test case for s_1 from Δ_{s_1} for the implementation relation $\mathbf{ioconf} = \mathbf{ioconf}_{traces(s)}$ (figure 3). We start with giving an input:
$but_{\text{in}} \in init(\Delta_{s_1}) \cap L_I$, so $t := but_{\text{in}}; t'$ and $\nu(t) = \mathbf{pass}$.
In the next step we generate the test case t' from $\Delta' = liq_{\text{out}}; \delta; \mathbf{stop}$, where we check the outputs:
$t' := \Sigma\{x; \mathbf{stop} \mid x \in L_U, x \notin \{liq_{\text{out}}\}\} \,\Box\, \Sigma\{x; t_x \mid x \in L_U, x \in \{liq_{\text{out}}\}\}$
$=\ choc_{\text{out}}; \mathbf{stop} \,\Box\, liq_{\text{out}}; t_{liq_{\text{out}}}$.
Since $\delta \notin out(\Delta')$, we have $\nu(t') = \mathbf{fail}$. Moreover, $\nu(\mathbf{stop}) = \mathbf{fail}$.
Now generating $t_{liq_{\text{out}}}$ from $\Delta'' = \delta; \mathbf{stop}$ we again check the outputs:
$t_{liq_{\text{out}}} := \Sigma\{x; \mathbf{stop} \mid x \in L_U, x \notin \{\delta\}\} \,\Box\, \Sigma\{x; t_x \mid x \in L_U, x \in \{\delta\}\}$

$= choc_{\text{out}}$; **stop** $\Box$ liq_{out}; **stop**,
with for both branches $\nu(\textbf{stop}) =$ **fail**, and $\nu(t_{liq_{\text{out}}}) =$ **pass**.

Combining $t_{liq_{\text{out}}}$ and t' into t we obtain the test case t of figure 4 as a sound test case for s_1, which is consistent with the results found in examples 4.12 and 5.3: q_1 **ioconf** s_1, q_2 **ioc$\not{o}$nf** s_1, and q_3 **ioc$\not{o}$nf** s_1, and indeed q_1 **passes** t, q_2 **fails** t, and q_3 **fails** t. $\Box$

7 Concluding Remarks

This paper presented a theory for conformance testing of implementations that communicate via inputs and outputs. The main ingredients of this theory are the implementation relations $\leq_{iot}$, **ioconf**, and **ioconf**$_{\mathcal{F}}$, and a sound and exhaustive test generation algorithm. The resulting theory and algorithm are somewhat simpler than the corresponding theory and algorithms for testing with symmetric interactions (e.g., compare proposition 4.3 with 3.3, and compare algorithm 6.3 with the **conf**-based test generation algorithm in [15]). The theory and the algorithm can form the basis for the development of test generation tools. They can be applied to those domains where implementations can be assumed to communicate via inputs and outputs, which is the case for many realistic systems, and where specifications can be expressed in labelled transition systems, which also holds for many specification formalisms.

It was indicated that input-output transition systems only marginally differ from input/output automata [12], having weaker requirements on input-enabling and on quiescence. We think that in a few cases these weaker requirements are easier and more intuitive. This was indicated for quiescence with the example in section 4, just above definition 4.2, but it was also indicated that for strongly-converging systems the two coincide. For a precise comparison a more elaborate investigation of divergence in input-output transition systems is necessary. The weaker requirement on input enabling allows some systems that are $\mathcal{IOTS}$ but not IOA. For example, when the communication between an IOA system and a bounded input buffer is hidden, then the whole system is not IOA anymore: when the buffer is full, no input actions are possible anymore without first performing an internal event. Such a system is $\mathcal{IOTS}$.

The model of input-output transition systems is also very much related to the model of input-output state machines [13]. The idea for the δ-trace automaton is inspired by the way the absence of output is treated in [13], but there are subtle differences in the way the δ-transitions are added.

The implementation relations and algorithm in this paper generalize those for queue systems [17]. Queue systems are transition systems in a queue context, i.e., to which two unbounded queues are attached to model asynchronous communication, one queue for inputs, and one for outputs.

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.

Another open issue is the atomicity of actions. Although we allow specifications to be labelled transition systems, the actions are classified as inputs and outputs, and they have a one-to-one correspondence to those of the implementation. An interesting area for further investigation occurs if implementation relations are combined with action refinement, so that one abstract symmetric interaction of the specification is implemented using multiple inputs and outputs, e.g., implementing an abstract interaction by means of a hand-shake protocol. Tests could be derived from the specification using symmetric algorithms (section 3) and then refined, or the specification could be refined after which the input-output based algorithm is used. The precise relation between testing, inputs and outputs, and action refinement needs further investigation.

A second open problem is the well-known test selection problem (test-suite size reduction [10]). Algorithm 6.3 can generate infinitely many sound test cases, but which ones shall be really executed? Solutions can be sought by defining coverage measures, fault models, stronger test hypotheses, etc. [1, 10, 13, 15]. 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 in test tools will probably not be feasible. A last, more practical problem is the implementation of the observation of quiescence. In practical test execution tools, timers will have to be used, for which the time-out values need to be chosen carefully, in order not to observe quiescence where there is none.

8 REFERENCES

[1] G. Bernot. Testing against formal specifications: A theoretical view. In S. Abramsky and T. S. E. Maibaum, eds., *TAPSOFT'91*, 99–119. LNCS 494, Springer-Verlag, 1991.

[2] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, eds., *Prot. Spec., Test., and Ver. VIII*, 63–74. North-Holland, 1988.

[3] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. A formal approach to conformance testing. In J. de Meer, et al., eds., *Protocol Test Systems II*, 349–363. North-Holland, 1990.

[4] ITU-T. SDL. Recommendation Z.100, 1992.

[5] R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.

[6] R. De Nicola and R. Segala. A process algebraic view of input/output automata. *TCS*, 138:391–423, 1995.

[7] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, eds., *CONCUR'90*, LNCS 458, 278–297. Springer-Verlag, 1990.

[8] ISO. Estelle – International Standard IS-9074, 1989.

[9] ISO. LOTOS – International Standard IS-8807, 1989.

[10] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Formal Methods in Conformance Testing, working draft.* September 1995.

[11] G. Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25(1):23–41, 1992.

[12] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[13] M. Phalippou. *Relations d'Implantation et Hypothèses de Test sur des Automates à Entrées et Sorties.* PhD thesis, L'Université de Bordeaux I (F), 1994.

[14] R. Segala. Quiescence, fairness, testing, and the notion of implementation. In E. Best, ed., *CONCUR'93*, 324–338. LNCS 715, Springer-Verlag, 1993.

[15] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente (NL), 1992.

[16] J. Tretmans. Testing labelled transition systems with inputs and outputs. Memorandum INF-95-26, University of Twente (NL), 1995.

[17] J. Tretmans and L. Verhaard. A queue model relating synchronous and asynchronous communication. In R.J. Linn and M.Ü. Uyar, eds., *Prot. Spec., Test., and Ver. XII*, 131–145. North-Holland, 1992.

[18] F. Vaandrager. On the relationship between process algebra and input/output automata. In *Logic in Computer Science*, 387–398. Sixth Annual IEEE Symposium, 1991.

[19] C. D. Wezeman. The CO-OP method for compositional derivation of conformance testers. In E. Brinksma, et al., eds., *Prot. Spec., Test., and Ver. IX*, 145–158. North-Holland, 1990.