Test Generation with Inputs, Outputs, and Repetitive Quiescence

Jan Tretmans
Tele-Informatics and Open Systems Group
Department of Computer Science
University of Twente
P.O. Box 217, NL-7500 AE Enschede
tretmans@cs.utwente.nl

Abstract

This paper studies testing based on labeled 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 correctness of these implementations with respect to labeled transition system specifications, are defined analogous to the theories of testing equivalence and -preorder, and refusal testing. 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 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 [Ber91, Tre92, ISO96]. Thus the test hypothesis allows to reason about implementations as if they were formal objects, and, consequently, to express the correctness of implementations with respect to specifications by a formal relation between such models of implementations and specifications. Such a relation is called an implementation relation [BAL+90, ISO96]. Conformance testing now consists of performing experiments to decide whether the unknown model of the implementation relates to the specification according to the implementation relation. 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 formal languages, e.g., ACP [BK85], CCS [Mil89], and CSP [Hoa85]. Also (most parts of) the semantics of standardized languages like LOTOS [ISO89b], SDL [CCI92], and Estelle [ISO89a] 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 system \( p \) is equivalent to a system \( q \) if any test case in the class 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 \( q \)). Such a definition of an implementation relation by explicit use of the tests and observations that can discern them, is referred to as an extensional definition. Many different relations can be defined by variations of the class of tests, the way they are executed, and the required relation between observations [DNH84, Abr87, DN87, Phi87, Gl89, Gl93].

Once an implementation relation has been defined, conformance testing involves finding a set of tests for one particular specification, that is in some sense minimal, and that can discriminate between correct and erroneous implementations of that specification. Conformance testing for labelled transition systems has been studied especially in the context of testing communication protocols with the language LOTOS, e.g., [BSS87, Bri88, PF90, We90, Led92, Tre92]. This paper uses both kinds of testing theories: first an implementation relation is defined extensionally, and then test generation from specifications for this particular relation is investigated.

Almost all of the testing theory for labelled transition systems mentioned above is based on synchronous, symmetric communication between 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 [LT89], Input-Output State Machines [Pha94], and Queue Contexts [TV92]. 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 in 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 transition systems.

The aim of this paper is to study implementation relations, conformance testing, and test gener-
ation algorithms for labelled transition systems that communicate via inputs and outputs. The
implementations are modelled by input-output transition systems, a special kind of labelled transi-
tion systems, where inputs are always enabled, and specifications are described as normal labelled
transition systems. Input-output transition systems differ only marginally from the Input/Output
Automata of [LT89]. These models are introduced in section 2. Implementation relations with
inputs and outputs are defined extensionally following the ideas of testing equivalence and refusal
testing [DNH84, DN87, Phi87, Lan90]. First, these existing relations, which are based on sym-
metric interactions, are recalled in section 3, and then their input-output versions are discussed in
section 4. The first input-output relation, called input-output testing relation, is defined following
a testing scenario à la [DNH84, DN87]. It is analogous to the scenario used in [Seg93] to ob-
tain a testing characterization of the relation quiescent trace preorder on Input/Output Automata
[Vaa91], and analogous results are obtained. The second relation, which is called input-output
refusal relation, is defined with the testing scenario for refusal testing [Phi87, Lan90]. Weaker
variants of both input-output relations are defined to allow for partial specifications. It will be
shown that all defined relations can be simply and intuitively characterized in terms of only traces,
if a special action, explicitly modelling the absence of outputs (repetitive quiescence, cf. [Vaa91]),
is added. This special action has all the properties of, and can be considered as, a normal output
action. The current paper generalizes [Seg93, Tre96a], which considered only testing preorder with
inputs and outputs, by also considering refusal testing, and by showing that all relations can be
expressed as special instances of a class of refusal-like implementation relations.

After having discussed the relevant implementation relations in section 4, section 5 starts formaliz-
ing 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. Analogous to the generalization
of implementation relations, the algorithm of section 6 generalizes the one given in [Tre96a] for
refusal testing. Some concluding remarks and open problems are discussed in section 7. Elaborated
proofs can be found in the corresponding technical report [Tre96b].
2 Models

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 actions of a system; the special label \( \tau \not\in L \) represents an unobservable, internal action. A transition \( (s, \mu, s') \in T \) is denoted as \( s \overset{\mu}{\rightarrow} s' \). A computation is a (finite) composition of transitions:

\[
s_0 \overset{\mu_1}{\rightarrow} s_1 \overset{\mu_2}{\rightarrow} s_2 \overset{\mu_3}{\rightarrow} \ldots \overset{\mu_{n-1}}{\rightarrow} s_n \overset{\mu_n}{\rightarrow} s_n
\]

A trace captures the observable aspects of a computation; it is the sequence of observable actions of a computation. The set of all finite sequences of actions over \( L \) is denoted by \( L^* \), with \( \epsilon \) denoting the empty sequence. If \( \sigma_1, \sigma_2 \in L^* \), then \( \sigma_1 \sigma_2 \) is the concatenation of \( \sigma_1 \) and \( \sigma_2 \).

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 convergent, i.e., ones that do not have infinite compositions of transitions with only internal actions. 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 \( \mu_1, \mu_2 \in L \cup \{\tau\} \), \( a_1, a_2 \in L \), and \( \sigma \in L^* \).

\[
\begin{align*}
  s \overset{\mu_1 \ldots \mu_n}{\rightarrow} s' &\overset{\text{def}}{=} \exists s_0, \ldots, s_n : s = s_0 \overset{\mu_1}{\rightarrow} s_1 \overset{\mu_2}{\rightarrow} \ldots \overset{\mu_n}{\rightarrow} s_n = s' \\
  s \overset{\mu_1 \ldots \mu_n}{\rightarrow} &\overset{\text{def}}{=} \exists s' : s \overset{\mu_1 \ldots \mu_n}{\rightarrow} s' \\
  s \overset{\mu_1 \ldots \mu_n}{\rightarrow} &\overset{\text{def}}{=} \text{not } \exists s' : s \overset{\mu_1 \ldots \mu_n}{\rightarrow} s' \\
  s \overset{\sigma}{\Rightarrow} &\overset{\text{def}}{=} \exists s' : s = s' \text{ or } s \overset{\sigma}{\Rightarrow} s' \\
  s \overset{\sigma}{\Rightarrow} &\overset{\text{def}}{=} \exists s_1, s_2 : s \overset{\sigma}{\Rightarrow} s_1 \overset{\sigma}{\Rightarrow} s_2 \overset{\sigma}{\Rightarrow} s' \\
  s \overset{\sigma}{\Rightarrow} &\overset{\text{def}}{=} \exists s_0 \ldots, s_n : s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} s_n = s' \\
  s \overset{\sigma}{\Rightarrow} &\overset{\text{def}}{=} \exists s' : s \overset{\sigma}{\Rightarrow} s' \\
  \sigma &\overset{\text{def}}{=} \text{not } \exists s' : s \overset{\sigma}{\Rightarrow} s'
\end{align*}
\]

We will not always distinguish between a transition system and its initial state: if \( p = \langle S, L, T, s_0 \rangle \), we will identify the process \( p \) with its initial state \( s_0 \), e.g., we write \( p \overset{\sigma}{\Rightarrow} \) instead of \( s_0 \overset{\sigma}{\Rightarrow} \).

Definition 2.3
1. \( \text{init}(p) \overset{\text{def}}{=} \{ \mu \in L \cup \{\tau\} \mid p \overset{\mu}{\rightarrow} \} \)
2. \( \text{traces}(p) \overset{\text{def}}{=} \{ \sigma \in L^* \mid p \overset{\sigma}{\Rightarrow} \} \)
3. \( p \text{ after } \sigma \overset{\text{def}}{=} \{ p' \mid p \overset{\sigma}{\Rightarrow} p' \} \)
4. \( p \) has finite behaviour if there is a natural number \( n \), such that all traces in \( \text{traces}(p) \) have length smaller than \( n \).
5. \( p \) is deterministic if for all \( \sigma \in L^* \), \( p \text{ after } \sigma \) has at most one element. If \( \sigma \in \text{traces}(p) \), then \( p \text{ after } \sigma \) is overloaded to denote this element.
We represent a labelled transition system in the standard way, either by a tree or a graph, where nodes represent states and edges represent transitions (e.g., figure 1), or by a process-algebraic behaviour expression, with a syntax inspired by LOTOS [ISO89b]:

\[ B = \text{def } \text{stop} | a : B | i : B | B \Box B | B \parallel B | \Sigma B | P \]

Here \( a \in L, B \) is a countable set of behaviour expressions, and \( P \in P \) is a process variable. The operational semantics of a behaviour expression with respect to an environment \( \{P := B_P | P \in P\} \) of process definitions, is given in the standard way by the following axioms and inference rules, which define for each behaviour expression, in finitely many steps, all its possible transitions (\text{stop} has no transitions, and note that not every behaviour expression represents a transition system in \( LTS(L) \), e.g., the transition system defined by \( P := i; P \) is not strongly convergent):

\[
\begin{align*}
\vdash a; B &\rightarrow B \\
\vdash i; B &\rightarrow \tau \\
B_1 &\rightarrow B'_1, \ \mu \in L \cup \{\tau\} \quad \vdash B_1 \Box B_2 &\rightarrow B'_1 \\
B_2 &\rightarrow B'_2, \ \mu \in L \cup \{\tau\} \quad \vdash B_1 \parallel B_2 &\rightarrow B'_2 \\
B_1 &\rightarrow B'_1 \\
B_2 &\rightarrow B'_2 \\
B_1 &\rightarrow B'_2, \ \mu \in L \cup \{\tau\} \quad \vdash B_1 \parallel B_2 &\rightarrow B'_1 \\
B &\rightarrow B', \ B \in B, \ \mu \in L \cup \{\tau\} \quad \vdash \Sigma B &\rightarrow B' \\
B_P &\rightarrow B', \ P := B_P, \ \mu \in L \cup \{\tau\} \quad \vdash P &\rightarrow B' 
\end{align*}
\]

Communication between a process and its environment, both modelled as labelled transition systems, is based on symmetric interaction, as expressed by the composition operator \( \parallel \). An interaction can occur if both the process and its environment are able to perform that interaction, implying that they can also both block the occurrence of an interaction. If 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 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 [LT89], 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 2.4**

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:  

$$\text{whenever} \quad p \xrightarrow{a} p' \quad \text{then} \quad \forall a \in L_I : \quad p' \xrightarrow{a} p$$

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 2.5**

Figure 1 gives some input-output transition systems with $L_I = \{\text{but}_i\}$ and $L_U = \{\text{liq}_u, \text{choc}_u\}$. 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 $\tau$-transitions, cf. definition 2.4).

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 $\parallel$, 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 [LT89]: instead of requiring strong input enabling as in [LT89] ($\forall a \in L_I : \quad p' \xrightarrow{a} p$), input-output transition systems allow input enabling via internal transitions (weak input enabling, $\forall a \in L_I : \quad p' \xrightarrow{a} p$).

---

### 3 Implementation Relations 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 recall implementation relations and conformance testing 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 $\text{imp} \subseteq \mathcal{LTS}(L) \times \mathcal{LTS}(L)$. Many different relations have been studied in the literature, e.g., bisimulation equivalence [Mil89], failure equivalence and preorder [Ho85], testing equivalence and preorder [DN84, DN87], refusal testing [Phi87], and many others [Gla89, Gla87]. A straightforward example is $\text{trace preorder} \leq_{tr}$, which requires inclusion of sets of traces. 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 $s \in \mathcal{LTS}(L)$. 

---

Figure 1: Input-output transition systems
Define 3.1
Let \( i, s \in \mathcal{LTS}(L) \), then
\[
i \leq_{tr} s \quad \overset{\text{def}}{=} \text{traces}(i) \subseteq \text{traces}(s)
\]
\[\square\]

Many implementation relations can be defined in an extensional way, which means that they are defined by explicitly comparing an implementation with a specification in terms of comparing the observations that an external observer can make of them [DNH84, DN87]. The intuition is that an implementation \( i \) correctly implements a specification \( s \), if any observation that can be made of \( i \) in any possible environment can be related to, or explained from, an observation of \( s \) in the same environment:

\[
i \text{imp} s =_{\text{def}} \forall u \in \mathcal{U} : \text{obs}(u, i) \ast \text{obs}(u, s)
\]

By varying the class of external observers \( \mathcal{U} \), the observations \( \text{obs} \) that an observer can make of \( i \) and \( s \), and the relation \( \ast \) between observations of \( i \) and \( s \), many different implementation relations can be defined.

One of the relations that can be expressed following (1) is testing preorder \( \leq_{te} \), which we formalize in a slightly different setting from the one in [DNH84, DN87]. It is obtained if labelled transition systems are chosen as observers \( \mathcal{U} \), the relation between observations is set inclusion, and the observations are traces. These traces are obtained from computations of \( i \), respectively \( s \), in parallel with an observer \( u \), where a distinction is made between normal traces and completed traces, i.e., traces which correspond to a computation after which no more actions are possible.

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

1. \( p \text{ after } \sigma \text{ refuses } A =_{\text{def}} \exists p' : p \xrightarrow{\sigma} p' \text{ and } \forall a \in A : p' \xrightarrow{a} \)
2. \( p \text{ after } \sigma \text{ deadlocks } =_{\text{def}} p \text{ after } \sigma \text{ refuses } L \)
3. The sets of observations, \( \text{obs}_c \) and \( \text{obs}_t \), respectively, that an observer \( u \in \mathcal{LTS}(L) \) can make of process \( p \in \mathcal{LTS}(L) \) are given by the completed traces and the traces, respectively, of their synchronized parallel communication \( u \parallel p \):

   \[
   \begin{align*}
   \text{obs}_c(u, p) & =_{\text{def}} \{ \sigma \in L^* \mid (u \parallel p) \text{ after } \sigma \text{ deadlocks} \} \\
   \text{obs}_t(u, p) & =_{\text{def}} \{ \sigma \in L^* \mid u \parallel p \xrightarrow{\sigma} \}
   \end{align*}
   \]

4. \( i \leq_{te} s =_{\text{def}} \forall u \in \mathcal{LTS}(L) : \text{obs}_c(u, i) \subseteq \text{obs}_c(u, s) \) and \( \text{obs}_t(u, i) \subseteq \text{obs}_t(u, s) \)
\[\square\]

The definitions in 3.2 are based on the occurrence, or absence, of observable actions. It is straightforward to show that on our class of strongly convergent transition systems these definitions correspond to those sometimes found in the literature, which also take internal actions into account:

\[
p \text{ after } \sigma \text{ refuses } A \text{ iff } \exists p' : p \xrightarrow{\sigma} p' \text{ and } \forall a \in A \cup \{\tau\} : p' \xrightarrow{a}/p
\]

The extensional definition of \( \leq_{te} \) in definition 3.2 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, is known to coincide with failure preorder for strongly convergent transition systems [DN87, Tre92].

Proposition 3.3
\( i \leq_{te} s \text{ iff } \forall \sigma \in L^*, \forall A \subseteq L : i \text{ after } \sigma \text{ refuses } A \text{ implies } s \text{ after } \sigma \text{ refuses } A \)
\[\square\]

A weaker implementation relation that is strongly related to \( \leq_{te} \) is the relation \( \text{conf} \) [BSS87]. It is a modification of \( \leq_{te} \) by restricting all observations to only those traces that are contained in the specification \( s \). This restriction is in particular used in conformance testing. It 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, \textit{conf} requires that an implementation does what it should do, not that it does not do what it is not allowed to do. So a specification only partially prescribes the required behaviour of the implementation. It is for the relation \textit{conf} that several test generation algorithms have been developed [Bri88, PF90, Wez90, Tre92].

**Definition 3.4**

\[ i \text{conf} \ s \ =_{\text{def}} \quad \forall u \in LTS(L) : \quad (\text{obs}_s(u, i) \cap \text{traces}(s)) \subseteq \text{obs}_s(u, s) \quad \text{and} \quad (\text{obs}_{\bar{s}}(u, i) \cap \text{traces}(s)) \subseteq \text{obs}_{\bar{s}}(u, s) \]

\[ \square \]

**Proposition 3.5**

\[ i \text{conf} \ s \iff \forall \sigma \in \text{traces}(s), \forall A \subseteq L : \quad i \text{after } \sigma \text{ refuses } A \text{ implies } s \text{ after } \sigma \text{ refuses } A \]

\[ \square \]

A relation with more discriminating power than testing preorder is obtained, following (1), by having more powerful observers: observers that cannot only detect the occurrence of actions, but also the absence of actions, i.e., refusals [Phi87]. We follow [Lan90] in modelling the observation of a refusal by adding a special label \( \theta \not \in L \) to observers: \( \mathcal{U} = LTS(L_\theta) \), where we write \( L_\theta \) for \( L \cup \{ \theta \} \). While observing a process, a transition labelled with \( \theta \) can only occur if no other transition is possible. In this way the observer knows that the process under observation cannot perform the other actions it offers. A parallel synchronization operator \( || \) is introduced, which models the communication between an observer with \( \theta \)-transitions and a normal process, i.e., a transition system without \( \theta \)-transitions. The implementation relation defined in this way is called refusal preorder \( \leq_{rf} \).

**Definition 3.6**

1. The operator \( || \) : \( LTS(L_\theta) \times LTS(L) \to LTS(L_\theta) \) is defined by the following inference rules:

\[ u \xrightarrow{a} u' \quad \vdash \quad u||p \xrightarrow{a} u'||p \]
\[ p \xrightarrow{a} p' \quad \vdash \quad u||p \xrightarrow{a} u'||p' \]
\[ u \xrightarrow{\theta} u', \quad p \xrightarrow{\theta} p', \quad a \in L \quad \vdash \quad u||p \xrightarrow{\theta} u'||p' \]
\[ u \xrightarrow{\theta} u', \quad p \xrightarrow{\theta} p', \quad \forall a \in L : \quad u \xrightarrow{\theta} p \quad \text{or} \quad p \xrightarrow{\theta} \quad \vdash \quad u||p \xrightarrow{\theta} u'||p \]

2. The sets of observations, \( \text{obs}_c^\theta \) and \( \text{obs}_\bar{s}^\theta \), respectively, that an observer \( u \in LTS(L_\theta) \) can make of process \( p \in LTS(L) \) are given by the completed traces and the traces, respectively, of the synchronized parallel communication \( || \) of \( u \) and \( p \):

\[ \text{obs}_c^\theta(u, p) =_{\text{def}} \quad \{ \sigma \in L_\theta^\circ \mid (u||p) \text{after } \sigma \text{ deadlocks} \} \]
\[ \text{obs}_\bar{s}^\theta(u, p) =_{\text{def}} \quad \{ \sigma \in L_\theta^\circ \mid (u||p) \xrightarrow{\sigma} \} \]

3. \( i \leq_{rf} s \quad =_{\text{def}} \quad \forall u \in LTS(L_\theta) : \quad \text{obs}_c^\theta(u, i) \subseteq \text{obs}_c^\theta(u, s) \quad \text{and} \quad \text{obs}_\bar{s}^\theta(u, i) \subseteq \text{obs}_\bar{s}^\theta(u, s) \]

\[ \square \]

A corresponding intensional characterization of refusal preorder can be given in terms of failure traces [Gla90, Lan90]. A failure trace is a trace in which both actions and refusals, represented by sets of refused actions, occur. To express this, the transition relation \( \xrightarrow{} \) is extended with refusal transitions: self-loop transitions labelled with a set of actions \( A \subseteq L \), expressing that all actions in \( A \) can be refused. The transition relation \( \xrightarrow{\varphi} \) (definition 2.2) is then extended analogously to \( \xrightarrow{\varphi} \) with \( \varphi \in (L \cup \mathcal{P}(L))^* \).

**Definition 3.7**

Let \( p \in LTS(L) \) and \( A \subseteq L \).

1. \( p \xrightarrow{A} p' =_{\text{def}} \quad p = p' \quad \text{and} \quad \forall \mu \in A \cup \{ \tau \} : \quad p \xrightarrow{\mu} \)

2. The failure traces of \( p \) are:

\[ \text{Ftraces}(p) =_{\text{def}} \quad \{ \varphi \in (L \cup \mathcal{P}(L))^* \mid p \xrightarrow{\varphi} \} \]

\[ \square \]
We conclude this section with relating the implementation relations based on symmetric interactions, and with illustrating them using the candy machines over $L = \{\text{but}, \text{choc}, \text{liq}, \text{bang}\}$ in figure 2. These examples also illustrate the inequalities of proposition 3.9.

**Figure 2**: Implementation relations with symmetric interactions

**Proposition 3.9**

1. $\leq_{tr}, \leq_{tr}, \leq_{rf}$ are preorders; $\text{conf}$ is reflexive, but not transitive.
2. $\leq_{rf} \subseteq \leq_{tr} = \leq_{tr} \cap \text{conf}$

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 $\text{imp} \subseteq TOTS(L_I, L_U) \times LTS(L_I \cup L_U)$. Like the relations based on symmetric interactions in section 3, we define them extensionally following (1).

4.1 Input-output testing relation

The implementation relations $\leq_{te}$ and $\text{conf}$ were defined by relating the observations, made of the implementation by a symmetrically interacting observer $u \in LTS(L)$, 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 \( IOTS(L_I, L_U) \) communicates with an 'output-input' observer in \( IOTS(L_U, L_I) \). In this way the input-output testing relation \( \leq_{int} \) is defined between \( i \in IOTS(L_I, L_U) \) and \( s \in LTS(L_I \cup L_U) \) 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**

Let \( i \in IOTS(L_I, L_U) \), \( s \in LTS(L_I \cup L_U) \), then

\[
i \leq_{int} s \iff \forall u \in IOTS(L_U, L_I) : \text{obs}_c(u, i) \subseteq \text{obs}_c(u, s) \text{ and } \text{obs}_t(u, i) \subseteq \text{obs}_t(u, s)
\]

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 \( \text{obs}_c \), \( \text{obs}_t \), and \( \text{obs}_b \), and after deadlocks apply to labelled transition systems, not only to input-output transition systems. Below we will elaborate on this possibility to have \( s \not\in IOTS \).

In [Seg93] the testing scenario of testing preorder [DNH84, DN87] 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 [Vaa91]. Although we are more liberal with respect to the specification, \( s \in LTS(L_I \cup L_U) \), and input-output transition systems differ marginally from Input/Output Automata, exactly the same intensional characterization is obtained: \( \leq_{int} \) is fully characterized by trace inclusion and inclusion of quiescent traces. A trace is quiescent if it may lead to a state from which the system cannot proceed autonomously, without inputs from its environment, i.e., a state from which no outputs or internal actions are possible.

**Definition 4.2**

Let \( p \in LTS(L_I \cup L_U) \).

1. A state \( s \) of \( p \) is quiescent, denoted by \( \delta(s) \), if \( \forall u \in L_U \cup \{r\} : s \xrightarrow\delta u \).
2. A quiescent trace of \( p \) is a trace \( \sigma \) that may lead to a quiescent state: \( \exists s' \in p \text{after } \sigma : \delta(s') \).
3. The set of quiescent traces of \( p \) is denoted by \( Qtraces(p) \).

**Proposition 4.3**

\( i \leq_{int} s \iff \text{traces}(i) \subseteq \text{traces}(s) \text{ and } Qtraces(i) \subseteq Qtraces(s) \)

**Sketch of the proof**

Comparing with the analogous definition and proposition for \( \leq_{te} \) (definition 3.2 and proposition 3.3) we see that the observation of deadlock of \( u \parallel i \) can only occur if \( i \) is in a state where it cannot produce any output (a quiescent state), and \( u \) is in a state where it cannot produce any input (input with respect to \( i \)). It follows then that for inclusion of \( \text{obs}_c \) it is sufficient to consider only quiescent traces. Inclusion of \( \text{obs}_t \) corresponds to inclusion of traces.

Comparing the intensional characterization of \( \leq_{int} \) 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.

Another characterization of \( \leq_{int} \) can be given by collecting for each trace all possible outputs that a process may produce after that trace, including a special action \( \delta \) that indicates possible quiescence. The special label \( \delta \not\in L \) indicates the absence of output actions in a state, i.e., it makes the observation of absence of outputs (quiescence) into an explicitly observable event. Proposition 4.5 then states that an implementation is correct according to \( \leq_{int} \) if all outputs it can produce after
any trace $\sigma$ can also be produced by the specification. Since this also holds for $\delta$, the implementation may show no outputs only if the specification can do so.

**Definition 4.4**
Let $p$ be a state in a transition system, and let $P$ be a set of states, then

1. $\text{out}(p) \stackrel{\text{def}}{=} \{ x \in L_U \mid p \xrightarrow{x} \} \cup \{ \delta \mid \delta(p) \}$
2. $\text{out}(P) \stackrel{\text{def}}{=} \bigcup \{ \text{out}(p) \mid p \in P \}$

**Proposition 4.5**

$i \leq_{i.o} s$ iff $\forall \sigma \in L^* : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$

**Sketch of the proof**
Using the facts that $\sigma \in Q\text{traces}(p)$ iff $\delta \in \text{out}(p \text{ after } \sigma)$ and $\sigma \in \text{traces}(p)$ iff $\text{out}(p \text{ after } \sigma) \neq \emptyset$, the proposition follows immediately from proposition 4.3.

![Figure 3: Two non-input-output specifications](image)

**Example 4.6**
Using proposition 4.5, it follows that $q_1 \leq_{i.o} q_2$ (figure 1): 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 $\text{but}_i ; \text{liquoric e} ; \text{stop} \sqcup \text{but}_i ; \text{choc} ; \text{stop}$ (omitting the input self-loops), an equivalence which does not hold for $\leq_{e}$ 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:

$\text{liquoric e } \sqcup \text{ stop } ; \text{ but}_i ; \text{stop} \sqcup \text{ but}_i ; \text{choc} ; \text{stop}$

On the other hand, $q_2 \not\leq_{i.o} q_1, q_3$: if the specification prescribes to produce only *liquorice*, then an implementation should not have the possibility to produce *chocolate*. $\text{choc} \in \text{out}(q_2 \text{ after } \text{but}_i)$, while $\text{choc} \not\in \text{out}(q_1 \text{ after } \text{but}_i)$, $\text{choc} \not\in \text{out}(q_3 \text{ after } \text{but}_i)$. We have $q_1 \leq_{i.o} q_3$, but $q_3 \not\leq_{i.o} q_1, q_2$. 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: $\delta \in \text{out}(q_3 \text{ after } \text{but}_i)$, while $\delta \not\in \text{out}(q_1 \text{ after } \text{but}_i), \text{out}(q_2 \text{ after } \text{but}_i)$.

Figure 3 presents two non-input-output transition system specifications, 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: $\text{out}(q_1 \text{ after } \text{but}_i \text{but}_i), \text{out}(q_2 \text{ after } \text{but}_i \text{but}_i), \text{out}(q_3 \text{ after } \text{but}_i \text{but}_i) \neq \emptyset$, while $\text{but}_i \text{but}_i \not\in \text{traces}(s_1), \text{traces}(s_2)$, so $\text{out}(s_1 \text{ after } \text{but}_i \text{but}_i), \text{out}(s_2 \text{ after } \text{but}_i \text{but}_i) = \emptyset$.

For the relation $\leq_{i.o}$ 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 specifier does not care about the responses of an implementation to 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 4.6. The implementation freedom, intended with non-specified inputs, cannot be expressed with the relation $\leq_{iot}$. From proposition 4.5 the reason can be deduced: since any implementation can always perform any sequence of input actions, and since from definition 4.4 it is easily deduced that $\text{out}(p \text{ after } \sigma) \neq \emptyset$ iff $p \xrightarrow{\sigma} q$, we have that an $\leq_{iot}$-implementable specification must at least be able to perform any sequence of input actions. So the specification must be an input-output transition system, too, otherwise no $\leq_{iot}$-implementation can exist.

For Input/Output Automata a solution to this problem is given in [DNS95], using the so-called demonic semantics for process expressions. In this semantics a transition to a demonic process $\Omega$ is added for each non-specified input. Since $\Omega$ 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. The discussion above immediately suggests how such a relation can be defined: instead of requiring inclusion of $\text{out}$-sets for all traces in $L^*$ (proposition 4.5), the weaker relation requires only inclusion of $\text{out}$-sets for traces that are explicitly specified in the specification. This relation is called $i/o$-conformance $\text{ioconf}$, and, analogous to $\text{conf}$, it allows partial specifications which only state requirements for traces explicitly specified in the specification (cf. the relation between $\leq_{te}$ and $\text{conf}$, definitions 3.2 and 3.4, and propositions 3.3 and 3.5).

**Definition 4.7**

Let $i \in IOTS(L_I, L_U)$, $s \in LTS(L_I \cup L_U)$, then

\[
i \text{i oconf} s =_{\text{def}} \forall \sigma \in \text{traces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)
\]

\[\square\]

**Example 4.8**

Consider again figures 1 and 3. Indeed, we have $q_1 \text{ i oconf} s_1$, whereas we had $q_1 \not\leq_{iot} s_1$. According to $\text{ioconf}$, $s_1$ specifies only that after one button, liquorice must be produced; with $\text{ioconf} s_1$ does not care what happens if the button is pushed twice, as was the case with $\leq_{iot}$.

On the other hand, $q_2 \text{ i oconf} s_1$, since $q_2$ can produce more than liquorice after the button has been pushed: $\text{out}(q_2 \text{ after } \text{but}_1) = \{\text{liq}_u, \text{choc}_u\} \not\subseteq \{\text{liq}_u\} = \text{out}(s_1 \text{ after } \text{but}_1)$. Moreover, $q_1, q_2 \text{ i oconf} s_2$, but $q_3 \text{ i oconf} s_1, s_2$, since $\delta \in \text{out}(q_3 \text{ after } \text{but}_1)$, while $\delta \not\in \text{out}(s_1 \text{ after } \text{but}_1)$, $\text{out}(s_2 \text{ after } \text{but}_1)$.

\[\square\]

### 4.2 Input-output refusal relation

We have seen implementation relations with symmetric interactions based on observers without, and with $\theta$-label, which resulted in the relations $\leq_{te}$ and $\leq_{rf}$, respectively, and we have seen an implementation relation with inputs and outputs based on observers without $\theta$-label. Naturally, the next step is an implementation relation with inputs and outputs based on observers with the power of the $\theta$-label. The resulting relation is called the *input-output refusal relation* $\leq_{ior}$.

**Definition 4.9**

Let $i \in IOTS(L_I, L_U)$, $s \in LTS(L_I \cup L_U)$, then
A quiescent trace was introduced as a trace ending in the absence of outputs. Taking the special action \( \delta \) which was used in \( \text{out}-\text{sets} \) to indicate the absence of outputs, a quiescent trace \( \sigma \in L^* \) can be written as a \( \delta \)-ending trace \( \sigma \cdot \delta \in (L \cup \{\delta\})^* \). Here, the special action \( \delta \) appears always as the last action in the trace. If this special action \( \delta \) is now treated as a completely normal action, which can occur at any place in a trace, we obtain traces with repetitive quiescence. For example, the trace \( \delta \cdot a \cdot \delta \cdot b \cdot x \) would mean intuitively that initially no outputs can be observed, then after input action \( a \) there is again no output, and then after input \( b \) is performed the output \( x \) can be observed. We write \( L_\delta \) for \( L \cup \{\delta\} \), and we call traces over \( L_\delta \) suspension traces. The implementation relation \( \leq_{i_r} \) turns out to be characterized by inclusion of these suspension traces (and hence it could also be called repetitive quiescence relation). Since quiescence corresponds to a refusal of \( L_U \) (definition 3.7), it follows that suspension traces are exactly the failure traces in which only \( L_U \) occurs as refusal set, i.e., failure traces restricted to \( (L \cup \{L_U\})^* \), and where \( \delta \) is written for the refusal \( L_U \).

**Definition 4.10**

The suspension traces of process \( p \in \text{LTS}(L) \) are:

\[
\operatorname{Straces}(p) = \text{def } \text{Ftraces}(p) \cap (L \cup \{L_U\})^*.
\]

For \( L_U \) occurring in a suspension trace we write \( \delta \), so that a suspension trace \( \sigma \in L_\delta^* \).

**Proposition 4.11**

\[
i \leq_{i_r} s \iff \operatorname{Straces}(i) \subseteq \operatorname{Straces}(s)
\]

**Sketch of the proof**

Analogous to the proof of proposition 4.3, and comparing with the corresponding situation for \( \leq_{i_f} \) (definition 3.6 and proposition 3.8), we have that a refusal can only be observed if \( i \) is in a state where it cannot produce any output (a quiescent state), and \( u \) is in a state where it cannot produce any input (input with respect to \( i \)). So the only refusals of \( i \) that can be observed are \( L_U \). As opposed to proposition 4.3, the normal traces are included in the suspension traces, so they need not be mentioned separately in the proposition.

An intensional characterization of \( \leq_{i_r} \) in terms of \( \text{out}-\text{sets} \), analogous to proposition 4.5, is easily given by generalizing the definition of \( \text{after} \) (definition 2.3) in a straightforward way to suspension traces.

**Proposition 4.12**

\[
i \leq_{i_r} s \iff \forall \sigma \in L_\delta^* : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)
\]

Again, completely analogous to the definitions of \( \text{conf} \) and of \( \text{iocconf} \), an implementation relation, called \( \text{ioco} \), is defined by restricting inclusion of \( \text{out}-\text{sets} \) to suspension traces of the specification.

**Definition 4.13**

\[
i \text{ioco } s \equiv \text{def } \forall \sigma \in \operatorname{Straces}(s) : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)
\]

**Example 4.14**

Examples 4.6 and 4.8 illustrated the implementation relations \( \leq_{i_o} \) and \( \text{iocconf} \), respectively, using the processes in figures 1 and 3. Replacing \( \leq_{i_o} \) by \( \leq_{i_r} \), and \( \text{iocconf} \) by \( \text{ioco} \), the same results are obtained for the processes in figures 1 and 3.

The difference between \( \leq_{i_o} \) and \( \leq_{i_r} \), and between \( \text{iocconf} \) and \( \text{ioco} \) is illustrated with the processes \( r_1 \) and \( r_2 \) in figure 4: \( r_1 \leq_{i_o} r_2 \), but \( r_1 \not\leq_{i_r} r_2 \); \( \text{out}(r_1 \text{ after } \text{but}, \delta \cdot \text{but}_d) = \{\text{liq}_u, \text{choc}_u\} \) and \( \text{out}(r_2 \text{ after } \text{but}_d, \delta \cdot \text{but}_d) = \{\text{choc}_u\} \). Intuitively, after pushing the \text{button}, we observe that nothing is produced by the machine, so we push the \text{button} again. Machine \( r_1 \) may then produce either
liquorice or chocolate, while machine $r_2$ will always produce chocolate. When using the relation $\leq_{loi}$ the observation always terminates after observing that nothing is produced. Hence, there is no way to distinguish between entering the left or the right branch of $r_1$ or $r_2$; after pushing the button twice both machines may produce either liquorice or chocolate: $\text{out}(r_{1,2} \text{ after but}_1 \text{ but}_2) = \{\text{liq}_u, \text{choc}_u\}$.

\[ \]

### 4.3 Relating relations with inputs and outputs

Two kinds of observations were used in the extensional definitions of testing preorder $\leq_{te}$ (definition 3.2), refusal preorder $\leq_{rf}$ (definition 3.6), the input-output testing relation $\leq_{iot}$ (definition 4.1), and the input-output refusal relation $\leq_{ior}$ (definition 4.9): the traces and the completed traces of the composition of a process and an observer, expressed by $\text{obs}_t(u,p)$ and $\text{obs}_c(u,p)$, respectively. The varying parameters in defining these four relations were the distinction between inputs and outputs (and associated input-enabledness), and the ability to observe refusals by adding the $\theta$-label to the class of observers.

<table>
<thead>
<tr>
<th></th>
<th>$u \in \text{LTS}(L)$</th>
<th>$u \in \text{LTS}(L_\theta)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>no inputs and no outputs</td>
<td>$\leq_{te}$ or $\leq_{rf}$</td>
<td>$\leq_{rf}$ or $\leq_{oi}$</td>
</tr>
<tr>
<td>inputs and outputs</td>
<td>$\leq_{iot}$ or $\leq_{ior}$</td>
<td>$\leq_{ior}$</td>
</tr>
</tbody>
</table>

Table 1: Observations $\text{obs}_c$ and $\text{obs}_t$

Although all four relations were defined by requiring inclusion of both $\text{obs}_c$ and of $\text{obs}_t$, some of the relations only need observations of one kind. This is indicated in table 1 by mentioning the necessary and sufficient observations. Adding the ability to observe refusals to observers, using the $\theta$-action, makes observation of completed traces with $\text{obs}_c$ superfluous: for $\leq_{rf}$ and $\leq_{ior}$ it suffices to consider observations of the kind $\text{obs}_t$. If no distinction between inputs and outputs is made, any observation of a trace in $\text{obs}_c$ can always be simulated in $\text{obs}_t$ with an observer which can perform only this particular trace and then terminates: for $\leq_{te}$ and $\leq_{rf}$ it suffices to consider observations of the kind $\text{obs}_c$. Only for $\leq_{iot}$ both kinds of observations are necessary, as shows the example in figure 5. Let $L_I = \emptyset$ and $L_U = \{x, y\}$, then, to define both intuitively incorrect implementations $i_1$ and $i_2$ as not $\leq_{iot}$-related, we need both $\text{obs}_c$ and $\text{obs}_t$: 

$$
The input-output implementation relations defined so far, viz. \( \leq_{iol} \), \( \mathit{ioconf} \), \( \leq_{ior} \), and \( \mathit{ioco} \), are easily related using their characterizations in terms of \( \mathit{out} \)-sets. The only difference between the relations is the set of (suspension) traces for which the \( \mathit{out} \)-sets are compared, cf. propositions 4.5 (\( \leq_{iol} \)) and 4.12 (\( \leq_{ior} \)), and definitions 4.7 (\( \mathit{ioconf} \)) and 4.13 (\( \mathit{ioco} \)). So, if we introduce the following class of relations \( \mathit{ioco}_F \) with \( F \subseteq L_i^g \):

\[
i \mathit{ioco}_F \ s = \mathsf{def} \ \forall \sigma \in F \colon \mathit{out}(i \ \mathsf{after} \ \sigma) \subseteq \mathit{out}(s \ \mathsf{after} \ \sigma)
\]

then they can all be expressed as instances of \( \mathit{ioco}_F \):

\[
\begin{align*}
\leq_{iol} &= \mathit{ioco}_{L_i} & \mathit{ioconf} &= \mathit{ioco}_{\mathit{trace}}(s) \\
\leq_{ior} &= \mathit{ioco}_{L_i^g} & \mathit{ioco} &= \mathit{ioco}_{\mathit{Strace}}(s)
\end{align*}
\]

Using (3) and (4) the input-output implementation relations are easily related by relating their respective sets \( F \) (proposition 4.15). The inequalities follow from the candy machines in examples 4.8 and 4.14. The generalized implementation relation \( \mathit{ioco}_F \) is the relation for which conformance testing and test derivation will be studied in section 6.

**Proposition 4.15**

\[
\leq_{ior} \subset \left\{ \leq_{iol}, \mathit{ioco} \right\} \subset \mathit{ioconf}
\]

\[\square\]

### 4.4 Suspension automata

The characterizations of \( \leq_{iol} \) in proposition 4.3, and of \( \leq_{ior} \) in proposition 4.11 in terms of traces suggest to transform a labelled transition system into another one representing exactly the respective sets of traces, so that the relations can be characterized by trace preorder \( \leq_t \) (definition 3.1) on the results of this transformation. Such a transformed transition system can be obtained by taking the deterministic transition system representing exactly these sets of traces, if care is taken to correctly include possible quiescence. For \( \leq_{ior} \) such a transition system is referred to as the *suspension automaton* \( \Gamma \), and it is obtained from a transition system by adding loops \( s \xrightarrow{\delta} s \) for all quiescent states, and then determinizing the resulting automaton. It is easily seen that the implementation relation \( \leq_{ior} \) reduces to trace preorder \( \leq_t \) on suspension automata. Moreover, \( \mathit{out} \)-sets can be directly transposed to suspension automata.

**Definition 4.16**

Let \( L \) be partitioned into \( L_I \) and \( L_U \), and let \( p = \langle S, L, T, s_0 \rangle \in \mathit{LTS}(L) \) be a labelled transition system, then the *suspension automaton* of \( p, \Gamma_p \), is the labelled transition system \( \langle S_\delta, L_\delta, T_\delta, q_0 \rangle \in \mathit{LTS}(L_\delta) \), where

\[
\begin{align*}
S_\delta &= \mathsf{def} \ {\mathcal{P}}(S) \wedge \emptyset \quad (\mathcal{P}(S) \text{ is the powerset of } S, \text{i.e., the set of its subsets}) \\
T_\delta &= \mathsf{def} \{ q \xrightarrow{a} q' \mid a \in L_I \cup L_U, \ q, q' \in S_\delta, \ q' = \{s' \in S \mid \exists s \in q : s \xrightarrow{a} s'\} \} \\
&\quad \cup \{ q \xrightarrow{\delta} q' \mid q, q' \in S_\delta, \ q' = \{s \in q \mid \delta(s)\} \}
\end{align*}
\]
Proposition 4.17
Let \( p \in \mathcal{LTSA}(L) \) with inputs in \( L_I \) and outputs in \( L_U \), let \( \sigma \in L_s^\delta \), and consider \( \delta \) as an output action of \( \Gamma_p \), i.e., \( \Gamma_p \) has inputs in \( L_I \) and outputs in \( L_U \cup \{\delta\} \), then

1. \( \Gamma_p \) is deterministic.
2. \( \text{traces}(\Gamma_p) = \text{Straces}(p) \)
3. \( \text{out}(\Gamma_p \text{after } \sigma) = \text{out}(p \text{after } \sigma) \)
4. \( \sigma \in \text{traces}(\Gamma_p) \) iff \( \text{out}(\Gamma_p \text{after } \sigma) \neq \emptyset \)

Sketch of the proof
The first term of \( T_\delta \) in definition 4.16 corresponds to the standard algorithm for determinization of automata. The second term adds \( \delta \)-transitions for states in \( S_\delta \) containing a quiescent state, thus precisely creating the suspension traces of \( p \), while not affecting determinism. The third part follows from the fact that \( \Gamma_p \text{after } \sigma = p \text{after } \sigma \), and the last part is clear from the construction of \( \Gamma'_p \); if there is no transition labelled with an output from \( q \in S_\delta \), then there must be at least one quiescent state in \( q \), so a \( \delta \)-transition is added.

Corollary 4.18
\( i \leq_{ior} s \) iff \( \Gamma_i \leq_{tr} \Gamma_s \)

Figure 6: Suspension automata for figure 1

So we have that \( \leq_{ior} \) is completely characterized by \( \leq_{tr} \) on the corresponding suspension automata. Using (3) and (4) also the other implementation relations can be expressed in suspension automata restricting to suitable sets of traces, e.g., to \( L^\varepsilon \cdot \{\varepsilon, \delta\} \) for \( \leq_{iot} \). In [Tre96a] a variant of the suspension automaton (called \( \delta \)-trace automaton) was defined to characterize \( \leq_{iot} \) directly. In that automaton transitions of the form \( q \xrightarrow{\delta} \text{stop} \) were added for quiescent states, so that its traces are automatically restricted to \( L^\varepsilon \cdot \{\varepsilon, \delta\} \).

Suspension automata are deterministic transition systems so that the transition relations \( \xrightarrow{\sigma} \) and \( \xrightarrow{\sigma} \) coincide, and each trace \( \sigma \) always goes to a unique state, denoted by \( \Gamma_p \text{after } \sigma \). The action \( \delta \), modelling quiescence, occurs as an explicit action in suspension automata. The action \( \delta \) has all the properties of an output action. This leads us to the conclusion that input-output transition systems can be considered modulo trace equivalence, if quiescence is added as an additional, observable output action. Because of these properties, the suspension automaton of a specification will be the basis for the derivation of tests in section 6.
Example 4.19
Figures 6 and 7 show the suspension automata for the processes of figures 1 and 4, respectively. For $\Gamma_{q_3}$ 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 $\delta$-transition, since there is a state in $\{s_1, s_2\}$, i.e. $s_2$, that refuses all outputs. Using corollary 4.18 it is now easily checked that $r_2 \leq_{ior} r_1$, but $r_1 \not\leq_{ior} r_2$ (example 4.14).

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 $\text{ioco}_F$, we can start the discussion of conformance testing. This means that the statement of (1) is reversed: instead of defining an implementation relation by choosing a set of observers, we look for a minimal (or at least reduced) set of observers or test cases, which fully characterizes all $\text{ioco}_F$-correct implementations of a given specification. The first point of discussion is how these test cases look like, how they are executed, and when they are successful.

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. In particular, since we consider the relations $\text{ioco}_F$, tests will be processes in $\text{CTS}(L_I \cup L_O \cup \{\theta\})$. 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 from the perspective of 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 a state of a test case either is a terminal state, or offers one particular input to the implementation, or accepts all possible outputs from the implementation, including the $\delta$-action which is accepted by a $\theta$-action in the
test case. Finally, to be able to decide about the success of a test, the terminal states of a test case are labelled with the verdict pass or fail. Altogether, we come to the following definition of a test case.

**Definition 5.1**

1. A **test case** $t$ is a labelled transition system $(S, L_I \cup L_U \cup \{\theta\}, T, s_0)$ such that
   - $t$ is deterministic and has finite behaviour;
   - $S$ contains the terminal states pass and fail, with $\text{init}(\text{pass}) = \text{init}(\text{fail}) = \emptyset$;
   - for any state $t' \in S$ of the test case, $t' \neq \text{pass}, \text{fail}$, either $\text{init}(t') = \{a\}$ for some $a \in L_I$, or $\text{init}(t') = L_U \cup \{\theta\}$.

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

Note that $L_I$ and $L_U$ refer to the inputs and outputs from the point of view of the implementation under test, so $L_I$ denotes the outputs, and $L_U$ the inputs of the test case.

A test run of an implementation with a test case is modelled by the synchronous parallel execution $[\cdot]$ 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). Since for each state $t' \neq \text{pass, fail}$ of a test case either $\text{init}(t') = \{a\}$ for some $a \in L_I$, in which case no deadlock can occur because of input-enabledness of implementations, or $\text{init}(t') = L_U \cup \{\theta\}$, in which case no deadlock can occur because of the $\theta$-transition, it follows that deadlock can only occur in the terminal states pass and fail. An implementation passes a test run if and only if the test run deadlocks in pass. Since an implementation can behave nondeterministically different test runs of the same test case with the same implementation may lead to different terminal 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 $t \in \text{TEST}(L_U, L_I)$ with an implementation $i \in \text{OTS}(L_I, L_U)$ is a trace of the synchronous parallel composition $t \parallel i$ leading to a terminal state of $t$:

   $$\sigma$$ is a test run of $t$ and $i =_{def} \exists t': t \parallel i \overset{\sigma}{\Rightarrow} \text{pass} \lor t \parallel i \overset{\sigma}{\Rightarrow} \text{fail}$

2. An implementation $i$ **passes** a test case $t$, if all their test runs lead to the pass-state of $t$:

   $$i \text{ passes } t =_{def} \forall \sigma \in L_I^*, \forall t': t \parallel i \overset{\sigma}{\Rightarrow} \text{pass} \lor t \parallel i \overset{\sigma}{\Rightarrow} \text{fail}$$

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

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

If $i$ does not pass the test suite, it **fails**: $i \text{ fails } T =_{def} \exists t \in T : i \text{ fails } t$.

**Example 5.3**

For $r_1$ (figure 4) there are three test runs with $t$ in figure 8:

$$t \parallel r_1 \overset{\text{pass}}{\Rightarrow} r'_1$$
$$t \parallel r_1 \overset{\theta; \text{fail}}{\Rightarrow} r''_1$$
$$t \parallel r_1 \overset{\theta; \text{pass}}{\Rightarrow} r''_1$$

where $r'_1$, $r''_1$, and $r''_1$ are the leaves of $r_1$ from left to right. Since the terminal state of $t$ for the second run is fail, we have $r_1$ **fails** $t$. Similarly, it can be checked that $r_2$ **passes** $t$. 

\[18\]
6 Test Generation for Input-Output Transition Systems

Now all ingredients are there to present an algorithm to generate test suites from labelled transition system specifications for any of the implementation relations $\text{ioco}_R$. A generated test suite must test implementations for conformance with respect to $s$ and $\text{ioco}_R$. Ideally, an implementation should pass the test suite if and only if it is conforming. In this case the test suite is called complete [ISO96]. 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 $\text{ioco}_R$:

- $T$ is complete if for all $i$: $\text{ioco}_R s$, iff $i$ passes $T$
- $T$ is sound if for all $i$: $\text{ioco}_R s$, implies $i$ passes $T$
- $T$ is exhaustive if for all $i$: $\text{ioco}_R s$, if $i$ passes $T$

We aim at producing sound test suites from a specification $s$, and we use for that purpose the suspension automaton $\Gamma_s$. To get some idea how such test cases will look like we consider the definition of $\text{ioco}$ in terms of suspension automata (definition 4.13 with proposition 4.17.3). We see that to test for $\text{ioco}$ we have to check for each $\sigma \in \mathcal{F} \subseteq L_\delta^*$ whether $\text{out}(\Gamma, \text{after } \sigma) \subseteq \text{out}(\Gamma_s, \text{after } \sigma)$. Basically, this can be done by having a test case $t$ that executes $\sigma$:

$$t \mid i \xrightarrow{\sigma} t' \mid i'$$

and then checks $\text{out}(\Gamma, \text{after } \sigma)$ by having transitions to pass-states for all allowed outputs (those in $\text{out}(\Gamma_s, \text{after } \sigma)$), and transitions to fail-states for all erroneous outputs (those not in $\text{out}(\Gamma_s, \text{after } \sigma)$). Special care should be taken for the special output $\delta$: $\delta$ actually models the absence of any output, so no transition will be made by the implementation if $i'$ 'outputs' $\delta$. This matches a $\theta$-transition in the test case, which exactly occurs if nothing else can happen. This $\theta$-transition will go the pass-state if quiescence is allowed ($\delta \in \text{out}(\Gamma_s, \text{after } \sigma)$), and to the fail-state if the specification does not allow 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 6.3.1), and all possible test cases
together form an exhaustive (and thus complete) test suite (theorem 6.3.2), 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
different test cases, which can be arbitrarily long, but which all have finite behaviour.

In the algorithm we use the notation $\overline{\sigma}$ for a trace in which all $\delta$-actions have been replaced by $\theta$-
actions (and others left unchanged), or vice versa, depending on the domain of $\sigma$. The interchange
of $\delta$- and $\theta$-actions is natural since both are in a certain sense complementary: $\delta$ models the absence
of outputs (the refusal $L_U$) which can be observed with $\theta$, and which is the only refusal which can
be observed when dealing with input-output transition systems (cf. complementary actions $a$ and
$\pi$ in CCS [Mil89]).

Algorithm 6.2
Let $\Gamma$ be the suspension automaton of a specification, and let $\mathcal{F} \subseteq L_0^\infty$, then a test case $t \in$
$\mathcal{T E S T}(L_U, L_I)$ is obtained by a finite number of recursive applications of one of the following three
nondeterministic choices:

1. (⋆ terminate the test case ⋆) 
   \[
   t := \text{pass} ;
   \]
2. (⋆ give a next input to the implementation ⋆) 
   \[
   t := \; a \; ; \; t' ;
   \]
   where $a \in L_I$, such that $\mathcal{F}' = \{ \sigma \in L_0^\infty \mid a \cdot \sigma \in \mathcal{F} \} \neq \emptyset$, and $t'$ is obtained by recursively
   applying the algorithm for $\mathcal{F}'$ and $\Gamma'$, with $\Gamma \leadsto \Gamma'$;
3. (⋆ check the next output of the implementation ⋆) 
   \[
   t := \sum \{ x \; ; \; \text{fail} \mid x \in L_U \cup \{ \emptyset \}, \; \sigma \notin \text{out}(\Gamma), \; x \in \mathcal{F} \} \]
   \[
   \bigoplus \sum \{ x \; ; \; \text{pass} \mid x \in L_U \cup \{ \emptyset \}, \; \sigma \notin \text{out}(\Gamma), \; x \in \mathcal{F} \} \]
   \[
   \bigoplus \sum \{ x \; ; \; t_x \mid x \in L_U \cup \{ \emptyset \}, \; \sigma \notin \text{out}(\Gamma) \} \}
   \]
   where $t_x$ is obtained by recursively applying the algorithm for $\{ \sigma \in L_0^\infty \mid x \cdot \sigma \in \mathcal{F} \}$ and $\Gamma'$,
   with $\Gamma \leadsto \Gamma'$.

Theorem 6.3
1. A test case obtained with algorithm 6.2 from $\Gamma_s$ and $\mathcal{F}$ is sound for $s$ with respect to $\text{ioço}_\mathcal{F}$.
2. The set of all possible test cases that can be obtained with algorithm 6.2 is exhaustive.

Sketch of the proof
1. Soundness can be proved using the following sufficient condition for soundness of a test case
   $t \in \mathcal{T E S T}(L_U, L_I)$ for a specification $s$ with respect to $\text{ioço}_\mathcal{F}$:
   \[
   \forall \sigma \in L_0^\infty : \quad t \overset{\sigma}{\rightarrow} \text{fail} \quad \implies \quad \exists \sigma' \in \mathcal{F}, \; x \in L_U \cup \{ \emptyset \} : \sigma = \overline{\sigma' \cdot x} \quad \text{and} \quad x \notin \text{out}(\Gamma_s \quad \text{after} \quad \sigma') \tag{5}
   \]
   This property is proved by contradiction: let $t$ be not sound, then $\exists i : \ i \overset{\text{ioço}_\mathcal{F}}{\rightarrow} s$, and
   $t \mid i \overset{\text{fail}}{\rightarrow} j \mid i'$. It follows that $t \overset{\sigma}{\rightarrow} \text{fail}$ and $i \overset{\text{fail}}{\rightarrow} i'$, so from the premiss: $\exists \sigma' \in \mathcal{F}, \; x \in L_U \cup \{ \emptyset \} : \sigma = \overline{\sigma' \cdot x} \quad \text{and} \quad x \notin \text{out}(\Gamma_s \quad \text{after} \quad \sigma')$. But since $i \overset{\sigma'}{\rightarrow} x \quad \text{and} \quad i \overset{\text{ioço}_\mathcal{F}}{\rightarrow} s$ we have
   $x \in \text{out}(\Gamma_s \quad \text{after} \quad \sigma')$, so a contradiction.
   By straightforward induction on the structure of $t$ it is then proved that each $t$ generated
with algorithm 6.2 from $\Gamma_s$ and $\mathcal{F}$ satisfies property (5).
2. For exhaustiveness we have to show that the set of all test cases $\mathcal{T}$ generated with al-
gorithm 6.2 satisfies $\forall t : \ i \overset{\text{ioço}_\mathcal{F}}{\rightarrow} s \quad \implies \quad \exists \mathcal{T} \in \mathcal{T} : \ i \overset{\text{fails}}{\rightarrow} t$. So let $\sigma \in \mathcal{F}$ such that
   $\text{out}(i \quad \text{after} \quad \sigma) \notin \text{out}(s \quad \text{after} \sigma)$, so $\exists z \in \text{out}(i \quad \text{after} \sigma)$ with $z \notin \text{out}(s \quad \text{after} \sigma)$. A test
case $t_{[\sigma]}$ can be constructed with algorithm 6.2 from $\Gamma_s$ and $\mathcal{F}$ as follows:
\(t_{[\sigma]}\) is obtained with the third choice in algorithm 6.2, followed by the first choice for each \(t_x\);

- \(t_{[\sigma,a]}\) (\(b \in L_I\)) is obtained with the second choice in algorithm 6.2, choosing \(a = b\), and followed by recursive application to obtain \(t' = t_{[\sigma]}\);

- \(t_{[\sigma,a]}\) (\(y \in L_I \cup \{\theta\}\)) is obtained with the third choice in algorithm 6.2, followed by the first choice for each \(t_x\) with \(x \neq y\), and recursive application to obtain \(t_y = t_{[\sigma]}\).

Now it can be shown that \(t_{[\sigma]}|i \xrightarrow{\sigma} t_{[c]}|' i' \xrightarrow{\sigma} \text{fail}|'\), so \(i\) fails \(t_{[\sigma]}\): \(\Box\)

**Example 6.4**
Consider the candy machines \(r_1\) and \(r_2\) in figure 4. We use algorithm 6.2 to derive a test case from \(r_2\) with respect to \(\text{ioco} = \text{ioco}_{\text{straces}(x)}\). In the derivation we use the suspension automaton of figure 7. The successive choices for the recursive steps of the algorithm are:

1. First choice 2 (giving an input to the implementation) is taken: \(t := \text{but}_1; t_1\)
2. To obtain \(t_1\), the next output of the implementation is checked (choice 3):
   \[t_1 := \text{li}_q u_1; t_2; \square \text{choc}_u; \text{fail} \square \theta; t_2\]
3. If the output was \(\text{li}_q u_1\) then we stop with the test case (choice 1): \(t_2 := \text{pass}\)
4. If no output was produced (output \(\theta\); we know that we are in the right branch of \(r_2\)), then we give a new input to the implementation (choice 2): \(t_2 := \text{but}_1; t_3\)
5. To obtain \(t_3\) we again check the outputs (choice 3): \(t_3 := \text{choc}_u; t_4; \square \text{li}_q u_1; \text{fail} \square \theta; \text{fail}\)
6. After the output \(\text{choc}_u\) we check again the outputs (choice 3) to be sure that nothing more is produced: \(t_4 := \text{choc}_u; \text{fail} \square \text{li}_q u_1; \text{fail} \square \theta; t_5\)
7. For \(t_5\) we stop with the test case (choice 1): \(t_5 := \text{pass}\)

After putting together all pieces we obtain \(t\) of figure 8 as a sound test case for \(r_2\), which is consistent with the results in examples 4.14 and 5.3: \(r_1\text{ioco} r_2\), \(r_2\text{ioco} r_2\), and indeed \(r_1\text{fails} t\), and \(r_2\text{passes} t\). So test case \(t\) will detect that \(r_1\) is not \(\text{ioco}\)-correct with respect to \(r_2\): \(\Box\)

### 7 Concluding Remarks

This paper presented implementation relations, conformance testing, and test generation for implementations that communicate via inputs and outputs. The implementation relations were defined by applying the theory of testing equivalences and refusal testing to systems in which inputs and outputs can be distinguished, and in which inputs are always enabled. The defined relations \(\leq_{iof}\), \(\leq_{los}\), \(\text{ioconf}\), and \(\text{ioco}\), are particular instances of a class of implementation relations \(\text{ioconf}\_F\), which are most easily characterized if the refusal of outputs, i.e., quiescence, is considered as an explicitly observable event, represented by a special output action \(\delta\). Traces over input actions, output actions, and \(\delta\), are called suspension traces, and the parameter \(F\) in \(\text{ioconf}\_F\) is a set of them. Processes modulo these input-output implementation relations are fully characterized by (a subset of) their suspension traces. The action \(\delta\) is no different from a normal output action. This means that the addition of \(\delta\) to represent quiescence makes it possible to reason about systems using only linear properties, i.e., traces.

The characterization in terms of suspension traces formed the basis for a test generation algorithm, which was proved to derive test cases from a specification, which can detect, by means of conformance testing, all and only implementations which are not correct for that specification with respect to any of the implementation relations \(\text{ioconf}\_\mathcal{F}\). 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 6.2 with the \text{conf}-based test generation algorithm in [Tre92]). 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 [LT9], having a weaker requirement on input-enabling. This allows for some systems that are $TOTS$ but not IOA (e.g., communication with systems via explicitly modelled bounded buffers: when the buffer is full, no input actions are possible anymore without first performing an internal event. Such a system is $TOTS$).

Another model which is very much related to input-output transition systems, is that of Input-Output State Machines (IOSM) [Pha94]. Our suspension automaton was inspired by the way the absence of output is treated in [Pha94]. Like IOA, IOSMs have strong input-enabling (called completeness). Absence of outputs ("bloque de sortie") is also considered observable, and an implementation relation $R_1$ is defined, which strongly resembles $ioco$.

The interesting point about the relation $R_1$ is that it was defined without reference to an underlying theory of testing equivalence or refusal testing, but that it was defined as the result of formalizing existing protocol testing practice with an existing testing tool (TVEDA) based on formal specifications in Estelle and SDL. This may be an indication that relations like $ioco$ not only have a nice theoretical basis, but also have practical applicability. A first trial to apply the theory of $ioco$ to conformance testing of a very simple protocol looks promising [TPHT96].

The implementation relations and algorithm in this paper generalize those for queue systems [TV92]. 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.

An 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.

Adding the absence of outputs as a special observable event makes it easier to compare transition systems with other models in which the absence of outputs is treated in the same way, such as in the realm of Mealy Machines (Finite State Machines – FSM). FSMs are often used in the area of communication protocol conformance testing [BU91, YL95], where the absence of outputs is usually denoted by a special null-output. The precise relation between the testing theories based on labelled transition systems and those based on FSMs is left for further study.

More attention is also needed for the topic of efficiently and effectively obtaining suspension automata and test cases. In particular, a compositional method for deriving them from process-algebraic specifications would be helpful. Also equational and congruence properties and axiomatization are left for further study.

Among the more practically oriented problems are the well-known test selection problem (test-suite size reduction [ISO96]). Algorithm 6.2 can generate infinitely many sound test cases, but which ones shall be really executed? Solutions can be sought by defining coverage measures, fault models, stronger test hypotheses, etc. [Ber91, ISO96, Pha94, Tre92]. 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 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.

Acknowledgements

Stimulating discussions with, and comments from the Formal-Methods people within the Tele-Informatics and Open Systems Group greatly helped in writing this paper. Also the comments from participants in the European Research Action COST 247 'Verification and Validation Methods for Formal Descriptions', from anonymous reviewers, and from TACAS'96 participants are acknowledged.

References


