Repetitive Quiescence in Implementation and Testing

(Extended Abstract*)

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

Abstract

This paper studies implementation relations and 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. Implementation relations, which formalize the notion of correctness of these implementations with respect to labelled 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

In the development of concurrent systems with formal methods the notion of correctness of an implementation with respect to a specification plays a crucial role. Depending on the kind of formalizations of the implementation and the specification the definition of correctness may have different forms, e.g., a logical satisfaction relation in case of a logical specification, or an equivalence on a behavioural specification formalism if both the implementation and the specification are given as instances of such a formalism. We use the term \textit{implementation relation} for a relation between a domain of implementations and a domain of specifications, not necessarily the same, which formally defines the correctness of an implementation with respect to a specification.

On the domain of labelled transition systems many such implementation relations have been defined and studied, e.g., bisimulation equivalence [Mil89], failure equivalence and preorder [Hoa85], testing equivalence and preorder [DNH84], refusal preorder [Phi87], and many others [Gla90, Gla93]. Many of these implementation relations can be defined extensionally, i.e., by explicitly defining the observations that an external observer, or tester, can make of a system. A transition system \( p \) is then equivalent to a system \( q \) if any possible observer can make exactly the same observations with \( p \) as with \( q \) (or more generally, \( p \) relates to \( q \) if for all possible observers, the observations made of \( p \) are related in some sense to the observations made of \( q \)). Different relations can be defined by varying the class of observers, the observations that observers can make, and the required relation between these observations.

Once an implementation relation has been defined, the problem of \textit{conformance testing} reverses the above question: given a specification and an implementation relation determine a set of ob-

*The full version appeared in [Tre96b]*
servers that can exactly discriminate between correct and erroneous implementations. Such a set of observers is called a test suite, and the observers are referred to as tests or test cases. Test suites are used in conformance testing to decide in an operational way, by performing the tests, whether a black-box implementation correctly implements the specification. Preferably, a test suite is derived algorithmically from the specification. Conformance testing for labelled transition systems has been studied especially in the context of testing communication protocols with the process-algebraic language LOTOS [ISO89], e.g., [Bri88, PF90, Tre92, Wez90].

Most of the theory of implementation relations and testing for labelled transition systems 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. We will refer to such kind of communication as symmetric interaction.

Another way of looking at the communication between transition systems is by explicitly making a distinction between input actions and output actions. 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, 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.

This paper studies implementation relations and conformance testing for systems that communicate via inputs and outputs. Implementation relations with inputs and outputs are defined extensionally following the ideas of testing equivalence and refusal testing [DNH84, DN87, Phi87, Lan90]. It will be shown that the defined relations allow for a simple and intuitive characterization which can be given in terms of only traces, if a special action modelling the absence of outputs (quiescence, cf. [Vaa91]) is added. This special action has all the properties of, and can be considered as, a normal output action. This paper generalizes [Seg93, Tre96a], which considered testing preorder with inputs and outputs, by also considering refusal testing, and by showing that all relations can be considered as special cases of a class of refusal-like implementation relations. Consequently, the test derivation algorithm for conformance testing also generalizes the one given in [Tre96a] by dealing with this whole class of implementation relations.

The next section introduces the models of our concern: labelled transition systems and input-output transition systems. Section 3 briefly reviews the symmetric implementation relations testing preorder (must preorder) and refusal preorder (failure-trace preorder), which form the basis for the ones with inputs and outputs. The latter are presented in section 4. First the input-output analogue of testing preorder is recalled from [Seg93, Tre96a], and then the analogue of refusal preorder is defined. Some properties and a model for this relation will be discussed, and all the input-output based relations are generalized to a class of relations of the same form. This class of relations forms the basis for the discussion of conformance testing and test derivation in section 5. Some concluding remarks are given in section 6.
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 \((S, L, T, s_0)\), 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 \(\text{LTS}(L)\). For technical reasons we restrict \(\text{LTS}(L)\) 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 \(\epsilon\) denoting the empty sequence. If \(\sigma_1, \sigma_2 \in L^*\), then \(\sigma_1 \cdot \sigma_2\) is the concatenation of \(\sigma_1\) and \(\sigma_2\).

Let \(p = (S, L, T, s_0)\) 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 \(s_0\).

\[
\begin{align*}
  s \xrightarrow{\mu} s' & \quad \text{def} \quad (s, \mu, s') \in T \\
  s \xrightarrow{\mu_1 \ldots \mu_n} s' & \quad \text{def} \quad \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s' \\
  s \xrightarrow{\tau} s' & \quad \text{def} \quad s = s' \text{ or } s \xrightarrow{\tau \ldots \tau} s' \\
  s \xrightarrow{a} s' & \quad \text{def} \quad \exists s_1, s_2 : s \xrightarrow{a} s_1 \xrightarrow{a} s_2 = s' \\
  s \xrightarrow{a_1 \ldots a_n} s' & \quad \text{def} \quad \exists s_0 \ldots s_n : s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n = s' \\
  p \xrightarrow{\sigma} & \quad \text{def} \quad (s' : s \xrightarrow{\sigma} s') \\
  s = \text{stop} & \quad \text{def} \quad \text{not } \exists s' : s \xrightarrow{a} s' \\
  \text{traces}(p) & \quad \text{def} \quad \{ \sigma \in L^* \mid p \xrightarrow{\sigma} \} \\
  \text{init}(p) & \quad \text{def} \quad \{ \mu \in L \cup \{\tau\} \mid p \xrightarrow{\mu} \} \\
  p \text{ after } \sigma & \quad \text{def} \quad \{ p' \mid p \xrightarrow{\sigma} p' \}
\end{align*}
\]

A process \(p\) has finite behaviour if there is a natural number \(n\), such that all traces in \(\text{traces}(p)\) have length smaller than \(n\); \(p\) is deterministic if for all \(\sigma \in L^*\), there is at most one \(p'\) such that \(p \xrightarrow{\sigma} p'\); if \(\sigma \in \text{traces}(p)\), then \(p \text{ after } \sigma\) is overloaded to denote this unique \(p'\).

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 [ISO89]:

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

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

\[
\begin{align*}
  \vdash a : B \xrightarrow{a} B \\
  \vdash i : B \xrightarrow{\tau} B \\
  B_1 \xrightarrow{\mu} B_1', \mu \in L \cup \{\tau\} & \quad \vdash B_1 \boxdot B_2 \xrightarrow{\mu} B_1' \\
  B_2 \xrightarrow{\mu} B_2', \mu \in L \cup \{\tau\} & \quad \vdash B_1 \parallel B_2 \xrightarrow{\mu} B_1' \parallel B_2' \\
  B_1 \xrightarrow{\tau} B_1', B_2 \xrightarrow{\tau} B_2', \mu \in L & \quad \vdash B_1 \parallel B_2 \xrightarrow{\tau} B_1' \parallel B_2' \\
  B \xrightarrow{\tau} B', B \in B, \mu \in L \cup \{\tau\} & \quad \vdash \Sigma B \xrightarrow{\mu} B'
\end{align*}
\]
Communication between two processes modelled as labelled transition systems is described by the parallel composition operator $\parallel$. Communication is based on synchronous, symmetric interactions between the communicating processes: an interaction can occur if both processes are able to perform that interaction, implying that both processes can also block the occurrence of an interaction. This kind of symmetric interaction abstracts from inputs and outputs, and from initiative or direction in the communication.

Since many realistic systems do make a distinction between inputs and outputs, we introduce input-output transition systems 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. Communication between two input-output transition systems takes place by an interaction consisting of an input of one system with an output of the other. The interaction is initiated by the process performing the output, and the other system is always prepared to participate in such an interaction: all the inputs of an input-output system are always enabled; they can never be refused. 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:

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

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

When studying input-output transition systems we use the notational convention 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] ($\forall a \in L_I : p' \xrightarrow{a}$), input-output transition systems allow input enabling via internal transitions (weak input enabling, $\forall a \in L_I : p' \xrightarrow{a}$).

### 3 Implementation Relations with Symmetric Interactions

An implementation relation captures the notion of correctness of an implementation with respect to a specification. Let $s$ be a specification and $i$ an implementation, both expressed as a labelled transition system, and let $\text{imp} \subseteq LTS(L) \times LTS(L)$ be an implementation relation, then $i \text{ imp} s$ expresses that implementation $i$ is correct with respect to specification $s$. An implementation relation can be defined extensionally, which means that it is 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:

$$i \text{ imp} s \quad = \text{def} \quad \forall u \in U : \text{obs}(u, i) \ast \text{obs}(u, s)$$

By varying the class of external observers $U$, 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 these relations is testing preorder \( \leq_{te} \) (must preorder), 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 \( U \), the relation between observations is set inclusion, and the observations are traces obtained from \( i \), respectively \( s \), when running in parallel with an observer \( u \), where a distinction is made between normal traces and completed traces (traces after which no more actions are possible).

**Definition 3.1**

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

1. \( p \) after \( \sigma \) refuses \( A \) \( \overset{\text{def}}{=} \) \( \exists p' : p \xrightarrow{\sigma} p' \) and \( \forall a \in A : p' \xrightarrow{a} \)

2. \( p \) after \( \sigma \) deadlocks \( \overset{\text{def}}{=} \) \( p \) after \( \sigma \) refuses \( L \)

3. The sets of observations \( obs_c \) and \( obs_i \) that an observer \( u \in \mathcal{LTS}(L) \) can make of process \( p \in \mathcal{LTS}(L) \), are

\[
\begin{align*}
obs_c(u,p) & = \{ \sigma \in L^* \mid (u \parallel p) \text{ after } \sigma \text{ deadlocks} \} \\
obs_i(u,p) & = \{ \sigma \in L^* \mid (u \parallel p) \text{ implies } \}
\end{align*}
\]

4. \( i \leq_{te} s \) \( \overset{\text{def}}{=} \) \( \forall u \in \mathcal{LTS}(L) : obs_c(u,i) \subseteq obs_c(u,s) \) and \( obs_i(u,i) \subseteq obs_i(u,s) \)

The extensional definition of \( \leq_{te} \) in definition 3.1.4 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 on our class of strongly converging transition systems.

**Proposition 3.2**

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

A weaker implementation relation that is strongly related to \( \leq_{te} \) is the relation \( \text{conf} \) [Bri88]. 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, \( \text{conf} \) 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 \( \text{conf} \) that several test generation algorithms have been developed [Bri88, PF90, Woz90, Tre92].

**Definition 3.3**

\[ i \text{ conf } s = \text{def } \forall u \in \mathcal{LTS}(L) : \text{ ( } obs_c(u,i) \cap \text{ traces}(s) \text{ ) } \subseteq \text{ obs}_c(u,s) \text{ and } \text{ ( } obs_i(u,i) \cap \text{ traces}(s) \text{ ) } \subseteq \text{ obs}_i(u,s) \]

**Proposition 3.4**

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

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 \( \theta \notin L \) to observers: \( U = \mathcal{LTS}(L_\theta) \), 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 \( \parallel \) 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 \textit{refusal preorder} \( \leq_{rf} \).

**Definition 3.5**

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

\[
\begin{align*}
& u \overrightarrow{\theta} u' \quad \vdash \quad u| | p \overrightarrow{\theta} u' | | p \\
& p \overrightarrow{\theta} p' \quad \vdash \quad u| | p \overrightarrow{\theta} u | | p' \\
& u \overrightarrow{i} u', p \overrightarrow{a} p', a \in L \quad \vdash \quad u| | p \overrightarrow{a} u | | p' \\
& u \overrightarrow{\theta} u', u \overrightarrow{\theta} p', \forall a \in L : u \overrightarrow{a} / \quad \text{or} \quad p \overrightarrow{a} / \quad \vdash \quad u| | p \overrightarrow{a} u | | p
\end{align*}
\]

2. The sets of \textit{observations} \( \text{obs}_c^\theta \) and \( \text{obs}_t^\theta \) that an observer \( u \in \mathcal{LTS}(L_\theta) \) can make of process \( p \in \mathcal{LTS}(L) \), are

\[
\begin{align*}
\text{obs}_c^\theta(u, p) & =_{\text{def}} \{ \sigma \in L_\theta^* \mid (u| | p) \text{ after } \sigma \text{ deadlocks} \} \\
\text{obs}_t^\theta(u, p) & =_{\text{def}} \{ \sigma \in L_\theta^* \mid (u| | p) \mathrel{\overset{\sigma}{\Rightarrow}} \}
\end{align*}
\]

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

A corresponding intensional characterization of refusal preorder is given in terms of failure traces. A failure trace is a trace in which both actions and refusals, represented by sets of actions, occur. To express this, the transition relation \( \rightarrow \) 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 \( \Rightarrow \) is then extended as expected to \( \mathrel{\overset{\varphi}{\Rightarrow}} \) with \( \varphi \in (L \cup \mathcal{P}(L))^\ast \).

**Definition 3.6**

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

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

2. The failure traces of \( p \) are: \( \text{Ftraces}(p) =_{\text{def}} \{ \varphi \in (L \cup \mathcal{P}(L))^\ast \mid p \mathrel{\overset{\varphi}{\Rightarrow}} \} \)

**Proposition 3.7**

\( i \leq_{rf} s \iff \text{Ftraces}(i) \subseteq \text{Ftraces}(s) \)

### 4 Implementations Relations with Inputs and Outputs

We now consider implementation relations for implementations with inputs and outputs: \( \text{imp} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U) \). We define such implementation relations, like the symmetric relations, in an extensional manner following (1). The first relation, the \textit{input-output testing relation} \( \leq_{iot} \), is defined completely analogous to testing preorder \( \leq_{te} \) in definition 3.1 by requiring that any possible observation made of \( i \) is a possible observation of \( s \). However, since systems are considered to communicate via inputs and outputs with outputs and inputs, respectively, of their environment, it is natural to consider only observers that are input-output transition systems, but with inputs and outputs reversed: \( u \in \mathcal{IOTS}(L_U, L_I) \) [Seg93, Tre96a].

**Definition 4.1**

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

\( i \leq_{iot} s =_{\text{def}} \forall u \in \mathcal{IOTS}(L_U, L_I) : \text{obs}_c(u, i) \subseteq \text{obs}_c(u, s) \) and \( \text{obs}_t(u, i) \subseteq \text{obs}_t(u, s) \)
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, \( s \in \mathcal{LTS}(L_I \cup L_U) \), and input-output transition systems differ marginally from Input/Output Automata, exactly the same intensional characterization is obtained: \( \leq_{iot} \) 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 \( p \in \mathcal{LTS}(L_I \cup L_U) \).

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

**Proposition 4.3**
\( i \leq_{iot} s \) iff \( \mathsf{traces}(i) \subseteq \mathsf{traces}(s) \) and \( \mathsf{Qtraces}(i) \subseteq \mathsf{Qtraces}(s) \).

Another characterization of \( \leq_{iot} \) can be given based on \( \mathsf{out} \)-sets [Tre96a]. The set \( \mathsf{out}(p \after \sigma) \) gives all possible output actions which can be observed after the trace \( \sigma \). This includes a special action \( \delta \), which actually indicates the absence of all outputs, i.e., it indicates quiescence after \( \sigma \). The action \( \delta \) makes the absence of outputs into an explicitly observable event. Proposition 4.5 states that an implementation is correct according to \( \leq_{iot} \) if all outputs it can produce after any trace \( \sigma \) can also be produced by the specification. Since this also applies to the special output \( \delta \), the implementation may show no outputs, i.e., quiescence, 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. \( \mathsf{out}(p) = \text{def} \{ x \in L_U | p \xrightarrow{x} \} \cup \{ \delta | \delta(p) \} \)
2. \( \mathsf{out}(P) = \text{def} \bigcup \{ \mathsf{out}(p) | p \in P \} \)

**Proposition 4.5**
\( i \leq_{iot} s \) iff \( \forall \sigma \in L^* : \mathsf{out}(i \after \sigma) \subseteq \mathsf{out}(s \after \sigma) \).

Analogous to the way the definition of \( \mathsf{conf} \) was obtained from the one for \( \leq_{te} \) (definitions 3.1 and 3.3 and propositions 3.2 and 3.4), the implementation relation \( \mathsf{ioconf} \) is obtained from \( \leq_{iot} \) by restricting the traces in proposition 4.5 to those of the specification [Tre96a]. As for \( \mathsf{conf} \), this allows partial specifications which only state requirements for traces explicitly specified in the specification.

**Definition 4.6**
\( i \mathsf{ioconf} s = \text{def} \forall \sigma \in \mathsf{traces}(s) : \mathsf{out}(i \after \sigma) \subseteq \mathsf{out}(s \after \sigma) \).

Up to this point we have seen implementation relations with symmetric interactions based on observers without \( \theta \)-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 to define 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 4.7
Let \( i \in \operatorname{IOTS}(L_I, L_U) \), \( s \in \operatorname{LTS}(L_I \cup L_U) \), then
\[
i \leq_{\text{ior}} s \quad \text{=def} \quad \forall u \in \operatorname{IOTS}(L_U, L_I \cup \{\emptyset\}) : \quad \text{obs}^\emptyset_\epsilon(u, i) \subseteq \text{obs}^\emptyset_\epsilon(u, s) \quad \text{and} \quad \text{obs}^\emptyset_\iota(u, i) \subseteq \text{obs}^\emptyset_\iota(u, s)
\]
\[\Box\]

A quiescent trace was introduced in definition 4.2 as a trace ending in the absence of outputs. Using the special action \( \delta \) which was used in \( \text{out} \)-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 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 intuitively mean 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 \( \leq_{\text{ior}} \) is characterized by inclusion of suspension traces (and hence it could also be called repetitive quiescence relation). Since quiescence corresponds to a refusal of \( \text{out} \)-sets to indicate the absence of outputs, a quiescent trace \( \sigma \in L^* \) can be written as a \( \delta \)-ending trace \( \sigma \cdot \delta \in (L \cup \{\delta\})^* \), and where \( \delta \) is written for the refusal of \( L_U \).

Definition 4.8
The suspension traces of process \( p \in \operatorname{LTS}(L) \) are: \( \text{Straces}(p) = \text{def} \quad \text{F traces}(p) \cap (L \cup \{L_U\})^* \).

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

Proposition 4.9
\( i \leq_{\text{ior}} s \quad \text{iff} \quad \text{Straces}(i) \subseteq \text{Straces}(s) \)
\[\Box\]

An intensional characterization of \( \leq_{\text{ior}} \) in terms of \( \text{out} \)-sets, analogous to proposition 4.5, is easily given by generalizing the definition of \text{after} \ (section 2) in a straightforward way to suspension traces.

Proposition 4.10
\( i \leq_{\text{ior}} s \quad \text{iff} \quad \forall \sigma \in L_\delta^* : \quad \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma) \)
\[\Box\]

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

Definition 4.11
\( i \text{ ioco } s = \text{def} \quad \forall \sigma \in \text{Straces}(s) : \quad \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma) \)
\[\Box\]

The input-output implementation relations defined so far, viz. \( \leq_{\text{iots}}, \text{iocof}, \leq_{\text{ior}}, \) and \( \text{ioco} \), are easily related using their characterizations in terms of \( \text{out} \)-sets. The only difference between the relations is the set of (suspension) traces for which the \( \text{out} \)-sets are compared, cf. propositions 4.5 (\( \leq_{\text{iots}} \)) and 4.10 (\( \leq_{\text{ior}} \)), and definitions 4.6 (\text{iocof}) and 4.11 (\text{ioco}). So, if we introduce the following class of relations \( \text{ioco}_F \) with \( F \subseteq L_\delta^* \):
\[
i \text{ioco}_F s = \text{def} \quad \forall \sigma \in F : \quad \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)
\]
then they can all be expressed as instances of \( \text{ioco}_F \):
\[
\leq_{\text{iots}} = \text{ioco}_{L^*} \quad \text{iocof} = \text{ioco}_{\text{Straces}(s)} \quad \leq_{\text{ior}} = \text{ioco}_{L_\delta^*} \quad \text{ioco} = \text{ioco}_{\text{Straces}(s)}
\]

Using (2) and (3) the input-output implementation relations are easily related by relating their respective sets \( F \), see proposition 4.12. Figure 1 presents candy machines with \( L_I = \{\text{but}\} \) and
$L_U = \{ \text{liq}, \text{choc} \}$, exemplifying the similarities and differences between the four relations. The generalized implementation relation $\text{ioco}$ will be the basis for the study of conformance testing and test derivation in section 5.

**Proposition 4.12**

$$\leq_{i\sigma} \subset \left\{ \leq_{i\tau} \right\} \subset \text{ioco}$$

![Figure 1: The implementation relations $\leq_{i\tau}$, $\text{iocnf}$, $\leq_{i\sigma}$, and $\text{ioco}$](image)

The characterization of $\leq_{i\sigma}$ in proposition 4.9 in terms of suspension traces suggests to transform a labelled transition system into another one representing exactly the suspension traces. Such a transition system is referred to as a **suspension automaton**, and it is obtained from a process $p$ by adding self-loops $s \xrightarrow{\delta} s$ to all states that are quiescent, and then determinizing the resulting automaton. The implementation relation $\leq_{i\sigma}$ then easily reduces to trace preorder on suspension automata, while also out-sets are easily computed from suspension automata. Figure 2 shows the suspension automata of the two right-most examples in figure 1, which clearly demonstrates the difference between the two systems in terms of traces: $?\text{but} \cdot ?\text{but} \cdot \text{liq}$ is a trace of $\Gamma_{q_2}$, so a suspension trace of $q_2$, but not of $q_1$.

**Definition 4.13**

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

- $S_\delta = \overset{\text{def}}{=} \mathcal{P}(S) \setminus \{\emptyset\}$  \hspace{1em} ($\mathcal{P}(S)$ is the powerset of $S$, i.e., the set of its subsets)
\[ T_\delta = \text{def} \left\{ q \xrightarrow{a} q' \mid a \in L_I \cup L_U, q, q' \in S_\delta, q' = \{ s' \in S \mid \exists s \in q : s \xrightarrow{a} s' \} \right\} \]

\[ q_0 = \text{def} \left\{ s' \in S \mid s_0 \xrightarrow{\cdot} s' \right\} \]

\[ \text{Proposition 4.14} \]

Let \( p \in \mathcal{LTS}(L) \) with inputs in \( L_I \) and outputs in \( L_U \), let \( \sigma \in L_\delta^* \), 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 \( \text{out}(\Gamma_p \text{ after } \sigma) \neq \emptyset \)

\[ \text{Corollary 4.15} \]

\( i \leq \text{iocor } s \) iff \( \text{traces}(\Gamma_i) \subseteq \text{traces}(\Gamma_s) \)

\[ \text{Figure 2: Suspension automata} \]

\section{5 Conformance Testing with Inputs and Outputs}

We now consider the conformance testing problem for the class of implementation relations \( \text{ioco}_F \) with \( F \subseteq \text{Straces}(s) \). This means that the problem expressed by (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, i.e., a test suite, which fully characterizes all \( \text{ioco}_F \)-correct implementations of a given specification. Subsequently, such a test suite can be used for conformance testing of black-box implementations.

Since we consider the class of implementation relations \( \text{ioco}_F \) tests will be processes in \( \mathcal{LTS}(L_I \cup L_U \cup \{ \theta \}) \). It is desirable, although not necessary, that tests used for conformance testing have certain properties. First, to allow for practically feasible test execution, tests should have finite behaviour to guarantee that any observation is bounded. Secondly, we would like to have as much control as possible in the conformance testing process, i.e., test execution should proceed as deterministically as possible. This implies, first of all, that test cases themselves must be
deterministic. But also test cases with a choice between an input and an output, or with a choice between multiple input actions (input and output from the perspective of the implementation), are undesirable. Both introduce unnecessary nondeterminism in a 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 δ-action, which is accepted by the θ-action in the test case. So a test case t is a deterministic process in $LTS(L_I \cup L_U \cup \{\emptyset\})$ with finite behaviour, such that for any state $t'$ of t either $init(t') = \emptyset$, or $init(t') = \{a\}$ for some $a \in L_I$, or $init(t') = L_U \cup \{\emptyset\}$.

It suffices for a test case t with θ-action to consider only observations $obs^\theta(t, i)$, since observations in $obs^\theta_c(t, i)$ can always be simulated with a test case with additional θ’s at certain states. Hence we only need to check whether $obs^\theta_i(t, i) \subseteq obs^\theta_i(t, s)$. Since for conformance testing the specification $s$ is known, the set $obs^\theta_i(t, s)$ can be calculated beforehand, and we can simply label the states of t as pass or as fail depending on whether the unique trace (because of determinism in the test case) leading to that state is in $obs^\theta_i(t, s)$. But since the traces in $obs^\theta_i(t, i)$ are prefix-closed, combined with the fact that test cases are finite, it suffices to consider only the completed traces in $obs^\theta_i(t, i)$. Moreover, the input-enabledness of implementations, together with the requirements on states of test cases, imply that these completed traces of $obs^\theta_i(t, i)$ will always end in a terminal state of $t$, so the terminal states are the only ones which need to be labelled. Combining all desirable properties and their consequences, we come to the following definition of a test case. Moreover, we can define a notion of passing a test case by an implementation, if all completed observations end in the state pass of the test case.

**Definition 5.1**

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

   The class of test cases over $L_I$ and $L_U$ is denoted as $\text{TEST}(L_U, L_I)$.

2. A test suite $T$ is a set of test cases: $T \subseteq \text{TEST}(L_U, L_I)$.

3. An implementation $i \in \text{IOTS}(L_I, L_U)$ passes a test case $t \in \text{TEST}(L_U, L_I)$, if all completed traces of the parallel composition of $t$ and $i$ lead to the pass-state of $t$:

   $$i \text{ passes } t =_{\text{def}} \forall \sigma \in L^*_i, \forall i' : t \parallel i \Rightarrow \sigma | \text{fail} \Rightarrow i'$$

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

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

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

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

The conformance testing problem can now be stated as

Given $s \in LTS(L_I \cup L_U), \mathcal{F} \subseteq \text{traces}(s)$, find a test suite $T \subseteq \text{TEST}(L_U, L_I)$, such that for any $i \in \text{IOTS}(L_I, L_U)$: $i \text{ ioco}_\mathcal{F} s$ iff $i \text{ passes } T$.
Ideally, an implementation should pass a test suite if and only if it is correct. However, although theoretically interesting, such a test suite would be infinitely large. Practically applicable conformance testing is restricted to finite test suites, implying that only the left-to-right implication holds, i.e., incorrect implementations may be detected, but correctness cannot be assured. Hence, we split the above requirement in the left-to-right implication, called soundness, and the right-to-left implication, called exhaustiveness, where practical testing is performed with test suites which are only sound [ISO96]. We state sufficient conditions for soundness and exhaustiveness of our test suites. In these conditions we use the notation $\sigma$ for a trace in which all $\delta$-actions have been replaced by $\theta$-actions, or vice versa, depending on the context. The interchange of $\delta$- and $\theta$-actions is natural since both are in a certain sense complementary: $\delta$ models the absence of outputs which can be observed with $\theta$ (cf. complementary actions $a$ and $\sigma$ in CCS [Mil89]).

**Proposition 5.2**

1. A test case $t \in T(EST(L_U, L_I))$ is sound, i.e., $t$ fails implies $i \text{ioco}_{X} s$, if
   \[
   \forall \sigma \in L_{1}^{*} : t \xrightarrow{\sigma} \text{fail} \implies \exists \sigma' \in F, x \in L_{U} \cup \{\delta\} : \sigma = \sigma'^{\sigma} \text{ and } x \notin \text{out}(s \text{ after } \sigma')
   \]

2. A test suite $T \subseteq T(EST(L_U, L_I))$ is exhaustive, i.e., $t$ passes $T$ implies $i \text{ioco}_{X} s$, if
   \[
   \forall \sigma \in F, \exists t \in T, \exists t' : t \xrightarrow{\sigma} t' \text{ and } \text{init}(t') = L_{U} \cup \{\theta\} \text{ and } \forall x \notin \text{out}(s \text{ after } \sigma) : t' \xrightarrow{x} \text{fail}
   \]

With these conditions it is straightforward to show that the following nondeterministic test-case derivation algorithm produces only sound test cases, and that any incorrect implementation can be detected by a test case derived with this algorithm, i.e., the set of all possibly derived test cases is exhaustive.

**Algorithm 5.3**

Let $\Gamma$ be the suspension automaton of a specification, and let $F \subseteq \text{traces}(\Gamma)$, then a test case $t \in T(EST(L_U, L_I))$ 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 $F' = \{ \sigma \in L_{1}^{*} | a \cdot \sigma \in F \} \neq \emptyset$, and $t'$ is obtained by recursively applying the algorithm for $F'$ and $\Gamma'$, with $\Gamma \xrightarrow{a} \Gamma'$;

3. (⋆ check the next output of the implementation ⋆)
   
   $t := \Sigma \{ x : \text{fail} | x \in L_{U} \cup \{\theta\}, \pi \notin \text{out}(\Gamma), \epsilon \in F \} \quad \Box \quad \Sigma \{ x : \text{pass} | x \in L_{U} \cup \{\theta\}, \pi \notin \text{out}(\Gamma), \epsilon \notin F \} \quad \Box \quad \Sigma \{ x : t_{x} | x \in L_{U} \cup \{\theta\}, \pi \notin \text{out}(\Gamma) \}$
   where $t_x$ is obtained by recursively applying the algorithm for $\{ \sigma \in L_{1}^{*} | x \cdot \sigma \in F \}$ and $\Gamma'$, with $\Gamma \xrightarrow{x} \Gamma'$.

**Corollary 5.4**

1. A test case obtained with algorithm 5.3 from $\Gamma_{s}$ and $F$ is sound for $s$ with respect to $i \text{ioco}_{X}$.

2. The set of all possible test cases that can be obtained with algorithm 5.3 is exhaustive.

**Example 5.5**

Consider the right-most candy machines $q_1$ and $q_2$ of figure 1. We use algorithm 5.3 to derive a
Figure 3: Test case $t$

test case from $q_1$. The test case can detect that $q_2$ is not a correct implementation of $q_1$ for the
implementation relation $\mathrm{ioco}$. In the derivation we use the suspension automaton $\Gamma_{q_1}$ of figure 2.
The successive choices of the recursive steps of the algorithm, leading to the test case $t$ in figure 3
(with inputs and outputs reversed), are:

choice 2: $t := ?\text{but}; t_1$
choice 3: $t_1 := !\text{liq}; t_2, ?!\text{choc}; \text{fail} ? \theta; t_2$
choice 1: $t_2, := \text{pass}$
choice 2: $t_2 := ?\text{but}; t_3$
choice 3: $t_3 := ?!\text{choc}; t_4 ? !\text{liq}; \text{fail} ? \theta; \text{fail}$
choice 1: $t_4 := \text{pass}$

It can easily be checked that $t\mid_{q_2} \xrightarrow{\text{but} ? \theta - \text{but} - \text{liq}} \text{fail}\mid_{q'_2}$, so indeed $q_2 \text{ fails } t$. On the other hand,
if $t$ is applied to the specification $q_1$ itself it passes: $q_1 \text{ passes } t$, which is consistent with the fact
that $\mathrm{ioco}$ is reflexive.

\section{Concluding Remarks}

The theory of testing equivalences and refusal testing has been applied to define implementation
relations for systems in which inputs and outputs can be distinguished, and in which inputs are
always enabled. The defined relations are particular cases of a class of implementation relations $\mathrm{ioco}_F$, which is most easily defined 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,
outputs actions, and $\delta$ are called suspension traces, and the parameter $F$ in $\mathrm{ioco}_F$ is a set of them.
Processes modulo these input-output implementation relations are fully characterized by their
suspension traces. Whereas for input-output relations based on testing equivalence the special
action $\delta$ has a special position (it can only occur at the end of a trace), it can occur at any
place in suspension traces resulting from refusal testing with inputs and outputs: refusal testing
leads to repetitive quiescence, and the action $\delta$ is no different from a normal output action. The
characterization in terms of suspension traces leads to the definition of the suspension automaton,
which is the deterministic transition system over inputs, outputs and $\delta$, which just represents the
suspension traces. The suspension automaton forms 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 implementations which are not correct with respect to that specification and $\mathrm{ioco}_F$. 
and only such implementations.

It was noted that input-output transition systems only marginally differ from Input/Output Automata (IOA) [LT89], having a weaker requirement on input-enabling. 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 (‘blocage de sortie’) is also considered observable, and an implementation relation $R_1$ is defined, which strongly resembles $\text{ioco}$.

The interesting point about $R_1$ 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 [CGPT96]) based on formal specifications in Estelle and SDL. This may be an indication that relations like $\text{ioco}$ not only have a nice theoretical basis, but also practical applicability. In this respect, a first trial to apply the theory of $\text{ioconf}$ to conformance testing of a very simple protocol looks promising [TFPHT96].

Adding the absence of outputs as a special observable event facilitates reasoning about systems with inputs and outputs because only linear properties, i.e., traces, have to be considered. It also 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 [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.

Another issue for further study is how to obtain effectively the suspension automaton. In particular, a compositional method for deriving it from a process-algebraic specification would be helpful. Also congruence properties and axiomatization are left for further study.

References


