Developments in testing transition systems

Ed Brinksma, Lex Heerink and Jan Tretmans
Tele-Informatics and Open Systems group, Dept. of Computer Science
University of Twente, 7500 AE Enschede, The Netherlands
{brinksma,heerink,tretmans}@cs.utwente.nl

Abstract

This paper discusses some of the developments in the theory of test generation from labelled transition systems over the last decade, and puts these developments in a historical perspective. These developments are driven by the need to make testing theory applicable to realistic systems. We illustrate the developments that have taken place in a chronological order, and we discuss the main motivations that led to these developments. In this paper the claim is made that testing theory (slowly) narrows the gap with testing practice, and that progress is made in designing test generation algorithms that can be used in realistic situations while maintaining a sound theoretical basis.

1 INTRODUCTION

Testing and verification Testing and verification are complementary techniques that are used to increase the level of confidence in the correct functioning of systems as prescribed by their specifications. 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, in practice being based on observing only a small subset of all possible instances of system behaviour, is usually incomplete: testing shows the presence of errors, not their absence. Since test-

ing can be applied to the real implementation, it is useful in those cases when a valid and reliable model is not present.

There is an apparent paradox between the attention that verification and testing get in usage and research. Whereas most of the research in the area of distributed systems is concentrated on verification, testing is the predominant technique in practice. People from the realm of verification very often consider testing as inferior, because it can only detect some errors, but it cannot prove correctness; on the other hand, people from the realm of testing consider verification as impracticable and not applicable to realistically-sized systems.

Protocol conformance testing Protocol conformance testing is concerned with checking protocol implementations against their specifications by means of experimentation. Tests are derived from the protocol specification, then 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. Since conformance testing is a mainly manual, laborious and time-consuming process, automating the testing process has always received much attention. To automate the generation of test cases the protocol specification must be in a form amenable to manipulation by tools. Natural language specifications do not serve this purpose; formal languages do. The availability and increasing use of formal methods has resulted in theories, methods and pragmatics for the (semi-)automatic derivation of tests from formal specifications. In the area of test execution there are currently commercial protocol-tester tools available that can execute tests for many different protocols. For such tools to work properly it is important that test cases can be specified precisely and unambiguously. The standardised test specification language TTCN [22, part 3] is widely used for this purpose.

Conformance testing and formal methods Starting point for protocol conformance testing based on formal methods is a formal specification, e.g., a specification written in one of the currently standardised formal description techniques Estelle [20], LOTOS [21], or SDL [10]. Correctness and validity of this specification is assumed, and is not considered as part of conformance testing. Furthermore, there is an implementation, referred to as the implementation under test (IUT), which is treated as a black box, exhibiting external behaviour. The IUT is a physical, real object that is in principle not amenable to formal reasoning. We can only deal with implementations in a formal way, if we make the assumption that any real implementation has a formal model with which we could reason formally. This formal model is only assumed to exist, but it need not be known a priori. This assumption is referred to as the *test hypothesis* [3, 39, 23]. The test hypothesis allows to reason about implementations as if they were formal objects, and, consequently, to express conformance of implementations with respect to specifications by means of a formal relation between such models of implementations and specifications.

Such a relation is called an *implementation relation* [8, 23]. 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, called a test suite. The test suite must be sound, i.e., it must give a negative verdict only if the implementation is incorrect. Additionally, the test suite must be as complete as possible, i.e., if the implementation is incorrect, it must have a high probability to give a negative verdict.

Many different approaches to algorithmic test generation, based on different protocol specification formalisms, have been undertaken. Two main approaches can be distinguished: those based on Finite State Machines (FSM) and those based on Labelled Transition Systems (LTS). FSM-based protocol testing has been inspired by functional hardware testing and is based on modelling the behaviour of a protocol as a Mealy machine (Finite State Machine FSM) [5, 16, 27, 26, 30, 37, 46].

Goal and overview LTS-based testing has its basis in the formal theory of testing equivalences for labelled transition systems and process algebras, which is based on the formalisation of the notion of test and observation in [13, 12], and which continues with [1, 33, 24, 17].

The goal of this paper is to describe the developments in the theory for test generation for labelled transition systems, as they have led to the current status. We will show that the approach that started from practice and the one that started from theory are now at the point of meeting each other, leading to practical test generation algorithms that have a sound theoretical basis. One indication for this claim is that the algorithm implemented in TVEDA can be given a theoretical basis in the theory of refusal testing [33, 24] by adding to this theory a distinction between input and output actions. This was shown using the theory of Input/Output Transition Systems (IOTS) in [41]. The model of IOTS can be used very well to describe SDL and TTCN processes. Recent results [19] also link the notion of channel (as in SDL) or Point of Control and Observation (PCO) into the LTS-based testing theory.

Section 2 introduces LTS and fixes notation, and section 3 introduces testing concepts for LTS as described by, e.g., [13, 12]. Next, section 4 presents a testing theory for LTS that uses these concepts, and shows how tests can be constructed that are able to check correctness of implementations. Since this theory assumes that implementations communicate in a symmetric manner with their environment, which is unrealistic in practice, a more refined testing theory, based on IOTS, is presented in section 5. Section 6 discusses a refinement of the IOTS model that takes the distribution of PCOs of implementations into account. This theory can serve as an unified model in which both the traditional testing theory of section 3, and the refined theory of section 5, can be expressed. Section 7 ends with conclusions and further work.

2 LABELLED TRANSITION SYSTEMS

In this paper we will concentrate on a testing theory for labelled transition systems. We will use this formalism to model the behaviour of specifications, implementations and tests. A labelled transition consists of nodes and transitions between nodes that are labelled with actions. Formally, a (labelled) transition system (LTS) over L is a quadruple $\langle S, L, \rightarrow, s_0 \rangle$ where

- S is a (countable) set of states;
- L is a (countable) set of observable actions;
- $\rightarrow \subseteq S \times (L \cup \{\tau\}) \times S$ is a set of transitions; and
- $s_0 \in S$ is the initial state.

The special action $\tau \notin L$ represents an unobservable, internal action. We restrict to (strongly) convergent transition systems, i.e., transition systems that are not able to perform an infinite sequence of internal transitions. The class of all convergent transition systems over L is denoted by $\mathcal{LTS}(L)$, and the set of all finite words over L is denoted by L^*. In order to describe the sequences of actions in L and $\mathcal{P}(L)$ that can be performed from a given state (where $\mathcal{P}(\cdot)$ denotes the powerset operator on sets) we use the following abbreviations, with $p = \langle S, L, \rightarrow, s_0 \rangle$ a labelled transition system such that $s, s' \in S, \lambda, \lambda_i \in \mathcal{P}(L) \cup L \cup \{\tau\}, \alpha, \alpha_i \in \mathcal{P}(L) \cup L$ and $\sigma \in (\mathcal{P}(L) \cup L)^*$.

$$
\begin{array}{lll}
s \xrightarrow{\lambda} s' & =_{def} & \begin{cases} (s, \lambda, s') \in \rightarrow, & \text{if } \lambda \in L \cup \{\tau\} \\ s = s' \text{ and } \forall \mu \in \lambda \cup \{\tau\}, \forall s'' : \neg(s \xrightarrow{\mu} s''), & \\ & \text{if } \lambda \in \mathcal{P}(L) \end{cases} \\
s \xrightarrow{\lambda_1 \cdot \lambda_2 \cdot \ldots \cdot \lambda_n} s' & =_{def} & \exists s_0, s_1, \ldots, s_n : s = s_0 \xrightarrow{\lambda_1} s_1 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_n} s_n = s' \\
s \xrightarrow{\lambda_1 \cdot \lambda_2 \cdot \ldots \cdot \lambda_n} & =_{def} & \exists s' : s \xrightarrow{\lambda_1 \cdot \lambda_2 \cdot \ldots \cdot \lambda_n} s' \\
s \stackrel{\epsilon}{\Longrightarrow} s' & =_{def} & s = s' \text{ or } s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \stackrel{\alpha}{\Longrightarrow} s' & =_{def} & \exists s_1, s_2 : s \stackrel{\epsilon}{\Longrightarrow} s_1 \xrightarrow{\alpha} s_2 \stackrel{\epsilon}{\Longrightarrow} s' \\
s \stackrel{\alpha_1 \cdot \alpha_2 \cdot \ldots \cdot \alpha_n}{\Longrightarrow} s' & =_{def} & \exists s_0, s_1, \ldots, s_n : s = s_0 \stackrel{\alpha_1}{\Longrightarrow} s_1 \stackrel{\alpha_2}{\Longrightarrow} \ldots \stackrel{\alpha_n}{\Longrightarrow} s_n = s' \\
s \stackrel{\sigma}{\Longrightarrow} & =_{def} & \exists s' : s \stackrel{\sigma}{\Longrightarrow} s'
\end{array}
$$

Self-loop transitions of the form $s \xrightarrow{A} s$ where $A \subseteq L$ are called refusal transitions. In this case A is called a refusal of s. Such a refusal transition explicitly encodes the inability to perform any action in $A \cup \{\tau\}$ in state s. A failure trace consists of a sequence over refusal transitions $\xrightarrow{A}$ with $A \subseteq L$ and 'normal' transitions $\xrightarrow{\mu}$ with $\mu \in L \cup \{\tau\}$ where an abstraction from internal actions τ is made. For readability we do not distinguish between a labelled transition system and its initial state, e.g., $p \stackrel{\sigma}{\Longrightarrow} =_{def} s_0 \stackrel{\sigma}{\Longrightarrow}$ where s_0 is the initial state of labelled transition system p. If $p \stackrel{\sigma}{\Longrightarrow}$ where $\sigma \in L^*$ then σ is called a trace of p. For $p \in \mathcal{LTS}(L)$ we will use the following definitions.

1. $f\text{-}traces(p) =_{def} \{\sigma \in (\mathcal{P}(L) \cup L)^* \mid p \stackrel{\sigma}{\Longrightarrow}\}$
2. $traces(p) =_{def} \{\sigma \in L^* \mid p \stackrel{\sigma}{\Longrightarrow}\}$
3. p **after** σ **refuses** $A =_{def} \exists p' : p \stackrel{\sigma}{\Longrightarrow} p'$ and $\forall \mu \in A \cup \{\tau\} : \neg(p' \stackrel{\mu}{\longrightarrow})$
4. p **after** σ **deadlocks** $=_{def}$ p **after** σ **refuses** L
5. $der(p) =_{def} \{p' \mid \exists \sigma \in L^* : p \stackrel{\sigma}{\Longrightarrow} p'\}$
6. $init(p) =_{def} \{\mu \in L \cup \{\tau\} \mid \exists p' : p \stackrel{\mu}{\longrightarrow} p'\}$
7. P **after** σ $=_{def} \{p' \mid \exists p \in P : p \stackrel{\sigma}{\Longrightarrow} p'\}$ where P is a set of states
8. p is *deterministic* iff $\forall \sigma \in L^* : \mid \{p\}$ **after** $\sigma \mid \leq 1$
9. p has *finite behaviour* iff $\exists N \in \mathbf{N} : \forall \sigma \in traces(p) : \mid \sigma \mid \leq N$

In testing, an external observer experiments on an implementation in order to unravel its (unknown) behaviour. A test specifies the behaviour of an observer, and we assume that tests are modelled as LTS. Tests can be run, or executed, against implementations. From the execution of a test against an implementation observations can be made. These observations are then compared with the expected observations that can be obtained by running the same test against the specified behaviour, and a verdict (success or failure) is assigned. Failure should indicate that there is evidence that the implementation did not behave correct, otherwise success should be assigned. Section 5 treats test execution in more detail.

3 TESTING RELATIONS FOR TRANSITION SYSTEMS

In order to decide the correctness of implementations a clear correctness criterion is needed: when is an implementation considered correct with respect to its specification? In the context of labelled transition systems many proposals for such correctness criteria in the form of implementation relations have been made [17]. One of the first significant implementation relations was *observation equivalence* [29]. Observation equivalence is defined as a relation over states of transition systems by means of (weak) bisimulation relations. Informally, two systems $p, q \in \mathcal{LTS}(L)$ are called observation equivalent, denoted by $p \approx q$, if for every trace $\sigma \in L^*$ every state that is reachable from p after having performed trace σ is itself observation equivalent to some state of q that is also reachable after having performed trace σ, and similarly with p and q interchanged. Observation equivalence intuitively captures the notion of equivalent external behaviour of systems; two systems are observation equivalent if they exhibit "exactly the same" external behaviour. See [29] for a formal definition of observation equivalence.

Instead of relating behaviours intensionally in terms of relations over states and transitions between states, it is also possible to relate system behaviour in an extensional way; what kind of systems can be distinguished from each from each other by means of experimentation? [13, 12] were first in compar-

ing system behaviour in this way by explicitly modelling the behaviour of experiments, and relating the observations that can be made when these experiments are applied to systems. In general, for a set of experiments $\mathcal{U}$, and a set of observations $obs(u, p)$ that experiment $u \in \mathcal{U}$ may cause when system p is tested, they define a so-called *testing relation* over systems by relating the observations $obs(u, i)$ and $obs(u, s)$ that are made when experiments $u \in \mathcal{U}$ are carried out against the systems i and s. Formally, such testing relations are defined as follows

$$i \textbf{ conforms-to } s \quad =_{def} \quad \forall u \in \mathcal{U} : obs(u, i) \sqsubseteq obs(u, s) \tag{1}$$

where **conforms-to** denotes the testing relation that is defined. By varying the set of experiments $\mathcal{U}$, the set of observations obs and the relation $\sqsubseteq$ between these sets of observations, different testing equivalences can be defined. [13, 12] discuss, and compare, several different testing relations by varying the set of observations obs and the relation $\sqsubseteq$ between these sets of observations. The theory described in [13, 12] forms the basis for testing theories for transitions systems. We will discuss three instances of such testing relations that are relevant for the remainder of this paper, viz., observation equivalence, testing preorder and refusal preorder, and use a formalisation following [39] that slightly differs from the original formalisation given in the seminal work of [13, 12].

Observation equivalence [1] shows that observation equivalence can be characterised in an extensional way (i.e., following the characterisation of equation (1)), under the assumption that at each stage of a test run infinitely many local copies of the internal state of the system under test can be made, and infinitely many experiments can be conducted on these local copies. Intuitively, this means that at each stage of a test run the implementation must be tested against all possible operating environments. These assumptions are quite strong and too difficult to meet in practice. Therefore, observation equivalence is, in general, too fine to serve as a realistic implementation relation, and weaker notions of correctness between implementations and specifications have to be defined.

Testing preorder In testing preorder it is assumed that the behaviour of external observers can, just as the behaviour of implementations and specifications, be modelled as transition systems (that is, $\mathcal{U} \equiv \mathcal{LTS}(L)$) and these observers communicate in a synchronous and symmetric way with the system under test [13, 12]. From an observer u and system under test p, the binary infix operator $||$ creates a transition system $u \;||\; p$ that models the behaviour of u experimenting on p in a synchronous way. The transitions that $u \;||\; p$ can perform are defined by the smallest set of transitions induced by the following inference rules

$$\frac{u \xrightarrow{\tau} u'}{u \mid\mid p \xrightarrow{\tau} u' \mid\mid p} \qquad \frac{p \xrightarrow{\tau} p'}{u \mid\mid p \xrightarrow{\tau} u \mid\mid p'} \qquad \frac{u \xrightarrow{a} u', p \xrightarrow{a} p'}{u \mid\mid p \xrightarrow{a} u' \mid\mid p'} \ (a \in L)$$

Using $\mid\mid$ a testing preorder on transition systems [13] can be defined in an extensional way following equation (1). Intuitively, an implementation i is testing preorder related to specification s, denoted as $i \ \leq_{te} \ s$, if for every external observer u that is modelled as a transition system, each trace that $u \ \mid\mid \ i$ can perform is preserved by $u \ \mid\mid \ s$, and each deadlock of $u \ \mid\mid \ i$ is preserved by $u \mid\mid s$. Formally, testing preorder $\leq_{te}$ is defined by

$$i \ \leq_{te} \ s \quad =_{def} \quad \forall u \in \mathcal{LTS}(L): \quad \begin{array}{r} obs_t(u,i) \subseteq obs_t(u,s) \\ \text{and } obs_c(u,i) \subseteq obs_c(u,s) \end{array}$$

where $obs_t(u,p) =_{def} \{\sigma \in L^* \mid (u \ \mid\mid \ p) \stackrel{\sigma}{\Longrightarrow}\}$ and $obs_c(u,p) =_{def} \{\sigma \in L^* \mid \ (u \ \mid\mid \ p)$ **after** σ **deadlocks**$\}$. The relation $\leq_{te}$ can be intensionally characterised by $i \leq_{te} s$ iff $\forall \sigma \in L^*, \forall A \subseteq L : i$ **after** σ **refuses** A implies s **after** σ **refuses** A. Testing preorder allows implementations to be "more deterministic" than their specification, but it does not allow that implementations "can do more" than is specified; in this sense the specification not only prescribes what behaviour is allowed, but also what behaviour is not allowed! The relation $\leq_{te}$ serves as the basic implementation relation in many testing theories for transition systems.

Refusal preorder Refusal preorder can be seen as a refinement of testing preorder, and is defined extensionally in the theory of refusal testing [33]. Instead of administrating the successful actions that are conducted on an implementation by an observer, refusal testing also takes the unsuccessful actions into account. The difference between refusal preorder and testing preorder is that observers can detect deadlock, and act on it, i.e., in refusal preorder observers are able to continue after observation of deadlock. Formally, we model this as in [24] by using a special deadlock detection label $\theta \notin L$ (i.e., $\mathcal{U} \equiv \mathcal{LTS}(L \cup \{\theta\})$, cf. equation (1)) that is used to detect the inability to synchronise between the observer u and system under test p. The θ-action is observed if there is no other way to continue, i.e., when p is not able to interact with the actions offered by u. The transition system $u \rceil\mid p \in \mathcal{LTS}(L \cup \{\theta\})$ that occurs as the result of communication between a deadlock observer $u \in \mathcal{LTS}(L \cup \{\theta\})$ and a transition system $p \in \mathcal{LTS}(L)$ is defined by the following inference rules.

$$\frac{u \xrightarrow{\tau} u'}{u \rceil\mid p \xrightarrow{\tau} u' \rceil\mid p} \qquad \frac{u \xrightarrow{a} u', p \xrightarrow{a} p'}{u \rceil\mid p \xrightarrow{a} u' \rceil\mid p'} \ (a \in L)$$

$$\frac{p \xrightarrow{\tau} p'}{u \rceil\mid p \xrightarrow{\tau} u \rceil\mid p'} \qquad \frac{u \xrightarrow{\theta} u', u \xrightarrow{\tau} \not\rightarrow, p \xrightarrow{\tau} \not\rightarrow, init(u) \cap init(p) = \emptyset}{u \rceil\mid p \xrightarrow{\theta} u' \rceil\mid p}$$

Observations made by an observer u by means of the operator $\rceil\!|$ now may include the action θ. The testing preorder induced for observers in $\mathcal{LTS}(L \cup \{\theta\})$ is called refusal preorder, and is defined in the style of equation (1):

$$i \leq_{rf} s \quad =_{def} \quad \forall u \in \mathcal{LTS}(L \cup \{\theta\}) : \quad obs_c^{\theta}(u,i) \subseteq obs_c^{\theta}(u,s) \text{ and } obs_t^{\theta}(u,i) \subseteq obs_t^{\theta}(u,s)$$

where $obs_c^{\theta}(u,p) \quad =_{def} \quad \{\sigma \in (L \cup \{\theta\})^* \mid (u \rceil\!| \; p)$ **after** σ **deadlocks**$\}$ and $obs_t^{\theta}(u,p) \quad =_{def} \quad \{\sigma \in (L \cup \{\theta\})^* \mid (u \rceil\!| \; p) \stackrel{\sigma}{\Longrightarrow}\}$. Informally, $i \leq_{rf} s$ if, for every observer $u \in \mathcal{LTS}(L \cup \{\theta\})$, every sequence of actions that may occur when u is run against i (using $\rceil\!|$) is specified in $u \rceil\!| \; s$; i is not allowed to accept, or reject, an action when communicating with u, if this is not specified by s. Refusal preorder is strictly stronger than testing preorder, i.e., $\leq_{rf} \subset \leq_{te}$. Refusal preorder is characterised by inclusion of failure traces: $i \leq_{rf} s$ iff $\mathit{f\text{-}traces}(i) \subseteq \mathit{f\text{-}traces}(s)$.

We emphasize that implementation relations that abstract from the nondeterministic characteristics of protocols (e.g., trace preorder or trace equivalence) are, in general, *not* sufficient to capture the intuition behind correctness of systems. Even if protocols are defined as deterministic automata, their joint operation with underlying layers, such as operating systems, generally will behave in a nondeterministic manner.

4 CONF TESTING

As shown in section 3 [13, 12] define a correctness criterion (in terms of a testing relation) by providing a set of experiments ($\mathcal{U}$), a notion of observation (obs), and a way to relate observations of different systems ($\sqsubseteq$) (equation (1)). In test generation the opposite happens: for some implementation relation a set of tests $\mathcal{U}$ has to be designed that is able to distinguish between correct and incorrect implementations by comparing the observations that the implementation produces with the expected observations when the same test is applied to the specification. The first testing theory that treats the problem of test generation in this way is [6, 7].

In [6, 7] a method is presented to derive test cases from a specification that is able to discriminate between correct and incorrect implementation with respect to the implementation relation **conf**. The relation **conf** can be seen as a liberal variant of $\leq_{te}$. The difference with $\leq_{te}$ is that the implementation may do things that are not specified; in **conf** there is no need to perform any robustness tests! Since in **conf** there is no need to check how the implementation behaves for unspecified traces, test generation algorithms for **conf** are better suited for automation than test generation algorithms for $\leq_{te}$. In particular, for a finite behaviour specification this means that only a finite

number of traces have to be checked. Formally, the relation **conf** is defined as $\leq_{te}$ restricted to the traces of the specification.

$$i \textbf{ conf } s \quad =_{def} \quad \forall u \in \mathcal{LTS}(L): \quad obs_t(u,i) \cap traces(s) \subseteq obs_t(u,s) \text{ and } obs_c(u,i) \cap traces(s) \subseteq obs_c(u,s)$$

In literature, this relation is usually known in its intentional characterisation: i **conf** s iff $\forall \sigma \in traces(s), \forall A \subseteq L^*: i$ **after** σ **refuses** A implies s **after** σ **refuses** A. Informally, the **conf** relation indicates that an implementation is correct with respect to its specification if, after executing a specified trace, the implementation is not able to reach an unspecified deadlock when synchronised with an arbitrary test process. [6, 7] develops a theory for the construction of a so-called *canonical tester* from a specification. The canonical tester $T(s)$ of s is a process that preserves the traces of s (i.e., $traces(T(s)) = traces(s)$) and that is able to decide unambiguously whether an implementation i is **conf**-correct with respect to specification s, i.e.,

$$\forall i \in \mathcal{LTS}(L): i \textbf{ conf } s \quad \textit{iff} \quad i \textit{ conf-passes } T(s)$$

where i *conf-passes* $T(s) =_{def} \forall \sigma \in L^*: (i \mid\mid T(s))$ **after** σ **deadlocks** implies $T(s)$ **after** σ **deadlocks**. This is done by running $T(s)$ against implementation i until it deadlocks, and checking that every deadlock of $i \mid\mid T(s)$ can be explained by a deadlock of $T(s)$; if $T(s)$ did not end in a deadlock state, evidence of non-conformance with respect to **conf** has been found. The elegance of **conf**-testing is nicely illustrated by the fact that the canonical tester of a canonical tester is testing equivalent with the original specification; $T(T(s)) \approx_{te} s$ (where $\approx_{te}$ is the symmetric reduction of $\leq_{te}$) [6].

In [2, 45] a procedure to construct canonical testers has been implemented for finite Basic LOTOS processes, that is, from a finite behaviour LOTOS specification s without data a tester $T(s)$ is constructed that is again represented as a finite behaviour Basic LOTOS process. [35] has extended this to Basic LOTOS processes with infinite behaviour. A procedure for the construction of tests from a specification related to the theory of canonical testers in such a way that these tests preserve the structure of the specification is sketched in [34]. In [25] a variant of the theory of canonical testers is discussed for a transitive version of the **conf** relation. [15] derives, and simplifies, canonical testers using refusal graphs. Figure 1 presents an example of a process and its canonical tester.

The theory of canonical testers is applicable to situations where the system under test communicates in a symmetric and synchronous manner with an external observer; both the observer and the system under test have to agree on an action in order to interact, and there is no notion of initiative of actions. Since asynchronously communicating systems can be modelled in terms of synchronously communicating systems by explicitly modelling the intermedi-

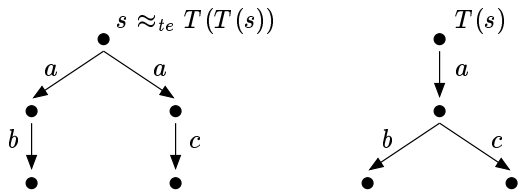

Figure 1 Canonical testers.

ate communication medium between these two systems **conf**-testing can also be applied to asynchronously communicating systems (e.g., the queue systems discussed in section 5). Consequently, **conf**-testing is widely applicable to a large variety of systems.

However, the theory of canonical testers also has some difficulties that restricts its applicability in practice. We will mention the two important ones in our view. The first difficulty has to do with the large application scope of the theory of canonical testers. In general, the more widely applicable a theory becomes, the less powerful this theory becomes for specific situations. In particular, communication between realistic systems is, in practice, often asymmetric. By exploiting the characteristics of such asymmetric communication, a more refined testing theory can be developed. The next section discusses in detail how this can be done.

Another drawback of the theory of canonical testers is its difficulty to handle data in a symbolic way. Since in most realistic applications data is involved, it is necessary to deal with data in a symbolic way in order to generate canonical testers in an efficient way. In [14, 39] some problems with the derivation of canonical testers for transition systems that are specified in full LOTOS (i.e., LOTOS with data) have been identified, such as an explosion in the data part of the specification. In particular, the derivation of canonical testers in a symbolic way is complicated by the fact that not only the data domains and the constraints imposed on the data values that are communicated need to be composed in a correct way, but also the branching structure of the specification (and thus of the canonical tester itself) needs to be taken into account. The problem is that the test generation algorithm for **conf** uses powerset constructions that are, in principle, able to transform countable branching structures into uncountable branching structures.

5 CHANGING THE INTERFACES

Several approaches have been proposed to model the interaction between implementations and their environment more faithfully, e.g., by explicitly considering the asymmetric nature of communication with the aim to come to a

testing theory that is better suited for test generation in realistic situations. Moreover, since the standardised test notation TTCN [22, part 3] uses inputs and outputs to specify tests, theories that incorporate such asymmetric communication allow the generation of tests in TTCN. In this section we present a short overview of some of the approaches that have been proposed in this area, and we will elaborate on one of them.

Apply asynchronous theory to transition systems Much research has been done in systems that communicate in an asynchronous manner (e.g., [4]), and some languages used in protocol conformance testing are based on asynchronous paradigms (e.g., SDL [10] , Estelle [20], TTCN [22, part 3]). [9] gives a short overview of translation between labelled transition systems and Mealy machines, which can be used as an underlying semantic model for, e.g., SDL [10]. In particular, research has been done in transforming transition systems without inputs and outputs into FSMs with inputs and outputs, and deriving tests for these FSMs (e.g., [18]). However, many of these developments lack a solid, formal basis, and their use in practice is restricted.

Queue systems In [42] asynchronous communication between an implementation and its environment is modelled explicitly by the introduction of an underlying communication layer. This layer essentially consists of two unbounded FIFO queues, one of which is used for message transfer from the implementation to the environment, and the other for message transfer in the opposite direction (figure 2). Such systems are called queue systems.

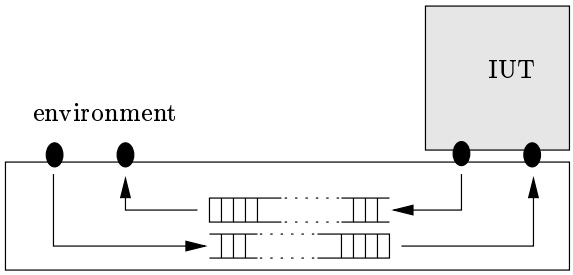

Figure 2 Architecture of a queue system.

In order to formalise the notion of queue systems the set of labels L is partitioned in a set of input labels L_I and a set of output labels L_U (i.e., $L = L_I \cup L_U, L_I \cap L_U = \emptyset$). Input labels are supplied from the environment via the input queue to the IUT, and, similarly, output labels run via the output queue. In particular, [42] is interested in what kind of systems can be distinguished from each other in the asynchronous setting sketched above, and how this compares to the synchronous setting. They therefore define a

new implementation relation $\leq_{te}^{Q}$ that captures whether two systems are $\leq_{te}$-related when tested through the queues. Formally,

$$i \leq_{te}^{Q} s \quad =_{def} \quad \mathcal{Q}(i) \leq_{te} \mathcal{Q}(s)$$

where $Q(p)$ denotes the transition system that is induced when p is placed in an environment where communication runs via two queues as sketched above.

They also define classes of asynchronous implementation relations called *queue preorders* $\leq_{\mathcal{Q}}^{\mathcal{F}}$ as preorders that disallow the implementation to produce unspecified outputs (where the inability to produce outputs is considered observable) after having performed arbitrary trace in some specified $\mathcal{F} \subseteq L^*$, i.e.,

$$i \leq_{\mathcal{Q}}^{\mathcal{F}} s \quad =_{def} \quad \forall \sigma \in \mathcal{F} : \mathcal{O}_i(\sigma) \subseteq \mathcal{O}_s(\sigma) \tag{2}$$

where $\mathcal{O}_p(\sigma) =_{def} \{x \in L_U \mid \mathcal{Q}(p) \stackrel{\sigma \cdot x}{\Longrightarrow}\} \cup \{\delta \mid \mathcal{Q}(p) \textbf{ after } \sigma \textbf{ refuses } L_U\}$ and $\delta \notin L$. By restricting the set $\mathcal{F}$ to sets of traces that depend on the specification s asynchronous **conf**-like relations can be defined, and their properties can be investigated. [44] presents an algorithm that is able to derive a complete test suite for such classes of queue implementation relations.

The asynchronous testing theory for queue systems can be seen as an attempt to narrow the gap between testing based on synchronous theories (such as the theory for canonical testers, section 4) and testing based on asynchronous theories via inputs and outputs (e.g., testing based on systems specified in SDL [10]). However, queue systems are restricted in their use; the theory is only appropriate for systems that explicitly communicate with each other via two unbounded FIFO queues, and other communication architectures (such as having more than two queues, allowing media to be non-FIFO, etc.) cannot be described in this model. Fortunately, the requirement that systems communicate with each other via unbounded FIFO queues turns out not to be necessary in order to apply the ideas discussed before: the only essential requirements are that the set of actions can be partitioned in a set of input actions L_I and a set of output actions L_U, and that implementations can never refuse input actions, whereas the environment is always prepared to accept output actions (where input actions and output actions are viewed from the perspective of the system under test). By considering in figure 2 the input queue as part of the implementation, and the output queue as part of the environment, queue systems are just a special case of systems satisfying this requirement. This observation has triggered research on systems that are never able to refuse input actions. We discuss three of such (marginally) different system models: input/output automata (IOA), input/output state machines (IOSM), and input/output transition systems (IOTS).

Input/Output Automata (IOA) Formally, a transition system p where the set of labels L is partitioned in a set of input labels L_I and a set of output labels L_U (i.e., $L = L_I \cup L_U$ and $L_I \cap L_U = \emptyset$), and that satisfies

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

is called an input/output automaton (IOA) [28]. By explicitly distinguishing between inputs and outputs, implementations and their observers are allowed to communicate in a complementary manner; observers control and supply the input actions, while implementations control and produce output actions. [36] applies the ideas from [13] to implementations that are assumed to be modelled as IOA.

Input/Output State Machines (IOSM) [32] introduces a model called (complete) input/output state machines (IOSM) that differs from IOA by requiring that IOSM must have a finite number of states. This model is used as a semantic underpinning for test derivation in the tool TVEDA [11].

Input/Output Transition Systems (IOTS) According to [40, 41] an input/output transition system (IOTS) is a transition system that marginally differs from IOA and IOSM. Like in IOA the set of labels is partitioned in a set of input labels L_I and a set of output labels L_U, but the difference is that instead of requiring that inputs are always strongly enabled, we require for IOTS that inputs are weakly enabled, i.e., $p \in \mathcal{LTS}(L_I \cup L_U)$ is IOTS iff

$$\forall p' \in der(p), \forall a \in L_I : p' \stackrel{a}{\Longrightarrow} \tag{3}$$

The above condition is strictly weaker than the one imposed on IOA. Consequently, test theory for IOTS is more general than for IOA. Note that queue systems can be seen as subclass of IOTS: every implementation in a queue context satisfies the condition imposed on IOTS, but not vice versa.

Although IOA, IOSM and IOTS differ marginally, we concentrate here on the most liberal one, namely IOTS, and discuss testing theory for implementations that can be modelled as IOTS in the same way as [40, 41]. We denote the universe of IOTS with input set L_I and output set L_U by $\mathcal{IOTS}(L_I, L_U)$.

Inputs and outputs are complementary: inputs for IUT are outputs from the perspective of the environment, and outputs produced by the IUT are inputs for the environment (figure 3). By convention, we will use the terms inputs and outputs always from the perspective of the IUT. Many existing implementations satisfy the test assumption that inputs are always enabled (that is, they can be modelled as an IOTS), and that inputs are initiated and controlled by the environment, whereas outputs are initiated and controlled by the implementation. From now on we will assume that implementations can

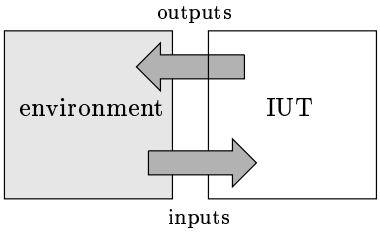

Figure 3 Asymmetric communication between IUT and its environment.

be modelled as members of $\mathcal{IOTS}(L_I, L_U)$. However, if the implementation is not able to refuse inputs initiated by the environment, then it is reasonable to assume that the environment is not able to refuse outputs produced by the implementation. If we allow the environment to also observe the inability of implementations to produce any output by means of θ (see section 3), then this means that the behaviour of the environment can be modelled as a member of $\mathcal{IOTS}(L_U, L_I \cup \{\theta\})$. By instantiating the set of observers with $\mathcal{IOTS}(L_U, L_I \cup \{\theta\})$ and the set of implementations with $\mathcal{IOTS}(L_I, L_U)$, *input/output refusal preorder*, $\leq_{ior}$ [41], is defined following the extensional characterisation given in equation (1))

$$i \leq_{ior} s \quad =_{def} \quad \forall u \in \mathcal{IOTS}(L_U, L_U \cup \{\theta\}) : \quad obs_c^{\theta}(u,i) \subseteq obs_c^{\theta}(u,s) \text{ and } obs_t^{\theta}(u,i) \subseteq obs_t^{\theta}(u,s) \quad (4)$$

Since implementations are, by assumption, always prepared to accept input actions and the environment is always prepared to accept output actions, the only way to deadlock for these kind of systems is if the environment does not provide an input action, and the IUT does not produce an output action. The inability to produce outputs is an important characteristic of implementations that is observable by observers that are equipped with a θ-label. Following terminology introduced in [43] we call a state *quiescent* if no output action or internal transition can be produced from this state; $\delta(s) =_{def} s \xrightarrow{L_U} s$. Observing quiescence can be made explicit by means of a special event with label $\delta \notin L$; δ can be observed if the implementation is in a quiescent state. [41] proves that $\leq_{ior}$ can also be characterised intensionally in terms of inclusion between the sets of output actions, including δ, that the implementation and the specification can perform. Formally, $i \leq_{ior} s$ iff after all failure traces in $\sigma \in (L \cup \{L_U\})^*$ the outputs produced by the implementation are specified, and the implementation may only refuse to produce outputs if the specification does so, viz.,

$$i \leq_{ior} s \quad \textit{iff} \quad \forall \sigma \in (L \cup \{L_U\})^* : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$$

where $out(S) =_{def} \{x \in L_U \mid \exists s \in S : s \stackrel{x}{\Longrightarrow}\} \cup \{\delta \mid \exists s \in S : \delta(s)\}$ for S a set of states. A failure trace in $(L \cup \{L_U\})^*$ is called a *suspension trace*; $s\text{-}traces(p) =_{def} f\text{-}traces(p) \cap (L \cup \{L_U\})^*$.

Since checking $out(\, i \textbf{ after } \sigma\,) \subseteq out(\, s \textbf{ after } \sigma\,)$ for all suspension traces is hard to achieve by means of testing, the above characterisation can be relaxed by checking this condition for fewer suspension traces. In general, for each $\mathcal{F} \subseteq (L \cup \{L_U\})^*$ an implementation relation $\textbf{ioco}_{\mathcal{F}}$ can be defined that only checks the condition $out(\, i \textbf{ after } \sigma\,) \subseteq out(\, s \textbf{ after } \sigma\,)$ for $\sigma \in \mathcal{F}$, viz.,

$$i \textbf{ ioco}_{\mathcal{F}}\ s \quad =_{def} \quad \forall \sigma \in \mathcal{F} : out(\, i \textbf{ after } \sigma\,) \subseteq out(\, s \textbf{ after } \sigma\,) \tag{5}$$

Note the correspondence in structure between equation (5) and equation (2).

Validating a system by means of testing involves, in practice, checking how the system reacts to stimuli from the environment. The relation $\textbf{ioco}_{\mathcal{F}}$ captures this intuitive notion of correctness [41]: correct implementations may only give reactions that are specified. From now on we focus on the generation of tests for implementation relation $\textbf{ioco}_{\mathcal{F}}$ with $\mathcal{F} \subseteq s\text{-}traces(s)$.

For testing implementations in $\mathcal{IOTS}(L_I, L_U)$ it suffices to restrict the class of tests to a specific subclass of $\mathcal{IOTS}(L_U, L_I \cup \{\theta\}) \subseteq \mathcal{LTS}(L \cup \{\theta\})$ in order to check whether systems are $\leq_{ior}$-related or not. In particular, [41] shows that it suffices to restrict to deterministic members with finite behaviour of $\mathcal{LTS}(L_U, L_I \cup \{\theta\})$, such that either a single input action is supplied, or all output actions, including θ, can be observed. There is no need to introduce additional nondeterminism in the test, and, since all errors occur within a finite depth of a transition system, they can be found using a finite series of experiments. Formally, a test case t for $\textbf{ioco}_{\mathcal{F}}$ is a labelled transition system over $L_I \cup L_U \cup \{\theta\}$ such that (i) t is deterministic and has finite behaviour, (ii) there exists two states **pass**, **fail** such that $init(\textbf{ pass }) = init(\textbf{ fail }) = \emptyset$, and (iii) for all states $t' \in der(t)$ with $t' \neq \textbf{ pass }, \textbf{ fail }$ we have $init(t') = \{a\}$ for some $a \in L_I$, or $init(t') = L_U \cup \{\theta\}$. The universe of tests over L_U and L_I is denoted as $TESTS(L_U, L_I)$, and a test suite T is a set of tests: $T \subseteq TESTS(L_U, L_I)$. We denote test cases with a LOTOS-like syntax: $t := a; t \mid \sum T \mid \textbf{ pass } \mid \textbf{ fail }$ where $\emptyset \neq T \subseteq TESTS(L_U, L_I)$. The semantics of these expressions is the obvious one: $a;T$ is able to do action a, after which it behaves as t, $\sum T$ behaves make a choice between the behaviours of T, and **pass**, **fail** cannot perform any action at all. Instead of $\sum\{B_1, B_2\}$ we also write $B_1 + B_2$.

In order to give an indication about the (in)correctness of implementations based on observations made after execution of a test case, a verdict (success or failure) is assigned to implementations. For brevity we will identify the verdicts success and failure with the states **pass** and **fail**, respectively. The execution of a test is modelled in terms of test runs. A test run $\sigma \in L^*$ of test t and implementation i is a trace that $t \rceil\!| \, i$ can perform such that test

t ends in **pass** or **fail**: $\exists i' : t \rceil\!| \; i \stackrel{\sigma}{\Longrightarrow} \textbf{pass} \; \rceil\!| \; i'$ or $t \rceil\!| \; i \stackrel{\sigma}{\Longrightarrow} \textbf{fail} \; \rceil\!| \; i'$. An implementation i is said to fail test t if there exists a test run of $t \rceil\!| \; i$ that ends in **fail**, i.e., $i \; \textit{fails} \; t =_{def} \exists \sigma \in L^*, \exists i' : t \rceil\!| \; i \stackrel{\sigma}{\Longrightarrow} \textbf{fail} \; \rceil\!| \; i'$. Dually, an implementation passes test t if it does not fail test t: $i \; \textit{passes} \; t =_{def} \neg(i \; \textit{fails} \; t)$. We shall say that an implementation passes a set of test cases T, denoted as $i \; \textit{passes} \; T$, if it passes all tests in test suite T. The failing of a test suite is defined conversively. To link the passing and failing of an implementation to the correctness and incorrectness of this implementation, respectively, the verdicts **pass** and **fail** in the test case have to be assigned carefully. Ideally, test cases are designed in such a way that correct implementations always pass this set of tests (soundness), and incorrect implementations always fail this set of tests (exhaustiveness). Since exhaustiveness is difficult (if not impossible) to reach in practice we require soundness when designing test suites, and strive for exhaustiveness; erroneous behaviour is likely to be detected by the test suite. A test suite that is both sound and exhaustive is called complete.

Now we can give a test generation algorithm that is able to produce test cases in $TESTS(L_U, L_I)$ from a specification $s \in \mathcal{LTS}(L_I \cup L_U)$ with respect to implementation relation $\mathbf{ioco}_{\mathcal{F}}$, and where it is assumed that implementations can be modelled as members of $\mathcal{IOTS}(L_I, L_U)$. The algorithm is inspired by the one presented in [41] and given in figure 4. In the algorithm we use the notation $\overline{\sigma}$ for a trace in which all occurrences of δ are replaced by the deadlock detection symbol θ that is used to observe this output deadlock, and vice versa: $\overline{\sigma}$ leaves other actions unchanged.

The algorithm is parameterised over a set of suspension traces $\mathcal{F}$ and a specification $s \in \mathcal{LTS}(L_I \cup L_U)$. For each suspension trace in $\mathcal{F}$ the algorithm produces a test case that is able to check that the implementation produces an valid output(cf. equation (5)). The algorithm keeps track of the current states of the specification that are exercised by means of the variable S, which is initialised with $\{s_0\}\,\textbf{after}\,\epsilon$ (where s_0 is the initial state of specification s). Tests are constructed by recursive application of three different steps. Step 1 is used to terminate a test case by assigning **pass**. Step 2 supplies an input $a \in L_I$, that is specified by some trace in $\mathcal{F}$, to the implementation, updates the set of possible current states S of the specification and the set of suspension traces $\mathcal{F}$ that need to be verified, and recursively proceeds. In step 3 the output actions that the implementation produces are checked for validity: a **fail** is assigned if the implementation produces an output that cannot be produced by the specification, and we have already executed a trace in $\mathcal{F}$, i.e., $\epsilon \in \mathcal{F}$. In this case there is evidence that the implementation violates equation (5). If the implementation produces an unspecified output for which no checking is required ($\epsilon \notin \mathcal{F}$) a **pass** is assigned: there is no evidence of incorrectness with respect to $\mathbf{ioco}_{\mathcal{F}}$. In case the implementation produces a specified output then checking needs to be continued, i.e., the algorithm recursively proceeds, where S and $\mathcal{F}$ are updated accordingly.

Input: specification $s \in \mathcal{LTS}(L_I \cup L_U)$
Input: set of failure traces $\mathcal{F} \subseteq (L \cup \{L_U\})^*$
Output: test case $\Pi_{\mathcal{F},S} \in TESTS(L_U, L_I)$.

Initial value: $S = \{s_0\}$ **after** ϵ, where s_0 is the initial state of s.

Apply one of the following non-deterministic choices recursively.

1. (* terminate the test case *)

   $\Pi_{\mathcal{F},S} :=$ **pass**

2. (* supply an input to the implementation *)
   Take $a \in L_I$ such that $\mathcal{F}' \neq \emptyset$, then

   $\Pi_{\mathcal{F},S} := a; \Pi_{\mathcal{F}',S'}$

   where $\mathcal{F}' = \{\sigma \mid a \cdot \sigma \in \mathcal{F}\}$ and $S' = S$ **after** a
3. (* check the next output of the implementation *)

$$\begin{aligned}\Pi_{\mathcal{F},S} := \quad & \sum\{x;\ \textbf{fail} \mid x \in L_U \cup \{\theta\}, \overline{x} \notin out(S), \epsilon \in \mathcal{F}\} \\ & + \sum\{x;\ \textbf{pass} \mid x \in L_U \cup \{\theta\}, \overline{x} \notin out(S), \epsilon \notin \mathcal{F}\} \\ & + \sum\{x; \Pi_{\mathcal{F}'_x,S'_x} \mid x \in L_U \cup \{\theta\}, \overline{x} \in out(S)\}\end{aligned}$$

where $\mathcal{F}'_x = \{\sigma \mid \overline{x} \cdot \sigma \in \mathcal{F}\}$ and $S'_x = S$ **after** $\overline{x}$

Figure 4 Test generation algorithm.

When executing tests obtained using the algorithm in figure 4, implementations that are **ioco**$_{\mathcal{F}}$-correct will never be considered erroneous, i.e., there is no test run that will lead to a **fail**-state when these tests are executed against **ioco**$_{\mathcal{F}}$-correct implementations (soundness). Moreover, executing *all* (usually infinitely many) test cases that are generated by the algorithm can detect all erroneous implementations (exhaustiveness) [41].

Theorem 1 Let $s \in \mathcal{LTS}(L_I \cup L_U)$ and $\mathcal{F} \subseteq s\text{-}traces(s)$.

1. A test case obtained with the algorithm depicted in figure 4 is sound for s with respect to **ioco**$_{\mathcal{F}}$.
2. The set of all test cases that can be obtained by the algorithm depicted in figure 4 is exhaustive for s with respect to **ioco**$_{\mathcal{F}}$.

The testing theory for IOTS is expected to be more useful, due to the distinction between inputs and outputs, than theories that do not make such explicit distinction. This is also motivated by the existence of the tool TVEDA, that originated from protocol testing experience. TVEDA can derive tests that are similar to tests that can be derived by algorithm 4. In [32] an attempt to provide an theoretical foundation behind TVEDA was given, which resulted in an implementation relation R_1 that is very similar to $\mathbf{ioco}_{traces(s)}$. Moreover, since algorithm 4 abstracts from the branching structure of implementations and only deals with trace structures, it is expected that data aspects are more easy to incorporate than in the algorithm for the construction of canonical testers (section 4): in testing for $\mathbf{ioco}_{\mathcal{F}}$ the explosion from countably branching structures to uncountably branching structures (that is present in the construction of canonical testers) is avoided.

6 CLOSING THE CIRCLE

Although the shift from symmetric to asymmetric communication allows for a more realistic modelling of the testing process, still some criticism can be ventilated towards the asymmetric model. As indicated in [36] the requirement that implementations must be modelled as members of $\mathcal{IOTS}(L_I, L_U)$ is still restrictive; not all implementations satisfy the requirement that inputs are always enabled (e.g., systems that communicate with each other via *bounded* queues; if the queue is full, no input can be accepted any more). Furthermore, observers for IOTS are forced to accept *all* outputs, even if these outputs occur at geographically dispersed places, and thereby a possible distribution of the environment itself is ignored. As, in practice, many distributed implementations communicate with their environment via distributed locations, or PCOs (Point of Control and Observation [22]), the distributed nature of the interfaces should be taken into account when testing these systems. For example, the standardised language SDL [10] explicitly incorporates the different locations through which an implementation communicates with its environment by means of channels, and the standardised test notation TTCN [22, part 3] is also able to express the sending and reception of messages to specific locations. In the IOTS model it is not possible to exploit the distributed nature of interfaces. An example of a system that cannot be described as an IOTS is depicted in figure 5.

In order to overcome these deficiencies recent research has lead to a model that refines the IOTS model, and, at the same time, unifies both the symmetric and asymmetric communication paradigm in a single framework. Basically, this is done by making two refinements to the IOTS model that are sufficient to model systems like the one in figure 5, i.e., (i) distinguishing between different locations, or channels, through which an implementation communicates with its environment, and (ii) weaken the requirement for IOTS that all input

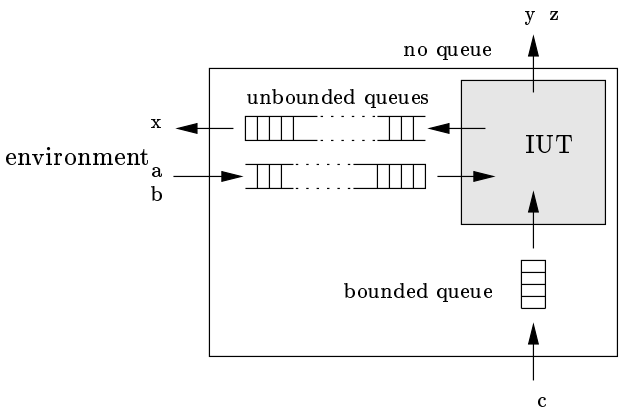

Figure 5 Example of a multi input/output queue system.

actions have to be continuously enabled. We will briefly elaborate on both refinements.

Ad (i) Instead of partitioning the label set L in an input set L_I and an output set L_U, these sets themselves are partitioned in one or more groups (i.e., sets) of actions; $L_I = \bigcup_{1 \leq i \leq n} L_I^i$ and $L_U = \bigcup_{1 \leq j \leq m} L_U^j$. Each group of actions defines a *channel* where these actions may occur. By distinguishing between the different channels of an implementation an external observer is (potentially) able to observe the inability of an implementation to produce an output at some output channel, while at another output channel the implementation can produce an output. Note that this is not possible in the IOTS model: observers of an IOTS are not able to check that subsets of outputs cannot occur.

Ad (ii) Instead of requiring that input actions must always be enabled, it is required for each input channel that "if an input in a channel is enabled, then all inputs at this channel should be simultaneously enabled", i.e., For each input channel L_I^i we require

$$\forall p' \in der(p),\ \text{if } \exists a \in L_I^i : p' \stackrel{a}{\Longrightarrow} \text{ then } \forall b \in L_I^i : p' \stackrel{b}{\Longrightarrow} \tag{6}$$

This requirement is strictly weaker than the one imposed on IOTS where all inputs are always enabled. In particular, this requirement allows us to model communication by means of bounded queues; all inputs in a channel are only enabled if the queue is not full.

Systems that are modelled with these two refinements are called *multi input/output transition systems* (MIOTS) [40, 41]. The class of MIOTS under

consideration depends on the specific partitioning of the channels, that is, the set of implementations in MIOTS is parameterised by the location of interfaces through which these implementations communicate with their environment. Such systems can be tested by means of observers that are also modelled as MIOTS (where input channels of the system are output channels for the observer, and vice versa). This yields an implementation relation $\leq_{mior}$ defined similarly as $\leq_{ior}$ (equation (4)), and a characterisation in terms of $\mathbf{mioco}_{\mathcal{F}}$ similarly as $\mathbf{ioco}_{\mathcal{F}}$ (equation (5)), that are parameterised over the distribution of the interfaces of the implementation with the environment. [19] investigates testing theory for MIOTS, and they relate the different instances of $\leq_{mior}$ for the specific distributions of interfaces.

MIOTS allow to relate synchronous and asynchronous testing theories by varying the granularity of the interfaces, and thus close the circle with refusal testing (section 1, [33]). Moreover, the different instances of $\leq_{mior}$ for the specific distributions of the interfaces are related. If all inputs run via a single channel and all outputs run via a single channel, and requirement (6) is strengthened to requirement (3) for inputs on this single input channel, then $\leq_{mior}$ corresponds to $\leq_{ior}$. On the other hand, if each action runs through a separate channel, i.e., the sets L_I and L_U are partitioned in singletons, then $\leq_{mior}$ equals $\leq_{rf}$ [19]. This means that the symmetric testing theory discussed in section 3 and the asymmetric testing theory discussed in section 5 are unified in a single testing framework, and the test algorithm presented in [19] is able to generate tests for $\leq_{rf}$, $\leq_{ior}$, $\leq_{mior}$, $\mathbf{ioco}_{\mathcal{F}}$ and $\mathbf{mioco}_{\mathcal{F}}$.

7 CONCLUSIONS

History In this paper we sketched the developments that have taken place (and still take place) in testing based on labelled transition systems. The seminal work in [13, 12] introduces a testing theory for labelled transition systems based on the assumption that communication between systems and their environments is symmetric. They define, and compare, many testing relations by varying the class of tests, and the class of observations. [33] discusses a refinement of [13, 12] by allowing observers to continue after observation of deadlock. The first mature theory based on [13, 12] that presents an algorithm to derive tests from a specification is presented in [6, 7]. They discuss how to generate a test suite that can distinguish between correct and incorrect implementations (with respect to implementation relation **conf**).

As, in practice, communication between implementations and testers is often asymmetric, many approaches that incorporate such asymmetric communication have been done with the aim to apply testing theory to realistic systems (SDL [10], TTCN [22, part 3]). One of these approaches is [42]. They assume that communication between implementations and testers runs via two unbounded queues, and they define, and analyse, testing relations (so-

called queue preorders) for systems that communicate with their environment through these queues. A more general approach is taken by assuming that implementations can be modelled as input/output transition systems (IOTS). An IOTS is a LTS that makes an explicit distinction between input actions and output actions, and assumes that input actions are weakly enabled. In this way it isolates the relevant aspects of queue systems without requiring that communication with the environment is done via queues.

[41] applies the ideas of [33] to IOTS, and defines a testing theory for implementations that can be modelled as IOTS. They assume that the inability to produce output actions, i.e., quiescence, is observable, and define an implementation relation $\mathbf{ioco}_{\mathcal{F}}$ that captures the intuition of correctness in practice. They also present an algorithm that is able to derive a sound and complete set of tests from a specification. These tests resemble tests generated by the tool TVEDA [11] that originated from practical testing experience.

[19] refines the theory of IOTS by taking the distribution of the interfaces of implementations into account. They explicitly model the locations (also: PCOs or channels) where actions can take place, and they require that input actions per input channel are either simultaneously enabled or simultaneously disabled. Such systems are called multi input/output transition systems (MIOTS). For implementations that can be modelled as MIOTS refusal testing [33] is applied, and quiescence is assumed to be observable (cf. [36, 40, 41]). Similar to $\mathbf{ioco}_{\mathcal{F}}$ they define an implementation relation $\mathbf{mioco}_{\mathcal{F}}$ relative to the distribution of interfaces of implementations, and present an algorithm that is able to derive sound and complete test cases for $\mathbf{mioco}_{\mathcal{F}}$. This results in a testing theory that is parameterised over the granularity of interfaces of implementations. [19] shows that specific instances yield the traditional refusal testing theory of [33], and the refusal testing for IOTS [41], and hence incorporates both theories in a single framework.

Future The theory and the algorithm for IOTS/MIOTS can form the basis for the development of test generation tools. In order to use such tools in realistic testing experiments several aspects need elaboration. One of these aspects involves data handling. In many realistic applications data is involved. To deal with data in an efficient way the test generation algorithm has to incorporate such data aspects in a symbolic way; otherwise automation of tests is not feasible due to explosion in the data part. Another aspect concerns the well-known problem of test selection. As test suites grow in size, execution of all of the tests in the test suite becomes too expensive, and selections have to made; which tests are executed, and which are not? (Partial) solutions can be found in defining coverage measures, fault models, strengtening test assumptions, etc. [3, 23, 31]. Experiments in applying the algorithm to realistic problems have to be conducted in order to show the strengths and weaknesses in the testing theory for IOTS. A first trial in which a preliminary version of the theory for IOTS was applied to a simple protocol looks promising [38], but more experiments are needed to draw meaningful conclusions. Finally, the

relation between formalisms that incorporate channels (e.g., SDL, TTCN), and MIOTS needs further investigation.

8 REFERENCES

[1] S. Abramsky. Observational equivalence as a testing equivalence. *Th. Comp. Sc.*, 53(3):225–241, 1987.

[2] R. Alderden. COOPER, the compositional construction of a canonical tester. In S.T. Vuong, editor, *FORTE'89*, pages 13–17. North-Holland, 1990.

[3] G. Bernot. Testing against formal specifications: A theoretical view. In S. Abramsky et al., editor, *TAPSOFT'91, Volume 2*, pages 99–119. LNCS 494, Springer-Verlag, 1991.

[4] F. de Boer, J.W. Klop, and J. Rutten. Asynchronous communication in process algebra. In J.W. Bakker et al., editor, *REX Workshop on Semantics: Foundations and Applications*. LNCS 666, Springer-Verlag, 1993.

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

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

[7] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal et al., editor, *PSTV VIII*, pages 63–74. North-Holland, 1988.

[8] E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. A formal approach to conformance testing. In J. de Meer et al., editor, *Second International Workshop on Protocol Test Systems*, pages 349–363. North-Holland, 1990.

[9] A. Cavalli and M. Philippou. Some issues on testing theory and its applications. In T. Mizuno et al., editor, *Seventh International Workshop on Protocol Test Systems*. Chapman & Hall, 1995.

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

[11] M. Clatin, R. Groz, M. Phalippou, and R. Thummel. Two approaches linking test generation with verification techniques. In A. Cavalli et al., editor, *Eight International Workshop on Protocol Test Systems*. Chapman & Hall, 1996.

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

[13] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Th. Comp. Sc.*, 34:83–133, 1984.

[14] P. Doornbosch. Test derivation for Full LOTOS. Memorandum INF-91-51, University of Twente, Enschede, The Netherlands, 1991.

[15] K. Drira, P. Azéma, and F. Vernadat. Refusal graphs for conformance

tester generation and simplification: A computational framework. In A. Danthine et al., editor, *PSTV XIII*. North-Holland, 1993.

[16] S. Fujiwara et al. Test selection based on finite state models. *IEEE Trans. on Soft. Eng.*, 17(6):591–603, 1991.

[17] R.J. van Glabbeek. The linear time – branching time spectrum II (The semantics of sequential systems with silent moves). In E. Best, editor, *CONCUR'93*, LNCS 715, pages 66–81. Springer-Verlag, 1993.

[18] D. Gueraichi and L. Logrippo. Derivation of test cases for lap-b from a LOTOS specification. In S.T. Vuong, editor, *FORTE'89*. North-Holland, 1990. Also: technical report TR-89-18.

[19] L. Heerink and J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. CTIT technical report, University of Twente, Enschede, The Netherlands, 1997.

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

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

[22] ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework*. International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.

[23] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Proposed ITU-T Z.500 and Committee Draft on "Formal Methods in Conformance Testing"*. CD 13245-1. ISO – ITU-T, Geneve, 1996.

[24] R. Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma et al., editor, *PSTV IX*, pages 87–98. North-Holland, 1990.

[25] G. Leduc. Conformance relation, associated equivalence, and minimum canonical tester in LOTOS. In B. Jonsson et al., editor, *PSTV XI*. North-Holland, 1991.

[26] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Trans. on Comp.*, 43(3):306–320, 1994.

[27] G. Luo, A. Petrenko, and G. von Bochmann. Selecting test sequences for partially-specified nondeterministic finite state machines. Publication N.864, Université de Montréal, Montréal, Canada, 1993.

[28] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989. Also: Technical Report MIT/LCS/TM-373 (TM-351 revised), MIT, U.S.A., 1988.

[29] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[30] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das. Nondeterministic state machines in protocol conformance testing. In O. Rafiq, editor, *Sixth International Workshop on Protocol Test Systems*, pages 363–

378. North-Holland, 1994.

[31] M. Phalippou. Executable testers. In O. Rafiq, editor, *Sixth International Workshop on Protocol Test Systems*, pages 35–50. North-Holland, 1994.

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

[33] I. Phillips. Refusal testing. *Th. Comp. Sc.*, 50(2):241–284, 1987.

[34] D. H. Pitt and D. Freestone. The derivation of conformance tests from LOTOS specifications. *IEEE Trans. on Soft. Eng.*, 16(12):1337–1343, 1990.

[35] G. Schoemakers. Generation of canonical testers from recursive LOTOS specifications. Master's thesis, University of Twente, Enschede, The Netherlands, 1994.

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

[37] D.P. Sidhu and T.K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Trans. on Soft. Eng.*, 15(4):413–426, 1989.

[38] R. Terpstra, L. Ferreira Pires, L. Heerink, and J. Tretmans. Testing theory in practice: A simple experiment. In Z. Brezočnik et al., editor, *COST 247 International Workshop on Applied Formal Methods in System Design*, pages 168–183, Maribor, Slovenia, 1996.

[39] J. Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[40] J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[41] J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software – Concepts and Tools*, 17:103–120, 1996.

[42] J. Tretmans and L. Verhaard. A queue model relating synchronous and asynchronous communication. In R.J. Linn and M.Ü. Uyar, editors, *PSTV XII*, IFIP Transactions, pages 131–145. North-Holland, 1992.

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

[44] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In G. Bochmann et al., editor, *Fifth International Workshop on Protocol Test Systems.* North-Holland, 1993.

[45] C. D. Wezeman. The CO-OP method for compositional derivation of conformance testers. In E. Brinksma et al., editor, *PSTV IX*, pages 145–158. North-Holland, 1990.

[46] M. Yannakakis and D. Lee. Testing finite state machines: Fault detection. *Journal of Computer and System Sciences*, 50(2):209–227, 1995.