Testing Concurrent Systems: A Formal Approach

Jan Tretmans

University of Twente *
Faculty of Computer Science, Formal Methods and Tools research group
P.O. Box 217, 7500 AE Enschede, The Netherlands
tretmans@cs.utwente.nl

Abstract. This paper discusses the use of formal methods in testing of concurrent systems. It is argued that formal methods and testing can be mutually profitable and useful. A framework for testing based on formal specifications is presented. This framework is elaborated for labelled transition systems, providing formal definitions of conformance, test execution and test derivation. A test derivation algorithm is given and its tool implementation is briefly discussed.

1 Introduction

During the last decades much theoretical research in computing science has been devoted to formal methods. This research has resulted in many formal languages and in verification techniques, supported by prototype tools, to verify properties of high-level, formal system descriptions. Although these methods are based on sound mathematical theories, there are not many systems developed nowadays for which correctness is completely formally verified using these methods.

On the other hand, the current practice of checking correctness of computing systems is based on a more informal and pragmatic approach. Testing is usually the predominant technique, where an implementation is subjected to a number of tests which have been obtained in an ad-hoc or heuristic manner. A formal, underlying theory for testing is mostly lacking.

The combination of testing and formal methods is not very often made. Sometimes it is claimed that formally verifying computer programs would make testing superfluous, and that, from a formal point of view, testing is inferior as a way of assessing correctness. Also, some people cannot imagine how the practical, operational, and 'dirty-hands' approach of testing could be combined with the mathematical and 'clean' way of verification using formal methods. Moreover, the classical biases against the use of formal verification methods, such as that formal methods are not practical, that they are not applicable to any real system

* This research is supported by the Dutch Technology Foundation STW under project STW TIF.4111: *Côte de Resyste* – COnformance TEsting of REactive SYSTEms; URL: http://fmt.cs.utwente.nl/CdR.

Jos C.M. Baeten, Sjouke Mauw (Eds.): CONCUR'99, LNCS 1664, pp. 46–65, 1999.

but very simple toy systems, and that they require a profound mathematical training, do not help in making test engineers adopt formal methods.

Fortunately, views are changing. Academic research on testing is increasing, and even the most formal verifyer admits that a formally verified system should still be tested. (Because: Who verified the compiler? And the operating system? And who verified the verifyer?). On the other hand, formal methods are used in more and more software projects, in particular for safety critical systems, and also the view that a formal specification can be beneficial during testing is getting more support.

The aim of this paper is to strengthen this process of changing views. To that purpose, this paper discusses how testing can be performed based on formal specifications, and how advantage can be obtained in terms of precision, clarity and consistency of the testing process by adopting this formal approach. Also, it will be shown how the use of formal methods helps automating the testing process, in particular the automated derivation of tests from formal specifications. The discussion about testing and formal methods will support the following claims: (*i*) formal methods and testing are a perfect couple; (*ii*) testing and formal verification are both necessary; (*iii*) a formally verified specification is a good starting point for testing; (*iv*) formal testing is a good starting point for introducing formal methods in software development.

The structure of this paper is as follows. In the next section we start with some informal discussion on classical software testing, see, e.g., [38,3]. Section 3 then discusses a formal, generic framework for testing with formal methods. Section 4 makes this framework more specific by instantiating it for the formalism of labelled transition systems. Section 5 discusses tool support and a concrete application of testing a simple protocol based on the labelled transition system testing theory. Finally, section 6 comes back to the claims made above and discusses some open issues.

The intention of this paper is to give an idea about how testing and formal methods can be mutually beneficial. A complete overview of formal approaches to testing is outside the scope of this paper. Other approaches exist, e.g., [21] for Abstract Data Type testing, and certainly other instantiations of the generic framework of section 3 are possible, e.g., with Finite-State Machines (Mealy machines) [7,35]. Also it is not the intention to give a complete and precise overview of testing for labelled transition systems. However, the branch of testing theory, which is elaborated in section 4, is shown to be a realistic and practically applicable approach in section 5. Moreover, many pointers to the literature are provided which allow to explore alternatives and to study further details.

2 Software Testing

What Is Testing? Testing is an operational way to check the correctness of a system implementation by means of experimenting with it. Tests are applied to the implementation under test in a controlled environment, and, based on observations made during the execution of the tests, a verdict about the correct

functioning of the implementation is given. The correctness criterion that is to be tested is given by the system specification; the specification is the basis for testing.

Testing is an important technique to increase confidence in the quality of a computing system. In almost any software development trajectory some form of testing is included.

Sorts of Testing. There are many different kinds of testing. In the first place, different aspects of system behaviour can be tested: Does the system have the intended functionality and does it comply with its functional specification (functional tests or conformance tests)? Does the system work as fast as required (performance tests)? How does the system react if its environment shows unexpected or strange behaviour (robustness tests)? Can the system cope with heavy loads (stress testing)? How long can we rely on the correct functioning of the system (reliability tests)? What is the availability of the system (availability tests)?

Moreover, testing can be applied at different levels of abstraction and for different levels of (sub-)systems: individual functions, modules, combinations of modules, subsystems and complete systems can all be tested.

Another distinction can be made according to the parties or persons performing (or responsible for) testing. In this dimension there are, for example, system developer tests, factory acceptance tests, user acceptance tests, operational acceptance tests, and third party (independent) tests, e.g., for certification.

A very common distinction is the one between black box and white box testing. In black box testing, or functional testing, only the outside of the system under test is known to the tester. In white box testing, also the internal structure of the system is known and this knowledge can be used by the tester. Naturally, the distinction between black and white box testing leads to many gradations of grey box testing, e.g., when the module structure of a system is known, but not the code of each module.

In this paper, we concentrate on black box, functional testing, also called *conformance testing*. We do not care about the level of (sub-)systems or who is performing the testing. Key points are that there is a system implementation exhibiting behaviour and that there is a specification. The specification is a prescription of what the system should do; the goal of testing is to check, by means of testing, whether the implemented system indeed satisfies this prescription. In particular, the rest of this paper will consider a conformance testing process based on specifications which are given in a formal notation.

Confusion of Tongues. Sometimes the term testing is also used for performing static checks on the program code, e.g., checking declarations of variables using a static checker, or code inspections. This kind of testing is then denoted by *static testing*. However, we restrict to *dynamic testing*, i.e., testing consisting of really executing the implemented system, as described above. Another broader use of the term testing is to include *monitoring*. Monitoring is then called *passive testing* as opposed to *active testing* as described above, where the tester

has active control over the test environment, and a set of predefined tests is executed. A third extension of the term testing, sometimes made, is to include all checking activities in the whole software development trajectory, e.g., reviews and inspections.

The Testing Process. In the conformance testing process there are two main phases: *test generation* and *test execution*. Test generation involves analysis of the specification and determination of which functionalities will be tested, determining how these can be tested, and developing and specifying test scripts. Test execution involves the development of a test environment in which the test scripts can be executed, the actual execution of the test scripts and analysis of the execution results and the assignment of a verdict about the well-functioning of the implementation under test.

Other important activities in the testing process are test management and test maintenance. In particular, test maintenance is often underestimated. It involves recording and documenting the test scripts, test environments, used test tools, relating test sets to versions of specifications and implementations, with the aim of making the testing process repeatable and reusable, in particular for regression testing. Regression testing is the re-testing of unmodified functionality in case of a modification of the system. It is one of the most expensive (and thus often deliberately neglected) aspects of testing.

Test Automation. Testing is a difficult, expensive, time-consuming and labour-intensive process. Moreover, testing is (should be) repeated each time a system is modified. Hence, testing would be an ideal candidate for automation.

The main class of commercially available test tools are *record & playback* tools (capture and replay tools) which support the test execution process. Record & playback tools are able to record user actions at a (graphical) user interface, such as keyboard and mouse actions, in order to replay these actions at a later point in time. In this way a recorded test can be replayed several times, which may be advantageous during regression testing.

For the test generation phase there are tools which are able to generate large amounts of input test data. However, these tools are mainly used for performance and stress tests and hence, are outside the scope of this paper. Some tools exist that are able to generate a set of tests with the same structure based on a template of a test case by only varying the input parameters in this template. In the area of communication protocol testing there exist some (prototype) test tools that can (semi-) automatically generate test cases for conformance testing from a formal specification. Some of these tools will be briefly described in section 5.

To relate test cases to the requirements that they test, standard requirements management tools can be used, but such tools are not specific for testing. The main functionality of such tools is to relate high level system requirements to (lower level) sub-system requirements and to relate requirements to test cases.

A kind of test tools which are used during test execution, but which (should) influence test generation, are code coverage tools. Code coverage tools calculate the percentage of the system code executed during test execution according

to some criterion, e.g., "all paths", "all statements", or "all definition-usage combinations" of variables. They give an indication about the completeness of a set of tests. Note that this notion of completeness refers to the implemented code (white box testing); it does not say anything about the extent to which the requirements or the specification were covered.

3 Formal Framework for Testing

In section 2 the software testing process was described from a traditional perspective. Conformance testing was introduced as a kind of testing where the behaviour of a system is systematically tested with respect to the system's specification of functional behaviour.

In this section a framework is presented for the use of formal methods in conformance testing [10,44,32]. The framework can be used for testing of an implementation with respect to a formal specification of its functional behaviour. It introduces, at a high level of abstraction, the concepts used in a formal conformance testing process and it defines a structure which allows to reason about testing in a formal way. The most important part of this is to link the informal world of implementations, tests and experiments with the formal world of specifications and models. To this extent the framework introduces the concepts of conformance, i.e., functional correctness, testing, sound and exhaustive test suites, and test derivation. All these concepts are introduced at a generic level; sections 4 and 5 will show how to instantiate and apply these concepts.

Conformance. For talking about conformance we need implementations and specifications. The specifications are formal, so a universe of formal specifications denoted *SPECS* is assumed. Implementations are the systems that we are going to test, henceforth they will be called IUT, implementation under test, and the class of all IUT's is denoted by *IMPS*. So, conformance could be introduced by having a relation **conforms-to** $\subseteq$ *IMPS* $\times$ *SPECS* with IUT **conforms-to** s expressing that IUT is a correct implementation of specification s.

However, unlike specifications, implementations under test are real, physical objects, such as pieces of hardware or software; they are treated as black boxes exhibiting behaviour and interacting with their environment, but not amenable to formal reasoning. This makes it difficult to give a formal definition of **conforms-to** which should be our aim in a formal testing framework. In order to reason formally about implementations, we make the assumption that any real implementation IUT $\in$ *IMPS* can be modelled by a formal object $i_{\text{IUT}} \in$ *MODS*, where *MODS* is referred to as the universe of models. This assumption is referred to as the *test hypothesis* [6]. Note that the test hypothesis only assumes that a model i_{IUT} exists, but not that it is known a priori.

Thus the test hypothesis allows to reason about implementations as if they were formal objects, and, consequently, to express conformance by a formal relation between models of implementations and specifications. Such a relation is called an *implementation relation* **imp** $\subseteq$ *MODS* $\times$ *SPECS* [10,32]. Implementation IUT $\in$ *IMPS* is said to be correct with respect to $s \in$ *SPECS*,

IUT **conforms-to** s, if and only if the model $i_{\text{IUT}} \in \mathit{MODS}$ of IUT is **imp**-related to s: i_{IUT} **imp** s.

Observation and Testing. The behaviour of an implementation under test is investigated by performing experiments on the implementation and observing the reactions that the implementation produces to these experiments. The specification of such an experiment is called a *test case*, and the process of applying a test to an implementation under test is called *test execution*.

Let test cases be formally expressed as elements of a domain TESTS. Then test execution requires an operational procedure to execute and apply a test case $t \in \mathit{TESTS}$ to an implementation under test $\text{IUT} \in \mathit{IMPS}$. This operational procedure is denoted by $\text{EXEC}(t, \text{IUT})$. During test execution a number of observations will be made, e.g., occurring events will be logged, or the response of the implementation to a particular stimulus will be recorded. Let (the formal interpretation of) these observations be given in a domain of observations OBS, then test execution $\text{EXEC}(t, \text{IUT})$ will lead to a subset of OBS. Note that EXEC is not a formal concept; it captures the action of "pushing the button" to let t run with IUT. Also note that $\text{EXEC}(t, \text{IUT})$ may involve multiple runs of t and IUT, e.g., in case nondeterminism is involved.

Again, since $\text{EXEC}(t, \text{IUT})$ corresponds to the physical execution of a test case, we have to model this process of test execution in our formal domain to allow formal reasoning about it. This is done by introducing an observation function $\mathit{obs} : \mathit{TESTS} \times \mathit{MODS} \to \mathcal{P}(\mathit{OBS})$. So, $\mathit{obs}(t, i_{\text{IUT}})$ formally models the real test execution $\text{EXEC}(t, \text{IUT})$.

In the context of an *observational framework* consisting of TESTS, OBS, EXEC and obs, it can now be stated more precisely what is meant by the test hypothesis:

$$\forall \text{IUT} \in \mathit{IMPS}\ \exists i_{\text{IUT}} \in \mathit{MODS}\ \forall t \in \mathit{TESTS} :\ \text{EXEC}(t, \text{IUT}) = \mathit{obs}(t, i_{\text{IUT}}) \quad (1)$$

This could be paraphrased as follows: for all real implementations that we are testing, it is assumed that there is a model, such that if we would put the IUT and the model in black boxes and would perform all possible experiments defined in TESTS, then we would not be able to distinguish between the real IUT and the model. Actually, this notion of testing is analogous to the ideas underlying testing equivalences [15,14], which will be elaborated for transition systems in section 4.

Usually, we like to interpret observations of test execution in terms of being right or wrong. So we introduce a family of *verdict functions* $\nu_t : \mathcal{P}(\mathit{OBS}) \to \{\mathbf{fail}, \mathbf{pass}\}$ which allows to introduce the following abbreviation:

$$\text{IUT}\ \mathbf{passes}\ t \quad \Longleftrightarrow_{\text{def}} \quad \nu_t(\text{EXEC}(t, \text{IUT})) = \mathbf{pass} \quad (2)$$

This is easily extended to a *test suite* $T \subseteq \mathit{TESTS}$: IUT **passes** $T \Leftrightarrow \forall t \in T :$ IUT **passes** t. Moreover, an implementation fails test suite T if it does not pass: IUT **fails** $T \Leftrightarrow$ IUT **passes** T.

Conformance Testing. Conformance testing involves assessing, by means of testing, whether an implementation conforms, with respect to implementation relation **imp**, to its specification. Hence, the notions of conformance, expressed by **imp**, and of test execution, expressed by EXEC, have to be linked in such a way that from test execution an indication about conformance is obtained. So, ideally, we would like to have a test suite T_s such that for a given specification s

$$\text{IUT } \textbf{conforms-to } s \iff \text{IUT } \textbf{passes } T_s \tag{3}$$

A test suite with this property is called *complete*; it can distinguish exactly between all conforming and non-conforming implementations. Unfortunately, this is a very strong requirement for practical testing: complete test suites are usually infinite, and consequently not practically executable. Hence, usually a weaker requirement on test suites is posed: they should be *sound*, which means that all correct implementations, and possibly some incorrect implementations, will pass them; or, in other words, any detected erroneous implementation is indeed non-conforming, but not the other way around. Soundness corresponds to the left-to-right implication in (3). The right-to-left implication is called *exhaustiveness*; it means that all non-conforming implementations will be detected.

To show soundness (or exhaustiveness) for a particular test suite we have to use the formal models of implementations and test execution:

$$\forall i \in \mathit{MODS} : \quad i \ \textbf{imp} \ s \iff \forall t \in T : \ \nu_t(\mathit{obs}(t,i)) = \textbf{pass} \tag{4}$$

Once (4) has been shown it follows that

$$\begin{array}{ll} & \text{IUT } \textbf{passes } T \\ \text{iff} & (* \text{ definition } \textbf{passes } T \ *) \\ & \forall t \in T : \ \text{IUT } \textbf{passes } t \\ \text{iff} & (* \text{ definition } \textbf{passes } t \ *) \\ & \forall t \in T : \ \nu_t(\text{EXEC}(t, \text{IUT})) = \textbf{pass} \\ \text{iff} & (* \text{ test hypothesis (1) } *) \\ & \forall t \in T : \ \nu_t(\mathit{obs}(t, i_{\text{IUT}})) = \textbf{pass} \\ \text{iff} & (* \text{ completeness on models (4) applied to } i_{\text{IUT}} \ *) \\ & i_{\text{IUT}} \ \textbf{imp} \ s \\ \text{iff} & (* \text{ definition of conformance } *) \\ & \text{IUT } \textbf{conforms-to } s \end{array}$$

So, if the completeness property has been proved on the level of models and if there is ground to assume that the test hypothesis holds, then conformance of an implementation with respect to its specification can be decided by means of a testing procedure.

Now, of course, an important activity is to devise algorithms which produce sound and/or complete test suites from a specification given an implementation relation. This activity is known as *test derivation*. It can be seen as a function $\mathit{der}_{\textbf{imp}} : \mathit{SPECS} \rightarrow \mathcal{P}(\mathit{TESTS})$. Following the requirement on soundness of test suites, such a function should only produce sound test suites for any specification $s \in \mathit{SPECS}$, so the test suite $\mathit{der}_{\textbf{imp}}(s)$ should satisfy the left-to-right implication of (4).

Extensions. Some extensions to and refinements of the formal testing framework can be made. Two of them are mentioned here. The first one concerns the *test architecture* [44,32]. A test architecture defines the environment in which an implementation is tested. It gives an abstract view of how the tester communicates with the IUT. Usually, an IUT is embedded in a test context, which is there when the IUT is tested, but which is not the object of testing. In order to formally reason about testing in context, the test context must be formally modelled. Sometimes, the term SUT – system under test – is then used to denote the implementation with its test context, whereas IUT is used to denote the bare implementation without its context.

The second extension is the introduction of *coverage* within the formal framework. The coverage of a test suite can be introduced by assigning to each erroneous implementation that is detected by a test suite a value and subsequently integrating all values. This can be combined with a stochastic view on erroneous implementations and a probabilistic view on test execution [9,26].

4 Labelled Transition Systems

One of the formalisms studied in the realm of conformance testing is that of *labelled transition systems*. A labelled transition system is a structure consisting of states with transitions, labelled with actions, between them. The formalism of labelled transition systems can be used for modelling the behaviour of processes, such as specifications, implementations and tests, and it serves as a semantical model for various formal languages, e.g., ACP [5], CCS [37], and CSP [28]. Also (most parts of) the semantics of standardized languages like LOTOS [30] and SDL [12], and of the modelling language PROMELA [29] can be expressed in labelled transition systems. We assume the basic definitions of labelled transition systems to be familiar; they can be found in many of the given references, e.g., in [45] the definitions are given in the same notation as they are used here (however, we will not consider internal actions τ in this section).

This section instantiates the generic, formal testing framework of section 3 with labelled transition systems. This means that the formal domains *SPECS*, *MODS* and *TESTS* will now consist of (some kind of) transition systems. In particular, it will be shown how the **ioco**-testing theory based on inputs, outputs and repetitive quiescence fits within the testing framework [45].

Traditionally, for labelled transition systems the term testing theory does not refer to conformance testing. Instead of starting with a specification to find a test suite that characterizes the class of its conforming implementations, these testing theories aim at defining implementation relations, given a class of tests: a transition system p is equivalent to a system q if any test case in the class leads to the same observations with p as with q (or more generally, p relates to q if for all possible tests, the observations made of p are related in some sense to the observations made of q). In terms of an observational framework as introduced in section 3, an implementation relation **imp** is defined by

$$p \ \mathbf{imp} \ q \quad \Longleftrightarrow_{\text{def}} \quad \forall t \in \mathit{TESTS} : \ \mathit{obs}(t,p) \ \otimes \ \mathit{obs}(t,q) \tag{5}$$

Many different relations can be defined by variations of the class of tests *TESTS*, the way observations *obs* are obtained, and the required relation between observations $\otimes$ [15,1,14,40,23,24].

Once an implementation relation has been defined, conformance testing involves finding a test derivation algorithm such that test suites can be derived from a specification which are sound, and, in some sense, minimal. Conformance testing for labelled transition systems has been studied especially in the context of testing communication protocols with the language LOTOS, e.g., [11,8,41,49,34,47,45,27].

For the discussion of the **ioco**-testing theory both kinds of testing theory are used: firstly, the implementation relation **ioco** is defined following the principle of (5); secondly, test derivation from specifications for **ioco** is investigated resulting in a sound and exhaustive test derivation algorithm.

In the remainder of this section we will successively instantiate all the ingredients of the formal testing framework of section 3 for **ioco**-based testing. These include *SPECS*, *IMPS*, *MODS*, **imp**, *TESTS*, *OBS*, ν_t, EXEC, *obs* and $der_{\mathbf{imp}}$. The description of the different concepts will be done semi-formally; full technical details can be found in [45]. The next section, section 5, will briefly discuss the use of this **ioco**-testing theory for building of software tools and for testing some simple communication protocol implementations based on LOTOS and PROMELA specifications.

Specifications. For specifications we allow to use labelled transition systems, or any formal language with a labelled transition system semantics. We require that the actions of the transition system are known and can be partitioned into inputs and outputs, denoted by L_I and L_U, respectively. However, we do not impose any restrictions on inputs or outputs. For $\mathcal{LTS}(L)$ the class of labelled transition systems over action alphabet L, $SPECS := \mathcal{LTS}(L_I \cup L_U)$.

Implementations and their Models. We assume implementations to be modelled by a special class of transition systems called *input-output transition systems*, which, inspired by Input/Output Automata (IOA) [36], have the property that any input action is always enabled in any state. For $\mathcal{IOTS}(L_I, L_U)$ the class of input-output transition systems with inputs in L_I and outputs in L_U, $MODS := \mathcal{IOTS}(L_I, L_U)$.

For *IMPS* we allow any computer system or program which can be modelled as an input-output transition system, i.e., a system which has distinct inputs and outputs, where inputs can be mapped 1:1 on L_I and outputs on L_U, and where inputs can always occur.

Implementation Relation. The implementation relation is instantiated with the relation $\mathbf{ioco} \subseteq \mathcal{IOTS}(L_I, L_U) \times \mathcal{LTS}(L_I \cup L_U)$, which is briefly discussed here.

The relation **ioco** inherits many ideas from other relations defined in the literature. Its roots are in the theory of testing equivalence and preorders [15,14], where testing preorder on transitions systems is defined following (5) using transition systems as tests, traces and completed traces of the synchronized parallel

composition of t and p as observations, and inclusion of observations as comparison criterion. Three developments, which build on these testing preorders, are of importance for **ioco**.

Firstly, a relation with more discriminating power than testing preorder was defined in [40] by having more powerful testers which can detect not only the occurrence of actions but also the absence of actions, i.e., refusals. We follow [33] in modelling the observation of a refusal by adding a special label $\theta \notin L$ to observers: $TESTS = \mathcal{LTS}(L \cup \{\theta\})$. While observing a process, a transition labelled with θ 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. This is modelled using a parallel operator $\rceil\!|\!|$ which is the usual synchronized parallel composition operator extended with the following inference rule to cope with the refusal-detecting features of θ:

$$u \xrightarrow{\theta} u', \quad \forall a \in L : \; u \xrightarrow{a} \!\!\!\!/ \;\; \text{or} \;\; p \xrightarrow{a} \!\!\!\!/ \qquad \vdash \qquad u \rceil\!|\!| \, p \xrightarrow{\theta} u' \rceil\!|\!| \, p$$

The implementation relation defined in this way is called *refusal preorder*.

A second development was the definition of a weaker implementation relation **conf** that is strongly related to testing preorder [11]. It is a modification of testing preorder 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. In other words, **conf** requires that an implementation does what it should do, not that it does not do what it is not allowed to do. Several test generation algorithms have been developed for the relation **conf** [41,49], among which the canonical tester theory [8], corresponding tools have been implemented [17,2], and extensions have been studied [34,16].

The third development of importance for **ioco** was the application of the principles of testing preorder to Input/Output Automata in [42]. It was shown that testing preorder coincides with quiescent trace preorder introduced in [46] when requiring that inputs are always enabled.

The relation **ioco** inherits from all these developments. The definition of **ioco** follows the principles of testing preorder (5) with tests that can also detect the refusal of actions as in refusal preorder. Outputs and always enabled inputs are distinguished analogous to IOA, and, moreover, a restriction is made to only the traces of the specification as in **conf**. The resulting relation **ioco** can be defined semi-formally as follows.

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

$$i \;\mathbf{ioco}\; s \quad \Longleftrightarrow_{\text{def}} \quad \forall \sigma \in \mathit{Straces}(s) : \quad \mathit{out}(\, i \;\mathbf{after}\; \sigma \,) \;\subseteq\; \mathit{out}(\, s \;\mathbf{after}\; \sigma \,)$$

where

- $p \;\mathbf{after}\; \sigma$ is the set of states in which transition system p can be after having executed the trace σ.

- $out(\, p \,\mathbf{after}\, \sigma \,)$ is the set of output actions which may occur in some state of $p \,\mathbf{after}\, \sigma$. Additionally, the special action δ, indicating *quiescence*, may occur if there is a *quiescent state* in $p \,\mathbf{after}\, \sigma$.
- A state p is *quiescent*, denoted by $p \xrightarrow{\delta} p$, if no output action can occur: $\forall x \in L_U : p \xrightarrow{x} \!\!\!\!/\;$.
- *Straces*(s) are the *suspension traces* of specification s, i.e., the traces in which the special action δ may occur beside normal input and output actions.

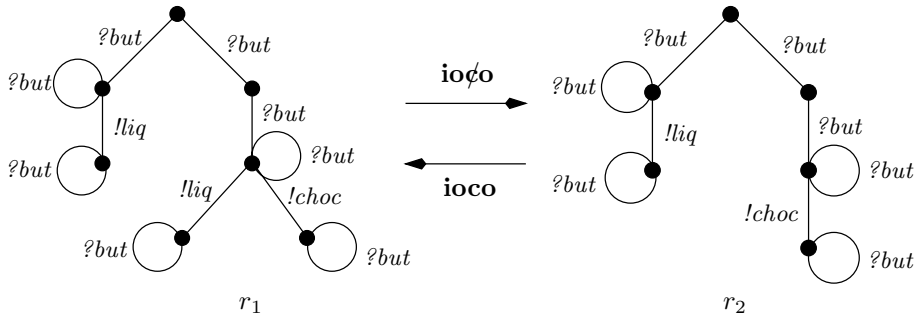

Fig. 1. (Non-)**ioco**-related input-output transition systems.

The relation **ioco** is chosen as implementation relation in our framework: **imp** := **ioco**. Informally, an implementation i is **ioco**-correct with respect to the specification s if i can never produce an output which could not have been produced by s in the same situation, i.e., after the same suspension trace. Moreover, i may only be quiescent, i.e., produce no output at all, if s can do so.

Example 1. Figure 1 gives two input-output transition systems with $L_I = \{\, ?but\}$ and $L_U = \{\, !liq, !choc\}$ and their **ioco**-relation.
r_1 **ioƒo** r_2 since $out(\, r_1 \,\mathbf{after}\, ?but{\cdot}\delta{\cdot}?but \,) = \{\, !liq, !choc\}$,
while $out(\, r_2 \,\mathbf{after}\, ?but{\cdot}\delta{\cdot}?but \,) = \{\, !choc\}$.

For more details about the relation **ioco**, argumentation for its use, and for more generic definitions we refer to [45]. New developments have led to a variant of **ioco**, called **mioco**, were explicit communication channels for actions are distinguished. Moreover, this **mioco**-theory allows to include all testing-based implementation relations, including refusal preorder, testing preorder, trace preorder, quiescent trace preorder and different variants of **ioco** and **mioco**, in a single lattice [27,25].

Tests. Also *TESTS* is instantiated with transition systems, but this time we add an extra label θ, as in [33], to model the detection of refusals, in particular the detection of the refusal of all outputs, i.e., quiescence. Moreover, we restrict tests to deterministic transition systems with finite behaviour, so that any test

execution is always finite and ends in a terminal state of the test case. We will denote these terminal states as either **pass** or **fail**. Finally, we require that for each non-terminal state s of a test case either $init(s) = \{a\}$ for some $a \in L_I$, or $init(s) = L_U \cup \{\theta\}$; $init(t)$ is the set of initial actions of t: $init(t) = \{a | t \xrightarrow{a}\}$. So, the behaviour of a test case is described by a (finite) tree where in each state either one specific input action can occur, or all outputs together with the special action θ. The special label $\theta \notin L \cup \{\delta\}$ will be used in a test case to detect quiescent states of an implementation, so it can be thought of as the communicating counterpart of a δ-action. It will usually be implemented by a kind of time-out.

Example 2. Figure 2 gives an example of a test case t.

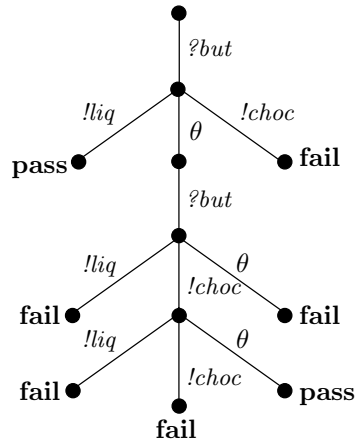

Fig. 2. A test case t

Observations. Observations are logs of actions, i.e., traces over $L \cup \{\theta\}$: $OBS := (L \cup \{\theta\})^*$.

Observation Function. The observation function *obs* is defined by the synchronized parallel composition of t and i ending in a final state of t:

$$obs(t, i) \quad =_{\text{def}} \quad \{\ \sigma \in (L \cup \theta)^* \ \mid\ t] | \, i \stackrel{\sigma}{\Longrightarrow} t'] | \, i',\ t' = \textbf{pass} \text{ or } t' = \textbf{fail}\ \}$$

Example 3. For r_1 (figure 1) there are three observations with t of figure 2:

$$t] | \, r_1 \stackrel{?but \cdot !liq}{\Longrightarrow} \textbf{pass}] | \, r_1'$$
$$t] | \, r_1 \stackrel{?but \cdot \theta \cdot ?but \cdot !liq}{\Longrightarrow} \textbf{fail}] | \, r_1''$$
$$t] | \, r_1 \stackrel{?but \cdot \theta \cdot ?but \cdot !choc \cdot \theta}{\Longrightarrow} \textbf{pass}] | \, r_1'''$$

where r_1', r_1'', and r_1''' are the leaves of r_1 from left to right.

Verdicts. The verdict assigned to a set of observations $O \subseteq OBS$ is **pass** if all traces in O lead to the terminal state **pass** of the test case:

$$\nu_t(O) \quad =_{\text{def}} \quad \begin{cases} \textbf{pass} & \text{if } \forall \sigma \in O : t \stackrel{\sigma}{\Longrightarrow} \textbf{pass} \\ \textbf{fail} & \text{otherwise} \end{cases}$$

Example 4. Continuing example 3 we have that, since the terminal state of t for the second run is **fail**, the verdict for r_1 is **fail**. Similarly, it can be checked that the verdict for r_2 is **pass**.

Test Execution. Test execution $\text{EXEC}(t, \text{IUT})$ should be correctly implemented, i.e., it should be implemented such that it correctly reflects the semantics as expressed by $obs(t, i_{\text{IUT}})$ and establishes the test hypothesis.

Test Derivation. The following algorithm specifies the derivation of test cases from a labelled transition system specification for the implementation relation **ioco**. The test cases are denoted using a process-algebraic notation: ";" denotes action prefix; "+" denotes choice; "Σ" denotes generalized choice. Moreover, for S a set of states, S **after** a denotes the set of states which can be reached from any state in S via action a.

Algorithm – **ioco** *Test Derivation:* Let s be a specification with initial state s_0. Let S be a non-empty set of states, with initially $S = \{s_0\}$. Then a test case t is obtained from S by a finite number of recursive applications of one of the following three nondeterministic choices:

1. $(*$ terminate the test case $*)$
   $t := \textbf{pass}$
2. $(*$ give a next input to the implementation $*)$
   $t := a\ ;\ t'$, if S **after** $a \neq \emptyset$
   where $a \in L_I$, and t' is obtained by recursively applying the algorithm for $S' = S$ **after** a .
3. $(*$ check the next output of the implementation $*)$
   $t := \quad \Sigma \{\ x\ ;\ \textbf{fail} \mid x \in L_U,\ x \notin out(S)\ \}$
   $\quad + \Sigma \{\ \theta\ ;\ \textbf{fail} \mid \delta \notin out(S)\ \}$
   $\quad + \Sigma \{\ x\ ;\ t_x \mid x \in L_U,\ x \in out(S)\ \}$
   $\quad + \Sigma \{\ \theta\ ;\ t_\theta \mid \delta \in out(S)\ \}$
   where t_x and t_θ are obtained by recursively applying the algorithm for S **after** x and S **after** δ, respectively.

Given a specification $s \in \mathcal{LTS}(L_I \cup L_U)$, this algorithm was proved in [45] to produce only sound test cases, i.e., test cases which never produce **fail** while testing an **ioco**-conforming implementation. Formally, let *der* be any function satisfying the (nondeterministic) algorithm, then the following holds

$$\forall i \in \mathcal{IOTS}(L_I, L_U) :\ i\ \textbf{ioco}\ s \implies \forall t \in der(s) : \nu_t(obs(t, i)) = \textbf{pass}$$

Moreover, it was shown in [45] that any non-conforming implementation can always be detected by a test case generated with this algorithm, i.e., let T_s be the set of all test cases which can be generated by the algorithm from s, then

$$\forall i \in \mathcal{IOTS}(L_I, L_U) : \; i \; \mathbf{ioco} \; s \iff \forall t \in T_s : \nu_t(obs(t, i)) = \mathbf{pass}$$

Example 5. Using the **ioco**-test derivation algorithm the test case t of figure 2 can be derived from specification r_2 in figure 1. This is consistent with figure 1 and example 4: $r_1 \; \mathbf{ioco} \not\; r_2$, $r_2 \; \mathbf{ioco} \; r_2$ (**ioco** is reflexive), and indeed $\nu_t(obs(t, r_1)) = \mathbf{fail}$, and $\nu_t(obs(t, r_2)) = \mathbf{pass}$. So, test case t can be used to detect that r_1 is not **ioco**-correct with respect to r_2.

5 Tools and an Application

The algorithm for **ioco**-test derivation has a wider applicability than candy machines. Different tools have been built which implement, more or less strictly, this algorithm. These include TVEDA [39,13], TGV [18] and TORX [4].

TVEDA is a tool which is able to generate test cases in TTCN [31, part 3] from single-process SDL specifications. Actually, it is interesting to note that the test generation algorithm of TVEDA was not based on the algorithm for **ioco**-test derivation but on the intuition and heuristics of experienced test case developers at France Telecom CNET. Only careful analysis afterwards showed that this algorithm generates test cases for an implementation relation which was called "R_1" in [39] and which is almost the same as **ioco**.

The tool TGV generates tests in TTCN from LOTOS or SDL specifications. It implements a test derivation algorithm for **ioco** with an unfair extension for finite-state divergences. Moreover, it allows test purposes to be specified by means of automata, which makes it possible to identify the parts of a specification which are interesting from a testing point of view.

Whereas TVEDA and TGV only support the test derivation process by deriving test suites and expressing them in TTCN, the tool TORX combines **ioco**-test derivation and test execution in an integrated manner. This approach, where test derivation and test execution occur simultaneously, is called *on-the-fly testing*. Instead of deriving a complete test case, the test derivation process only derives the next test event from the specification and this test event is immediately executed. While executing a test case, only the necessary part of the test case is considered: the test case is derived *lazily* (cf. lazy evaluation of functional languages). This can reduce the effort needed for deriving a test case, see also [48].

TORX is currently able to derive test cases from LOTOS and PROMELA specifications, but since its implementation uses the OPEN/CÆSAR interface [20] for traversing through a labelled transition system, the tool can be easily extended to any formalism with transition system semantics for which there is an OPEN/CÆSAR interface implementation available.

A simple experiment was conducted to show the viability and the practical applicability of the **ioco** testing theory and the tool TorX [4]. For this experiment a simple protocol, *the Conference Protocol*, was considered [19]. The Conference Protocol resembles a "chatbox". It offers to users the possibility to join a group, to chat with the members of the group, and to leave the group. It is implemented on top of the UDP protocol from the TCP/IP protocol suite.

Specifications in LOTOS and in Promela were developed for the Conference Protocol. An implementation in the C programming language was developed, too. From this implementation 27 (erroneous) mutants were derived. Moreover, for benchmarking, an SDL specification was developed from which 13 TTCN test cases were generated using the tool Autolink which is part of the SDL tool set Tau [43].

The 28 different implementations were tested with respect to the LOTOS and Promela specifications using TorX with the on-the-fly approach. All 25 **ioco**-incorrect mutants could be detected, based on the LOTOS as well as on the Promela specification. The length of the test run, i.e., the number of test events before the defect was detected, varied between 2 and 498 test events. Two mutants, although differing from the specification, were **ioco**-correct, and indeed no errors were found in these implementations. (These implementations differed in traces not explicitly contained in the specification, i.e., traces σ, with $\sigma \notin \mathit{Straces}(s)$, cf. the definition of **ioco** in section 4).

While testing the **ioco**-correct implementations based on the LOTOS specification, we were able to execute test runs consisting of 28,000 test events without finding a discrepancy between implementation and specification. Then the infamous message "out of memory" occurred while consuming 1.4 Gb. of memory. Since our Promela implementation in TorX inherits the state-space exploration algorithm from the very efficient model checker Spin [29], much longer test runs could be made with Promela: 450,000 test events using 400 Mb.

Using the 13 SDL-derived test cases, 5 erroneous mutants slipped through the testing procedure: they obtained a verdict **pass**. Although this experiment was certainly not significant enough for a fair comparison between the tool TorX and the commercial tool Autolink, we dare conclude that the **ioco**-based test theory as implemented in TorX constitutes a sound, feasible, and practically applicable approach for conformance testing based on formal methods.

6 Concluding Remarks

We have shown in this paper how formal methods can be used in conformance testing. It can be concluded that the use of formal methods in testing has many advantages. These advantages include

- a formal, thus more precise and less ambiguous specification of what should be tested;
- formal preciseness and clarity in the properties that are being tested;
- formal reasoning about the validity of tests; and

– algorithmic generation of test cases, with the potential of automated test case derivation.

The first advantage is already present in the testing process even if the testing process itself is not formal. Analysis of practical testing processes shows that most of the problems encountered are not due to the testing process itself but to unclear, imprecise and ambiguous specifications. Formalizing these specifications helps in reducing testing problems even without any formal testing. This is also one of the main conclusions of testing in the Bos-project where specifications were written in Z and Promela and testing was performed systematically based on these formal specifications, but using manual, conventional techniques without any formal derivation steps [22].

The third advantage addresses another practical testing problem, viz. that the occurrence of a **fail** verdict does not always point to an error in the implementation. In many cases, sometimes up to 50%, the error is due to an erroneous test case. Formal reasoning about conformance and about the validity of test cases may help to alleviate this problem.

The second advantage opens ways to combine verification and testing in a systematic and precise way. Some properties of a system may be verified while others are tested.

The fourth advantage has the largest economic implications. By automation the testing effort in software projects, which may currently take up to 40% of software development costs, may be reduced significantly. And this can be a good starting point for the introduction of formal methods in software development: most likely, more people will invest in using formal methods if test cases are for free once a formal system specification has been developed.

Formal verification does not make testing superfluous, nor does testing make formal verification superfluous. They are complementary techniques for analysis and checking of correctness of systems. While verification aims at proving properties about systems by formal manipulation on a mathematical model of the system, testing is performed by exercising the real, executing implementation (or an executable simulation model). Verification can give certainty about satisfaction of a required property, but this certainty only applies to the model of the system: any verification is only as good as the validity of the system model. Testing, being based on observing only a small subset of all possible instances of system behaviour, can never be complete: testing can only show the presence of errors, not their absence. But since testing can be applied to the real implementation, it is useful in those cases when a valid and reliable model of the system is difficult to build due to complexity, when the complete system is a combination of formal parts and parts which cannot be formally modelled (e.g., physical devices), when the model is proprietary (e.g., third party testing), or when the validity of a constructed model is to be checked with respect to the physical implementation. Moreover, testing based on a formal specification only makes sense if this specification can be assumed to be valid, i.e., has been sufficiently verified.

A crucial point both in formal verification and in formal testing is the link with the non-formal reality. In verification this occurs when a model of reality is built for which informal arguments are given that it is a valid modelling of reality. Subsequently, reasoning occurs completely in the formal domain under the assumption that the formal results will also apply to reality if the model is valid. In formal testing the link with reality is established using the test hypothesis. Here, a model is assumed to exist in a particular formal domain. It is not necessary that this model is available (then we could perform formal verification), nor that we will ever be able to develop it. Moreover, it is assumed that the way of doing experiments on the real system is modelled in a valid way by the formal function *obs*. This incorporates, among others, that test cases are assumed to be correctly implemented. Whether in formal testing or in verification, somewhere the link to the non-formal reality has to be made. It is important to be aware of the assumptions on which this is based, so that results are interpreted in the right context and with the appropriate precautions.

The formal testing framework of section 3 and its instantiation in section 4 provide a good basis for testing with formal methods. But they also point to some open problems. One of the most important ones is the problem of *test selection*. The algorithm for **ioco** test derivation, and many other similar algorithms, allow to derive infinitely many sound test cases. But which ones shall be selected and executed? Can test suites be compared with respect to their error detecting capabilities? Can measures be assigned to test suites expressing their quality? Can the quality of an implementation passing a particular test suite be quantified? To these questions there are not many usable answers, yet. Solutions can be sought by defining coverage measures, fault models, quantifying test hypotheses, etc. [6,32,39,9].

Acknowledgement

Numerous people, in particular the participants in the ISO/ITU-T standardization group on "Formal Methods in Conformance Testing", the partners in the *Côte de Resyste* research project, testing engineers at CMG The Hague B.V. and at CMG Finance B.V., and the members of the Formal Methods and Tools group at the University of Twente, contributed to the developments described in this paper by means of stimulating discussions or commenting on earlier papers, for which I am grateful. Joost Katoen, René de Vries, Axel Belinfante and Jan Feenstra are thanked for proof-reading.

References

1. S. Abramsky. Observational equivalence as a testing equivalence. *Theoretical Computer Science*, 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. B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

4. A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, 12^{th} *Int. Workshop on Testing of Communicating Systems*. Kluwer Academic Publishers, 1999.
5. J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.
6. G. Bernot. Testing against formal specifications: A theoretical view. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT'91, Volume 2*, pages 99–119. Lecture Notes in Computer Science 494, Springer-Verlag, 1991.
7. 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.
8. E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 63–74. North-Holland, 1988.
9. E. Brinksma. On the coverage of partial validations. In M. Nivat, C.M.I. Rattray, T. Rus, and G. Scollo, editors, *AMAST'93*, pages 247–254. BCS-FACS Workshops in Computing Series, Springer-Verlag, 1993.
10. E. Brinksma, R. Alderden, R. Langerak, J. van de Lagemaat, and J. Tretmans. A formal approach to conformance testing. In J. de Meer, L. Mackert, and W. Effelsberg, editors, *Second Int. Workshop on Protocol Test Systems*, pages 349–363. North-Holland, 1990.
11. E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In G. von Bochmann and B. Sarikaya, editors, *Protocol Specification, Testing, and Verification VI*, pages 349–360. North-Holland, 1987.
12. CCITT. *Specification and Description Language (SDL)*. Recommendation Z.100. ITU-T General Secretariat, Geneve, Switzerland, 1992.
13. M. Clatin. Manuel d'utilisation de TVEDA V3. Manual LAA/EIA/EVP/109, France Télécom CNET LAA/EIA/EVP, Lannion, France, 1996.
14. R. De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24:211–237, 1987.
15. R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.
16. K. Drira. The refusal graph: a tradeoff between verification and test. In O. Rafiq, editor, *Sixth Int. Workshop on Protocol Test Systems*, pages 297–312. North-Holland, 1994.
17. H. Eertink. The implementation of a test derivation algorithm. Memorandum INF-87-36, University of Twente, Enschede, The Netherlands, 1987.
18. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming – Special Issue on COST247, Verification and Validation Methods for Formal Descriptions*, 29(1–2):123–146, 1997.
19. L. Ferreira Pires. Protocol implementation: Manual for practical exercises 1995/1996. Lecture notes, University of Twente, Enschede, The Netherlands, August 1995.
20. H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In B. Steffen, editor, *Fourth Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, pages 68–84. Lecture Notes in Computer Science 1384, Springer-Verlag, 1998.

21. M.-C. Gaudel. Testing can be formal, too. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, pages 82–96. Lecture Notes in Computer Science 915, Springer-Verlag, 1995.
22. W. Geurts, K. Wijbrans, and J. Tretmans. Testing and formal methods — Bos project case study. In *EuroSTAR'98: 6^{th} European Int. Conference on Software Testing, Analysis & Review*, pages 215–229, Munich, Germany, November 30 – December 1 1998.
23. R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90*, Lecture Notes in Computer Science 458, pages 278–297. Springer-Verlag, 1990.
24. 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*, Lecture Notes in Computer Science 715, pages 66–81. Springer-Verlag, 1993.
25. L. Heerink. *Ins and Outs in Refusal Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1998.
26. L. Heerink and J. Tretmans. Formal methods in conformance testing: A probabilistic refinement. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *Ninth Int. Workshop on Testing of Communicating Systems*, pages 261–276. Chapman & Hall, 1996.
27. L. Heerink and J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In T. Mizuno, N. Shiratori, T. Higashino, and A. Togashi, editors, *Formal Desciption Techniques and Protocol Specification, Testing and Verification FORTE X /PSTV XVII '97*, pages 23–38. Chapman & Hall, 1997.
28. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
29. G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall Inc., 1991.
30. ISO. *Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* International Standard IS-8807. ISO, Geneve, 1989.
31. ISO. *Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework.* International Standard IS-9646. ISO, Geneve, 1991. Also: CCITT X.290–X.294.
32. ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing.* Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, Geneve, 1996.
33. R. Langerak. A testing theory for LOTOS using deadlock detection. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification IX*, pages 87–98. North-Holland, 1990.
34. G. Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25(1):23–41, 1992.
35. D. Lee and M. Yannakakis. Principles and methods for testing finite state machines. *The Proceedings of the IEEE*, August 1996.
36. N.A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.
37. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
38. G.J. Myers. *The Art of Software Testing.* John Wiley & Sons Inc, 1979.
39. 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.
40. I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(2):241–284, 1987.

41. D. H. Pitt and D. Freestone. The derivation of conformance tests from LOTOS specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, 1990.
42. R. Segala. Quiescence, fairness, testing, and the notion of implementation. In E. Best, editor, *CONCUR'93*, pages 324–338. Lecture Notes in Computer Science 715, Springer-Verlag, 1993.
43. Telelogic. TAU *SDL Tool Set Documentation*. Telelogic AB, Malmö, Sweden, 1998.
44. J. Tretmans. A formal approach to conformance testing. In O. Rafiq, editor, *Sixth Int. Workshop on Protocol Test Systems*, number C-19 in IFIP Transactions, pages 257–276. North-Holland, 1994.
45. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
46. 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.
47. L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In G. von Bochmann, R. Dssouli, and A. Das, editors, *Fifth Int. Workshop on Protocol Test Systems*, IFIP Transactions. North-Holland, 1993.
48. R.G. de Vries and J. Tretmans. On-the-Fly Conformance Testing using SPIN. In G. Holzmann, E. Najm, and A. Serhrouchni, editors, *Fourth Workshop on Automata Theoretic Verification with the* SPIN *Model Checker*, ENST 98 S 002, pages 115–128, Paris, France, November 2 1998. Ecole Nationale Supérieure des Télécommunications. Also to appear in Software Tools for Technology Transfer.
49. C. D. Wezeman. The CO-OP method for compositional derivation of conformance testers. In E. Brinksma, G. Scollo, and C. A. Vissers, editors, *Protocol Specification, Testing, and Verification IX*, pages 145–158. North-Holland, 1990.