Factorized test generation for multi-input/output transition systems

Ed Brinksma, Lex Heerink and Jan Tretmans
University of Twente, NL-7500 AE Enschede
{brinksma,heerink,tretmans}@cs.utwente.nl

Abstract

In this paper we present factorized test generation techniques that can be used to generate test cases from a specification that is modelled as a labelled transition system. The test generation techniques are able to construct a sound (and complete) test suite for correctness criterion $\mathbf{mioco}_{\mathcal{F}}$ [5] by splitting up this correctness criterion into many simpler correctness criteria, and by generating tests for these simpler correctness criteria. By isolating the relevant part of the specification that is needed to generate tests for each of these simpler correctness criteria and using this part to generate tests from, test generation can be done more efficiently. These techniques are intended to keep the generation of tests from a specification feasible and manageable.

1 INTRODUCTION

Testing To assess the correctness of systems testing is a frequently applied technique. The aim of testing is to check whether an implementation is correct with respect to its specification. This is done by conducting experiments on the implementation, observing the responses of the implementation to these experiments, and comparing these responses with the ones that could be expected on the basis of the specification. An implementation is considered incorrect, or erroneous, in case the responses to an experiment are different from the ones that could be expected. Testing can only show the presence of errors in implementations, never their absence. However, it is commonly agreed that confidence in the correct operation of an implementation increases if more and more tests are conducted and no errors are found. To reason about testing in a formal framework a clear definition of the universe of experiments ($\mathcal{U}$), observations that occur when experiment $u \in \mathcal{U}$ is carried out on system p ($obs(u, p)$), and a comparison criterion ($\sqsubseteq$) for these observations must be defined. This results in an extensional definition of a correctness criterion **conforms-to** (implementation relation) between implementation i and specification s as follows

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

In this paper we assume that specifications and implementations can be modelled by (subclasses of) labelled transition systems. In [3, 12, 14] and others different instantiations of the relation **conforms-to** have been defined by varying $\mathcal{U}$, obs, and the comparison criterion $\sqsubseteq$.

Test generation Instead of defining implementation relations by varying $\mathcal{U}$, obs, and $\sqsubseteq$ (1) the problem in test generation is to obtain the set of experiments that are needed to distinguish between correct and incorrect implementations for a given implementation relation and given specification. Preferably, such experiments are calculated automatically from the specification and the implementation relation. Unfortunately, calculating such experiments is often too complex in space and time to be feasible.

Contribution of this paper In this paper we discuss a factorized test generation technique that can be used to generate tests from a transition system specification with respect to the correctness criterion *multi input/output conformance* $\mathbf{mioco}_{\mathcal{F}}$, which was introduced in [5]. The technique is factorized with respect to the implementation relation $\mathbf{mioco}_{\mathcal{F}}$ in the sense that the "complicated" correctness criterion $\mathbf{mioco}_{\mathcal{F}}$ can be split up in several independent "easier-to-check" correctness criteria. Moreover, for the generation of tests for each of these simpler criteria we use a specification that is obtained by "projecting" the original specification. Such a projected specification is usually smaller in size than the original specification, and thus easier to handle by tools. In this way test generation from large-sized specifications for complicated correctness criteria becomes feasible. Some existing test tools, such as TGV [4] and AUTOLINK [13], have implemented test generation techniques that are similar to the ones of which the underlying mathematical principles are explained in this paper.

Overview Section 2 recalls [5], defining the subclass of multi-input/output transition systems. Next, section 3 presents the correctness criterion $\mathbf{mioco}_{\mathcal{F}}$, and describes an algorithm that is able to generate tests from a specification with respect to $\mathbf{mioco}_{\mathcal{F}}$. Section 4 investigates under which conditions specifications can be safely reduced in size without loosing the ability to generate valid tests from it. Section 5 describes two factorized test generation techniques: the technique described in section 5.1 is able to produce a sound test suite, and the one in section 5.2 produces a complete test suite. In section 5.3 implementation techniques are discussed which make the factorized way of generating tests more efficient. Finally, section 6 contains conclusions and further research.

2 MULTI-INPUT/OUTPUT TRANSITION SYSTEMS

Many behaviour description languages use the formalism of labelled transition systems as their underlying semantic model (e.g., CCS [10], Lotos [6]). In this paper we use rigid transition systems, i.e., without internal actions, to specify and model the behaviour of systems.

Definition 1 *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, $\rightarrow \subseteq S \times L \times S$ is a set of transitions, and $s_0 \in S$ is the initial state.*

The universe of LTSs over L is denoted by $\mathcal{LTS}(L)$. Instead of $(s, a, s') \in \rightarrow$ we write $s \xrightarrow{a} s'$. We extend the relation $\longrightarrow$ with labels that are sets of actions: $s \xrightarrow{A} s$ means that s cannot perform actions $a \in A$, i.e., $s \xrightarrow{A} s =_{def} \forall \mu \in A : s \overset{\mu}{\not\longrightarrow}$. Such self-loop transitions are called refusal transitions [12]. A failure trace is a finite sequence $\mu_1 \cdot \ldots \cdot \mu_n$ of actions and refusals. We write $p \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} p_n$ for $\exists p_1, \ldots p_{n-1} : p \xrightarrow{\mu_1} p_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} p_n$, and $p \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n}$ for $\exists p_n : p \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} p_n$. The set of finite sequences over actions in L is denoted by L^*, and the set of derivates of p is defined as $der(p) =_{def} \{p' \mid \exists \sigma \in L^* : p \xrightarrow{\sigma} p'\}$. The set of failure traces of p over L is defined as $f\text{-}traces(p) =_{def} \{\sigma \in (L \cup \mathcal{P}(L))^* \mid p \xrightarrow{\sigma}\}$ where $\mathcal{P}(\cdot)$ denotes the powerset operator. For the notation of transition systems we use some standard process-algebraic operators (cf. Lotos [6]). For this paper it suffices to use action-prefix $\mu; B$ which can perform action μ and then behave as B, and unguarded choice $\sum \mathcal{B}$ which can behave as any of its members $B \in \mathcal{B}$. We abbreviate $\sum\{B_1, B_2\}$ by $B_1 + B_2$ and $\sum \emptyset$ by **stop**.

In traditional testing theory [1, 3] an LTS abstracts from the initiative of actions. In reality, however, many implementations communicate with their environment via clearly distinguishable input actions (actions that are initiated by the environment and consumed by the implementation, e.g., button push experiments) and output actions (actions that are initiated by the implementation and consumed by the environment, e.g., messages that occur on a display). In testing the distinction between input actions and output actions is crucial to model realistic implementations faithfully. This has triggered research in transition system models where the labelset L is partitioned in a set of input actions L_I and a set of output actions L_U, e.g., input/output automata (IOA, [9]), input/output state machines (IOSM, [11]), input/output labelled transition systems (IOLTS, [4]) and input/output transition systems (IOTS, [14]). These models additionally require that input actions are continuously enabled [2].

A further refinement with respect to distinguishing inputs and outputs was proposed in [5]. There not only a distinction between input actions and output actions is made, but also the interface of the implementation with its environment ("PCO") is explicitly modelled. This is done by partitioning the set of input actions L_I in a set of channels $\mathcal{L}_I =_{def} \{L_I^1, \ldots, L_I^n\}$, and the set of output actions L_U in a set of channels $\mathcal{L}_U =_{def} \{L_U^1, \ldots, L_U^m\}$. Each channel L_I^j or L_U^k represents a location on the interface of the implementation where the actions in L_I^j or L_U^k may occur, respectively. Moreover, to enlarge the diversity of systems that can be modelled (compared to IOA, IOSM, IOLTS and IOTS) a more liberal condition with respect to the enabling of inputs is imposed: inputs need not always be enabled, but for each input channel input actions must be simultaneously enabled. Such systems are called *multi-input/output transition systems* (MIOTS).

Definition 2 *A multi-input/output transition system (MIOTS) p over partitioning $\mathcal{L}_I$ of L_I and partitioning $\mathcal{L}_U$ of L_U is a transition system with inputs and outputs, $p \in \mathcal{LTS}(L_I \cup L_U)$, such that for all $L_I^j \in \mathcal{L}_I$*

$$\forall p' \in der(p),\ \ if\ \ \exists a \in L_I^j : p' \xrightarrow{a} \quad then\ \ \forall b \in L_I^j : p' \xrightarrow{b}$$

The universe of multi-input/output transition systems over $\mathcal{L}_I$ and $\mathcal{L}_U$ is denoted by $\mathcal{MIOTS}(\mathcal{L}_I, \mathcal{L}_U)$.

The formalisms LTS and IOTS are special classes of $\mathcal{MIOTS}(\mathcal{L}_I, \mathcal{L}_U)$ for specific instantiations of the sets $\mathcal{L}_I$ and $\mathcal{L}_U$ [5]. Because many realistic implementations can be modelled as MIOTS we will use MIOTS as the modelling formalism of implementations. The choice to model implementations as MIOTS still leaves freedom to choose the interface of these implementations with their environment by instantiating these MIOTS with the proper partitionings $\mathcal{L}_I$ and $\mathcal{L}_U$. Figure 1 depicts the interface of a multi-input/output transition system.

In [5] an extensional correctness criterion $\leq_{mior}$ has been defined that indicates when an implementation is correct with respect to its specification. This relation can be defined in the same style as equation (1), where the set of observers is taken as the set of *singular observers*. A singular observer acts as a special multi-input/output transition system which supplies inputs and observes outputs, where inputs for the implementation are outputs for the observer and vice versa. Moreover, these observers are equipped with labels to detect *input suspension*, i.e., the non-acceptance of an input action by the implementation (e.g., a button that is pressed but that does not go down), and with labels to detect *output suspension*, i.e., the inability of the implementation to produce an output action (e.g., a display that remains empty). By having the ability to detect input suspension a larger class of implementations can be tested than in, e.g., the testing theory of IOTS [14]. In the extensional

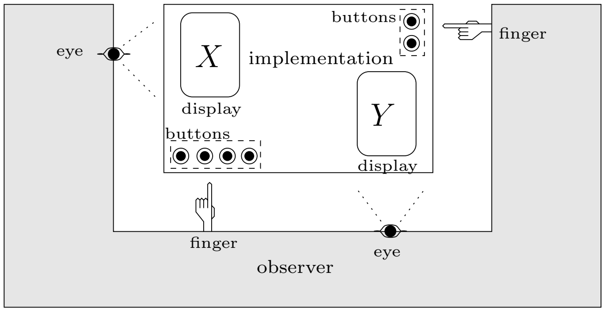


Figure 1 An interface for a multi-input/output transition system

definition of $\leq_{mior}$ an implementation i is related to its specification s if, for every singular observer u, all observations $obs(u,i)$ that u can make of i are included in the set of observations $obs(u,s)$ that u can make of s. We will not present the extensional definition of $\leq_{mior}$ here, but instead we will use an intensional characterization of $\leq_{mior}$ as its definition. For more details we refer to [5].

Definition 3 *$\leq_{mior} \subseteq \mathcal{MIOTS}(\mathcal{L}_I, \mathcal{L}_U) \times \mathcal{LTS}(L_I \cup L_U)$ is defined by*

$$\begin{array}{l} i \;\; \leq_{mior} \;\; s \\ =_{def} \forall \sigma \in (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^* : \;\; out(\, i \,\textbf{after}\, \sigma \,) \subseteq out(\, s \,\textbf{after}\, \sigma \,) \end{array}$$

where

$$\begin{array}{lll} out(\, p \,\textbf{after}\, \sigma \,) =_{def} & \{x \in L_U \mid \exists p' : p \xrightarrow{\sigma} p' \xrightarrow{x} \} & (i) \\ & \cup \{L_I^j \mid 1 \leq j \leq n, \exists p' : p \xrightarrow{\sigma} p' \text{ and } p' \xrightarrow{L_I^j} \} & (ii) \\ & \cup \{L_U^k \mid 1 \leq k \leq m, \exists p' : p \xrightarrow{\sigma} p' \text{ and } p' \xrightarrow{L_U^K} \} & (iii) \end{array}$$

The relation $\leq_{mior}$ states that an implementation is $\leq_{mior}$-incorrect if (i) the implementation produces an output, which cannot be produced by the specification after the same trace, or (ii) the implementation has an input suspension at some input channel L_I^j where the specification has none, or (iii) the implementation has an output suspension at some output channel L_U^k where the specification has none.

3 TEST GENERATION FOR MIOTS

Checking for $\leq_{mior}$ requires checking $out(\, i \,\textbf{after}\, \sigma \,) \subseteq out(\, s \,\textbf{after}\, \sigma \,)$ for all $\sigma \in (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$. Since this is too time consuming in practice, the relation $\textbf{mioco}_{\mathcal{F}}$ restricts $\leq_{mior}$ by checking this condition for all failure traces in $\mathcal{F}$ for particularly chosen $\mathcal{F}$ (cf. the conformance relation **conf** in [1]).

Definition 4 *The relation* $\mathbf{mioco}_{\mathcal{F}} \subseteq \mathcal{MIOTS}(\mathcal{L}_I, \mathcal{L}_U) \times \mathcal{LTS}(L_I \cup L_U)$, *where* $\mathcal{F} \subseteq (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$, *is defined by*

$$i \ \mathbf{mioco}_{\mathcal{F}} \ s \ =_{def} \ \forall \sigma \in \mathcal{F} : \ out(\, i \ \mathbf{after} \ \sigma \,) \subseteq out(\, s \ \mathbf{after} \ \sigma \,)$$

Proposition 1 *Let* $\mathcal{F}_1, \mathcal{F}_2 \subseteq (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$

1. $\mathbf{mioco}_{\mathcal{F}_1 \cup \mathcal{F}_2} = \mathbf{mioco}_{\mathcal{F}_1} \cap \mathbf{mioco}_{\mathcal{F}_2}$
2. $\mathcal{F}_1 \subseteq \mathcal{F}_2$ *implies* $\mathbf{mioco}_{\mathcal{F}_1} \supseteq \mathbf{mioco}_{\mathcal{F}_2}$

We use the parameterized relation $\mathbf{mioco}_{\mathcal{F}}$ as the class of correctness criteria for well-chosen $\mathcal{F}$. This is motivated by the fact that for specific instances of $\mathcal{L}_I$, $\mathcal{L}_U$ and $\mathcal{F}$ this relation coincides with well-known implementation relations such as **ioco** and **ioconf** advocated in [14]. The set $\mathcal{F}$ can be considered as a set of test purposes for which tests must be derived [7]. The selection of such traces could be based on testing heuristics or experience. For complex and critical applications, such as communication protocols, the set $\mathcal{F}$ can be very large.

In [5] a test generation algorithm Π has been presented that is able to generate tests from a specification $s \in \mathcal{LTS}(L_I \cup L_U)$. These tests can decide whether an implementation is $\mathbf{mioco}_{\mathcal{F}}$-correct with respect to its specification or not. Tests are modelled as singular observers and use special labels θ_i^j to detect input suspension at channel L_I^j, and the special labels θ_u^k to detect output suspension at channel L_U^k. Tests are built up recursively by either applying an input action a to channel L_I^j and detecting its acceptance or suspension ($t \ ::= \ a; t + \theta_i^j; t$), or observing an output channel L_U^k and detecting the occurrence of an output or output suspension ($t ::= \sum_{x \in L_U^k \cup \{\theta_u^k\}} x; t$), or ending the test [5]. The end states of a test are labelled with **pass** or **fail** and indicate success or failure of test execution.

Figure 2 depicts test algorithm Π from [5]. The algorithm takes a specification $s \in \mathcal{LTS}(L_I \cup L_U)$ and a set of failure traces $\mathcal{F}$ and computes a test case $\Pi_{\mathcal{F},S}$ by applying one of the steps in the algorithm. The set S keeps track of the possible current states of the specification, and initially contains the initial state of specification s. The set $\mathcal{F}$ contains the failure traces for which the condition $out(\, i \ \mathbf{after} \ \sigma \,) \subseteq out(\, s \ \mathbf{after} \ \sigma \,)$ has to be tested. Both sets S and $\mathcal{F}$ are updated in case the test generation algorithm proceeds recursively. We define $S \ \mathbf{after} \ \sigma \ =_{def} \{s' \mid \exists s \in S : s \xrightarrow{\sigma} s'\}$ as the set of states that are reachable from a state in S after σ. The trace $\overline{\sigma}$ denotes the trace σ where all occurrences of refusals L_I^j and L_U^k are replaced by there suspension detection labels θ_i^j and θ_u^k, and vice versa.

Running a test against an implementation means that the experiments prescribed in the test (supplying an input to an input channel, or observing an output from an output channel) are applied to the implementation. In case an input action is supplied, then it is either accepted or rejected, after which the

```
Input:   set of states $S$
Input:   set of failure traces $\mathcal{F} \subseteq (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$
Output:  test case $\Pi_{\mathcal{F},S}$

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

Apply one of the following non-deterministic choices recursively.
1. (* terminate the test case if there are no more specified traces in $\mathcal{F}$ *)
   if $\mathcal{F} = \emptyset$ then
         $\Pi_{\mathcal{F},S} := \textbf{pass}$
2. (* terminate the test case when a trace $\sigma \in \mathcal{F}$ has been performed *)
   if $\epsilon \in \mathcal{F}$ then take some $L_I^j \in \mathcal{L}_I$, and for some $a \in L_I^j$ (* supply input $a$ *)
         $\Pi_{\mathcal{F},S} := \begin{cases} a; \textbf{pass} + \theta_i^j; \textbf{fail} & \text{if } S \textbf{ after } L_I^j = \emptyset \\ a; \textbf{pass} + \theta_i^j; \textbf{pass} & \text{if } S \textbf{ after } L_I^j \neq \emptyset \end{cases}$
3. (* terminate the test case when a trace $\sigma \in \mathcal{F}$ has been performed *)
   if $\epsilon \in \mathcal{F}$ then take some $L_U^k \in \mathcal{L}_U$, then (* observe channel $L_U^k$ *)
         $\Pi_{\mathcal{F},S} := \sum\{x; \textbf{pass} \mid x \in L_U^k \cup \{\theta_u^k\} \text{ and } S \textbf{ after } \overline{x} \neq \emptyset\}$
                $+ \sum\{x; \textbf{fail} \mid x \in L_U^k \cup \{\theta_u^k\} \text{ and } S \textbf{ after } \overline{x} = \emptyset\}$
4. (* supply an input for which you want to test deeper *)
   take some $L_I^j \in \mathcal{L}_I$ and $a \in L_I^j$ such that $\{\sigma \mid a \cdot \sigma \in \mathcal{F}\} \neq \emptyset$, then
         $\Pi_{\mathcal{F},S} := a; \Pi_{\mathcal{F}',S'} + \theta_i^j; \textbf{pass}$
   where $S' = S \textbf{ after } a, \mathcal{F}' = \{\sigma \mid a \cdot \sigma \in \mathcal{F}\}$
5. (* supply some input and continue if it is refused *)
   take some $L_I^j \in \mathcal{L}_I$ such that $\{\sigma \mid L_I^j \cdot \sigma \in \mathcal{F}\} \neq \emptyset$, then
         $\Pi_{\mathcal{F},S} := a; \textbf{pass} + \theta_i^j; \Pi_{\mathcal{F}'',S''}$
   where $a \in L_I^j, S'' = S \textbf{ after } L_I^j, \mathcal{F}'' = \{\sigma \mid L_I^j \cdot \sigma \in \mathcal{F}\}$
6. (* Find a channel $L_U^k$ that produces an output for which to test deeper *)
   take some $L_U^k \in \mathcal{L}_U$ such that $\{\sigma \mid \exists x \in L_U^k \cup \{L_U^k\} : x \cdot \sigma \in \mathcal{F}\} \neq \emptyset$, then
         $\Pi_{\mathcal{F},S} := \sum\{x; \Pi_{\mathcal{F}',S'} \mid x \in L_U^k \cup \{\theta_u^k\}$   and $\mathcal{F}' = \{\sigma \mid \overline{x} \cdot \sigma \in \mathcal{F}\}$
                                                     and $S' = S \textbf{ after } \overline{x}\}$
```

Figure 2 Test generation algorithm.

test continues accordingly. Similarly, for any output channel either an output action is produced by the implementation and observed by the test, or output at this channel is suspended, and the test continues with its corresponding successor experiment. Consequently, when running a test against an implementation the test will always end up in one of its end states (i.e., a **pass** or a **fail** state).

We say that implementation i passes test t if t can only end up in a **pass** state after running against i. This is denoted by the predicate i *passes* t. The dual is denoted by i *fails* t, meaning that t *may* end up in a **fail** state after running against i. For test suite T we define i *passes* $T =_{def} \forall t \in T : i$ *passes* t, and i *fails* $T =_{def} \neg(i$ *passes* $T)$.

To assess the correctness of implementations by means of testing we have to link the passing and failing of these tests when run against implementations to the correctness of these implementations. For that we use the terms soundness, exhaustiveness and completeness of test suites [8]. A test suite T is *sound* (for specification s with respect to $\mathbf{mioco}_{\mathcal{F}}$) if every correct implementation will always pass this test suite: $i\ \mathbf{mioco}_{\mathcal{F}}\ s \implies i$ *passes* T. Test suite T is *exhaustive* if passing test suite T guarantees correctness: $i\ \mathbf{mioco}_{\mathcal{F}}\ s \impliedby i$ *passes* T. Test suite T is *complete* if it is both sound and exhaustive. A sound test suite is never able to reject correct implementations, and an exhaustive test suite is theoretically able to fail with all incorrect ones (which, in practice, may take an infinite amount of time).

It has been shown [5] that every test that can be generated by algorithm Π for $\mathcal{F}$ and s is sound for s with respect to $\mathbf{mioco}_{\mathcal{F}}$. Moreover, the set of all tests that can be generated by algorithm Π for $\mathcal{F}$ and s, denoted by $\Pi_{\mathcal{F}}(s)$, is complete for s with respect to $\mathbf{mioco}_{\mathcal{F}}$.

Proposition 2 *Let* $\mathcal{F} \subseteq (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$ *and* $s \in \mathcal{LTS}(L_I \cup L_U)$

1. *Any test case obtained from algorithm* Π *for* s *and* $\mathcal{F}$ *is sound for* s *with respect to* $\mathbf{mioco}_{\mathcal{F}}$.
2. *The set of all test cases that can be obtained from algorithm* Π *for* s *and* $\mathcal{F}$ *is complete for* s *with respect to* $\mathbf{mioco}_{\mathcal{F}}$.

The algorithm may generate tests that are not very efficient in detecting incorrect implementations, however, optimizations are not considered here.

4 LOOSER SPECIFICATIONS

We discuss a technique that can be used to isolate a part of the specification, and use this part to generate tests. This technique is called *loosening of specifications* [8]. We analyse the conditions under which it is valid to isolate such a part of the specification. Sections 5.1 and 5.2 will show how to generate a sound and complete test suite from these parts for implementation relation $\mathbf{mioco}_{\mathcal{F}}$ in a factorized way.

For correctness criterion $\mathbf{mioco}_{\mathcal{F}}$ only the behaviour after failure traces specified in $\mathcal{F}$ has to be investigated (definition 4). For analysing whether the responses to failure traces in $\mathcal{F}$ are valid or not there is no need to investigate the complete specification; responses to experiments that are not specified in $\mathcal{F}$ can be discarded. This argument shows that it may be possible to generate tests from a smaller specification (in size). The question is how to obtain such a smaller specification.

Such a smaller specification can be obtained by having the tester provide the input actions of the failure traces in $\mathcal{F}$ for which correctness has to be checked. Because a tester fully controls the input actions of an implementation but not the output actions, such a tester can "steer" the implementation towards checking a specific failure trace in $\mathcal{F}$ as much as possible by providing the input actions that are necessary to perform this failure trace. Deterministic processes that specify such sequences of input actions are called *selection processes*. Such processes can be seen as test purposes. From a selection process q and specification s a specification $s \,\|_{L_I} q$ is isolated that contains the responses to the input sequences specified in q, but discards all responses to input sequences that are not specified in q. The operator $\|_{L_I}$ describes how the part $s \,\|_{L_I} q$ is isolated from s, and its formal definition is given below (cf. Lotos [6])

Definition 5 *The universe of selection processes $\mathcal{SLTS}(L_I)$ over L_I is*

$$\mathcal{SLTS}(L_I) \quad =_{def} \quad \{p \in \mathcal{LTS}(L_I) \mid p \textit{ is deterministic}\}$$

Let $s \in \mathcal{LTS}(L_I \cup L_U)$ and $q \in \mathcal{SLTS}(L_I)$ then the transition system $s \;\|_{L_I}\; q \in \mathcal{LTS}(L_I \cup L_U)$ is inductively defined by the following inference rules.

$$\frac{s \xrightarrow{a} s',\, q \xrightarrow{a} q'}{s \;\|_{L_I}\; q \xrightarrow{a} s' \;\|_{L_I}\; q'} \;(a \in L_I) \qquad\qquad \frac{s \xrightarrow{x} s'}{s \;\|_{L_I}\; q \xrightarrow{x} s' \;\|_{L_I}\; q} \;(x \in L_U)$$

The operator $\|_{L_I}$ forces synchronization on actions in L_I, but allows actions not in L_I (i.e., actions in L_U) to occur independently.

We focus on what conditions need to be imposed on q in order to use $s \,\|_{L_I} q$ instead of s as the specification to generate tests from without running the risk to generate tests that are able to reject implementations that are $\mathbf{mioco}_{\mathcal{F}}$-correct for s, that is, what conditions need to be imposed on q in order to generate test suites from $s \,\|_{L_I} q$ that are sound with respect to $\mathbf{mioco}_{\mathcal{F}}$ for s.

As a first step in analysing these conditions we compare the input refusals and output refusals of s with the ones of $s \,\|_{L_I} q$. Because *all* and *only* output actions that $s \,\|_{L_I} q$ can perform are the ones that s is able to perform, any refusal $X \subseteq L_U$ of s is also a refusal of $s \,\|_{L_I} q$ and vice versa. For refusals of input actions $A \subseteq L_I$ the situation is slightly different. Because s and q need to synchronize on input actions it follows that the inability of s to perform an input action is reflected by the inability of $s \,\|_{L_I} q$ to perform this action, but not vice versa!

Proposition 3 *Let $A \subseteq L_I$ and $X \subseteq L_U$.*

1. $s \xrightarrow{A} s \quad \textit{implies} \quad s \;\|_{L_I}\; q \xrightarrow{A} s \;\|_{L_I}\; q$

2. $s \xrightarrow{X} s \quad \textit{iff} \quad s \;\|_{L_I}\; q \xrightarrow{X} s \;\|_{L_I}\; q$

Proposition 3 states that $s \,\|_{L_I} q$ preserves the refusals $A \subseteq L_I$ and $X \subseteq L_U$ of s. This result can be used to show that $s \,\|_{L_I} q$ preserves all failure traces of s in $(L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$ for which q specifies the sequences of input actions to be performed. We use $\sigma\lceil L_I$ to denote the sequence that arises from σ when restricted to actions in L_I.

Proposition 4 *For all* $\sigma \in (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$

$$s \xrightarrow{\sigma} s' \text{ and } q \xrightarrow{\sigma\lceil L_I} q' \quad \text{implies} \quad s \,\|_{L_I} q \xrightarrow{\sigma} s' \,\|_{L_I} q'$$

Combining the facts that all output actions of $s' \,\|_{L_I} q'$ are produced by s' (definition 5) and that all input refusals and output refusals of s' are preserved by $s' \,\|_{L_I} q'$ (proposition 3), it follows, together with proposition 4, that $out(\, s \textbf{ after } \sigma\,)$ is included in $out(\, s \,\|_{L_I} q \textbf{ after } \sigma\,)$ for those σ such that $\sigma\lceil L_I$ is specified by q.

Proposition 5 *Let* $\sigma \in (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$

$$q \xrightarrow{\sigma\lceil L_I} \quad \text{implies} \quad out(\, s \textbf{ after } \sigma\,) \subseteq out(\, s \,\|_{L_I} q \textbf{ after } \sigma\,)$$

By choosing suitable q it is possible to "steer" for which failure traces the inclusion $out(\, s \textbf{ after } \sigma\,) \subseteq out(\, s \,\|_{L_I} q \textbf{ after } \sigma\,)$ holds. In particular, if q contains the input sequences of the failures traces specified in $\mathcal{F}$, then this inclusion holds (at least) for all $\sigma \in \mathcal{F}$. But then, according to definition 4, any implementation that is $\mathbf{mioco}_{\mathcal{F}}$-correct for s is also $\mathbf{mioco}_{\mathcal{F}}$-correct for $s \,\|_{L_I} q$, or alternatively, any $\mathbf{mioco}_{\mathcal{F}}$-incorrect implementation for $s \,\|_{L_I} q$ is also $\mathbf{mioco}_{\mathcal{F}}$-incorrect for s. We define $\mathcal{F}\lceil L_I$ as the set-wise restriction on failure traces in $\mathcal{F}$: $\mathcal{F}\lceil L_I =_{def} \{\sigma\lceil L_I \mid \sigma \in \mathcal{F}\}$.

Proposition 6 *If* $traces(q) \supseteq \mathcal{F}\lceil L_I$, *then*

$$i \;\mathbf{mioco}_{\mathcal{F}}\; s \quad \text{implies} \quad i \;\mathbf{mioco}_{\mathcal{F}}\; (s \,\|_{L_I}\, q)$$

The significance of proposition 6 is that in order to obtain a sound test suite that can check whether an implementation is $\mathbf{mioco}_{\mathcal{F}}$-correct for "big" specification s, it is possible to generate a sound test suite from the "smaller" specification $s \,\|_{L_I} q$ as long as q specifies all input sequences of the failures traces in $\mathcal{F}$. If $traces(q) \supseteq \mathcal{F}$ we call the specification $s \,\|_{L_I} q$ the *projected* specification of s on $\mathbf{mioco}_{\mathcal{F}}$. The reverse implication of proposition 6 does not hold in general: erroneous implementations with respect to $\mathbf{mioco}_{\mathcal{F}}$ and s may pass a sound test suite that is generated from $s \,\|_{L_I} q$. This is caused by the fact that, due to the pruning of s using q, additional input refusals may be introduced in $s \,\|_{L_I} q$ that were not present in s itself (cf. proposition 3.1).

5 FACTORIZED TEST GENERATION

In this section we describe two techniques to generate tests for $\mathbf{mioco}_{\mathcal{F}}$ in a factorized way. We do this by generating tests from a specification that is projected on the correctness criterion $\mathbf{mioco}_{\{\sigma\}}$ for all $\sigma \in \mathcal{F}$. Section 5.1 discusses how a sound test suite can be obtained in this way, and section 5.2 shows how to obtain a complete test suite.

5.1 Factorized test generation (soundness)

In practice, when generating tests for $\mathbf{mioco}_{\mathcal{F}}$, the set $\mathcal{F}$ may contain a large number of failure traces, and the specification s can be very large (e.g., measured in number of states and transitions). Consequently, the generation of tests $\Pi_{\mathcal{F}}(s)$ directly from $\mathcal{F}$ and s can be a time and space consuming task, and tools may not be able to generate this set. In this subsection we present a technique to cope (at least partially) with this complexity.

In order to reduce the size of the specification s selection processes can be used (see section 4). A special class of selection processes is the class consisting of linear sequences over the set of input actions L_I. Such selection processes are called *sticks.*

Definition 6 *Let $\sigma, \sigma' \in (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$, then $stick(\sigma)$ is the transition system that is inductively defined by*

$$\begin{array}{lll} stick(\epsilon) & =_{def} & \mathbf{stop} \\ stick(a\cdot\sigma') & =_{def} & \left\{ \begin{array}{ll} a; stick(\sigma') & \mathit{if}\ a \in L_I \\ stick(\sigma') & \mathit{otherwise} \end{array} \right. \end{array}$$

The universe of all sticks is denoted by $\mathcal{STICK}(L_I)$.

Note that $\mathcal{STICK}(L_I) \subset \mathcal{SLTS}(L_I)$. Figure 4(a) depicts the structure of a stick.

From the generalized version of proposition 1.1 it follows that checking for correctness with respect to $\mathbf{mioco}_{\mathcal{F}}$ can be expressed in terms of checking for $\mathbf{mioco}_{\{\sigma\}}$ for each $\sigma \in \mathcal{F}$, viz.

$$\mathbf{mioco}_{\mathcal{F}} = \bigcap_{\sigma \in \mathcal{F}} \mathbf{mioco}_{\{\sigma\}} \tag{2}$$

Each correctness check with respect to $\mathbf{mioco}_{\{\sigma\}}$ can be performed independently. For this correctness criterion it suffices to take a selection process that is able to perform the sequence $\sigma \lceil L_I$ according to proposition 5. The process $stick(\sigma)$ is such a selection process. Thus, according to proposition 6 we have

$$i\ \mathbf{mioco}_{\{\sigma\}}\ s \quad \text{implies} \quad i\ \mathbf{mioco}_{\{\sigma\}}\ (s \,\|_{L_I} stick(\sigma)) \tag{3}$$

Combining the results of equations (2) and (3) shows that instead of generating tests from s for $\mathbf{mioco}_{\mathcal{F}}$ we can generate tests from $s \parallel_{L_I} \mathit{stick}(\sigma)$ for $\mathbf{mioco}_{\{\sigma\}}$ without running the risk that $\mathbf{mioco}_{\mathcal{F}}$-correct implementations are rejected. For all $\sigma \in \mathcal{F}$ this can be done independently. This leads to the parallelization procedure sketched in figure 3.

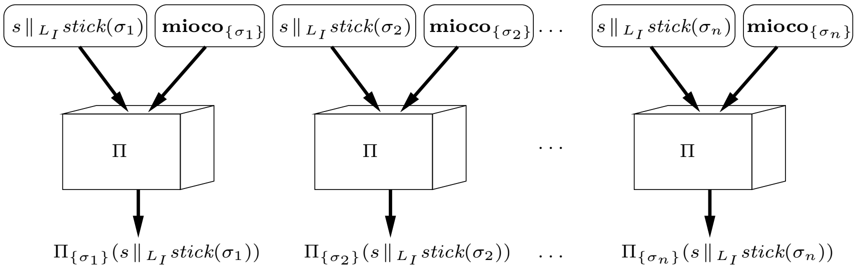

$s \parallel_{L_I} \mathit{stick}(\sigma_1)$
$\mathbf{mioco}_{\{\sigma_1\}}$
$s \parallel_{L_I} \mathit{stick}(\sigma_2)$
$\mathbf{mioco}_{\{\sigma_2\}}$
...
$s \parallel_{L_I} \mathit{stick}(\sigma_n)$
$\mathbf{mioco}_{\{\sigma_n\}}$
Π
Π
...
Π
$\Pi_{\{\sigma_1\}}(s \parallel_{L_I} \mathit{stick}(\sigma_1))$
$\Pi_{\{\sigma_2\}}(s \parallel_{L_I} \mathit{stick}(\sigma_2))$
...
$\Pi_{\{\sigma_n\}}(s \parallel_{L_I} \mathit{stick}(\sigma_n))$

Figure 3 Factorized test suite generation (soundness)

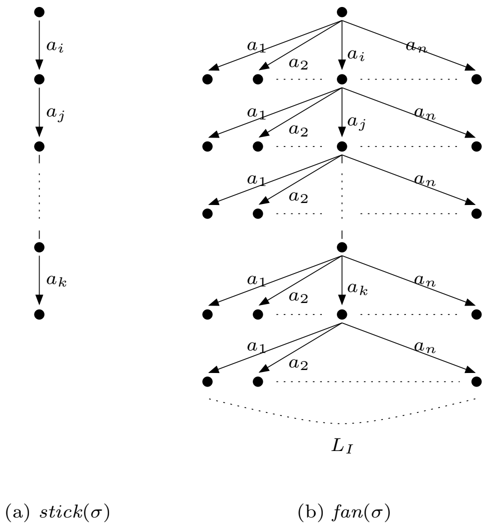
Proposition 7 $\;i\ \mathbf{mioco}_{\mathcal{F}}\ s\quad \textit{implies}\quad \forall \sigma \in \mathcal{F} : i\ \mathbf{mioco}_{\{\sigma\}}\ (s \parallel_{L_I} \mathit{stick}(\sigma))$

Algorithm Π can be applied to generate tests from $s \parallel_{L_I} \mathit{stick}(\sigma)$ for $\mathbf{mioco}_{\{\sigma\}}$. This gives us a test suite $\Pi_{\{\sigma\}}(s \parallel_{L_I} \mathit{stick}(\sigma))$ for each $\sigma \in \mathcal{F}$. The union of all these tests suites is sound for $\mathbf{mioco}_{\mathcal{F}}$. Now any implementation that fails a test in $\bigcup_{\sigma\in\mathcal{F}} \Pi_{\{\sigma\}}(s \parallel_{L_I} \mathit{stick}(\sigma))$ will also fail test suite $\Pi_{\mathcal{F}}(s)$, and hence is $\mathbf{mioco}_{\mathcal{F}}$-incorrect for s (remember the convention that $\Pi_{\mathcal{F}}(s)$ denotes the set of all tests that are generated by Π from s for $\mathbf{mioco}_{\mathcal{F}}$, and that this set is complete (proposition 2.2)).

Corollary 1 $\;i\ \textit{fails}\ \bigcup_{\sigma\in\mathcal{F}} \Pi_{\{\sigma\}}(s \parallel_{L_I} \mathit{stick}(\sigma))\quad \textit{implies}\quad i\ \textit{fails}\ \Pi_{\mathcal{F}}(s)$

Instead of generating tests from s for $\mathbf{mioco}_{\mathcal{F}}$ we can generate tests from $s \parallel_{L_I} \mathit{stick}(\sigma)$ for $\mathbf{mioco}_{\{\sigma\}}$. This reduces complexity in several ways. Specification $s \parallel_{L_I} \mathit{stick}(\sigma)$ is in most cases much smaller than s (in number of states and transitions) due to its projection on $\{\sigma\}$, and $\mathbf{mioco}_{\{\sigma\}}$ is less complex, and thus easier to check, than $\mathbf{mioco}_{\mathcal{F}}$. Although the number of test generation activities increases (for each $\sigma \in \mathcal{F}$ a test suite $\Pi_{\{\sigma\}}(s \parallel_{L_I} \mathit{stick}(\sigma))$ is generated) all these test generation activities are simpler and can be done independently.

Note that the reverse of corollary 1 does not hold; an implementation that passes test suite $\bigcup_{\sigma\in\mathcal{F}} \Pi_{\{\sigma\}}(s \parallel_{L_I} \mathit{stick}(\sigma))$ does not have to pass test suite $\Pi_{\mathcal{F}}(s)$. This is a direct consequence of the one-way implication of (3), that is a direct consequence of the one-way implication in proposition 3.1.



(a) *stick*(σ)

(b) *fan*(σ)

Figure 4 *stick*(σ) and *fan*(σ) with $\sigma\lceil L_I = a_i{\cdot}a_j{\cdot}\ldots{\cdot}a_k$

5.2 Factorized test generation (completeness)

The factorized test generation technique sketched in section 5.1 produces a sound but not necessarily complete test suite (corollary 1). In our attempts to develop a test suite that is able to reject as many faulty implementations as possible, we develop in this section a factorized test generation technique which can generate a complete test suite.

For arbitrary selection process q proposition 3.1 states that any input refusal of s is preserved in $s \,\|\,_{L_I} q$, but not necessarily vice versa: an input refusal of $s \,\|\,_{L_I} q$ can be caused by s itself, or by pruning of s with q. This also holds if the selection process q is a stick. Consequently, $s \,\|\,_{L_I} q$ may have input suspension where s has none. This exactly explains the absence of the reverse implication in proposition 6.

In order to prevent the unwanted introduction of input refusals in $s \,\|\,_{L_I} q$ we have to enforce that every input refusal of $s \,\|\,_{L_I} q$ is caused by s. This can be done by requiring that q is always able to offer any input action. In that case any input refusal of $s \,\|\,_{L_I} q$ must be caused by s itself, i.e., $s \xrightarrow{A} s$ iff $s \,\|\,_{L_I} q \xrightarrow{A} s \,\|\,_{L_I} q$. Selection processes that are prepared to synchronize on all input actions in states that lie on a particular sequence of input actions are called *fans*. A fan can be seen as a stick where in each state all input actions are offered. Figure 4(b) visualizes the structure of a fan.

Definition 7 *Let $\sigma, \sigma' \in (L_I \cup L_U \cup \mathcal{L}_I \cup \mathcal{L}_U)^*$, then $fan(\sigma)$ is the transition system that is inductively defined by*

$$
\begin{array}{lll}
fan(\epsilon) & =_{def} & \sum\{a; \mathbf{stop} \mid a \in L_I\} \\
fan(a \cdot \sigma') & =_{def} & \left\{ \begin{array}{ll} \sum\{b; \mathbf{stop} \mid b \in L_I, b \neq a\} + a; fan(\sigma') & \textit{if } a \in L_I \\ fan(\sigma') & \textit{otherwise} \end{array} \right.
\end{array}
$$

The universe of all fans is denoted by $\mathcal{FAN}(L_I)$.

We now claim that $s \,\|_{L_I} fan(\sigma)$ can be used for complete test generation from s for $\mathbf{mioco}_{\{\sigma\}}$:

$$i \;\mathbf{mioco}_{\{\sigma\}}\; s \quad \text{iff} \quad i \;\mathbf{mioco}_{\{\sigma\}}\; (s \,\|_{L_I} fan(\sigma)) \tag{4}$$

Since $s \,\|_{L_I} fan(\sigma)$ will, in most cases, be smaller than s itself it is profitable to use $s \,\|_{L_I} fan(\sigma)$ for the generation of tests. Together with equation (2) this procedure can be repeated for each $\sigma \in \mathcal{F}$, thereby obtaining a parallelization procedure for the generation of a complete test suite for $\mathbf{mioco}_{\mathcal{F}}$. This procedure can be visualized by replacing all $stick(\sigma_i)$ with $fan(\sigma_i)$ in figure 3.

Proposition 8 *$i \;\mathbf{mioco}_{\mathcal{F}}\; s$ iff $\forall \sigma \in \mathcal{F} : i \;\mathbf{mioco}_{\{\sigma\}}\; (s \,\|_{L_I} fan(\sigma))$*

As test generation algorithm Π is able to generate a test suite $\Pi_{\mathcal{F}}(s)$ from specification s that is complete with respect to $\mathbf{mioco}_{\mathcal{F}}$ (see proposition 2.2), it immediately follows from proposition 8 that an implementation is $\mathbf{mioco}_{\mathcal{F}}$-correct for s in case it passes every test in $\Pi_{\{\sigma\}}(s \,\|_{L_I} fan(\sigma))$ for all $\sigma \in \mathcal{F}$.

Corollary 2 *i fails $\bigcup_{\sigma \in \mathcal{F}} \Pi_{\{\sigma\}}(s \,\|_{L_I} fan(\sigma))$ iff i fails $\Pi_{\mathcal{F}}(s)$*

5.3 Efficient implementation of factorized test generation

Efficient implementation of factorized test generation techniques can be obtained by exploiting the special structure of the set of failure traces $\mathcal{F}$, and by sharing common parts of tests that are generated. In this subsection we briefly discuss (*i*) reducing the set $\mathcal{F}$ as far as possible without weakening the correctness criterion, and (*ii*) sharing common prefixes of test generation.

Reduction of $\mathcal{F}$: One way to reduce the set $\mathcal{F}$ of failure traces is to preprocess $\mathcal{F}$ and remove all "equivalent" failure traces that would apriori lead to the generation of identical test cases, that is, to reduce $\mathcal{F}$ to a smaller set $\mathcal{F}'$ without weakening the correctness criterion.

Find the smallest $\mathcal{F}' \subseteq \mathcal{F}$ such that $\{i \mid i \;\mathbf{mioco}_{\mathcal{F}'}\; s\} = \{i \mid i \;\mathbf{mioco}_{\mathcal{F}}\; s\}$

An example of such a reduction is the removal of failure traces that only differ in permutations of failures. Since the algorithm produces tests that check whether $out(\, i\, \mathbf{after}\, \sigma\,) \subseteq out(\, s\, \mathbf{after}\, \sigma\,)$ for each $\sigma \in \mathcal{F}$ and the set of states reachable by failure trace $\sigma_1 \cdot A \cdot X \cdot \sigma_2$ equals the set of states reachable by failure trace $\sigma_1 \cdot X \cdot A \cdot \sigma_2$ (where A and X are refusals), one of these tests is redundant.

Another example of such a reduction has to do with robustness testing. In case the behaviour of an implementation for a failure trace which is not in the specification is checked, then any implementation that accepts this failure trace is considered erroneous. Checking correctness for any longer failure trace would not be senseful, since the implementation was already considered erroneous. Consequently, it suffices to restrict to the smallest prefix of failure traces that occur in $\mathcal{F}$ and not in s.

Sharing common prefixes: For common prefixes of failures traces in $\mathcal{F}$ the application of Π can be done in a shared way, e.g., in case Π has to be applied for failure trace $\sigma \cdot \sigma_1$ and for $\sigma \cdot \sigma_2$, then the application for σ can be shared. So, test generation could be started by a single master test generation process which spawns new test generation processes at points where failure traces in $\mathcal{F}$ bifurcate.

6 CONCLUSIONS

In this paper, factorized test generation techniques were presented which can be used to generate test cases from a specification for implementation relation $\mathbf{mioco}_{\mathcal{F}}$. The factorized techniques consist of two steps. Firstly, the "complex" correctness criterion $\mathbf{mioco}_{\mathcal{F}}$ is split up in several independent and "easy-to-check" correctness criteria $\mathbf{mioco}_{\{\sigma\}}$. Secondly, the specification that is used for test generation for $\mathbf{mioco}_{\{\sigma\}}$ is reduced to a smaller specification than the original one by projecting the original specification on the correctness criterion $\mathbf{mioco}_{\{\sigma\}}$. In this way test generation from a specification for $\mathbf{mioco}_{\mathcal{F}}$ can be done more efficiently, which is necessary to make test generation for realistically-sized applications feasible. Depending on the type of selection process that is used (a stick or a fan) the factorized test generation proves to produce a sound or a complete test suite with respect to $\mathbf{mioco}_{\mathcal{F}}$, respectively.

Related work In the tool TGV [4] tests are generated with respect to a test purpose that is given as a IOLTS automaton. This test purpose acts as a selection process that is used to isolate the relevant part of the specification from which tests are generated. A similar facility is provided by the AUTOLINK tool [13] that supports the (semi-)automatic generation of tests from SDL specifications with respect to test purposes that are specified as Message Sequence Charts. This tool runs in cooperation with the SDL development environment SDT.

Further work The next step to be taken is the implementation of a tool on the basis of the theory presented here. This will require more than the direct translation into actual code of the algorithms presented in this paper. For example, algorithm Π is an abstract, generic algorithm that captures the essential idea of test generation for $\mathbf{mioco}_{\mathcal{F}}$, but which should be optimized. and made more efficient. Moreover, this paper considered factorized test generation from given $\mathcal{F}$. How to select $\mathcal{F}$, the *test selection problem*, was not discussed.

7 REFERENCES

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

[2] E. Brinksma, L. Heerink, and J. Tretmans. Developments in testing transition systems. In M. Kim et al. (eds.), *IWTCS X*, pages 143–166. Chapman & Hall, 1997.

[3] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[4] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur et al. (eds.), *CAV'96*. LNCS 1102, Springer-Verlag, 1996.

[5] L. Heerink and J. Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In T. Mizuno et al. (eds.), *FORTE X/PSTV XVII*, pages 23–38. Chapman & Hall, 1997.

[6] ISO. *LOTOS*. International Standard IS-8807. Geneve, 1989.

[7] ISO. *Conformance Testing Methodology and Framework*. International Standard IS-9646. Geneve, 1991.

[8] ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8. *Framework: Formal Methods in Conformance Testing*. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. Geneve, 1996.

[9] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

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

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

[12] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(2):241–284, 1987.

[13] M. Schmitt, B. Koch, J. Grabowski, and D. Hogrefe. – Autolink – a tool for the automatic and semi-automatic test generation. In A. Wolisz et al. (eds.), *Formale Beschreibungstechniken für verteilte Systeme*, number Nr. 315 in GMD-Studien, St. Augustin, 1997.

[14] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.