

Testing Processes from Formal Specifications with Inputs, Outputs and Data Types

Grégory Lestiennes

L.R.I Université de Paris-Sud et CNRS
Bât. 490, F-91405 Orsay-cedex, France
lestienn@lri.fr

Marie-Claude Gaudel

L.R.I Université de Paris-Sud et CNRS
Bât. 490, F-91405 Orsay-cedex, France
mcg@lri.fr

Abstract

Deriving test cases from formal specifications of communicating processes has been studied for a while. Several methods have been proposed for specifications based on FSM (Finite State Machines), LTS (Labelled Transition Systems), IOTS (Input Output Transition Systems), etc. However, most approaches are limited to a finite set of actions, excluding the possibility of communicating typed values between processes. This article presents a test derivation and selection method based on a model of communicating processes with inputs, outputs and data types, which is closer to actual implementations of communication protocols.

1 Introduction

Deriving tests from formal specifications of communicating processes has been studied for a while. Several effective methods have been proposed based on specifications corresponding to FSM (Finite State Machines), LTS (Labelled Transition Systems), etc (see [2] for an annotated bibliography).

However, these approaches are based on models limited to some finite set of actions, excluding the possibility of exchanging typed values between processes. Some generalization has been presented in [8] for full LOTOS, i.e. LOTOS with data types.

Moreover, in LOTOS, interactions between processes are modeled by undirected synchronization. More realistic models of communication between processes are provided by IOTS (Input Output Transition Systems), or IOSM (Input Output State Machines). Test methods based on such models have been proposed in [18] and [15].

This article presents a test derivation and selection method based on a model of communicating processes with inputs, outputs and data types which is closer to actual implementations of communication protocols. The approach

follows the pattern presented in [7] for test data selection from formal specifications. In a few words, a notion of “exhaustive test set” is derived from the semantics of the formal notation and from the definition of a correct implementation, assuming some “testability hypotheses” on the implementation under test. Then a finite test set is selected via some “selection hypotheses”. It turns out that making the testability conditions explicit allows the definition of an exhaustive test set, which is an improvement of the one proposed in [18] for IOTS in the case without data types. Besides, the selection hypotheses make it possible to deal with data types while keeping the test set finite.

The paper is organized as follows. Section 2 introduces the models that will be used for implementations and specifications. Section 3 recalls the definition of the implementation relation *ioco*. Section 4 recalls some generalities on testing. Section 5 summarizes the links between test hypotheses, test sets and implementation relation previously defined in [7]. In this section we also present the two most used selection hypotheses: regularity and uniformity. Section 6 and 7 present the new results: Section 6 gives the testability hypotheses and the exhaustive test set $exhaustive'_{ioco}$ for *ioco*. One gives an algorithm which generates $exhaustive'_{ioco}$ and a proof of validity and unbiased for the defined test set and hypotheses. This section improves in various ways Tretmans approach [18]. Section 7 deals with the introduction of data types: A method is described to cope with unbounded execution and the infinity of actions introduced by data types. We show how the choice of selection hypotheses can be guided by both the specifications of data types and processes. Finally, Section 8 concludes this paper with a summary of the selection method.

2 Models

Classically specifications of communicating systems are based on Labelled Transitions Systems (LTS). A LTS is a

made of a set of states Q and a set of transitions T between them labelled with actions belonging to a language L and an initial state $q_0 \in Q$. To test implementations modeled by LTS, one studies their behavior, i.e. the sequences of actions they can perform. Some actions are considered as not observable by the environment. Sequences of observable actions executable by a transition system are called its traces. Due to the possible non determinism of these systems, a trace can lead to several states. LTS have proven to be very useful models, especially at the specification level. However, when a system interacts with its environment, there is a difference between inputs and outputs. Thus, implementations are more adequately modeled by IOTS, which differs from LTS in that the language L is divided into two subsets of actions: L_I the set of input actions and L_U the set of output actions.

A specificity of IOTS is that a behavior can lead to state in which no output action can be performed. Such a state is called a quiescent state: The IOTS remains idle until some input action is executed by its environment.

Example:

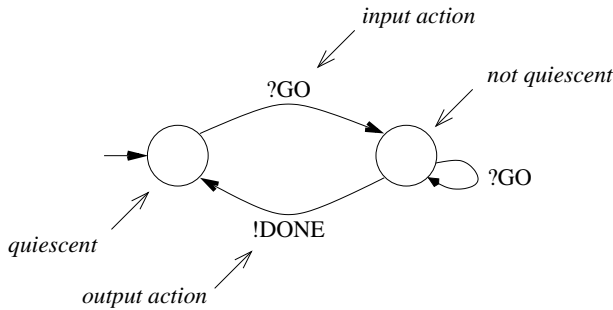


Figure 1. A very simple IOTS

The difference between input actions and output actions is modeled by the way the IOTS interacts with its environment: IOTS have the *input enabled property*, i.e. input actions are controlled by the environment and cannot be refused; output actions are controlled by the system and cannot be refused by the environment. IOTS considered in this paper are those that are *strongly convergent*, i.e. that cannot execute an infinite sequence of internal actions. Without this restriction, the observation of quiescences of implementations would not be possible.

IOTS are a development of Input Output Automata proposed by Lynch in [13]. The input enabled property has shown to be applicable in practice to a large class of interacting systems. Its pertinence is discussed by Segala in [17]. This model or some variants have been used as basis for test derivation by Phalippou [15] and Tretmans [18].

3 Implementation relations

In conformance testing, the two most used implementation relations when dealing with systems with inputs and outputs are *ioconf* and *ioco*. In this paper, we have chosen to use the second one, which is more general¹. The *ioco* relation relates specifications, which are expressed as LTS and implementations which are modeled as IOTS. This choice comes from the fact that using IOTS at the specification level turns out to be tedious. The input enabled property requires to describe the reaction of the system to any input, even in the case the input is ineffective.

The observation of quiescent states raises a problem. Practically it is done by a time-out mechanism² which checks that the system remains inactive when there is no input: In IOTS models of implementations, this idleness is described by a virtual action, classically noted δ , which loops on the quiescent states. Thus among the output actions of the IOTS modeling the implementation under test there is always this special action noted δ .

The *ioco* relation is then defined over the domain $LTS(L) \times IOTS(L_I, L_U \cup \{\delta\})$, with $L = L_I \cup L_U$, $L_I \cap L_U = \emptyset$ and $\delta \notin L$.

The definition of *ioco* is:

$$i \text{ ioco } s \text{ iff } \forall \sigma \in \text{Straces}(s): \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$$

Straces(s) are the traces of the specification s built on the language $L \cup \{\delta\}$.

The set of states i after σ is the set of all the states the implementation i can reach after an execution whose *Trace* is σ .

The set $\text{out}(S)$ with S a set of states is the set of all the actions in $L_U \cup \{\delta\}$ that can be performed, in the specification, in at least one of the states of S .

This relation forces the set of output actions of the implementation to be included in the set of the output actions of the specification, and this for every *Trace* of the specification the implementation can perform.

Considering the *Straces* of the specification makes it possible to check that after having remained in a quiescent state, the implementation can be reactivated by the arrival of some input.

As said above, the specification is modeled by an LTS to avoid specifying input actions that are ineffective. Thus the *Straces* used in the definition contain only those input actions mentioned by the specification in the current state. The implementation can have any reaction to an input action that is not specified in the current state.

¹*ioco* checks that the implementation is quiescent when the specification is. *ioconf* does not.

²Of course, it is an approximation. Note that this makes sense only if the IOTS is strongly convergent as mentioned above.

Remarks:

- As the absence of output action is considered to be an observable action δ , the implementation can be quiescent only if the specification also can.
- It is set inclusion and not set equality which is required, so the implementation can be more deterministic than the specification.
- After any Strace σ of the implementation there is always at least one action of $L_U \cup \{\delta\}$ which is executable in the IOTS model of the implementation: If no output action belonging to L_U is fireable, then δ is “performed”.

A study of *ioco* using IOTS (to define implementation and tests) has already been made by Tretmans in [18]. An implementation relation slightly different from *ioco* has been used to develop the tool TGV [6]. But these studies did not take into account data types for the generation of the tests. We show that such a generalization is possible, even if it often leads to LTS and IOTS with an infinity of actions and then an infinity of states.

4 Generalities on Process Testing

We consider the case where process testing aims at checking that the behavior of an implementation is conform to its specification. The behavior of the implementation corresponds to the sequences of observable actions (i.e. communications actions + quiescence) the implementation can perform. Internal actions cannot be seen by the environment and are thus ignored.

To test a process, one uses tests³ built on the same language of actions as the specification. As for the model of implementation, this language is divided into two subsets of actions: The output actions corresponding to the input actions of the implementation, and the input actions corresponding to the output actions of the implementation. Moreover, the special action δ of the specification corresponds to some time-out mechanism in the test.

Test execution consists in running the implementation and the test in parallel. During this execution, the implementation and the test synchronize on each action of L . This synchronization is quite natural since the input actions of the implementation are the output actions of the test, and vice versa. A verdict about the success of the test execution by the implementation is given, depending on the implementation relation and the observations made (usually traces and deadlocks) during the test execution.

When testing parallel systems, the considered traces are interleavings of the traces of each process constituting the system. Of course, such a trace makes sense only if the

³Strictly speaking, they are testers rather than tests. In this paper we use the term test as usually in the literature.

actions are atomic or at least observable as if they were atomic.

There are mainly two ways the tests can be designed: One can build a single global test which checks fully the conformity of the implementation the way Phalippou did in his thesis [15] following an approach similar to Brinskma’s canonical tester in [1].

The second way of designing tests is to define a test set⁴, each test corresponding to one (or several) particular execution(s). This makes selection easier and makes it possible to determine a test subset for a given test purpose. In this paper we chose this second kind of tests.

5 Exhaustivity and Testability

5.1 Test Experiments, Exhaustivity, Testability

The implementation relation *i impl s* is generally a large conjunction of elementary properties (for instance it may begin by “for all traces in the specification s ”). These elementary properties are the basis for the definition of what is a test experiment, a test data, and the verdict of a test experiment, i.e. the decision whether an implementation *i* passes a test *t*. The implementation relation as a whole is used for the definition of an exhaustive test set, *exhaustive(s)*. It is generally unrealistic, but it aims at providing a reference, as close as possible to correctness, for the definition of testing strategies and the selection of finite test sets. However, an implementation’s passing all the tests in the exhaustive test set does not necessarily mean that it satisfies the specification.

This is true for a class of reasonable implementations. But a totally erratic system or a diabolic one may pass the exhaustive test set and then fail. More formally, the implementation under test must fulfill some basic requirements coming from the kind of models considered for the specifications and the implementations. For instance, in the case of finite automata, the implementation must behave like an automata, without memory of its history: The new state after a transition *t* must depend on the previous state and *t* only. We call such a property a testability hypothesis, i.e. an hypothesis without which testing the implementation is meaningless⁵. Here we note the set of testability hypothesis *Htest(i)*.

Htest, *exhaustive*, and *impl* must satisfy:

$$Htest(i) \Rightarrow (i \text{ passes } exhaustive(s) \iff i \text{ impl } s). \quad (1)$$

There are some cases where several choices are possible for the pair $\langle Htest, exhaustive \rangle$. Intuitively, when

⁴Sometimes called a “test suite” even if the order is not relevant.

⁵This notion is different from the one given in the The IEEE Standard Glossary of Software Engineering Terminology (1990).

restricting the class of testable implementations, it is possible to weaken *exhaustive(s)*. Note that the verification of *Htest* may be ensured by various techniques, such as static analysis, proof, or other kind of testing.

5.2 Selection Hypotheses, Uniformity and Regularity

A black-box testing strategy can be formalized as the selection of a finite subset of *exhaustive(s)*. Let us consider as an example the classical partition testing strategy (more exactly, the sub-domain testing strategy). It consists in defining a collection of (possibly non-disjoint) subsets that covers the exhaustive test set. Then an element of each subset is selected and submitted to the implementation under test. The choice of such a strategy corresponds to stronger hypotheses than *Htest* on the implementation under test. We call such hypotheses selection hypotheses. In this case, they are called uniformity hypotheses. The implementation under test is assumed to uniformly behave on the test subsets UTS_i :

$$UTS_1 \cup \dots \cup UTS_p = \text{exhaustive}(s), \text{ and } \forall j = 1, \dots, p \\ \forall t \in UTS_j, i \text{ passes } t \implies i \text{ passes } UTS_j. \quad (2)$$

Another common kind of selection hypothesis is regularity hypotheses. In presence of infinite tests, it assumes that if the tests of size less than a certain bound are successful then the tests of any size also are.

Various selection hypotheses can be formulated and combined depending on some knowledge of the program, some coverage criteria of the specification and ultimately cost considerations. For example, Phalippou presented some independence and fairness hypotheses in [15] for the test of communication protocols against Input-Output State Machines descriptions. All these hypotheses are important from a theoretical point of view because they formalize common test practices and express the gap between the success of a test strategy and correctness. They are also important in practice because exposing them makes clear the assumptions made on the implementation. It gives some indication of complementary verifications [4, 12].

5.3 Validity and Unbias

A pair (H, T) of a set of hypotheses and one of tests is considered valid if H implies that if T is passed then *exhaustive(s)* is as well. It is considered unbiased if H implies that if *exhaustive(s)* is passed then T is as well. Assuming H , validity guarantees that all incorrect implementations are rejected, and being unbiased guarantees that no correct implementation is rejected.

From (1), $(Htest(i), \text{exhaustive}(s))$ is both valid and unbiased. Another extreme example that is both valid and unbiased is $(Htest(i) \wedge i \text{ passes } \text{exhaustive}(s), \emptyset)$, which indicates that if the implementation is assumed to be correct then no tests are needed! Interesting pairs are those that are valid and unbiased, with reasonable hypotheses stronger enough to reduce the set of tests to a tractable size.

6 Exhaustive test set and verdict for ioco

In this section we first give the testability hypotheses and then the exhaustive test set for *ioco*. Namely for a specification s , we give the definition of the test set $\text{exhaustive}'_{ioco}(s)$. Moreover, we give an algorithm which generates this test set. Finally we prove the equivalence between the conformance with respect to *ioco*, and success of $\text{exhaustive}'_{ioco}(s)$, whenever the test hypotheses are satisfied. Although hypotheses are presented first, they were defined concurrently with the test set.

6.1 Test Hypotheses

Our definitions of exhaustive test set for *ioco*, are based on the following testability hypotheses.

1) We make the assumption that the implementation is input enabled (cf Section 2). Thus, all the input actions must be implemented.

2) When testing a non deterministic implementation, one has to make the assumption that after a sufficient number of executions of the same test, all the paths corresponding to this test in the implementation have been taken. This classical assumption is known as the “complete testing assumption”. It ensures that the implementation will not have any other behaviors that those observed during the test executions. Thus each test t is performed several times. t is passed successfully by an implementation if and only if the verdict of all these executions is a success.

3) For implementations including some internal parallel executions, actions are supposed to be atomic or at least observable as if they were atomic. Moreover, they are supposed to conform to the interleaving model of parallelism. Therefore, such implementations can be tested as if they were sequential.

4) An hypothesis always done when the tests are a set of independent tests (as in this paper) is that the system can be correctly reset to be able to execute the tests starting from the initial state.

5) As said in Section 2, implementations have to be strongly convergent, to be able to check the quiescence of states.

These five hypotheses characterize the class of implementations for which the test sets built in the next sections

give a correct result. These hypotheses will be called $Hypo_{I/O}$.

6.2 Testing the ioco relation

The *ioco* relation is defined by:

$i \text{ ioco } s$ iff $\forall \sigma \in \text{Straces}(s)$: $\text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$

According to this definition, one can define an exhaustive test set $exhaustive_{ioco}$ for a specification s as:
 $exhaustive_{ioco}(s) = \{\sigma; a; stop \mid \sigma \in \text{Straces}(s)^6$
 $\text{and } a \in L_U \cup \{\delta\}\}$

None of the tests ends by an action belonging to L_I . This comes from the first hypothesis: Any input action can be performed by the implementation in any state. Thus such tests would be useless.

The verdict corresponding to this test set is: ⁷

- if $\sigma \cdot a$ cannot be performed by the implementation then success
- else if $\sigma \cdot a$ can be performed by the specification then success
- else failure

The first case above corresponds to some Straces of the specification not executable by the implementation. Remember that the implementation can be more deterministic than the specification. We come back to this point in the proof in Section 6.4.

The test set above can be improved. Let us consider the tests corresponding to a Strace $\sigma \cdot a$ of the specification. Due to the second case of the verdict, the associated test executions will lead to a success whatever the way the implementation behaves. Thus these tests can be removed. Moreover, testing the quiescence twice consecutively, or more, is useless because once an implementation is idle, it remains idle until an input action is executed by its environment. Thus tests corresponding to a Strace with two or more consecutive δ actions can also be removed from the exhaustive test set.

The test set resulting from the suppression of these two kinds of tests is called $exhaustive'_{ioco}$:

$$exhaustive'_{ioco}(s) = \{\sigma; a; stop \mid \sigma \in \text{Straces}(s)$$

$$\text{and } a \in (L_U \cup \{\delta\}) - \text{out}(s \text{ after } \sigma)$$

$$\text{and } \nexists \sigma', \sigma'' \mid \sigma; a = \sigma'; \delta; \delta; \sigma''\}$$

The set of actions $\text{out}(s \text{ after } \sigma)$ can be computed using the suspension automata defined by Tretmans [18]. This is the determinized automata corresponding to the traces of

⁶The syntax used to describe a test is similar to LOTOS: The “;” is used for the sequence and “stop” is a special symbol indicating termination.

⁷A sequence of observed actions is noted using “:”.

the specification, with the addition of the transition (s, δ, s) for any quiescent state s .

The verdict corresponding to this test set is:

- if $\sigma \cdot a$ cannot be performed by the implementation then success
- else failure

The intuition behind the tests of $exhaustive'_{ioco}(s)$ and the associated verdict is that after each correct behavior, one tries to provoke an incorrect output (or quiescence). Thus the execution corresponding to a test of $exhaustive'_{ioco}(s)$ is a failure if and only if it arrives to the end of the test. The definition of *ioco* naturally leads to such tests: specified outputs are not mandatory, but unspecified ones are forbidden.

6.3 Test generation algorithm

In this section, we describe a non deterministic algorithm which generates some tests belonging to the set $exhaustive'_{ioco}$.

Let s be a specification with initial state s_0 , let σ be a Strace initialized to ε , and let S be a non-empty set of states initialized to $\{s_0\}$. Then a test t is obtained from a given S and a given σ by a finite number of recursive applications of the following rules:

1. Terminate the test:
if σ has the form $\sigma' \cdot \delta$ then $t := a; stop$ where $a \in L_U - \text{out}(S) - \{\delta\}$
else $t := a; stop$ where $a \in L_U - \text{out}(S)$
2. Continue the test with an input action:
 $t := a; t'$ where $a \in L_I$, $S \text{ after } a \neq \emptyset$ and t' is obtained recursively by applying the algorithm for $S' = S \text{ after } a$ and $\sigma' = \sigma \cdot a$
3. Continue the test with an output action:
if σ has the form $\sigma' \cdot \delta$ then $t := a; t'$ where $a \in \text{out}(S) - \{\delta\}$ and t' is obtained by applying recursively the algorithm for $S' = S \text{ after } a$ and $\sigma' = \sigma \cdot a$
else $t := a; t'$ where $a \in \text{out}(S)$ and t' is obtained recursively by applying the algorithm for $S' = S \text{ after } a$ and $\sigma' = \sigma \cdot a$

This algorithm uses intensively the sets $\text{out}(S)$ and $S \text{ after } a$. The computation of these sets corresponds to the construction of the suspension automata mentioned in Section 6.2. Once this automata is computed, the complexity of the generation of a test is proportionnal to its length.

This algorithm improves a similar algorithm by Tretmans [19]. Tretmans' algorithm produces some tests that we do not have in $exhaustive'_{ioco}$: First there is the test

corresponding to the empty trace. This test is not interesting because it succeeds for any implementation. We have also discarded the tests ending with an input action. Under the assumption that the implementation is input enabled (hypothesis 1), those tests become useless. Finally, we removed the tests corresponding to Straces with successive δ actions.

Note that the extra tests generated by the algorithm of Tretmans check only partially the input enabled property since only those input actions mentioned by the specification after a Strace are tested.

6.4 Proof

In this proof, the pair $(Hypo_{I/O}, exhaustive'_{ioco})$ is considered. We prove:

- Validity: If the test hypotheses of Section 6.1 are satisfied then the success of $exhaustive'_{ioco}(s)$ by the implementation implies that the implementation is conform to the specification with respect to the *ioco* relation.
- Unbias: If the implementation conforms to the specification with respect to the *ioco* relation and if the hypotheses are satisfied then the implementation passes $exhaustive'_{ioco}(s)$ with success.

As said in Section 6.1, due to the possible non determinism of the implementation, we have to execute each test several times. So, for an implementation i and a test t , we will say that we have $success(i, t)$ if and only if every execution of the test t has been a success. Let us recall that: (\star) Tests corresponding to a Strace with successive δ actions behave the same way as the corresponding tests where each δ action suite is replaced by a single δ action.

Let i be an implementation and let s be a specification.

Let us suppose that:

- $\forall t \in exhaustive'_{ioco}(s) = \{\sigma; a; stop \mid \sigma \in Straces(s) \text{ and } a \in (L_U \cup \{\delta\}) - out(s \text{ after } \sigma) \text{ and } \nexists \sigma', \sigma'' \mid \sigma; a = \sigma'; \delta; \delta; \sigma''\}, success(i, t)$

and

- i satisfies the test hypotheses of Section 6.1

By construction of $exhaustive'_{ioco}(s)$, every Strace $\sigma \cdot a$ where $\sigma \in Straces(s)$ and $a \in (L_U \cup \{\delta\}) - out(s \text{ after } \sigma)$ is tested (assuming (\star)). As each test t with Strace $\sigma \cdot a$ is executed a sufficient number of times to know if the output action a is or not feasible after σ , every output action of $(L_U \cup \{\delta\}) - out(s \text{ after } \sigma)$ feasible by i after σ is found.

Then $\forall \sigma \in Straces(s)$, the set of actions $a \in (L_U \cup \{\delta\}) - out(s \text{ after } \sigma)$ observed after σ is exactly the set of

actions of $(L_U \cup \{\delta\}) - out(s \text{ after } \sigma)$ that i can perform after σ .

According to the verdict, for a test t with Strace $\sigma \cdot a$, one has $success(i, t)$ iff $\sigma \cdot a$ cannot be performed by i .

For any Strace $\sigma \in Straces(s)$, every test $\sigma; a; stop$ where $a \in (L_U \cup \{\delta\}) - out(s \text{ after } \sigma)$ is in $exhaustive'_{ioco}(s)$ (assuming (\star)). We know that every test in $exhaustive'_{ioco}(s)$ is successful.

Thus for a Strace $\sigma \in Straces(s)$ and each test $\sigma; a_k; stop$, such that $a_k \in (L_U \cup \{\delta\}) - out(s \text{ after } \sigma)$, we have $\sigma \cdot a_k$ cannot be performed by i , and then we have $a_k \notin out(i \text{ after } \sigma)$, because:

- if the test blocks right before a_k , then this implies that $a_k \notin out(i \text{ after } \sigma)$
- if the test blocks before the end of σ , then:

$$i \text{ after } \sigma = \emptyset$$

$$\Rightarrow out(i \text{ after } \sigma) = \emptyset$$

$$\Rightarrow a_k \notin out(i \text{ after } \sigma)$$

So we have $out(i \text{ after } \sigma) \subseteq out(s \text{ after } \sigma)$, for any Strace $\sigma \in Straces(s)$.

Thus if the implementation i satisfies the test hypotheses and if for all test $t \in exhaustive'_{ioco}(s)$ we have $success(i, t)$ then $i \text{ ioco } s$.

Now let us suppose that $i \text{ ioco } s$. According to the definition of *ioco*, for all Strace σ of s , $out(i \text{ after } \sigma)$ is included in $out(s \text{ after } \sigma)$. For any test $t = \sigma; a; stop$ of $exhaustive'_{ioco}(s)$, as $a \notin out(s \text{ after } \sigma)$ then $a \notin out(i \text{ after } \sigma)$ and then $\sigma \cdot a$ cannot be performed by i . This corresponds to a success.

Thus if $i \text{ ioco } s$ then for all test $t \in exhaustive'_{ioco}(s)$, we have $success(i, t)$.

7 Introduction of data types

In practice, most processes deal with values of some data types. This leads to infinite (or extremely large) LTS and IOTS, and consequently to infinite (or extremely large) exhaustive test sets. Thus there is a problem of choice of pertinent selection hypotheses, based both on data types properties and processes structures.

In this section we present such a selection method using the example of a specification of an unbounded buffer of messages with priorities. We give a finite set of tests for this specification selected from the corresponding $exhaustive'_{ioco}$ test set. This selection is based on regularity and uniformity hypotheses (cf Section 5.2) derived from the text of the specification.

7.1 Presentation of the buffer specification

The buffer has output actions of the form $outGate!m$ where m is a value of type `Message`, input actions derived from $inGate?M:Message$ where M is a variable of type `Message` and one input action $ready?$.

$$L_U = \{outGate!m \mid m : Message\}$$

$$L_I = \{inGate?M : Message, ready?\}$$

Besides, it uses the type `PriorityQueue` with the classical operations add , $remove$ and get , whose an algebraic specification is given in Figure 2.

The behavior of the Buffer is described on Figure 3 in a LOTOS-like syntax. It can be summarized as follows: Initially the queue is empty (line 5). Then the buffer can receive a message, which is added to the queue (line 8). It can also receive a signal that means that the consumer is ready (line 10). In this case it sends a message, which is removed from the queue (line 14).

From the `PriorityQueue` specification, if the queue is empty, then the empty message is sent.

type `PriorityQueue` is `String`, `Nat`, `Boolean`

sorts `Queue`, `Message`

opns

`emptyq`: \rightarrow `Queue`
`add`: `Message`, `Queue` \rightarrow `Queue`
`get`: `Queue` \rightarrow `Message`
`remove`: `Queue` \rightarrow `Queue`
`isEmpty`: `Queue` \rightarrow `Boolean`
 $(_,_)$: `Nat`, `String` \rightarrow `Message` {A message is a priority and a text }
`priority`: `Message` \rightarrow `Nat`
`text`: `Message` \rightarrow `String`

eqns

forall $s:String, n:Nat, q:Queue, m:Message$

definition of get

{ get returns the message whose priority is the higher and which is the oldest.}
`get`(`emptyq`)= $(0, \langle \rangle)$; { $\langle \rangle$ is the empty string}
`isEmpty`(q)=`true` \Rightarrow `get`(`add`(m, q)) = m ;
`isEmpty`(q)=`false`, `priority`(`get`(q)) ge `priority`(m) \Rightarrow `get`(`add`(m, q))=`get`(q);
`isEmpty`(q)=`false`, `priority`(`get`(q)) lt `priority`(m) \Rightarrow `get`(`add`(m, q))= m ;

definition of $remove$

{ $remove$ removes the message whose priority is the higher and which is the oldest.}
`remove`(`emptyq`)=`emptyq`;
`isEmpty`(q)=`true` \Rightarrow `remove`(`add`(m, q))=`emptyq`;
`isEmpty`(q)=`false`, `priority`(`get`(q)) ge `priority`(m) \Rightarrow `remove`(`add`(m, q))=`add`($m, \text{remove}(q)$);
`isEmpty`(q)=`false`, `priority`(`get`(q)) lt `priority`(m) \Rightarrow `remove`(`add`(m, q))= q ;

definition of $isEmpty$

`isEmpty`(`emptyq`)=`true`;

`isEmpty`(`add`(m, q))=`false`;

definition of $priority$

`priority`((n, s))= n ;

definition of $text$

`text`((n, s))= s ;

endtype

Figure 2. The `PriorityQueue` type in LOTOS [11]

1. **gates**
2. **in**: `inGate`, `ready`
3. **out**: `outGate`
4. **behavior**
5. `Buffer`(`emptyq`) {Initial state}
6. **where**
7. **process** `Buffer`($Q:Queue$):= {Parameterized process}
8. `inGate?M:Message; Buffer`(`add`(M, Q))
9. `[]` {or operator}
10. `ready?; Buffer`(`Ready`(Q))
11. **endproc**
12. **where**
13. **process** `Buffer`(`Ready`($Q:Queue$):=
14. `outGate!get`(Q); `Buffer`(`remove`(Q))
15. `[]`
16. `inGate?M:Message; Buffer`(`Ready`(`add`(M, Q))
17. **endproc**

Figure 3. Specification of the behavior of the Buffer process

The corresponding infinite LTS has the following set of states: $Q = \{Buffer(Q) \mid Q \in T_{\Sigma_{Queue}}\} \cup \{BufferReady(Q) \mid Q \in T_{\Sigma_{Queue}}\}$

Where $T_{\Sigma_{Queue}}$ is the set of closed terms of sort `Queue`.

Its initial state is: $q_0 = Buffer(emptyq)$

Its set of transitions is: $T = \{ \langle Buffer(Q), ready?, BufferReady(Q) \rangle \} \cup \{ \langle Buffer(Q), inGate?M, Buffer(add(M, Q)) \rangle \} \cup \{ \langle BufferReady(Q), outGate!get(Q), Buffer(remove(Q)) \rangle \} \cup \{ \langle BufferReady(Q), inGate?M, BufferReady(add(M, Q)) \rangle \} \forall Q \in T_{\Sigma_{Queue}}, \forall M \in T_{\Sigma_{Message}}$

Let's note that the construction of this LTS is quite simple since there is no guard on the transitions in the specification. We will come back to this point later.

7.2 The full test set

Given the previous LTS, $exhaustive'_{ioco}(Buffer(emptyq))$ is defined as in section 6.2. The problem is now to

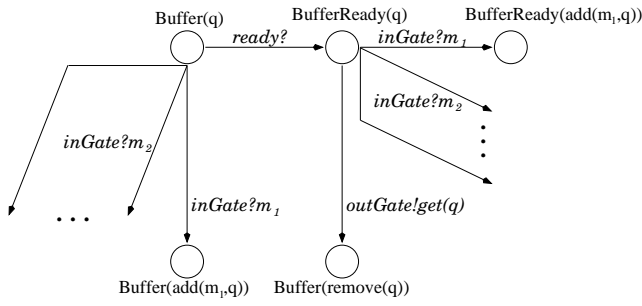


Figure 4. A part of the infinite LTS; q is any closed term of sort Queue

select an adequate subset of this set of tests. To this end, we build symbolic tests which correspond to classes of actual tests.

A symbolic test is a test which contains symbolic actions, i.e. actions where parameters and variables are not instantiated.

Syntactically, from the text of the specification, we see that there are two symbolic states (or state classes):

- Those which correspond to a recursive call: $Buffer(Q)$
- Those which correspond to states reached after the execution of the action $ready?$: $BufferReady(Q)$, where Q is any queue.

In the sequel, we note $Strace'(s)$ those traces of $Strace(s)$ without successive δ actions, which are prefixes of the tests of $exhaustive'_{ioco}(s)$.

Let us consider the traces in $Strace'(Buffer(emptyq))$ ending in a state of the class $Buffer(Q)$. This symbolic state has no output action thus it is a class of quiescent states. Therefore, for any $s \in Buffer(Q)$, $out(s) = \{\delta\}$. So we have to check that in such states, the implementation cannot produce any message. Forbidden output actions in the states in $Buffer(Q)$ are:

- $outGate!m$ where m is any message

Thus the tests for such a trace σ are:

- $\sigma; outGate? m; stop | m \in T_{\Sigma_{Message}}$

These tests can be factorized using a symbolic action to detect non conform outputs on the gate $outGate$:

- $\sigma; outGate? M:Message; stop$

As defined in Section 6.2, if the test execution reaches the end of the test then it is a failure.

Now let us consider the traces in $Strace'(Buffer(emptyq))$ ending in a state of the second class. The symbolic state $BufferReady(Q)$ has one output action: $outGate!get(Q)$. So the states of this class are not quiescent. The allowed output actions is:

- $outGate!get(Q)$

And the forbidden actions are:

- $outGate!m$ where m is any message different from $get(Q)$
- δ

Thus for such a trace σ , we have tests whose form is $\sigma; outGate?M : Message [M \neq get(Q)]; stop$ and in the case where σ does not end by δ we have the tests $\sigma; \delta; stop$.

In conclusion, we have a partition of the set $exhaustive'_{ioco}(Buffer(emptyq))$ where the classes are represented by the following symbolic tests:

- For all Straces' σ leading to a state of $Buffer(Q)$:
 - $\sigma; outGate?M:Message; stop$
- For all Straces' σ leading to a state of $BufferReady(Q)$:
 - $\sigma; outGate?M:Message [M \neq get(Q)]; stop$
- For all Straces' σ leading to a state of $BufferReady(Q)$ and ending by an action different from δ :
 - $\sigma; \delta; stop$

7.3 Selection Hypotheses

The partition above has obviously an infinite number of classes because of the infinite number of traces: Thus to define a finite test set we have to choose selection hypotheses of two kinds: Regularity hypotheses which allow to bound the length of the traces; Uniformity hypotheses which define equivalence classes on the values occurring in the actions of the traces. In our approach, uniformity hypotheses are deduced from the properties of the data types once the regularity hypotheses have been chosen. The choice of the regularity hypotheses is based on the recursive calls.

7.3.1 How to choose Selection Hypotheses

Let us study some criteria for limiting the length of the tests. It seems interesting to look at the evolution of the parameter which in our example involves the operations add and $remove$. The operation add is a "free constructor", i.e. it is not defined by any equation. Thus the data type definition gives no special case to be considered for this operation. On the contrary, there are four equations defining the operation $remove$, and these equations introduce three cases: The empty queue, queues with one element and queues with two or more elements. This operation is used in the (syntactically) first nested call of $Buffer$ in $BufferReady$ and a reasonable criteria is to cover all the cases mentioned in its definition. For that it is sufficient to consider queues with 0,1 or 2 elements. To reach this call with a queue of two elements it is necessary to have tests of length 4: Two messages inputs, one ready input and the symbolic action rejecting non conform outputs. As this is the only parameter evolution involving various cases, this leads to a regularity hypothesis where the length of the tests is less or equal to 4.

After this step, the test set is reduced but still infinite. Let us consider the equivalence classes based on the paths occurring in the behavior part of the specification and selected

by the regularity hypothesis. A predicate on messages and queues is associated to each path that conditions its execution. These predicates give a hint for choosing the uniformity hypotheses in order to cover the cases occurring in the process description. For example if one considers the symbolic path $inGate!M1; inGate!M2; ready!; outGate?M3$, the predicate is: $M3=get(add(M2,add(M1,emptyq)))$

Thus for each path predicate, we select one test. For our example, the test is an instantiation of the following symbolic test:

$inGate!M1; inGate!M2; ready!; outGate?M3:Message [M3 \neq get(add(M2,add(M1,emptyq)))]$.

This uniformity hypothesis on path predicates and the regularity hypothesis above lead to a finite test set.

The number of tests depends on the number of paths in the specification, which is rarely very big, and of the bound introduced by the regularity hypothesis. For usual data types, this bound is often small. However it is not difficult to invent a data type specification with a special case that requires very long tests (for instance, some sort of overflow). In this case, other selection methods than regularity must be considered [5].

7.3.2 Operation unfolding

So far we have used the behavior part for stating uniformity hypotheses. It is possible to refine them by using the data type part of the specification. One may perform “operation unfolding” in the paths’ predicates. Unfolding is a classical technique [3]. It is done by replacing an operation by its definitions determined by the equations and adding the condition of their application to the predicate of the path.

For example, in the path $inGate!M1; inGate!M2; ready!; outGate?M3:Message [M3=get(add(M2, add(M1, emptyq)))]$, one can unfold the operation *get*: the two first equations cannot be applied because of the structure of the current queue (2 elements). But the third and fourth equations can be applied. Unfolding the operation *get* in the previous case leads to the following interesting sub-cases:

1. $inGate!M1; inGate!M2; ready!; outGate?M3:Message [M3=M2 \wedge priority(get(add(M1,emptyq))) \text{ lt } priority(M2)]$
2. $inGate!M1; inGate!M2; ready!; outGate?M3:Message [M3=M1 \wedge priority(get(add(M1,emptyq))) \text{ ge } priority(M2)]$

Unfolding leads to weaker uniformity hypotheses. A large (possibly infinite) number of unfoldings can often be done for a given predicate so it is necessary to limit unfoldings in order to avoid useless value enumerations. The tool Gatel [14] developed by Marre proposes an assistance to control unfoldings for the case of Lustre language. This approach can be used in our case.

In the example, unfolding *get* as above, and *ge* (into *gt* and *eq* cases) yields to interesting nominal and boundary cases which correspond to refined uniformity hypotheses on

queues. As message texts are just passed without any treatment, unfolding keeps a full uniformity hypotheses on the sort String.

7.3.3 Predicate Resolution

The last step in the test generation is to instantiate variables with actual values. This is done by solving the predicates of the previous cases. Powerful tools are now available for solving large classes of constraints and predicates [14, 16, 9]. One of these tools, Gatel[14] which is based on randomized constraint solving and sophisticated heuristics, is used in an industrial context. In our example, the previous paths 1. and 2. lead to the following tests:

- $inGate!(2, "ab"); inGate!(8, "bc"); ready!; outGate?M:Message [M \neq (8, "bc")]; stop$
- $inGate!(14, "a"); inGate!(5, "b"); ready!; outGate?M:Message [M \neq (14, "a")]; stop$

In these tests the last action is still a symbolic one. This comes from the verdict: The idea is that we take the system to a specific state using precise actions and then we detect the production of any non conform action. This is the factorization of some of the tests $\{\sigma; a; stop | a \notin out(S \text{ after } \sigma)\}$. Namely those where *a* uses the same gate.

The general case is slightly more complex. For example if after a Strace σ , we are in a state where the two symbolic actions $out!X[cond1]$ and $out!Y[cond2]$ are fireable, then in the suspension automata (which is deterministic), the fireable actions will be: $out!X[cond1 \wedge \neg cond2]$, $out!X[cond1 \wedge \neg cond1]$ and if there are values fitting *cond1* and *cond2*, there will be the symbolic action $out!X[cond1 \wedge cond2]$.

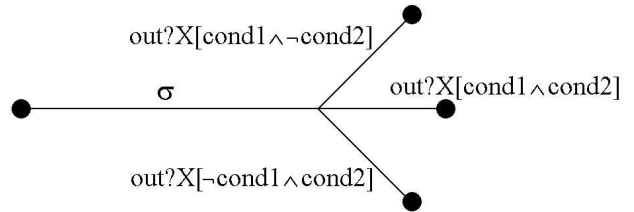


Figure 5. Possible choices in the suspension automata after a Strace σ

The values that shall not be emitted are those conditioned by the predicate $\neg cond1 \wedge \neg cond2$. Assuming the variables *X* and *Y* have type *t*, the test set will contain the following test:

$\sigma; out?x:t [\neg cond1 \wedge \neg cond2]$

This can be generalised to N actions using the same gate,

each of them having a different condition: The corresponding test's condition will be the conjunction of the negation of each condition.

A well-known problem in structural testing is the possibility of infeasible paths. Here, we are faced to the same difficulty since a symbolic trace can be infeasible due to contradictory guards in its symbolic actions. A specificity here is that some states may be quiescent even if, syntactically, some output action seems to be fireable.

Let us consider a process definition of the form:

```

process P(x:T):=
  in1?I1;...
  [] ...
  [] inn?In;...
  [] [guard1(x)]→ out1!O1;...
  [] ...
  [] [guardm(x)]→ outm!Om;...
endproc

```

If there exists some x such that the conjunction of the negation of all the guards is true then the states corresponding to these values of the parameters are quiescent. It means that we must consider the additional case for $P(x)$: $[\neg\text{guard}_1(x)\wedge\dots\wedge\neg\text{guard}_m(x)]\rightarrow\delta$ in the generation algorithm.

Clearly, a constraint solver may not terminate when searching solutions to these predicates. As said above there exist now sufficiently reliable tools to make such methods applicable.

7.3.4 Some tests for the example

Using our method on the example of the message buffer leads to the following 45 tests:

- $|\sigma|=0$:
outGate?M;stop
- $|\sigma|=1$:
inGate!(1,"a");outGate?M;stop
 δ ;outGate?M;stop
ready!;outGate?M[M \neq (0,<>)];stop
ready!; δ ;stop
- $|\sigma|=2$:
inGate!(1,"a");inGate!(2,"b");outGate?M;stop
inGate!(1,"a"); δ ;outGate?M;stop
inGate!(1,"a");ready!;outGate?M[M \neq (1,"a")];stop
inGate!(1,"a");ready!; δ ;stop

 δ ;inGate!(1,"a");outGate?M;stop
 δ ;ready!;outGate?M[M \neq (0,<>)];stop
 δ ;ready!; δ ;stop

ready!;inGate!(1,"a");outGate?M[M \neq (1,"a")];stop
ready!;inGate!(1,"a"); δ ;stop

ready!;outGate?(0,<>);outGate?M;stop

- $|\sigma|=3$:
inGate!(1,"a");inGate!(2,"b");inGate!(3,"c");
outGate?M;stop
inGate!(1,"a");inGate!(2,"b"); δ ;outGate?M;stop
inGate!(1,"a");inGate!(2,"b");ready!;
outGate?M[M \neq (2,"b")];stop
inGate!(1,"a");inGate!(0,"b");ready!;
outGate?M[M \neq (1,"a")];stop
inGate!(1,"a");inGate!(1,"b");ready!;
outGate?M[M \neq (1,"a")];stop
inGate!(1,"a");inGate!(2,"b");ready!; δ ;stop

inGate!(1,"a"); δ ;inGate!(2,"b");outGate?M;stop
inGate!(1,"a"); δ ;ready!;outGate?M[M \neq (1,"a")];stop
inGate!(1,"a"); δ ;ready!; δ ;stop

inGate!(1,"a");ready!;inGate!(2,"b");
outGate?M[M \neq (2,"b")];stop
inGate!(1,"a");ready!;inGate!(0,"b");
outGate?M[M \neq (1,"a")];stop
inGate!(1,"a");ready!;inGate!(1,"b");
outGate?M[M \neq (1,"a")];stop
inGate!(1,"a");ready!;inGate!(2,"b"); δ ;stop
inGate!(1,"a");ready!;outGate?(1,"a");outGate?M;stop

 δ ;inGate!(1,"a");inGate!(2,"b");outGate?M;stop
 δ ;inGate!(1,"a"); δ ;outGate?M;stop
 δ ;inGate!(1,"a");ready!;outGate?M[M \neq (1,"a")];stop
 δ ;inGate!(1,"a");ready!; δ ;stop

 δ ;ready!;inGate!(1,"a");outGate?M[M \neq (1,"a")];stop
 δ ;ready!;inGate!(1,"a"); δ ;stop
 δ ;ready!;outGate?(0,<>);outGate?M;stop

ready!;inGate!(1,"a");inGate!(2,"b");
outGate?M[M \neq (2,"b")];stop
ready!;inGate!(1,"a");inGate!(0,"b");
outGate?M[M \neq (1,"a")];stop
ready!;inGate!(1,"a");inGate!(1,"b");
outGate?M[M \neq (1,"a")];stop
ready!;inGate!(1,"a");inGate!(2,"b"); δ ;stop
ready!;inGate!(1,"a");outGate?(1,"a");outGate?M;stop

ready!;outGate?(0,<>);inGate!(1,"a");outGate?M;stop
ready!;outGate?(0,<>); δ ;outGate?M;stop
ready!;outGate?(0,<>);ready!;outGate?M[M \neq (0,<>)];stop
ready!;outGate?(0,<>);ready!; δ ;stop

7.4 Test factorization

Because of the input enabled property (cf Section 2), while performing a test on a Strace σ , the implementation may produce an incorrect output before the end of σ . With the previous tests and the associated verdict, such an output

leads to a success because the end of σ has not been reached. It seems then interesting to factorize all the tests that are a prefix of another (Figure 6)⁸. This reduces the number of tests in a significant way: The test set we get contains only factorized tests whose length is maximum (according to the regularity hypothesis). The verdict has to be modified to take into account erroneous outputs at any time during the test execution. One of the effect of such factorizations is that we have to pass each test a greater number of times to be sure that every sub-Strace has been tested. This way we obtain tests similar to those of Tretmans [19, 18]. The difference is that our tests always end by an output, and do not test quiescence twice.

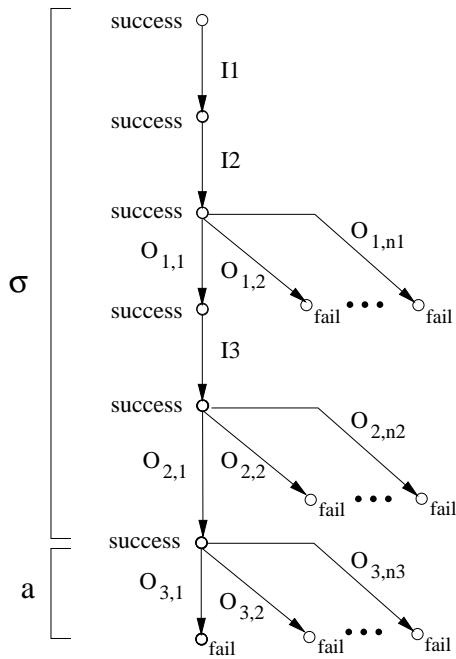


Figure 6. A factorized test

- I_i represents an input action
- $O_{i,j}$ represents an output action
- The output $O_{i,1}$ is some expected (for this test) correct output in the current state
- The outputs $O_{i,2}, \dots, O_{i,n_i}$ are the incorrect outputs in the current state

8 Conclusion

In this paper we have proposed a first approach towards a selection method for specifications of processes with inputs,

⁸The success label on Figure 6 corresponds to the notion given in Section 6.2.

outputs and data types. The considered implementation relation is *ioco* and we defined the set of tests $exhaustive'_{ioco}$, and the testing hypotheses in order to ensure validity and unbiased. Then the selection method consists of 4 steps:

- *Symbolic definition of $exhaustive'_{ioco}$* : During this step, one abstracts data types using symbolic states and actions. This leads to the definition of symbolic traces. Thus we get a set of symbolic tests which is an abstraction of the whole set $exhaustive'_{ioco}$.
- *Limitation of the length of these tests*: The next step is the limitation of the length of the symbolic traces of the tests. The minimum length, m , is determined using the parameters of the tested process: One has to cover every possible combination of the parameter's forms. The possible forms of parameters are deduced from the equations of data types definition. Thus acceptable regularity hypotheses are those where the length of the tests is bounded by some integer greater or equal to m . The set obtained after this step is an abstraction of a subset of $exhaustive'_{ioco}$, but it is often still infinite due to the data types.

Then, one may choose to instantiate the symbolic actions of each test with one set of values fitting the predicate that conditions the execution of this test. Doing so supposes a full uniformity hypothesis on the domain of the predicate associated to a test.

- *Selection of operation unfoldings in the predicates*: To weaken the previous uniformity hypotheses, one may unfold operations in the predicates of tests. To do so, for every applicable equation defining the concerned operation, one replaces the operation by its definition and one adds the condition of its application to the predicate. Unfolding is usually infinite too, so one has to limit the number of recursive unfoldings. This limitation defines our final uniformity hypotheses.
- *Instantiation of the symbolic actions*: Solving the predicate of the previous tests, one obtains a finite subset of $exhaustive'_{ioco}$ which coupled to the hypotheses determines a valid and unbiased testing strategy.

This method uses in an integrated way the behavior description of the process and the properties of the data types. It is worth to note that formal definition of the data type is not mandatory. Some algorithmic definition of the operations could be used as well as basis for unfolding.

The fact that the testability hypotheses are made explicit reduces in a significant way the exhaustive test set that serves as basis for the selection of some finite test set.

We have proposed a few guidelines for choosing the selection hypotheses. More elaborated ones could be based

on the analysis of the regularities in the underlying LTS. It is the subject of further work.

The current guidelines lead to test sets where nominal and boundary cases are covered. Adjusting the level of unfolding provides some flexibility with respect to the number of tests resulting from the selection. This makes it possible to modulate the choice of the selection hypotheses depending on the criticality of the system under test, or cost and delay considerations.

We plan to implement this method on the basis of some existing systems: The construction of the suspension automata of the TGV system [6] and the constraint solving package of Marre [14] developed for LUSTRE and reused recently for statistical structural testing [10].

References

- [1] E. Brinksma. A theory for the derivation of tests. Proceedings of Eighth International Conference of Protocol Specification, Testing and Verification, Atlantic City, North-Holland, 1988.
- [2] E. Brinskma and J. Tretmans. Testing transition systems: An annotated bibliography. In *LNCS*, volume 2067, pages 187–195, 2001.
- [3] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24, N°1:44–67, 1977.
- [4] F.T. Chan, H.Y. Chen, T.H. Tse, and T.Y. Chen. An integrated approach to class-level testing of object-oriented programs. In *ACM transactions on Software Engineering and Methodology*, 7(3):250-295,1998.
- [5] R.-K. Doong and P.G. Frankl. The ASTOOT Approach to Testing Object-Oriented Programs. In *ACM Transactions on Software Engineering and Methodologie*, pages 101–130, 1994.
- [6] 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*, 29 (1-2):123–146, 1997.
- [7] M.-C. Gaudel. Testing from formal specifications, a generic approach. In *LNCS*, volume 2043, pages 35–48. ADA-Europe Conference, 2001.
- [8] M.-C. Gaudel and P.R. James. Testing algebraic data types and processes: a unifying theory. In *Formal Aspects of Computing*, 10(5-6), pages 436–451, 1999.
- [9] A. Gotlieb, B. Botella, and M. Rueher. A CLP framework for computing structural test data. In *Constraints Stream*. 1st International Conference on Computational Logic CL 2000, Londres U.K., 2000.
- [10] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Automated Software Engineering Conference, IEEE*, pages 5–12, 2001.
- [11] ISO. LOTOS, a formal description technique based on the temporal ordering of observational behaviour. Technical Report 8807, 1989.
- [12] B. Littlewood, P.T. Popov, L. Strigini, and N. Shryane. Modeling the effects of combining diverse software fault detection techniques. In *IEEE Transactions on Software Engineering*, 26(12):1157-1167,2000.
- [13] N. Lynch and M. Tuttle. An introduction to input/output automata. Technical report, MIT/LCS/TM-373, 1988.
- [14] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: Gatel. In *Automated Software Engineering Conference, IEEE*, pages 229–237, 2000.
- [15] M. Phalippou. *Relations d’implantation et hypothèses de test sur des automates à entrées et sorties*. PhD thesis, Université de Bordeaux I, 1994.
- [16] A. Pretschner. Classical search strategies for test case generation with constraint logic programming. In *Proceedings of the workshop on Formal Approaches to Testing of Software (FATES’01)*, pages 47–60, 2001.
- [17] R. Segala. Quiescence, fairness, testing and the notion of implementation (extended abstract). In *LNCS*, volume 715. Proceedings of the 4th International Conference on Concurrency Theory (CONCUR ’93), Hildesheim, Germany, 1993.
- [18] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. In T. Margaria and B. Steffen, editors, *LNCS*, volume 1055, pp 127-146. 2nd Int. Workshop TACAS’96, 1996.
- [19] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *LNCS*, volume 1664, pages 46–65. CONCUR’99 - 10th Int. Conference on Concurrency Theory, 1999.