



405, avenue Ogilvy, bureau 101
Montréal (Québec) H3N 1M3
Tél. : 514 840-1234; Téléc. : 514 840-1244
Place de la Cité – Tour de la Cité
2600, boul. Laurier, bureau 625
Québec (Québec) G1V 2W1
Tél. : 418 648-8080; téléc. : 418 648-8141
www.crim.ca

CRIM - Documentation/Communications

Technical Report
A Transformational Approach for Asynchronous Test Case
Generation from Test Purposes

Final Version

CRIM-10/01-01

Adenilso Simao

On a leave from Instituto de Ciências Matemáticas e de Computação/USP

Alexandre Petrenko

Lead Researcher and Director, Distributed Systems Analysis

January 2010

Scientific and Technical Collection

ISBN-13 : 978-89522-130-2

TABLE OF CONTENTS

LIST OF FIGURES	3
ABSTRACT	4
1. INTRODUCTION	4
2. DEFINITIONS	6
3. GENERATING TEST CASES FROM TEST PURPOSES.....	10
4. EXAMPLE	17
5. SYNCHRONOUS VS ASYNCHRONOUS TEST CASES	20
6. PRELIMINARY EXPERIMENTAL RESULTS	22
7. CONCLUSION	24
8. REFERENCES	25

LIST OF FIGURES

<i>Figure 1 - Architecture for queued testing.</i>	8
<i>Figure 2 - A fragment of the unbounded queue.</i>	9
<i>Figure 3 - A fully specified input-progressive IOTS S.</i>	17
<i>Figure 4 - Test Purpose TP.</i>	17
<i>Figure 5 - $L_3 = B(TP)$.</i>	18
<i>Figure 6 - $L_3 = C(\text{delay}_\delta[I](B(TP)))$</i>	18
<i>Figure 7 - $L_4 = L_3 \parallel S_\delta$.</i>	18
<i>Figure 8 - Controllable q-sound test case TC.</i>	20
<i>Figure 9 - Synchronous test case which is not q-sound for S.</i>	21
<i>Figure 10 - Variation of the number of states explored w.r.t the number of states of test purpose.</i>	23
<i>Figure 11 - Variation of the number of states w.r.t. the number of output transitions in states with enabled inputs.</i>	24

ABSTRACT

In this paper, we investigate the problem of constructing a test case for a given test purpose and specification input/output transition system (IOTS). The communication between the tester and the implementation under test is assumed to be asynchronous, performed via queues. Differently from synchronous tests, when issuing verdicts, asynchronous tests should take into account the distortion caused by the queues in the observed interactions. We propose an algorithm which constructs a sound test case, by transforming the test purpose prior to composing it with the specification without queues, mitigating the state explosion problem which usually occurs when queues are directly involved in the composition. Experimental results confirm the resulting state space reduction. The proposed approach can also be used to obtain sound asynchronous tests from synchronous tests. We identify a class of IOTS specifications for which synchronous and asynchronous tests coincide.

1. INTRODUCTION

Generation of tests from state-based models, where next input can arrive even before outputs are produced in response to a previous input, relies on the input/output automaton model [10], also known as the input/output transition system (IOTS) model (the difference between them is marginal, at least from the testing perspective). Usually, it is assumed that the interaction between the tester and an implementation under test (IUT) is synchronous, implicitly assuming that the IUT can either refuse or accept inputs [2] [7] [12]. On the other hand, as the IUT can produce outputs at any moment, the tester should be prepared to accept all outputs from the IUT, or else be able to block (refuse) outputs of the implementation [2]. Moreover, the test context between the tester and the IUT cannot be always ignored. Testing distributed, remote applications under the assumptions that communication is synchronous and actions can be blocked is unrealistic. Synchronous communication for such applications can only be achieved if special protocols are used, see, e.g., [11] [18]. In this context, asynchronous tests, i.e., tests where the implementation's outputs are never blocked, can be more appropriate, reflecting the underlying test architecture which includes queues. Telecommunication systems, multi-thread multi-core programs and Web services are examples of applications requiring asynchronous tests.

Following [6] and [3], we consider that a test case is an elementary test designed to testing a particular functionality, which is represented as a test purpose. The test purpose specifies a finite set of traces which are related to a given functionality or some property. Designing correct asynchronous test cases is more complex than synchronous ones. The tester's observation of IUT's actions can be distorted by the test context; hence the tester should take into account any possible distortion when issuing verdicts. As a result, the

transformation of a synchronous test case into a proper asynchronous test case is usually a difficult task. An important property of test cases of either type is soundness. Informally, a sound test case only declares incorrect an implementation if it really is. Unsound test cases are in most situations useless, since their verdicts cannot be trusted.

In this paper, we investigate the problem of constructing an asynchronous test case for a given test purpose and a given specification IOTS which is sound when the tester and the IUT communicate via queues. The case when an IOTS implementation is tested via queues is investigated by Verhaard et al [16] and Huo and Petrenko [5]. The approach proposed in [16] relies on an explicit combined specification of a given IOTS and infinite queue context, so it is not clear how this approach could be implemented in practice. The method proposed in [5] generates tests to cover transitions of the specification, while avoiding the explicit composition with queues; however, test purposes are not considered in that work. The queued context is also considered by Jard et al. in [8], where a stamping mechanism is proposed to correct the queues' distortion, by ordering the outputs w.r.t. inputs, while quiescence is ignored. A stamping process has to communicate synchronously with the IUT as the ioco tester in [12]. Asynchronous testing based on ioco theory [12] is investigated by Lestiennes and Gaudel [9] and Weiglhofer and Wotawa [17]. In order to avoid blocking implementation's outputs, both approaches assume that the specification and implementations do not have states where both inputs and outputs are enabled, thus inputs are applied only when the system is quiescent: [9] calls such IOTS "IO-exclusive", while [17] "internal choice LTS". It is well known that any finite IOTS without input-output conflict can be folded into a Mealy FSM (we provide a formal definition later in the paper) and several existing test generation methods developed for FSMs become applicable. In this paper, we are interested in a more general setting, where both the specification and implementations are represented by a general type of IOTS which may not be converted into FSM model and the communication between the implementations and the tester are intermediated by queues, allowing for fully asynchronous communication.

The problem of (synchronous) test case generation for a test purpose is also investigated in [7]. The authors propose the on-the-fly synchronous composition of the test purpose and the specification to obtain an IOTS modelling a synchronous test case. Asynchronous test cases can be generated in a similar way; nonetheless, the distortion caused by queues should be taken into account. In [16] and [8], this is achieved by composing the specification with the queues. The test purpose can then be composed with the resulting composition. However, in this approach, which we call here a queue composing approach, the number of states to be explored to generate a test case increases rapidly as the test purpose grows.

The main contribution of this paper is a method for generating a sound asynchronous test case which avoids composition of the specification with queues. Instead of composing the specification with the queues, a transformation of the test purpose prior to composing it with the specification which reflects the queues' distortion is proposed (this is a transformational approach). The number of states in the resulting composition is much

smaller than that built by the queue composing approach, as confirmed by preliminary experimental results presented in this paper. We identify a class of IOTS specifications for which synchronous test cases are sound for queued testing, i.e., can be treated as asynchronous test cases. Synchronous, e.g., ioco, test cases unsound for queued testing can still be used as test purposes for the proposed method to produce asynchronous test cases.

The rest of the paper is organized as follows. In Section II, we introduce the required notions and concepts. In Section III, the transformational approach for generating an asynchronous test case from a test purpose is presented. An example is used to illustrate the method in Section IV. In Section V, we relate synchronous and asynchronous test cases. In Section VI, we present preliminary experimental results comparing the proposed transformational method with the queue composing approach. Finally, Section VII concludes the paper and points to future work.

2. DEFINITIONS

We use *input/output transition systems* (IOTS, a.k.a. input/output automata [10]) for modelling systems. Formally, an IOTS L is a quintuple $\langle S, I, O, \lambda, S_0 \rangle$, where S is a set of states; I and O are disjoint sets of input and output actions, respectively; $\lambda \subseteq S \times (I \cup O \cup \{\tau\}) \times S$ is the transition relation, with the symbol $\tau \notin (I \cup O)$ denoting internal actions; and S_0 is the set of initial states. We use $IOTS(I, O)$ to denote the set of IOTS with input set I and output set O .

For IOTS L , a *path* from state s_1 to state s_{n+1} is a sequence of transitions $p = (s_1, a_1, s_2)(s_2, a_2, s_3) \dots (s_n, a_n, s_{n+1})$, where $(s_i, a_i, s_{i+1}) \in \lambda$ for $i = 1, \dots, n$.

Let ε denote the empty sequence of actions. The *projection operator* $\downarrow_A$, which projects sequences of actions onto the set $A \subseteq I \cup O \cup \{\tau\}$, is recursively defined as $\varepsilon \downarrow_A = \varepsilon$, $(va) \downarrow_A = v \downarrow_A a$ if $a \in A$, and $(va) \downarrow_A = v \downarrow_A$ otherwise, where $v \in (I \cup O \cup \{\tau\})^*$ and $a \in I \cup O \cup \{\tau\}$.

A sequence $u \in (I \cup O)^*$ is called a *trace* of IOTS L from state $s_1 \in S$ if there exists path $(s_1, a_1, s_2)(s_2, a_2, s_3) \dots (s_n, a_n, s_{n+1})$, such that $u = (a_1 \dots a_n) \downarrow_{(I \cup O)}$. We use $traces(T)$ to denote the set of traces from states in $T \subseteq S$.

For trace $u = vw$, we say that v is a *prefix* of u , written $v \in \text{pref}(u)$; if w is not the empty sequence, v is a proper prefix of u . For trace set U , we define $\text{pref}(U)$ as the union of the prefixes of each trace $u \in U$.

In subsequent definitions, where sets of states are used as parameters, we often use IOTS L as shorthand for its initial state set S_0 , and state s for the singleton set $\{s\}$. Therefore, $traces(L) = traces(S_0)$ and $traces(s) = traces(\{s\})$. When $traces(L)$ is a finite set, we say that

L has *finite behaviour*. A *maximal* trace of L is a trace which is not a proper prefix of another trace of L . We denote by $mtraces(L)$ the set of maximal traces of L .

We use $init_L(T)$ to denote the set of actions enabled in states belonging to set $T \subseteq S$ of the IOTS L , i.e., $init_L(T) = \{a \in (I \cup O \cup \{\tau\}) \mid \exists t \in T \text{ and } \exists s \in S \text{ such that } (t, a, s) \in \lambda\}$. We define $out_L(T) = init_L(T) \cap O$ and $inp_L(T) = init_L(T) \cap I$ to refer to outputs and inputs enabled in the states of T , respectively. We will omit L in those notations whenever no ambiguity arises.

IOTS L is *input-enabled* if all input actions are enabled in each state, i.e., $inp(s) = I$ for all $s \in S$. IOTS L is *fully specified* if, for each state, either all input actions are enabled or no input action is enabled, i.e., either $inp(s) = I$ or $inp(s) = \emptyset$ for all $s \in S$.

We define a usual operator **after**. For state set $T \subseteq S$ and trace set U , T **after** U denotes the set of states that are reachable from states in T when traces in U are executed. Notice that T **after** $\{\varepsilon\}$ includes all states reachable by internal transitions as well as the states in T itself. An IOTS L is *deterministic* if it has a transition function λ , a single initial state, and no internal transitions. L is *output-nondeterministic* if there is a state s such that $|out_L(s \text{ after } \{\varepsilon\})| > 1$. State $s \in S$ is *stable* (quiescent) if no output or internal actions are enabled in s , i.e., $init(s) \cap (O \cup \{\tau\}) = \emptyset$; otherwise, it is *unstable*. We use $\delta \notin (I \cup O \cup \{\tau\})$ to indicate quiescence in an IOTS. Quiescence can be encoded in L by adding self-looping δ transitions to the stable states, and we denote the resulting IOTS as L_δ , which has the output action set $O \cup \{\delta\}$. A trace of L_δ is a *suspension* trace of L . A suspension trace of L whose last action is δ is a *quiescent* trace. We denote by $qtraces(L_\delta)$ the set of quiescent traces of L .

State $s \in S$ *deadlocks* if there is no action enabled in s , i.e., $init(s) = \emptyset$. State $s \in S$ *oscillates* if there is a path with only output or internal transitions from s to itself. IOTS L *deadlocks* or *oscillates* if there is a deadlock or oscillating state reachable from an initial state of L , respectively.

On the other hand, IOTS L is *input-progressive* if it neither deadlocks nor oscillates. An input-progressive IOTS L must accept inputs and, after an input, must reach a stable state in less than $|S|$ output or internal transitions, where $|S|$ is number of states of L .

In this paper, we assume that specifications and implementations are fully specified input-progressive IOTS.

Composition of IOTS formalizes the interaction the tester and the IUT. Formally, for IOTS $L_1 = \langle S_1, I_1, O_1, \lambda_1, S_{10} \rangle$ and $L_2 = \langle S_2, I_2, O_2, \lambda_2, S_{20} \rangle$ such that $O_1 \cap O_2 = \emptyset$, the *parallel composition* $L_1 \parallel L_2$ is the IOTS $\langle S, (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, \lambda, S_{10} \times S_{20} \rangle$, where the set of states $S \subseteq S_1 \times S_2$ and the transition relation λ are the smallest sets obtained by applying the following inference rules:

- $S_{10} \times S_{20} \subseteq S$;
- for $s_1 t_1 \in S$,
 - if $a \in (I_1 \cup O_1) \cap (I_2 \cup O_2)$, $(s_1, a, s_2) \in \lambda_1$, and $(t_1, a, t_2) \in \lambda_2$, then $s_2 t_2 \in S$ and $(s_1 t_1, a, s_2 t_2) \in \lambda$;
 - if $a \in \{\tau\} \cup (I_1 \cup O_1) \setminus (I_2 \cup O_2)$ and $(s_1, a, s_2) \in \lambda_1$, then $s_2 t_1 \in S$ and $(s_1 t_1, a, s_2 t_1) \in \lambda$;
 - if $a \in \{\tau\} \cup (I_2 \cup O_2) \setminus (I_1 \cup O_1)$ and $(t_1, a, t_2) \in \lambda_2$, then $s_1 t_2 \in S$ and $(s_1 t_1, a, s_1 t_2) \in \lambda$.

The tester of L_δ should have input set $O \cup \{\delta\}$ and output set I . However, to simplify the discussion, we refer to inputs and outputs always taking the view of the implementation; thus, we say for instance that the tester sends an input to the implementation and receives outputs from it.

Assume a tester *Test* and an IUT *Imp*, both modeled by IOTSs, interact with each other through queues (Figure 1).

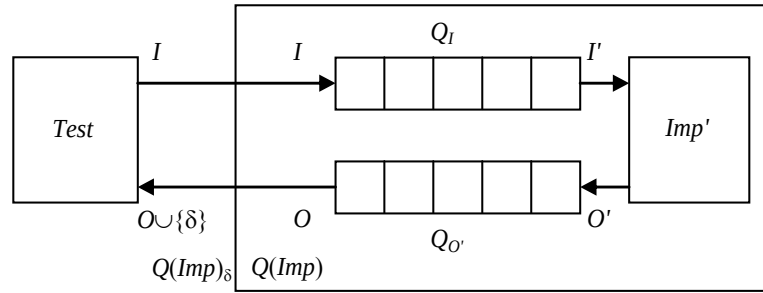


Figure 1 - Architecture for queued testing.

The action types of each component in Figure 1 are assigned in such a way that no action type is used at different interfaces, while the same action at the two ends of a queue are related by the relabeling operator ' [5]. In other words, if an action a is put in the queue, the action corresponding to removing it from the queue is a' . Formally, the operator ' is defined on actions: $(a)' = a'$, and $(a')' = a$. Thus, the relabeling operator applied to action which was already relabelled yields the original action. We lift the operator to sets of actions, traces, and IOTS: for action set A , $A' = \{a' \mid a \in A\}$; for traces, the operator ' is recursively defined as $\varepsilon' = \varepsilon$ and $(ua)' = u'a'$ for trace u and action a ; for IOTS $L \in IOTS(I, O)$, $L' \in IOTS(I', O')$ is derived from L by relabeling each action $a \in I \cup O$ to a' . Notice that $traces(L) = traces(L)'$.

For the queues, each input action a corresponds to an output action a' , and vice-versa. Formally, an *unbounded queue with input set A* , Q_A , is a deterministic IOTS $\langle S_{Q_A}, A, A', \lambda_{Q_A}, \{\varepsilon\} \rangle$, where the state set $S_{Q_A} = A^*$ and the transition relation $\lambda_{Q_A} = \{(u, a, ua) \mid u, ua \in S_{Q_A}\} \cup \{(av, a', v) \mid av, v \in S_{Q_A}\}$. As an example, Figure 2 shows a fragment of the unbounded queue $Q_{\{a, b\}}$ with the input actions a and b , where input actions are decorated with “?” and output actions with “!”.

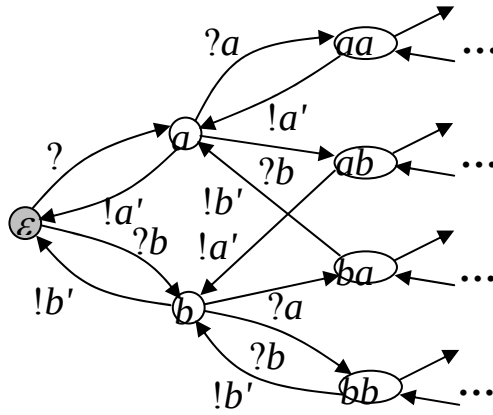


Figure 2 - A fragment of the unbounded queue.

For the closed system in Figure 1, the tester $Test \in IOTS(I, O \cup \{\delta\})$, the IUT $Imp' \in IOTS(I', O')$, input queue $Q_I \in IOTS(I, I')$, and output queue $Q_{O'} \in IOTS(O', O)$ (notice that $a'' = a$). In Figure 1, the system interacting with the tester is the composition of the IUT with the unbounded queues, which is defined by $Q(Imp) = \text{hide}[I' \cup O'](Q_I \parallel Imp' \parallel Q_{O'})$, where operator $\text{hide}[A]$ replaces transitions of actions in A with transitions of internal action τ . $Q(\)$ is similar to the queue operator in [15].

From the tester’s point of view, the behaviour of the IUT is $Q(Imp)_\delta$, and the specification of the system is $Q(Spec)_\delta$, where $Spec$ is the specification of Imp . Notice that the IOTS $Q(Spec)_\delta$ is input-enabled even if $Spec$ is not.

We find it convenient to use a generalized delay operator (see [1] for the definition of the delay operator) to describe the relation between $Q(L)_\delta$ and L_δ . The output actions of the IUT can be stored in the queue and thus delayed from the viewpoint of the tester, which chooses to send inputs rather than to read outputs. Similarly, the output actions of the tester (i.e., the input actions of the IUT) can be delayed from the viewpoint of the IUT. On the other hand, although the quiescent action δ can usually be treated as an output, it cannot be delayed. Thus, δ actions serve as boundaries within which the delay operator applies. We use the delay operator of [4] which takes into account the effect of δ actions.

Definition 1. For action set A , subset $B \subseteq A \setminus \{\delta\}$, and a set $U \subseteq A^*$ of action sequences, $\text{delay}_\delta[B](U)$ is the smallest set of action sequences obtained by applying the following inference rules:

- if $u \in U$, then $u \in \text{delay}_\delta[B](U)$; and
- if $u_1 a_1 a u_2 \in \text{delay}_\delta[B](U)$, then $u_1 a a_1 u_2 \in \text{delay}_\delta[B](U)$, where $u_1, u_2 \in A^*$, $a \in A \setminus (B \cup \{\delta\})$, and $a_1 \in B$.

According to the definition, $\text{delay}_\delta[B](U)$ derives a superset of U by shifting actions in B towards the end of each sequence in U while keeping the relative orders of actions in B and $A \setminus B$, respectively. Accordingly, the actions in $A \setminus B$ are pushed towards the beginning of the sequence. Actions in B cannot swap places with δ actions. Notice that, for each $\alpha \in \text{delay}_\delta[B](\{\beta\})$, it holds that $\beta \in \text{delay}_\delta[A \setminus B](\{\alpha\})$. Given a deterministic IOTS L with finite behaviour, we use $\text{delay}_\delta[B](L)$ to denote an IOTS whose trace set is $\text{delay}_\delta[B](\text{traces}(L))$. We can now relate the traces of L_δ and $Q(L_\delta)$.

Lemma 1. For any fully specified input-progressive IOTS L , it holds that $q\text{traces}(Q(L)_\delta) = \text{delay}_\delta[O](q\text{traces}(L_\delta))$.

Proof. First, if quiescent trace u is executed by L_δ , then $Q(L)_\delta$, when both queues are empty, can execute any quiescent trace in $\text{delay}_\delta[O](\{u\})$, i.e., $\text{delay}_\delta[O](\{u\}) \subseteq q\text{traces}(Q(L)_\delta)$ for each $u \in q\text{traces}(L_\delta)$. Thus, $\text{delay}_\delta[O](q\text{traces}(L_\delta)) \subseteq q\text{traces}(Q(L)_\delta)$.

Second, if trace u is executed by $Q(L)_\delta$ and both queues of $Q(L)_\delta$ are empty, then L_δ must have executed a trace in $\text{delay}_\delta[I](\{u\})$. Thus, for each $u \in q\text{traces}(Q(L)_\delta)$, there exists a trace $\alpha \in q\text{traces}(L_\delta)$, such that $\alpha \in \text{delay}_\delta[I](\{u\})$. Thus, we have that $u \in \text{delay}_\delta[O](\{\alpha\})$, and $u \in \text{delay}_\delta[O](q\text{traces}(L_\delta))$, for each $u \in q\text{traces}(Q(L)_\delta)$. Consequently, $q\text{traces}(Q(L)_\delta) \subseteq \text{delay}_\delta[O](q\text{traces}(L_\delta))$. ♦

3. GENERATING TEST CASES FROM TEST PURPOSES

Following [6] and [3], we consider that a test case is an elementary test designed to testing a particular functionality, which is represented as a test purpose. Usually, the test purpose consists of a set of traces which are somehow related to the given functionality the tester designer is interested in. As only finite tests are feasible, the set of traces is required to be finite. It is also desirable that unnecessary nondeterminism is avoided. In particular, any branching of a trace in the test purpose into several traces is caused only by outputs.

Definition 2. A test purpose is an IOTS $TP \in \text{IOTS}(I, O \cup \{\delta\})$ such that:

- TP is deterministic and has finite behaviour

- If $inp(t) \neq \emptyset$, then $|init(t)| = 1$, for all states t of TP .

A test purpose can be thought of as a script to be followed while testing the functionality. The result of a test case, the test verdict, is usually not specified in the test purpose. The test verdict should reflect how the interaction with the IUT occurred. If the observed output (from the IUT) is not allowed by the specification, the verdict should be **fail**, indicating that the IUT is incorrect. Otherwise, when no incorrect output is observed, two cases can be considered depending on whether the tester executing a test case was able to complete the interactions foreseen in the test purpose. If the IUT produces an output which is valid, i.e., allowed by the specification, but is not in the test purpose, the tester should issue the inclusive verdict **inc**, indicating that, although no misbehaviour from the IUT was observed, the objective of the test was not fulfilled. The tester issues the verdict **pass** if its interaction with the IUT occurs as the test purpose prescribes (and no incorrect output is observed). We formally define a test case as an IOTS. We require that the test case possesses the following properties. First, the tester should be deterministic and have at most one input action enabled in any state and when the tester expects one output from the implementation, it should be prepared to receive any possible output. Second, the tester should always reach a verdict in finite steps, and once a verdict is reached, it cannot be changed. These properties are expressed in the following definition.

Definition 3. Given an input set I and an output set O , an IOTS $TC = \langle T \cup \{\mathbf{fail}, \mathbf{pass}, \mathbf{inc}\}, I, O \cup \{\delta\}, \lambda_{TC}, t_0 \rangle$ is a test case, if:

- TC is deterministic and has finite behaviour.
- For each state $t \in T$, $|inp(t)| \leq 1$ and either $out(t) = \emptyset$ or $O \subseteq init(t)$.
- $\alpha \in mtraces(TC)$, if and only if $(TC \text{ after } \alpha) \subseteq \{\mathbf{fail}, \mathbf{pass}, \mathbf{inc}\}$.

We denote by $f\text{-traces}(TC)$ the traces of TC which lead to state **fail**, i.e., $f\text{-traces}(TC) = \{\alpha \in traces(TC) \mid TC \text{ after } \alpha = \{\mathbf{fail}\}\}$. Similarly, $p\text{-traces}(TC)$ and $i\text{-traces}(TC)$ are the traces of TC which lead to states **pass** and **inc**, respectively.

As a test case is deterministic, any of its traces leads to a single state. An input enabled in a given state indicates that the tester should send this input to the implementation. On the other hand, an enabled output indicates the tester should be ready to receive such an output from the implementation. If both inputs and outputs are enabled in a given state, the tester has to choose between either sending input or waiting for output. Such a situation is often undesirable, since it introduces unnecessary nondeterminism in the test execution. Test cases which avoid this situation are controllable. Formally, a test case TC is *controllable* if for each state $t \in T \setminus \{\mathbf{fail}, \mathbf{pass}, \mathbf{inc}\}$, either $init(t) = O \cup \{\delta\}$ or $init(t) = \{x\}$, for some $x \in I$.

An execution of TC corresponds to a maximal trace α of the composition $TC \parallel Q(Imp)_\delta$ which leads the test case to one of the verdict states. The verdict of the execution is given by the verdict state in $TC \text{ after } \alpha$. Even though the test case is deterministic, the

composition can be output-nondeterministic. Thus, the test case can have several executions leading to different verdicts.

An important property of test cases is *soundness*. Informally, a sound test case only issues verdict **fail** if the implementation is incorrect. Unsound test cases are usually useless, since their verdicts cannot be trusted. While the soundness of test cases is often defined for a particular conformance relation [13], in this paper, we adopt the definition based on the following observation. Suppose that implementation is a “copy” of the specification IOTS, differing from the specification just in state names. If a sound test case is executed with this implementation, the latter cannot fail.

Then the traces of the test purpose which are suspension traces of the specification should be present in a sound test case. It is also important that the tester does not issue a verdict prematurely, i.e., without having observed all the relevant outputs. If an incorrect output is observed, then the verdict **fail** can be issued immediately. On the other hand, **pass** and **inc** verdicts should only be issued when the specification with its queues is quiescent, thus all outputs are consumed by the tester from the queue. Notice that $Q(S)_\delta$ is quiescent, when both queues are empty and the specification S is in a quiescent state, since S is input-progressive. These properties are stated in the following definition.

Definition 4. A test case TC is *sound for queued testing* (or *q-sound*) w.r.t. a specification S and a test purpose TP , if

- $traces(TP) \cap traces(Q(S)_\delta) \subseteq traces(TC)$.
- $f\text{-}traces(TC) \cap traces(Q(S)_\delta) = \emptyset$.
- $i\text{-}traces(TC) \cup p\text{-}traces(TC) \subseteq qtraces(Q(S)_\delta)$.
- $mtraces(TP) \cap \text{pref}(i\text{-}traces(TC)) = \emptyset$.

The problem investigated in this paper is formulated as follows. Given a specification S and a test purpose TP , we want to generate a test case which is q-sound w.r.t. S and TP . As discussed in Section I, to solve this problem the queue composing approach uses the composition of the test purpose and the specification with queues. Indeed, if the test purpose is composed with $Q(S)_\delta$, it is possible to determine all the traces which lead the test case into **pass** and **inc** verdicts. However, $Q(S)_\delta$ has infinite number of states, since the queues are not bounded. Thus, a direct composition would not be feasible. Even though the states of $Q(S)_\delta$ can be computed on-the-fly for a given test purpose, the size of the composed IOTS may quickly become prohibitive.

We now present a method for generating a q-sound test case which avoids composition of the specification with queues. The idea is to find a way to transform the test purpose prior to composing it with the specification such that the queues’ distortion is taken into account

in the resulting composition without queues. We present first an informal overview of the method and then its formal version.

Assume for simplicity that a test purpose is a linear IOTS TP with a single sequence of input actions of the specification. The resulting test case should contain this sequence followed by all possible output sequences the specification can produce in response to it. All these sequences must lead the test case into **pass** state via the final quiescence action δ indicating the emptiness of the output queue. To determine them, one can compose the IOTSs TP and S without any queue. To take care of the specification which might not be input-enabled, in the composition, the test purpose should allow the specification to execute outputs needed to transfer between states with enabled inputs via unstable states with no input enabled. This can be realized by completing the IOTS TP with missing outputs, in this case, with all the outputs, which label self-looping transitions in each state. Thus, outputs may appear between inputs in the traces of the composition. These intermediate outputs are moved towards the end of the corresponding trace, simulating the effect of queues. Once all output sequences leading to **pass** state are determined, the test case has to be completed with **fail** state, by adding a transition to **fail** state from each state where at least one output or δ transition exists and for each output disabled in the state. The **inc** verdict is not used in this case, since a test purpose has no output actions.

It remains then to consider a test purpose with output actions. A test purpose with inputs and outputs foresees that an input action has to be executed only after output actions caused by other preceding inputs. In fact, this is the most general case. Due to the presence of queues, the outputs can actually be produced earlier than they are used in the test case and stored in the output queue. To account for this scenario, in the composition, the test purpose should allow the specification to execute the input action if the required outputs have already occurred in the composition and prohibit its execution if outputs unexpected by the test purpose occur. This can be achieved by adding to the test purpose such outputs terminating the executions and subsequently traces obtained by delaying inputs in the traces of TP with respect to the outputs, using the delay operator. The obtained IOTS is further completed with missing outputs, which label self-looping transitions in each state and composed with the specification, as explained earlier. The resulting composition contains all the possible executions driven by the given test purpose TP . The **inc** and **pass** traces of a test case can be determined as all possible output extensions of the traces of the test purpose TP which the composition contains. This is accomplished by delaying outputs (using again the delay operator) which are in the test purpose, but are preceded by inputs in the maximal traces of the composition. If a trace determined in this way corresponds to a maximal trace of the test purpose, a **pass** state should be reached. Otherwise, the test case should include the **inc** state reachable by those output sequences indicating that its execution deviates from what the test purpose requires. Once the test case is completed with **fail** state, its construction terminates.

We now formalize several operators transforming a test purpose and then finally the method itself. Given a deterministic IOTS $L = \langle T, I, O \cup \{\delta\}, \lambda_L, t_0 \rangle$, we denote by $B(L)$

the IOTS whose trace set is $traces(B(L)) = traces(L) \cup \{\alpha x \mid \alpha \in traces(L), init_L(L \text{ after } \alpha) \cap (O \cup \{\delta\}) \neq \emptyset, x \in (O \cup \{\delta\}) \setminus out_L(L \text{ after } \alpha)\}$. In each state of the IOTS $B(L)$, either all outputs and the quiescence are enabled or none of them is. Thus, the traces of $B(L)$ which are not traces of L end with an output which deviates from what is required by the test purpose. Let also $C(L)$ be the IOTS obtained by adding self-looping transitions for every output which is disabled in a given state, i.e., $C(L) = \langle T, I, O \cup \{\delta\}, \lambda, t_0 \rangle$, where $\lambda = \lambda_L \cup \{(t, x, t) \mid t \in T \text{ and } x \in O \setminus out_L(t)\}$. Thus, $C(delay_\delta[I](B(TP)))$ is the IOTS which will be used in the composition with the specification. Notice that for any maximal trace α of $B(TP)$ and any sequence $\gamma \in O^*$, all the sequences in $delay_\delta[I](\{\alpha\gamma\})$ are traces of the IOTS $C(delay_\delta[I](B(TP)))$. As the specification is input-progressive and the set of traces of TP is finite, the set of traces of the composition $C(delay_\delta[I](B(TP))) \parallel S_\delta$ is also finite.

A test case is an IOTS whose traces are obtained from the traces of the composition. The traces of the test purpose are used to determine when an output should be delayed in the trace of the composition to determine a required trace. Given $\alpha \in (I \cup O \cup \{\delta\})^*$, let $adj_{TP}(\alpha)$ be the longest trace $\beta\gamma$ in $pref(delay_\delta[O](\{\alpha\}))$, such that $\beta \in mtraces(B(TP))$ and $\gamma \in O^*$. For a set $A \subseteq (I \cup O \cup \{\delta\})^*$, let $adj_{TP}(A) = \{adj_{TP}(\alpha) \mid \alpha \in A\}$. The adjusted traces are then used to build an IOTS which has the **pass** and **inc** states. The test case is obtained by completing this IOTS with the state **fail** and transitions for the disabled outputs as follows. Given an IOTS $L = \langle S, I, O \cup \{\delta\}, \lambda_L, s_0 \rangle$, let $FailCompletion(L) = \langle S \cup \{\mathbf{fail}\}, I, O \cup \{\delta\}, \lambda, s_0 \rangle$, where $\lambda = \lambda_L \cup \{(s, x, \mathbf{fail}) \mid s \in S, out_L(s) \neq \emptyset \text{ and } x \in (O \cup \{\delta\}) \setminus out_L(s)\}$.

Algorithm for determining a test case

Input: a fully specified input-progressive $S \in IOTS(I, O)$, a test purpose $TP \in IOTS(I, O \cup \{\delta\})$

Output: A controllable q-sound test case TC w.r.t. S and TP .

$$L_1 = B(TP)$$

$$L_2 = delay_\delta[I](L_1)$$

$$L_3 = C(L_2)$$

$$L_4 = L_3 \parallel S_\delta$$

$$K = mtraces(L_4)$$

$$A = adj_{TP}(K)$$

Let $T = \langle pref(A) \cup \{\mathbf{pass}, \mathbf{inc}\}, I, O \cup \{\delta\}, \lambda, \{\varepsilon\} \rangle$ be an IOTS, where λ is defined as the smallest set such that

- $(\alpha, x, \alpha x) \in \lambda$, if $\alpha x \in \text{pref}(A)$ and $x \in (I \cup O \cup \{\delta\})$;
- $(\alpha, \delta, v) \in \lambda$, for each $\alpha \in A$, where $v = \mathbf{pass}$, if $\text{mtraces}(TP) \cap \text{pref}(\alpha\delta) \neq \emptyset$, and $v = \mathbf{inc}$ otherwise.

$TC = \text{FailCompletion}(T)$

Return TC .

Theorem 1. Let TC be a test case generated by the above Algorithm for a fully-specified input-progressive $S \in \text{IOTS}(I, O)$, and a test purpose $TP \in \text{IOTS}(I, O \cup \{\delta\})$. Then, TC is controllable and q-sound w.r.t. S and TP .

Proof. We first show that TC is controllable. Let T be the IOTS constructed in the algorithm from A . Notice that $\text{traces}(T) = \text{pref}(\{\alpha\delta \mid \alpha \in A\})$. For each trace $\phi \in \text{pref}(A)$, if there exists an input $x \in I$, such that $\phi x \in \text{pref}(A)$, then it follows that $\phi x \in \text{traces}(B(TP))$; thus, $\text{init}_T(t) = \{x\}$, where $t = (T \text{ after } \phi)$. Since $TC = \text{FailCompletion}(T)$, for each state t , if $\text{out}_T(t) \neq \emptyset$, it holds that $\text{init}_{TC}(t) = O \cup \{\delta\}$; and if $\text{out}_T(t) = \emptyset$ (i.e., if either $\text{init}_T(t) = \emptyset$ or $\text{init}_T(t) = \{x\}$, for some $x \in I$), it holds that $\text{init}_{TC}(t) = \text{init}_T(t)$. Thus, TC is controllable.

We next show that TC satisfies the Properties (a)-(d) in Definition 4. First, let $\alpha \in \text{traces}(TP) \cap \text{traces}(Q(S)_\delta)$. There exists $\beta \in \text{traces}(S_\delta)$, such that $\alpha \in \text{traces}(TP) \cap \text{delay}_\delta[O](\{\beta\})$. Thus, $\alpha \in \text{pref}(\text{adj}_{TP}(\beta))$. As $\beta \in K$ and $\text{adj}_{TP}(K) \subseteq \text{traces}(TC)$, it follows that $\alpha \in \text{traces}(TC)$ and, thus, Property (a) holds.

Second, suppose that $f\text{-traces}(TC) \cap \text{traces}(Q(S)_\delta) \neq \emptyset$. Thus, there exists $\alpha x \in f\text{-traces}(TC) \cap \text{traces}(Q(S)_\delta)$, where $\alpha \in \text{pref}(A)$ and $x \in O$. As $\alpha \in \text{pref}(A)$, there exists $\beta \in \text{mtraces}(B(TP))$ and $\mu \in O^*$, such that $\alpha = \beta\mu$. Since $\alpha x \in \text{traces}(Q(S)_\delta)$, there exists $\gamma \in O^*$, such that $\alpha x \gamma \delta \in \text{qtraces}(Q(S)_\delta)$. By Lemma 1, $\text{qtraces}(Q(S)_\delta) = \text{delay}_\delta[O](\text{qtraces}(S_\delta))$, therefore, $\alpha x \gamma \delta = \beta \mu x \gamma \delta \in \text{delay}_\delta[O](\{\chi\delta\})$. Hence, there exists $\chi \in \text{traces}(S_\delta)$, such that $\chi\delta \in \text{qtraces}(S_\delta)$ and $\beta \mu x \gamma \delta \in \text{delay}_\delta[O](\{\chi\delta\})$. Thus, $\chi\delta \in \text{delay}_\delta[I](\{\beta \mu x \gamma \delta\})$. As $\mu x \gamma \in O^*$ and $\beta \in \text{mtraces}(B(TP))$, it follows that $\chi \in \text{traces}(L_3)$ and, thus, $\chi \in \text{mtraces}(L_3 \parallel S_\delta) = K$. We have that $\beta \mu x \gamma = \alpha x \gamma = \text{adj}_{TP}(\chi) \in \text{adj}_{TP}(K) = A$. Thus, $\alpha x \notin f\text{-traces}(TC)$, a contradiction. It follows that $f\text{-traces}(TC) \cap \text{traces}(Q(S)_\delta) = \emptyset$ and, hence, Property (b) holds.

Third, for each trace $\chi\delta \in i\text{-traces}(TC) \cup p\text{-traces}(TC)$, we have that $\chi \in A$. There exist traces $\alpha \in K$, $\beta \in \text{mtraces}(B(TP))$ and a sequence $\gamma \in O^*$, such that $\beta\gamma = \chi = \text{adj}_{TP}(\alpha)$. As the trace α is a maximal trace of $L_3 \parallel S_\delta$ and L_3 contains self-looping transitions with output actions in all states which deadlock in $B(TP)$, the trace α leads to a stable state in S_δ . Thus, $\alpha\delta \in \text{qtraces}(S_\delta)$ and, consequently, it follows that $\beta\gamma\delta = \chi\delta \in \text{delay}_\delta[O](\text{qtraces}(S_\delta))$. By Lemma 1, we have that $\text{delay}_\delta[O](\text{qtraces}(S_\delta)) = \text{qtraces}(Q(S)_\delta)$ and, hence, $\chi\delta \in \text{qtraces}(Q(S)_\delta)$. Thus, Property (c) holds.

Finally, for each $\alpha \in i\text{-traces}(TC)$, it follows that $mtraces(TP) \cap pref(\alpha) = \emptyset$. Then $mtraces(TP) \cap pref(i\text{-traces}(TC)) = \emptyset$ and thus Property (d) holds. ♦

We now compare the complexity of the proposed algorithm and the queue composing approach [16] [8]. With unbounded queues, the IOTS $Q(S_\delta)$ has an infinite number of states. However, only finitely many states are reachable when it is composed with the test purpose, since the latter has a finite number of traces. Let k be the maximal number of inputs which appear in a maximal trace of TP , i.e., $k = \max\{|\alpha \downarrow_I| \mid \alpha \in mtraces(TP)\}$. The number of states of a queue with the set of actions I and the capacity k is $\sum_{i=0}^k |I|^i = \frac{|I|^{k+1}-1}{|I|-1}$.

If $|I| > 1$, the number of states is $\mathbf{O}(|I|^k)$. The minimal capacity of the output queue depends not only on the test purpose, but also on the length of output sequences that the specification can produce before reaching a stable state. As S is non-oscillating, no more than l outputs can be in the output queue when the test purpose is composed with the specification, where $l \leq k \times (n - 1)$, where n is the number of states of the specification. The number of states of a queue with set of action O and capacity l is $\mathbf{O}(|O|^l)$. Thus, the composition of $Q(S_\delta)$ with the test purpose TP may have $\mathbf{O}(n|I|^k|O|^l t)$ states, where t is the number of states of the test purpose.

On the other hand, the proposed algorithm avoids the composition with the queues. Instead, the test purpose is transformed and composed with S_δ . As TP has t states, $L_1 = B(TP)$ has $\mathbf{O}(t+|O|t) = \mathbf{O}(|O|t)$ states, since at most one new state is added for each state of the test purpose and an output in O . The number of states in the IOTS $L_2 = \text{delay}_\delta[I](L_1)$ can be estimated as follows. To represent all traces in $\text{delay}_\delta[I](L_1)$, it is sufficient to start with $L_2 = L_1$ and include into L_2 at most as many as states as the number of possible swaps between input and output actions in a trace of L_2 . Suppose that $\alpha xy\beta$ is a trace of L_2 , where $x \in I$ and $y \in O$, but $\alpha yx\beta$ is not. Let also s_1, s_2 and s_3 be the states of L_2 , such that $(L_2 \text{ after } \alpha) = s_1$ and there exists transitions (s_1, x, s_2) and (s_2, y, s_3) in L_2 . To represent the trace $\alpha yx\beta$, it is sufficient to include into L_2 a new state s' and transitions (s_1, y, s') and (s', x, s_3) . The action x may be swapped again with another output in $out_{L_1}(s_3)$. However, x can be swapped at most once per transition of L_1 with an output action. As there are at most $|O|t$ output transitions in L_1 , L_2 has at most $|O|^2 t^2$ more states than L_1 . Thus, L_2 has $\mathbf{O}(|O|^2 t^2)$ states. The IOTS $L_3 = C(L_2)$ has as many states as L_2 . Finally, $L_4 = L_3 \parallel S_\delta$ has $\mathbf{O}(n|O|^2 t^2)$ states. Notice that the composition built in the proposed algorithm may have exponentially fewer states than that in the queue composing approach, as it explores $\mathbf{O}(n|I|^k|O|^l t)$ states. Our algorithm nevertheless requires the additional task of adjusting the maximal traces of the L_4 . However, the adjustment can be done in time linear on the length of the trace.

In Section VI, we report on experiments assessing a state space reduction which can be achieved with the proposed approach.

4. EXAMPLE

We now illustrate the proposed method. Consider the IOTS S in Figure 3 and the test purpose TP with the maximal trace $?a?a?b!1?b$ (Figure 4).

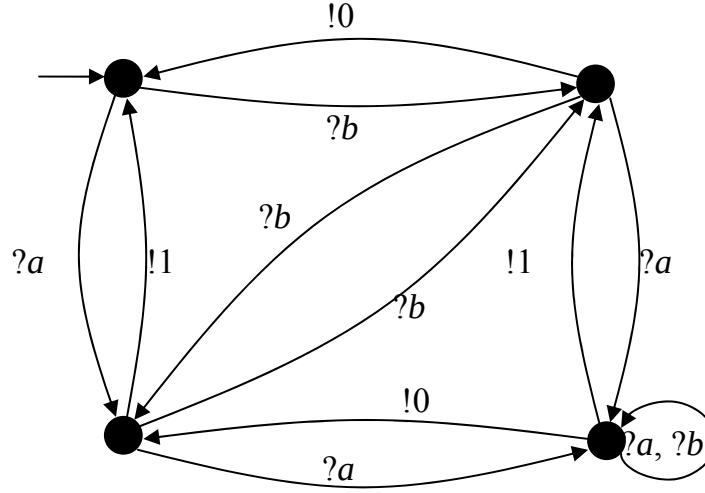


Figure 3 - A fully specified input-progressive IOTS S .

We have that $L_1 = B(TP)$ is a deterministic IOTS with the maximal traces $?a?a?b!1?b$, $?a?a?b!0$, and $?a?a?b\delta$ (Figure 5). Figures 6 and 7 show the IOTSs $L_3 = C(\text{delay}_\delta[I](L_1))$ and $L_4 = L_3 \parallel S_\delta$, respectively.

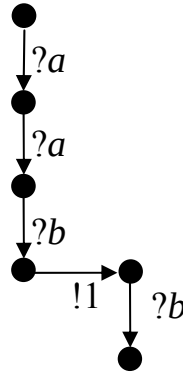


Figure 4 - Test Purpose TP .

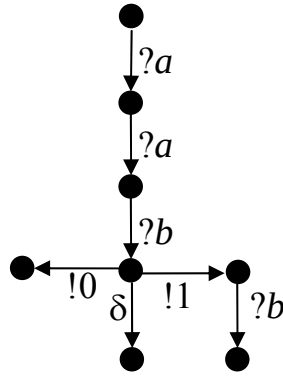


Figure 5 - $L_3 = B(TP)$.

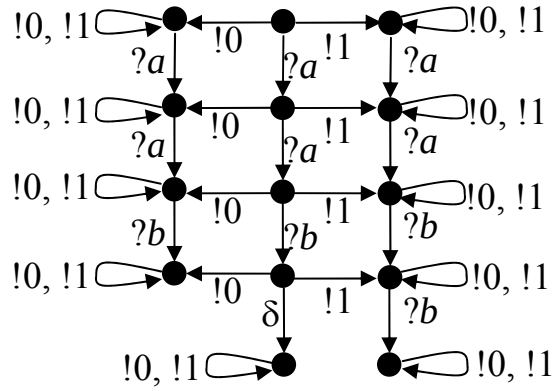


Figure 6 - $L_3 = C(\text{delay}_d[I])(B(TP))$

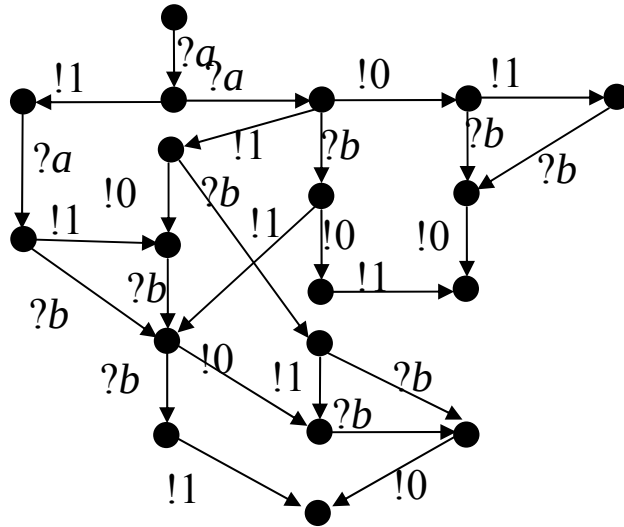


Figure 7 - $L_4 = L_3 \parallel S_\delta$

Each maximal trace of the IOTS L_4 is then adjusted to obtain the set A . For instance, outputs of the maximal trace $?a!1?a!1?b!0?b!0$ should be delayed according to the traces of the test purpose, obtaining the trace $?a?a?b!1?b!1!0!0$. The traces of L_4 and the respective adjusted traces are shown in Table 1. After adjusting all traces, the set A becomes $\{?a?a?b!0!0, ?a?a?b!0!1, ?a?a?b!0!1!0, ?a?a?b!1?b!0, ?a?a?b!1?b!0!0, ?a?a?b!1?b!0!0!0, ?a?a?b!1?b!0!1, ?a?a?b!1?b!1, ?a?a?b!1?b!1!0, ?a?a?b!1?b!1!0!0, ?a?a?b!1?b!1!1\}$.

$\alpha \in mtraces(L_4)$	$adj_{TP}(\alpha)$
$?a!1?a?b?b!1$	$?a?a?b!1?b!1$
$?a!1?a?b!0?b!0$	$?a?a?b!1?b!0!0$
$?a!1?a!1?b?b!1$	$?a?a?b!1?b!1!1$
$?a!1?a!1?b!0?b!0$	$?a?a?b!1?b!1!0!0$
$?a?a!1!0?b?b!1$	$?a?a?b!1?b!0!1$
$?a?a!1!0?b!0?b!0$	$?a?a?b!1?b!0!0!0$
$?a?a!1?b!1?b!0$	$?a?a?b!1?b!1!0$
$?a?a!1?b?b!0$	$?a?a?b!1?b!0$
$?a?a?b!1?b!1$	$?a?a?b!1?b!1$
$?a?a?b!1!0?b!0$	$?a?a?b!1?b!0!0$
$?a?a?b!0!1$	$?a?a?b!0!1$
$?a?a!0?b!0$	$?a?a?b!0!0$
$?a?a!0!1?b!0$	$?a?a?b!0!1!0$

Table 1. Maximal traces and respective adjusted traces.

The resulting test case is illustrated in Figure 8. For the sake of simplicity, we depict the test case as a tree and omit the transitions leading to the **fail** state.

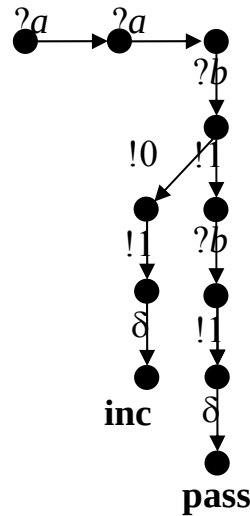


Figure 9 - Synchronous test case which is not q -sound for S .

On the other hand, the implementation of synchronous test cases requires that the tester and the IUT follow some protocol to deal with the situation when both the tester and the IUT have enabled actions. Otherwise, the tester should block outputs of the IUT when it is in a state with an enabled input. In [13], it is suggested that enabling outputs in all states of the test case solves this problem. Notice that the resulting test case is uncontrollable. Indeed, if in each state the test case has a transition for each output, from the theoretical viewpoint, the IUT outputs would not be blocked. However, for uncontrollable test case as in [13], the tester has to implement a protocol which allows detecting the first among two simultaneously offered by two systems for rendezvous actions [11] [18]. The tester for an uncontrollable test case then has some undesirable features. First, it has to repeatedly execute the test case (as opposed to queued testing with controllable test cases) to detect both possible winners of the race, even when the IUT is perfectly deterministic. Second, its behaviour is based on interleaved semantics and is not defined for the case when two independent actions execute simultaneously. The verdict of such execution is undefined. Third, the implementation of such a tester is more complex than that of a controllable one and requires some decisions which are not supported by the used untimed model. For instance, it is necessary to decide how long the tester should wait for outputs before deciding to send an input to the IUT. Even if a reasonable timeout can be identified, any output from the implementation should be blocked during the time passed between the moment the tester decides to send an input and the moment the input sending operation terminates.

Nonetheless, there is a situation when synchronous test cases generated by algorithms such as the one in [12] are q -sound. This is the case when the specification only allows inputs in stable states. Thus, in the resulting synchronous test case, any input will be sent only after

all the outputs which may result from the previous inputs are produced and observed by the tester.

Definition 5. An input-progressive IOTS S is a *Mealy* IOTS if inputs are enabled only in stable states, i.e., $inp(s) \neq \emptyset$ implies $out(s) = \emptyset$, for each state s of S .

By borrowing the name from the FSM world, we emphasize the fact that such IOTS behaves similarly to a Mealy machine; thus, several results from IOTS and FSM theories converge on this class of IOTS. Generating q-sound test cases for Mealy IOTS seems to be simpler than for non-Mealy, since queues cannot distort the order of executed actions. It turns out that when the specification is a Mealy IOTS and the traces of a test purpose are traces of the specification, a test case generated for synchronous test is also sound for asynchronous test.

Theorem 2. Let S be a Mealy IOTS, TP be a test purpose, such that $traces(TP) \subseteq traces(S_\delta)$, and TC be a synchronous test case for S and TP . Then, TC is also q-sound w.r.t. S and TP .

Albeit the problems discussed above, several methods for generation of synchronous test cases can be found in the literature, which are not restricted to Mealy IOTS. We refer the reader to a comprehensive review of methods based on **ioco** and its variations provided in [13]. The resulting test cases are usually not sound when queues are present, since incorrect **fail** verdict may be issued for a correct IUT. However, such test cases can be used as test purposes for the proposed algorithm. For instance, given a test case generated by the nondeterministic algorithm proposed in [12], a controllable and q-sound test case can be obtained by treating the **ioco** test case as a test purpose and using the proposed method. For the example in Figure 7, this leads to the test case as shown in Figure 6, except that the verdict **inc** is replaced by **pass**.

6. PRELIMINARY EXPERIMENTAL RESULTS

In this section, we present preliminary experimental results of the comparison of the proposed approach and the queue composing approach. In the queue composing approach, we compose on-the-fly the test purpose, the queues and the specification, as discussed in Section I. In the experiments, we used two specifications: the IOTS S in Figure 3 and the IOTS modelling the conference protocol [14], which has often been used to evaluate testing approaches for IOTS, e.g., [5] [14] [17]. The test purposes are randomly generated as follows. Initially, the test purpose is an IOTS whose trace set contains only the empty sequence. States and transitions are then iteratively added until a test purpose with the required number of states is obtained. In each iteration, a new state t' is added to the test purpose. Moreover, an existing state t which has no enabled input is randomly selected. Then, if there exist enabled output actions in the state t , an action x is randomly selected from $O \cup \{\delta\}$; otherwise, the action x is randomly selected from $I \cup O \cup \{\delta\}$. The

transition (t, x, t') is thus added to the test purpose. To compare the number of states used by the two approaches, in the case of the IOTS in Figure 3, we vary the number of states of a test purpose and keep the specification fixed; whereas in the case of the conference protocol, we use it as a starting point and vary the level of nondeterminism of the specification while keeping the number of states of a test purpose fixed. The results of both types of experiments indicate that the proposed approach provides a significant reduction of the explored state space.

In Figure 10, we plot the average number of states of the composed IOTSs obtained by the queue composing and the proposed approaches for the specification S in Figure 3. We note that the number of states of the composition used in the proposed approach increases much slower than that in the queue composing approach.

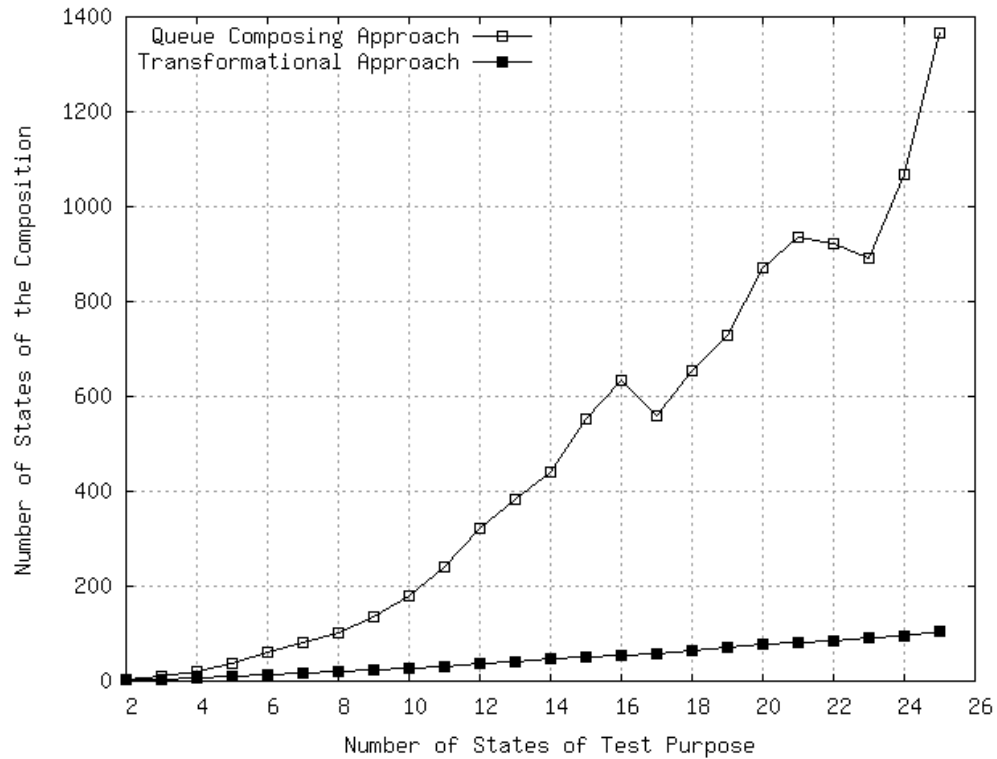


Figure 10 - Variation of the number of states explored w.r.t the number of states of test purpose.

As discussed in Section V, for Mealy IOTS specifications, synchronous test cases are q-sound. Thus, it is expected that generating test cases for Mealy IOTS is simpler than for non-Mealy, which is confirmed in the following experiment. A Mealy IOTS modelling the conference protocol [14] is used as the specification. The IOTS, which can be found, e.g., in [5], has 27 states, 11 input actions, 9 output actions and 80 transitions. We then randomly added output transitions to the stable states, such that a resulting IOTS is not

Mealy anymore. The more output transitions are added, the farther from being Mealy is the IOTS. We randomly generated test purposes with 15 states. In Figure 11, we show how the average number of states in the composition varies as the number of additional output transitions increases. The results confirm that for “quasi-Mealy” IOTS, the number of states in the composition used on the proposed method is close to that of the queue composing approach. At the same time, the proposed approach provides a significant reduction of the explored state space for specifications with many states possessing both inputs and outputs in unstable states.

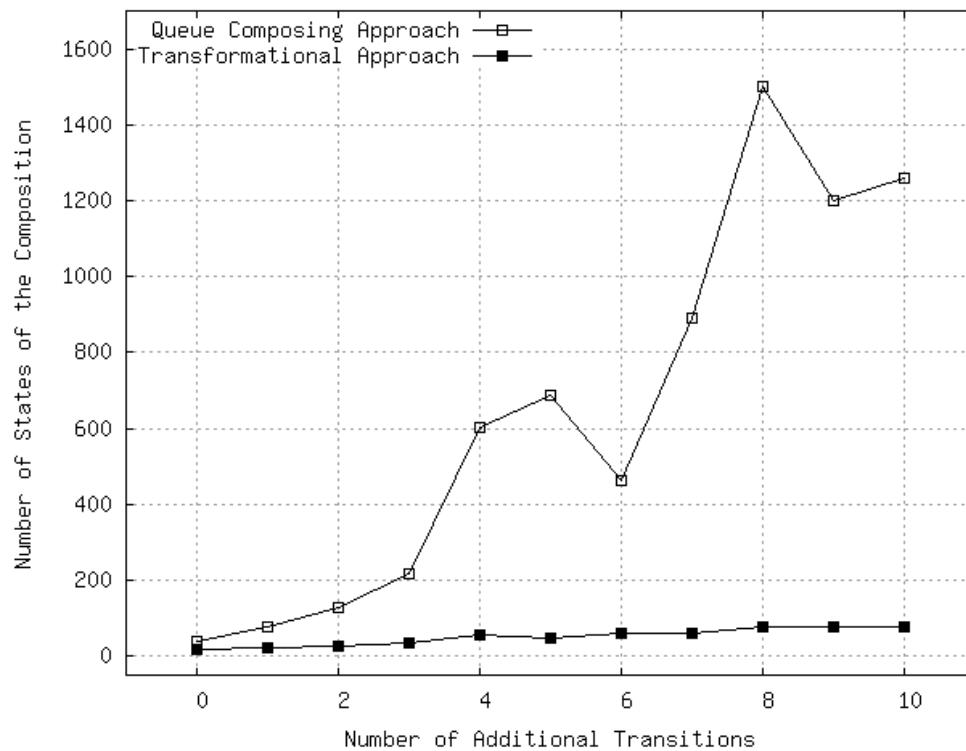


Figure 11 - Variation of the number of states w.r.t. the number of output transitions in states with enabled inputs.

7. CONCLUSION

In this paper, we investigated the problem of constructing an asynchronous test case for a given test purpose and a specification IOTS which is sound when the tester and the IUT communicate via queues, and elaborated an approach for solving it. The novelty of the approach lies in the idea of finding a transformation of the test purpose which reflects the distortion that queues creates, thus avoiding their composition with the specification which triggers the state explosion. Preliminary experimental results confirmed that the proposed

transformational approach requires a smaller state space compared to the existing queue composing approach. We identified a class of IOTS specifications for which synchronous and asynchronous test cases coincide. Specifically, we demonstrated that when the specification accepts inputs only in quiescent/stable states where it does not produce any output, behaving thus as a Mealy machine, a synchronous test case (e.g., generated for **ioco** conformance relation) is sound for queued testing. Moreover, the proposed method can still be used to obtain sound asynchronous test cases from synchronous ones, using the latter as test purposes. This is possible since the method accepts any type of test purposes, which may contain quiescence action and are not necessarily a part of the specification, differently from what is required in the previous work.

As future work, we can envisage the following possible extensions of the presented results. Firstly, we continue experimenting with the implementation of the proposed method to check its scalability for bigger specifications (since test purposes are usually crafted manually, they may still have a moderate size). Secondly, it is interesting to generalize the approach to the case of distributed testing, where the text context should include several queues.

8. REFERENCES

- [1] S. Balemi, “Input/output processes and communication delays”, Discrete Event Dynamic Systems: Theory and Applications, vol. 4, 1994, pp. 41–85.
- [2] I. B. Bourdonov, A. S. Kossatchev and V. V. Kuliainin, “Formal conformance testing of systems with refused inputs and forbidden actions”, Electronic Notes in Theoretical Computer Science, vol. 164, no. 4, 2006, pp. 83-96.
- [3] J.-C. Fernandez, C. Jard, T. Jéron and C. Viho, “Using on-the-fly verification techniques for the generation of test suites”, Proc. International Conference on Computer Aided Verification, 1996, pp. 348-359.
- [4] J. Huo and A. Petrenko, “On testing partially specified iots through lossless queues”, Proc. Testing of Communicating Systems, 2004, pp. 76-94.
- [5] J. Huo and A. Petrenko, “Transition covering tests for systems with queues”, Software Testing Verification and Reliability, vol. 19, 2009, pp. 55–83.
- [6] ITU (1996) ISO/IEC JTC1/SC21 WG7, “Information retrieval, transfer and management for OSI; Framework: Formal methods in conformance testing”, Committee Draft CD 13245-1, ITU-T proposed recommendation Z 500.
- [7] C. Jard and T. Jéron, “TGV: Theory, principles and algorithms”, Source International Journal on Software Tools for Technology Transfer, vol. 7, no. 4, 2005, pp. 297-315.

- [8] C. Jard, T. Jérón, L. Tanguy and C. Viho, “Remote testing can be as powerful as local testing, in Formal methods for protocol engineering and distributed systems”, Proc. Formal Description Techniques and Protocol Specification, Testing and Verification, Beijing, China, 1999, pp. 25-40.
- [9] G. Lestiennes et M-C. Gaudel, “Test de systèmes réactifs non réceptifs”, Journal Européen des Systèmes Automatisés, vol. 39. no. 1, 2005, pp. 255-270.
- [10] N. Lynch and M. R. Tuttle, “An introduction to input/output automata,” CWI Quarterly, vol. 2, no. 3, 1989, pp. 219-246.
- [11] A. Schiper, R. Simon, P. Desarzens and J.-A. Sengstag, “Efficient implementation of rendezvous”, The Computer Journal, vol. 32, no. 3, 1989, pp. 267-272.
- [12] J. Tretmans, “Test generation with inputs, outputs and repetitive quiescence”, Software Concepts and Tools, vol. 17, no. 3, 1996, pp. 103-120.
- [13] J. Tretmans, “Model based testing with labelled transition systems”, Formal Methods and Testing 2008, pp. 1-38.
- [14] J. Tretmans and E. Brinksma, “TorX: Automated model-based testing”, Proc. First European Conference on Model-Driven Software Engineering, 2003, pp. 31-43.
- [15] J. Tretmans and L. Verhaard, “A queue model relating synchronous and asynchronous communication”, Proc. International Symposium Protocol Specification, Testing and Verification, 1992, pp. 131–145.
- [16] L. Verhaard, J. Tretmans, P. Kim, and E. Brinksma, “On asynchronous testing”, Proc. Testing of Communicating Systems, 1992, pp. 55-66.
- [17] M. Weiglhofer and F. Wotawa, “Asynchronous Input-Output Conformance Testing”, 33rd Annual IEEE International Computer Software and Applications Conference, 2009.
- [18] L. Wischik and D. Wischik, “A reliable protocol for synchronous rendezvous”, Tech. Rep. UBLCS-2004-1, 2004.