550, rue Sherbrooke Ouest, bureau 100
Montréal (Québec) H3A 1B9
Téléphone : (514) 840-1234
Télécopieur : (514) 840-1244
http://www.crim.ca

CRIM - Documentation/Communications

Covering Transitions of Concurrent Systems through Queues

Version final

CRIM-05/08-09

Jiale Huo
Boursier
Alexandre Petrenko
Chercheur principal
Directeur de l'équipe
Analyse de systèmes distribués

31 août 2005

Collection scientifique et technique

ISBN 2-89522-067-0

Pour tout renseignement, communiquer avec:
CRIM Centre de documentation
CRIM
550, rue Sherbrooke Ouest, bureau 100
Montréal (Québec) H3A 1B9

Téléphone : (514) 840-1234
Télécopieur : (514) 840-1244

Bibliothèque nationale du Québec
Bibliothèque nationale du Canada
ISBN 2-89522-067-0

Abstract

By testing a concurrent system through queues, we seek a conclusive answer to whether a given transition of the specification of the system is executed. We define transition coverage through queues and find testing strategies to cover the transition in question with and without fairness, respectively.

1 Introduction

This paper studies the following problem: given a transition of the specification (model) of a concurrent system, design a strategy for a tester to steer an implementation under test (IUT) through a context of queues, so that the IUT is *guaranteed* to execute a transition corresponding to the aforementioned transition of the specification.

This problem is interesting to study for the following reasons. First, concurrent systems, such as distributed communicating systems or multithreaded programs, are notoriously difficult to test due to concurrency and non-determinism. Second, to buffer concurrent input/output actions, one usually inserts between a tester and IUT a pair of queues ([22], [11], and [8]), which add to the non-determinism. Third, transition coverage is an important and practical measurement in model-based testing, and even more so for complex systems such as concurrent ones.

Solving this problem can help improve the quality of test suites by increasing coverage of specifications.

The problem falls into a wider category of problems known as *specification-based testing*, a.k.a. model-based testing ([18], [12], [6], [13], [5], [1], [19], [8], and [17]).

Transition coverage was first used on specifications of sequential circuits ([18]), where the models are deterministic so that the correspondence between transitions of a specification and those of an IUT is straightforward.

Transition coverage for specification-based testing is later studied in testing of communication protocols ([12], [6], [13], [5], and [8]), where systems are often non-deterministic and tested through some contexts. As non-determinism makes it less evident the correspondence between transitions of specifications and those of IUT's, some works ([12] and [13]) assume white-box testing features such as observation of states, whereas others ([6] and [8]) define that a specification's transition is covered if it is one of many transitions that *may* be executed according to the behavior observed from the IUT. Moreover, to make the systems under test finite state, finite state contexts are usually considered ([12], [13], and [5]). We are aware of only [8] where infinite state contexts are considered.

Transition coverage for specifications is also related to test purposes ([6] and [10]) as covering a specific transition is a well-defined test purpose.

In software testing, however, transition coverage is usually related to *implementation-based testing* or *structural testing* ([20], [3, pp. 357-399], [14], and [7]), where coverage criteria are based on the structural features (statements, branches, paths, data flows, etc.) of a

program's code. Hence, structural testing covers implementations rather than specifications.

With the development of model-based testing, transition coverage for formal specifications is being studied in software testing ([1], [19], and [17]), and there are even commercial tools ([9]) that can automatically generate test cases for model coverage. General coverage criteria and techniques for test generation from state-based specifications are presented in [19]. Reference [1] presents a case study on covering the round-trip paths ([3]) of a container class's statechart. A method for covering transitions of a probabilistic non-deterministic FSM, without context, is introduced in [17]. It is noticed in [4], however, that there is a context of queues when multithreaded programs are tested.

Our work differs from previous ones in the following aspects.

First, we seek to obtain a conclusive answer from testing, namely, either a given transition of a non-deterministic specification *must* have been executed according to the trace observed from the IUT, which implies that the IUT has at least one transition that corresponds to ("implements") the specification's transition, or the IUT does not conform to the specification. In previous works, conclusive answers are only obtained for deterministic specifications ([18]), whereas for non-deterministic specifications, less strict criteria of transition coverage are used. For example, a transition is covered if it *may* have been executed ([6] and [8]) or if the probability that it is executed is close to 1 ([17]).

Second, we assume a context between a tester and IUT of unbounded queues, which have infinitely many states. Some previous works consider transition coverage through finite state contexts ([12], [13], and [5]) or no context at all ([17]). Unbounded queues can model storage with large capacity, such as computer memories, and unbounded communication delays.

Last, unlike [12] or [13], we do not assume that we can observe states of the IUT, but consider our problem under the assumption of black-box testing.

The main difficulty of obtaining the conclusive answer comes from non-determinism. An IUT (even a conforming one) acting as adversary can always make choices to avoid a tester's intension to make it execute certain transitions. As a result, the tester may not always succeed. For the tester, a winning strategy has to ensure that the transition is executed within finite steps no matter how the system maneuvers.

On the other hand, for a system that randomly picks its next moves, a transition can be covered under the so-called fairness assumption, which in our context means that a tester repeatedly applying a trace that covers the transition can eventually have it executed. In this case, we demonstrate that that is exactly what a winning strategy is supposed to do.

The rest of the paper is organized as follows. Preliminaries are given in Section 2. We present in Section 3 some results useful in later parts: a generalized delay operator and

some properties of traces under the operator. In Section 4, we study how to find winning strategies to cover given transitions. First, we define transition coverage and present a procedure to find traces covering transitions without context, to illustrate the basic ideas of our methods. Then, we define testing strategies and discuss how to obtain them. Last, we consider two cases of transition coverage through queues, one with the fairness assumption, the other without. Conclusions are given in Section 5, whereas proofs are provided in the appendix.

2 Preliminaries

2.1 Input/Output Transition Systems

For the model of systems, we use *input/output transition systems* (IOTS, a.k.a. input/output automata in [15]). Formally, an IOTS L is a quintuple $<S, I, O, \lambda, S_0>$, where S is a set of states; I and O are disjoint sets of input and output actions, respectively (i.e., $I \cap O = \emptyset$); $\lambda \subseteq S \times (I \cup O \cup \{\tau\}) \times S$ is the transition relation, with the symbol τ 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.

With transition relations (not functions), multiple initial states, and internal transitions, IOTS have many ways to express non-determinism. On the other hand, we call an IOTS *deterministic* if it has a transition function λ, a single initial state, and no internal transitions.

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

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 $(s_1, a_1, s_2)(s_2, a_2, s_3)\ldots(s_n, a_n, s_{n+1})$, such that $u = (a_1\ldots a_n)\downarrow_{(I\cup O)}$. We use $traces(T)$ to denote the set of traces from states in $T \subseteq S$.

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 $traces(s) = traces(\{s\})$.

For trace $u = vw$, we say that v is a *prefix* of u, written $v \in pref(u)$. Notice that u is its own prefix because $u = u\varepsilon$. For trace set U, we define $pref(U) = \{v \mid \exists u \in U \text{ s.t. } v \in pref(u)\}$.

We use $paths(T, u)$ to denote the set of paths from states in set $T \subseteq S$ with trace u, i.e., $paths(T, u) = \{(s_1, a_1, s_2)(s_2, a_2, s_3)\ldots (s_n, a_n, s_{n+1}) \mid s_1 \in T, (s_i, a_i, s_{i+1}) \in \lambda, \text{ and } (a_1a_2\ldots a_n)\downarrow_{I\cup O} = u\}$.

Traces represent the externally observable behavior of a system, whereas paths represent the internal functioning of the system.

We use $init(T)$ to denote the set of actions enabled in states belonging to set $T \subseteq S$, i.e., $init(T) = \{a \in (I \cup O \cup \{\tau\}) \mid \exists t \in T$ and $\exists s \in S$ s.t. $(t, a, s) \in \lambda\}$.

IOTS L is *fully specified* if either all input actions are enabled or no input action is enabled in each state, i.e., either $I \subseteq init(s)$ or $I \cap init(s) = \emptyset$ for all $s \in S$. In every state of a fully specified IOTS L, either L does not accept inputs at all if no input is enabled, or L's behavior after any input is defined. Since L's response to any input is predictable, we call it "fully specified".

An IOTS is *partially specified* if it is not fully specified.

State $s \in S$ is *stable* if no output or internal actions are enabled in s, i.e., $init(s) \cap (O \cup \{\tau\}) = \emptyset$. We use $S^{stable} \subseteq S$ to denote the set of all stable states.

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, i.e., there is $p \in paths(s, u)$ to s, where $u \in O^*$. 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 in stable states and, after an input, must reach a stable state in less than $|S|$ output or internal transitions, where $|S|$ is L's number of states.

In this paper, we consider only fully specified and input-progressive IOTS.

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**-ε includes all states reachable by internal transitions as well as the states in T itself.

Composition of IOTS formalizes the interaction between several systems. 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:

- if $s_1t_1 \in S_{10} \times S_{20}$, then $s_1t_1 \in S$;
- for $s_1t_1 \in S$,
  - 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 $s_2t_2 \in S$ and $(s_1t_1, a, s_2t_2) \in \lambda$;
  - 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 $s_2t_1 \in S$ and $(s_1t_1, a, s_2t_1) \in \lambda$;
  - 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 $s_1t_2 \in S$ and $(s_1t_1, a, s_1t_2) \in \lambda$.

2.2 Testing IOTS through Queues

Assume a tester *Test* and an IUT *Imp*, both modeled by IOTS, interact with each other through queues (Figure 1, the meaning of the symbols will be explained later in this section). The tester applies input stimuli to the IUT and observes its reaction. Therefore, output actions of the tester are input actions of the IUT, and vice versa.

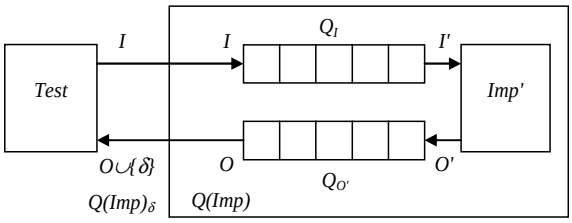

Figure 1 Architecture of queued testing

Besides outputs, absence of outputs, called *quiescence*, can also be a response of the IUT. Testers observing quiescence are used in so-called *quiescence testing* and *suspension testing* ([21] and [8]), where quiescence is treated as special output of the IUT, and observation of quiescence is made with a proper timer.

After [21], we use $\delta \notin I \cup O$ to indicate quiescence in an IOTS $L = <S, I, O, \lambda, S_0>$. 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 output set $O \cup \{\delta\}$. A tester of L_δ, therefore, should have input set $O \cup \{\delta\}$ and output set I.

The action types of each component in Figure 1 are assigned in such a way that no action types are used at different interfaces, while the same action at the two ends of a queue are related by operator $'$.

Formally, operator $'$ is defined on actions: $(a)' = a'$, and $(a')' = a$. We lift the operator to sets of actions, traces, and IOTS: for action set A, $A' = \{a' \mid a \in A\}$; for traces, $'$ 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'.

For the queues, each input action a (or a') corresponds to an output action a' (or a). Formally, an *unbounded queue with input set A*, Q_A, is a deterministic IOTS $<S_A, A, A', \lambda_A, \{\varepsilon\}>$, where the state set $S_A = A^*$ and the transition relation $\lambda_A = \{(u, a, ua) \mid u, ua \in S_A\} \cup \{(av, a', v) \mid av, v \in S_A\}$.

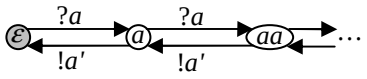

Figure 2. An unbounded queue

By definition, Q_A has infinitely many states, so it is "unbounded". As an example, Figure 2 shows an unbounded queue $Q_{\{a\}}$ with a single input action *a*, where input actions are decorated with "?" and output actions with "!".

In summary, for the closed system in Figure 1, the tester $Test \in IOTS(O \cup \{\delta\}, I)$, 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 queues, which is defined by $Q(Imp) = hide[I' \cup O'](Q_I \parallel Imp' \parallel Q_{O'})$, where operator $hide[A]$ replaces transitions of actions in A with transitions of internal action τ. $Q(\ )$ is similar to the queue operator in [22].

From the tester's point of view, the behavior of the system under test is $Q(Imp)_\delta$, and the specification of the system is $Q(Spec)_\delta$, where *Spec* is the specification of *Imp*.

3 Delay Operator and Properties of Traces

We find it convenient to use a generalized delay operator (see [2] for the definition of the delay operator) to describe the relation between $Q(L)_\delta$ and L_δ.

An IUT's output actions can be stored in the queue and thus *delayed* from the viewpoint of a tester, which chooses to send inputs rather than to read outputs. Similarly, the tester's output actions (i.e., the IUT's input actions) can be delayed from the viewpoint of the IUT.

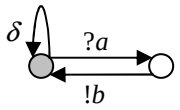

Figure 3. An IOTS with input *a* and output *b*

Example 1 Consider an IUT modeled by the IOTS shown in Figure 3. When the IUT is tested through queues, output actions in trace $?a!b?a!b$ can be delayed, so the tester may observe the trace $?a?a!b!b$. ◊

On the other hand, the quiescent action δ cannot be delayed, although it can be treated as a special output. In other words, δ actions serve as boundaries within which the delay operation applies.

Example 2 For the IUT in Figure 3, when trace $?a!b\delta?a!b\delta$ is executed, the tester cannot observe $?a!b?a\delta!b\delta$, because δ means absence of outputs and thus cannot be followed by an output. However, the tester can observe $?a?a!b!b\delta?a!b\delta$ if $?a!b?a!b\delta?a!b\delta$ is executed. ◊

Here, we generalize the delay operator in [2] to account for the effect of δ actions.

Definition 1 For action set A, subset $A_1 \subseteq A \setminus \{\delta\}$, and a set $U \subseteq A^*$ of action sequences, $delay_\delta[A_1](U)$ is the smallest set of action sequences obtained by applying the following inference rules:

- if $u \in U$, then $u \in delay_\delta[A_1](U)$; and
- if $u_1a_1au_2 \in delay_\delta[A_1](U)$, then $u_1aa_1u_2 \in delay_\delta[A_1](U)$, where $u_1, u_2 \in A^*$, $a \in A \setminus (A_1 \cup \{\delta\})$, and $a_1 \in A_1$.

According to the definition, $delay_\delta[A_1](U)$ derives a superset of U by shifting actions in A_1 towards the end of each sequence in U while keeping the relative orders of actions in A_1 and $A \setminus A_1$, respectively. Notice that actions in A_1 cannot swap places with δ actions.

As promised, traces of $Q(L)_\delta$ and L_δ can now be related with the help of the generalized delay operator.

First, if trace u is executed by L_δ and both queues of $Q(L)_\delta$ are empty, then $Q(L)_\delta$ can execute any trace in $delay_\delta[O](u\delta^*)$. According to Definition 1, we have $u \in delay_\delta[O](u\delta^*)$, so $traces(L_\delta) \subseteq traces(Q(L)_\delta)$.

Second, if trace u is executed by $Q(L)_\delta$ and the input queue of $Q(L)_\delta$ is empty, which we can always assume if L is input-progressive, then L_δ must have executed a trace in $delay_\delta[I](uO^*\delta^*) \cap traces(L_\delta)$.

Notice that $delay_\delta[I](uO^*\delta^*)$ is not regular (it is not even context-free), but $delay_\delta[I](uO^*\delta^*) \cap traces(L_\delta)$ is, because the number of actions left in the output queue is bounded when the input queue is empty. For input-progressive L, each input action can cause at most $|S| - 1$ output actions. Therefore, let $u = u_1\delta u_2\delta\ldots\delta u_n$, where $u_1, u_2, \ldots, u_n \in (I \cup O)^*$, u can cause at most $l(u) = |u_n{\downarrow_I}| \times (|S| - 1) - |u_n{\downarrow_O}|$ additional output actions in L_δ, where $|v|$ is the length of trace v. As a result, $delay_\delta[I](uO^*\delta^*) \cap traces(L_\delta) = delay_\delta[I](uO^{l(u)}\delta^*) \cap traces(L_\delta)$, where $O^{l(u)} \subseteq O^*$ has at most $l(u)$ actions in each element. Since $delay_\delta[I](uO^{l(u)}\delta^*)$ is regular, so is $delay_\delta[I](uO^*\delta^*) \cap traces(L_\delta)$ for input-progressive IOTS L.

With the delay operator, we can define some properties of traces, which will be useful in our search for winning strategies to cover transitions.

Trace $u \in traces(L_\delta)$ is *delay-insensitive* (DI) if $delay_\delta[I](uO^*\delta^*) \cap traces(L_\delta) = uO^*\delta^* \cap traces(L_\delta)$. A DI trace cannot be distorted by the delay operator. When we observe a DI trace from $Q(L)_\delta$, we know that the same trace is executed by L_δ. We use $DI\text{-}traces(L_\delta) \subseteq traces(L_\delta)$ to denote the set of DI traces of L_δ.

Delay-insensitivity as defined here is similar to that defined for macromodules (hardware) in [16].

Trace $u \in traces(L_\delta)$ is *input/output conflict-free* (IOCF) if $u = v_0a_1v_1a_2v_2\ldots a_nv_n$, where $v_i \in O^*\delta^*$ for $0 \le i \le n$, $a_j \in I$ for $0 < j \le n$, and L_δ-**after**-v_0, L_δ-**after**-$v_0a_1v_1\ldots a_kv_k \subseteq S^{stable}$ for $0 < k < n$. Input actions in an IOCF trace are only applied in stable states. We use $IOCF\text{-}traces(L_\delta) \subseteq traces(L_\delta)$ to denote the set of IOCF traces of L_δ.

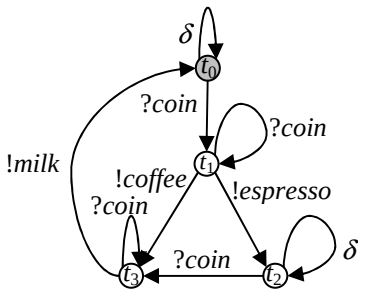

Figure 4 A coffee machine E_δ

We can verify that IOCF traces are DI because, under the delay operator, an input applied in a stable state cannot swap places with outputs or δ actions.

DI traces, however, do not have to be IOCF. In Figure 4, trace u = *coin coin espresso* $\delta \notin$ *IOCF-traces*(E_δ) because the second input *coin* is not applied in a stable state, but $u \in$ *DI-traces*(E_δ) because the input cannot swap places with δ.

4 Transition Coverage through Queues

Testing non-deterministic IOTS through queues ([8]), the tester only applies input stimuli, whereas the IUT decides whether to read the first input available in the queue or to produce an output and, in the later case, which output to produce. Moreover, it may end up in various states.

Due to the freedom of choices, we can conclusively declare that a given transition in the specification $Spec_\delta$ is covered if all paths of $Spec_\delta$ with the trace executed by $Q(Imp)_\delta$ traverses the transition.

Two problems have to be addressed by a tester trying to cover a specific transition. First, for reduction (preorder) conformance relations such as trace inclusion, the IUT is not required to implement every trace of the specification. Second, even if there is an implemented trace whose every path traverses the transition, the IUT can simply make moves to ensure that the trace itself is not executed.

Different solutions to these problems seem to be related to whether or not they assume *fairness*, which in our context means that an IUT eventually executes an implemented trace if its tester tries a sufficient number of times.

Assuming fairness, we can test equivalence conformance relations, such as trace equivalence. If a trace is not observed after a sufficient number of trials by the tester, we can conclude that *Imp* does not implement the trace, and thus does not conform to *Spec*.

It is also easy to find winning strategies to cover transitions under the fairness assumption. If there is a trace of $Q(Spec)_\delta$ that can only be caused by paths traversing the transition in question, the tester just have to repeatedly apply the trace, which a conforming IUT will execute eventually.

On the other hand, if we do not assume fairness, only reduction conformance relations can be tested. In this case, we need some adaptive strategies such that no matter how the IUT implements the specification or maneuvers, a tester can always force $Q(Imp)_\delta$ to execute a trace that can only be caused by paths traversing the given transition of $Spec_\delta$.

In light of this discussion, we consider in the following two cases of transition coverage, one with the fairness assumption and the other without.

Before that, we have to formally define transition coverage and strategies of testers.

4.1 Transition Coverage and Basic Algorithm

We first define transition coverage.

Definition 2 Consider IOTS $Spec_\delta$, where $Spec = <S, I, O, \lambda, S_0>$, and a given transition of $Spec_\delta$.

- A path *traverses* the transition if the transition is in the path.
- For trace $u \in traces(Spec_\delta)$,
  - u *covers* the transition if there exists path $p \in paths(Spec_\delta, u)$ that traverses the transition;
  - u *strongly covers* the transition if all paths in $paths(Spec_\delta, u)$ traverse the transition.
- For trace $v \in traces(Q(Spec)_\delta)$,
  - v *q-covers* the transition if there exists trace $u \in delay_\delta[I](vO^*\delta^*) \cap traces(Spec_\delta)$ that covers the transition;
  - v *strongly q-covers* the transition if all traces in $delay_\delta[I](vO^*\delta^*) \cap traces(Spec_\delta)$ strongly cover the transition.

Example 3 For the coffee machine E_δ in Figure 4, paths $(t_0, coin, t_1)(t_1, coffee, t_3)(t_3, milk, t_0)$ and $(t_0, coin, t_1)(t_1, espresso, t_2)(t_2, coin, t_3)(t_3, milk, t_0)$ traverse transition $(t_3, milk, t_0)$. As a result, traces *coin coffee milk* and *coin espresso coin milk* strongly cover the transition. Moreover, the reader can verify that these traces strongly q-cover the transition.

On the other hand, trace *coin coin espresso* strongly covers transition $(t_1, coin, t_1)$, but does not strongly q-cover it, because $delay_\delta[I](coin\ coin\ espresso\ \{coffee, espresso, milk\}^*\delta^*) \cap traces(E_\delta) = \{coin\ coin\ espresso, coin\ espresso\ coin\ milk\}\delta^*$. However, trace *coin coin espresso* δ strongly covers and strongly q-covers the transition. ◊

A trace of $Q(Spec)_\delta$ q-covers a transition if the transition is traversed by one of the paths causing the trace. If such a trace is executed by $Q(Imp)_\delta$, Imp_δ *may* have executed a transition corresponding to ("implementing") that of $Spec_\delta$.

Traces covering and q-covering transitions have been studied in [8], so here we concentrate on traces that strongly cover and strongly q-cover transitions.

A trace of $Q(Spec)_\delta$ strongly q-covers a transition if the transition is traversed by every paths causing the trace. Once such a trace is executed by $Q(Imp)_\delta$, Imp_δ *must* have executed a transition corresponding to that of $Spec_\delta$.

When $Q(Spec)_\delta$ executes strongly q-covering trace, all traces that $Spec_\delta$ can execute strongly cover the transition in question. We can find out whether a trace $u = a_1a_2\ldots a_n$ strongly covers a transition in the following way. We track the partition (S_i, T_i) of the state set $S_i \cup T_i$ reached after $a_1a_2\ldots a_i$, where S_i contains the states reached by paths that have not traversed the transition so far, and T_i contains the states reached by paths that have

already traversed the transition and not in S_i, i.e., $S_i \cap T_i = \emptyset$. Trace u strongly covers the transition if and only if $S_n = \emptyset$ and $T_n \neq \emptyset$, so that all paths that $Spec_\delta$ can take traverse the transition. The following procedure finds all traces *of* $Spec_\delta$ that strongly cover a given transition.

Procedure 1 To derive the set of traces that strongly cover a transition
Input: IOTS $Spec_\delta$, where $Spec = <S, I, O, \lambda, S_0>$, and transition (s_1, a, s_2) of $Spec_\delta$
Output: A set $U \subseteq traces(Spec_\delta)$ that contains all traces strongly covering (s_1, a, s_2)
Step 1: Let $X = <S^X, I, O \cup \{\delta\}, \lambda^X, S_0{}^X> = <\{(S_0, \emptyset)\}, I, O \cup \{\delta\}, \emptyset, \{(S_0, \emptyset)\}>$
Step 2: Do
Add (S_2, T_2) to S^X and $((S_1, T_1), b, (S_2, T_2))$ to λ^X, where
- $(S_1, T_1) \in S^X$, $b \in init(S_1 \cup T_1) \subseteq I \cup O \cup \{\delta\}$,
- $S_2 = \{s \mid$ there exists $p \in paths(S_1, b)$ to s that does not traverse $(s_1, a, s_2)\}$,
- $T_2 = \{s \mid$ any $p \in paths(S_1, b)$ to s traverses $(s_1, a, s_2)\} \cup \{s \mid$ there exists $p \in paths(T_1, b)$ to $s\} \setminus S_2$,

Until no more changes can be made to X.
Step 3: Return $U = \{u \in traces(X) \mid S_0{}^X$-**after**-$u = (\emptyset, T)$, where $T \neq \emptyset\}$.

If S, I, and O in Procedure 1 are finite sets, then the procedure is an algorithm because in this case X can be constructed in finite steps.

The computational complexity of Procedure 1 is $O(2^{2|S|})$. For deterministic IOTS $Spec_\delta$, the complexity is reduced to $O(|S|^2)$.

Procedure 1 contains the basic ideas of the methods that we will develop in this paper for finding winning strategies for transition coverage.

Procedure 1 cannot be generalized directly to find traces that strongly q-cover a transition, however. The difficulty is that the search space for strongly q-covering traces is $traces(Q(Spec)_\delta)$, a non-regular language even for finite-state $Spec$.

We will study how to find strongly q-covering traces in Section 4.4.

It is worth noticing that some transitions cannot be strongly q-covered although they can be strongly covered.

Example 4 Consider a two cell queue (the IOTS in Figure 2 with no outgoing input transitions defined in state *aa*). Transition (*a*, *a*, *aa*) is strongly covered by trace *aa*, but there is no trace to strongly q-cover the transition. Whenever the IOTS is in state *a* and the tester applies *a* through the input queue, the IOTS can always execute path $(a, a', \varepsilon)(\varepsilon, a, a)$, which has the same net effect as the IOTS executing path $(a, a, aa)(aa, a', a)$: an a' action is added to the output queue and the IOTS is in state *a*. ◊

The fact that even for deterministic IOTS as in Example 4 there are transitions that cannot be strongly q-covered indicates the impact of queues on the non-determinism of the system under test.

4.2 Testing Strategies

A tester needs some strategy to steer an IUT through queues to execute a trace that strongly q-covers a given transition in the specification.

Definition 3 For input set I and output set O, a *strategy* is a function f: $(I \cup O \cup \{\delta\})^* \rightarrow I \cup \{\varepsilon, stop\}$.
The *trace tree* of strategy f is the smallest set $TT(f)$ defined by the following inference rules:

- $\varepsilon \in TT(f)$;
- *if* $u \in TT(f)$ and $f(u) \in I$, then $uf(u) \in TT(f)$;
- *if* $u \in TT(f)$ and $f(u) = \varepsilon$, then $u(O \cup \{\delta\}) \subseteq TT(f)$.

In the definition of a strategy, $f(u) \in I$ means that a tester implementing the strategy applies input $f(u)$, $f(u) = \varepsilon$ means that the tester does not apply any input, but waits for an output, and $f(u) = stop$ indicates that the tester terminates.

The trace tree of a strategy is a set of traces that can be executed by a tester implementing the strategy.

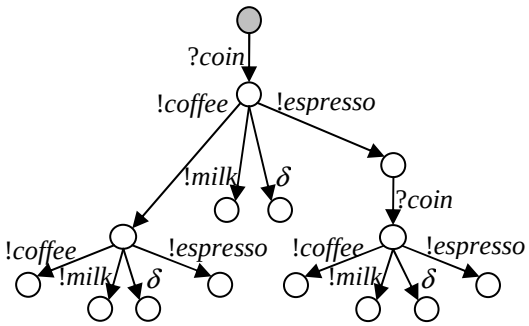

Figure 5 The trace tree of a strategy

Example 5 Figure 5 shows the trace trees of a strategy that can be implemented to test E_δ in Figure 4. ◊

On the other hand, trace trees of strategies are instances of a more general class of trace sets where traces fork on outputs.

Definition 4 For input set I and output set O, $U \subseteq (I \cup O \cup \{\delta\})^*$ is *output-branching* if u_1, $u_2 \in U$ implies $u_1 \in pref(u_2)$, or $u_2 \in pref(u_1)$, or $u_1 = va_1w_1$ and $u_2 = va_2w_2$, where v, w_1, $w_2 \in (I \cup O \cup \{\delta\})^*$, a_1, $a_2 \in (O \cup \{\delta\})$, and $a_1 \neq a_2$.

The trace tree of a strategy is an output-branching trace set, but the reverse is usually not true. A trace tree U is prefix closed, i.e., $U = pref(U)$, and "output-complete", i.e., $u(O \cup \{\delta\}) \subseteq U$ if $ub \in U$ for $u \in U$ and $b \in O \cup \{\delta\}$. An output-branching trace set does not necessarily have these properties.

It turns out that the more general trace sets can be used to define strategies. For an output-branching trace set U, we can uniquely define a strategy $f[U]$, where $f[U](u) = a \in I$ if $ua \in pref(U)$; $f[U](u) = \varepsilon$ if $u(O \cup \{\delta\}) \cap pref(U) \neq \emptyset$; and $f[U](u) = stop$, otherwise.

When we look for strategies, we usually find all traces according to a certain criteria and then prune the trace set into an output-branching trace set. We use a recursive function $prune(U)$ to prune a given finite trace set U into an output-branching subset of the shortest traces that are not prefixes of other traces.

Formally, for trace set U and action b, we define $height(U) = \mathbf{max}\{|u| \mid u \in U\}$, $U / b = \{v \mid bv \in U\}$, and $prune(U) =$

- $\{\varepsilon\}$, if $U = \{\varepsilon\}$; otherwise,
- $\cup_{b \in B} b \cdot prune(U / b)$, if $B = (O \cup \{\delta\}) \cap pref(U) \neq \emptyset$; otherwise,
- $b \cdot prune(U / b)$, where $b \in I$ s.t. $height(prune(U / b)) = \mathbf{min}\{height(prune(U / c)) \mid c \in I \cap pref(U)\}$.

By finding the trace set $prune(U)$ with the least height, we ensure that the strategy $f[prune(U)]$ is optimal in terms of test length.

4.3 Transition Coverage without Fairness

In the following, we try to find strategies that can steer $Q(Imp)_\delta$ to execute traces strongly q-covering a given transition of $Spec_\delta$. Depending on whether we make fairness assumption or not, different types of strategies can be used for the purpose of strongly q-covering the transition.

In this subsection, we do not assume fairness, and we only consider the following reduction conformance relation ([8]).

Definition 5 For $Spec$, $Imp \in IOTS(I, O)$, Imp is *queue-context suspension trace included into Spec* if $traces(Q(Imp)_\delta) \subseteq traces(Q(Spec)_\delta)$.

With the absence of fairness, we have to impose finiteness on strategies, i.e., *stop* must be reached within finite steps. Once *stop* is reached, we want the tester to witness a trace that either strongly q-covers the transition in question or does not belong to $traces(Q(Spec)_\delta)$, meaning that the IUT is non-conforming.

Definition 6 For IOTS $Spec_\delta$, where $Spec = <S, I, O, \lambda, S_0>$, strategy f is a *winning strategy over transition* (s_1, a, s_2) if there exists a number n such that, for any $u \in (I \cup O \cup \{\delta\})^*$, $|u| \geq n$ implies $f(u) = stop$, and $f(u) = stop$ implies either u strongly q-covers the transition or $u \notin traces(Q(Spec)_\delta)$.

Here, n is the bound of steps that a tester can execute to conclude that either the transition in question is strongly q-covered or the IUT is non-conforming.

Example 6 The strategy $f[\{coin\ coffee\ milk, coin\ espresso\ coin\ milk\}]$ (Figure 5) is a winning strategy over $(t_3, milk, t_0)$ of the coffee machine E_δ in Figure 4.

On the other hand, $f[\{coin\ coin\ espresso\ \delta\}]$ is not a winning strategy over transition $(t_1, coin, t_1)$ because, after the second *coin*, the tester may observe output sequences *espresso milk* δ, which means that the IUT produced an output instead of reading the second input *coin* in state t_1. ◊

The strategy shown in Figure 5 is of special interest. A tester implementing this strategy applies inputs to the IUT only when E_δ is supposed to be in stable states, so that only IOCF traces of the specification are executed. Proposition 1 states that such a winning strategy always exists if there is ever a winning strategy over a given transition.

Proposition 1 There is a winning strategy f_1 over a transition (s_1, a, s_2) of IOTS $Spec_\delta$ iff there is a winning strategy f_2 over the transition such that $TT(f_2) \cap traces(Q(Spec)_\delta) \subseteq IOCF\text{-}traces(Spec_\delta)$.

Intuitively, if a tester executes a trace of $Q(Spec)_\delta$, Imp_δ can postpone the consumption of input actions until the corresponding states in *Spec* are stable. If this is the case, traces executed by Imp_δ are IOCF traces of $Spec_\delta$. According to Definition 2, if a trace of $Q(Spec)_\delta$ strongly q-covers a transition, all corresponding traces possibly executed by $Spec_\delta$, including the IOCF ones, have to strongly cover the transition. Since IOCF traces are DI, they must be executed by $Spec_\delta$ if $Q(Spec)_\delta$ executes them. Thus, by Definition 2, if an IOCF trace strongly covers a transition, it also strongly q-covers it. Hence we can find a winning strategy by properly pruning the IOCF traces of $Spec_\delta$.

According to Proposition 1, we know that to derive a winning strategy over a transition, we only have to look into the IOCF traces of $Spec_\delta$ that strongly cover the transition. Therefore, we can generalize Procedure 1 to find such a winning strategy.

Procedure 2 To derive a winning strategy over a transition, if the strategy exists
Input: IOTS $Spec_\delta$, where $Spec = <S, I, O, \lambda, S_0>$, and transition (s_1, a, s_2) of $Spec_\delta$
Output: A winning strategy f over (s_1, a, s_2) or declare that f does not exist

Step 1: Let $X = < S^X, I, O \cup \{\delta\}, \lambda^X, S_0^X > = <\{(S_0, \emptyset)\}, I, O \cup \{\delta\}, \emptyset, \{(S_0, \emptyset)\}>$

Step 2: For $(S_1, T_1) \in S^X$, $(S_2, T_2) \in 2^S \times 2^S$, and $b \in init(S_1 \cup T_1)$ satisfying the following conditions:

- (S_1, T_1) is not marked as a leaf node;
- $b \in I$ only if $S_1 \cup T_1 \subseteq S^{stable}$;
- $S_2 = \{s \mid$ there is $p \in paths(S_1, b)$ to s that does not traverse $(s_1, a, s_2)\}$;
- $T_2 = \{s \mid$ any $p \in paths(S_1, b)$ to s traverses $(s_1, a, s_2)\} \cup \{s \mid$ there is $p \in paths(T_1, b)$ to $s\} \setminus S_2$;

Case 2.1: if (S_1, T_1) is not reachable from (S_2, T_2) in X, add (S_2, T_2) to S^X and $((S_1, T_1), b, (S_2, T_2))$ to λ^X, mark (S_2, T_2) as leaf node if $S_2 = \emptyset$;

Case 2.2: otherwise, if $b = \delta$ and $(S_1, T_1) \neq (S_2, T_2)$, or $b \in O$, mark (S_1, T_1) as a leaf node and delete all transitions from it;

until no more changes can be made to X.

Step 3: Do

- mark all states in S^X with no outgoing transitions as leaf nodes;
- delete the last input transition leading to a leaf node (S_1, T_1) where $S_1 \neq \emptyset$;

until no more changes can be made to X.

Step 4: If there is a leaf node (S_1, T_1) reachable from $(S_0, \emptyset)$, where $S_1 \neq \emptyset$, declare that f does not exist, otherwise,

Step 5: Return strategy $f[prune(traces(X))]$.

In Step 2, we exclude from further consideration transitions that introduce cycles in the construction of X because of the requirement that *stop* should be reached within finite steps. If such a transition is an output other than a self-looping δ transition (Case 2.2), we also have to cut off the last input transition before the output transition (in Step 3) because $Q(Imp)_\delta$ can choose to execute the output once the input is applied.

Like Procedure 1, Procedure 2 has complexity $O(2^{2|S|})$ for non-deterministic specifications and complexity $O(|S|^2)$ for deterministic ones.

The reader can verify that to strongly q-cover the transition $(t_3, coin, t_0)$ of E_δ in Figure 4, Procedure 2 returns the strategy shown in Figure 5.

4.4 Transition Coverage with Fairness

A winning strategy over a transition of $Spec_\delta$ forces $Q(Imp)_\delta$ to execute a trace that strongly q-covers the transition within finite steps, no matter how the IUT maneuvers. Such a strategy does not always exist for transitions that can nevertheless be strongly q-covered.

Example 7 There is no winning strategy over the input transition (t_1, *coin*, t_1) of E_δ in Figure 4. If we apply two *coin*s in a row, an IUT can always produce output *coffee* or *espresso* in state t_1 instead of accepting the second *coin*, so there is no guarantee that the IUT will execute a transition corresponding to (t_1, *coin*, t_1).

On the other hand, if we assume fairness, i.e., an implemented trace will be executed eventually if the tester tries a sufficient number of times, repeating a strategy with the trace u = *coin coin espresso* δ, i.e., $f[\{u\}]$, can eventually have the trace executed, which strongly q-covers the transition (t_1, *coin*, t_1). ◊

This example indicates that there are strategies, albeit not winning strategies, that can steer an $Q(Imp)_\delta$ to execute a trace that strongly q-covers the transition in question under the fairness assumption. A test implementing such a strategy has to be executed repeatedly, so that the trace can eventually be executed if it is implemented by the $Q(Imp)_\delta$. To ensure that this is the case, the IUT has to conform to the specification according to some equivalence relation.

Definition 7 For $Spec, Imp \in IOTS(I, O)$, *Imp* is *queue-context suspension trace equivalent to Spec* if $traces(Q(Imp)_\delta) = traces(Q(Spec)_\delta)$.

In this subsection, we assume fairness and consider only the queue-context suspension trace equivalence conformance relation.

For the winning strategy under fairness assumption, we still require finiteness, i.e., *stop* is reached within finite steps. Once *stop* is reached, there are three possible cases. First, a trace that strongly q-covers the transition in question is executed. In this case, we declare success and stop testing. Second, a trace that is not in $traces(Q(Spec)_\delta)$ is executed. In this case, we declare that the IUT *Imp* does not conform to *Spec* and stop testing. Third, a trace that is in $traces(Q(Spec)_\delta)$ but does not strongly q-cover the transition is executed. In this case, we reset *Imp* and restart the test. If the last case happens a sufficient (predefined) number of times, we declare that $Q(Imp)_\delta$ does not implement the trace strongly q-covering the transition, so that *Imp* does not conform to *Spec* under the queue-context suspension trace equivalence relation.

The formal definition of the winning strategy under fairness assumption is as follows.

Definition 8 For IOTS $Spec_\delta$, where $Spec = <S, I, O, \lambda, S_0>$, a strategy f is a *fair winning strategy over transition* (s_1, a, s_2), if there exists a number n such that, for any $u \in (I \cup O \cup \{\delta\})^*$, $|u| \geq n$ implies $f(u) = stop$, and there exists trace $v \in TT(f)$ that strongly q-covers the transition.

According to their definitions, a winning strategy is a fair winning strategy, but the reverse is usually not true.

A fair winning strategy only has to be finite and include a trace that strongly q-covers the transition in question. Therefore, for IOTS $Spec_\delta$, transition (s_1, a, s_2) of $Spec_\delta$ and a finite set $U \subseteq traces(Q(Spec)_\delta)$ of traces that strongly q-cover the transition, $f[prune(U)]$ is a fair winning strategy over the transition. In particular, if U is a singleton $\{u\}$, $f[prune(U)] = f[\{u\}]$.

Finding a fair winning strategy now reduces to finding a strongly q-covering trace.

However, it is unknown how to find such a trace as $traces(Q(Spec)_\delta)$ is usually not regular. The following proposition give us a better characterization of traces that strongly q-cover a given transition.

Proposition 2 For L_δ, there is a trace $u_1 \in traces(Q(L)_\delta)$ that strongly q-covers a given transition iff there is a trace $u_2 \in DI\text{-}traces(L_\delta)$ that strongly covers the transition and $|(u_2)\downarrow_\delta| \leq 2^{3|S|}$. Moreover, u_2 strongly q-covers the transition.

Proposition 2 tells us that, to find traces that strongly q-cover a transition, we only have to look into the DI traces of the IOTS itself, not the traces of the composition of the IOTS and the queues. Moreover, the proposition restricts the DI traces to those with a bounded number of quiescent actions.

Hence, if there are traces that strongly q-cover a transition (s_1, a, s_2) of $Spec_\delta$, we can find some in $U^{SC}(Spec_\delta, (s_1, a, s_2)) \cap U^{DI}(Spec_\delta)$, where $U^{SC}(Spec_\delta, (s_1, a, s_2))$ is the set of traces strongly covering the transition (the output of Procedure 1) and $U^{DI}(Spec_\delta) = \{u \in DI\text{-}traces(Spec_\delta) \mid |u\downarrow_\delta| \leq 2^{3|S|}\}$. As we know how to find $U^{SC}(Spec_\delta, (s_1, a, s_2))$, the problem of finding a fair winning strategy now boils down to deciding trace set $U^{DI}(Spec_\delta)$.

However, it is not easy to decide $U^{DI}(Spec_\delta)$, either. The problem is that regular languages are not closed under the delay operator, so it is unknown how to decide the set $DI\text{-}traces(Spec_\delta)$.

Here, we take a pragmatic approach by finding subsets (underestimations) of $U^{DI}(Spec_\delta)$. We define $U^{DI}{}_k(Spec_\delta) = \{u = v_1\delta v_2\delta\ldots v_n\delta v_{n+1} \in DI\text{-}traces(Spec_\delta) \mid v_i \in (I \cup O)^*$ and $|(v_i)\downarrow_I| \leq k, n \leq 2^{3|S|}\}$. One can check that $U^{DI}{}_k(Spec_\delta)$ is finite (because *Spec* is input-progressive), $U^{DI}{}_k(Spec_\delta) \subseteq U^{DI}(Spec_\delta)$, and $\lim_{k\to\infty} U^{DI}{}_k(Spec_\delta) = U^{DI}(Spec_\delta)$.

By limiting the number of input actions between two consecutive quiescence, we can decide $U^{DI}{}_k(Spec_\delta)$ because it is finite. We can improve our estimation of whether strongly q-covering traces exist by increasing k.

The reader can verify that for the coffee machine E_δ in Figure 4, trace $u = coin\ coin\ espresso\ \delta$ strongly covers transition $(t_1, coin, t_1)$ and is DI, so it belongs to $U^{SC}(E_\delta, (t_0, a, t_1)) \cap U^{DI}{}_2(E_\delta)$. Therefore, a fair winning strategy over the transition can be $f[\{u\}]$, which is

the one given in Example 7. When executing a test implementing this strategy, we can declare success if we observe the trace, but have to re-run the test otherwise.

5 Conclusions

Motivated by the need to improve specification coverage in testing, we considered in this paper the following problem: for the IUT of a concurrent system, give a conclusive answer by testing through a context of queues that either a transition corresponding to a given transition in the specification *must* have been executed or the IUT does not conform to the specification.

We defined transition coverage through queues and studied how to cover transitions with and without fairness assumption.

For transition coverage without the fairness assumption, we completely solved the problem by constructing winning strategies composed of only input/output conflict-free traces of the specification.

The computational complexity of the procedure for finding winning strategies (Procedure 2) is double exponential in terms of the specification's number of states, and is reduced to polynomial-time for deterministic specifications. Since the complexity of determinizing an IOTS alone is exponential, the impact of the infinite state context of unbounded queues seems to be minimized in our solution.

For transition coverage with the fairness assumption, the problem is partially solved because we only found underestimations for fair winning strategies, as it is unknown how to decide the delay-insensitive traces of an IOTS.

However, since a winning strategy is a fair winning strategy, one may first try to derive a winning strategy. If this fails, one may then examine delay-insensitive traces through underestimation to find a fair winning strategy.

Our solution also applies to other types of state machines, such as Mealy machines. Consider observable non-deterministic finite state machines (ONFSM), an important model for software systems ([23] and [17]). If we "decouple" transitions of an ONFSM into transition pairs of an input followed by an output, we obtain a deterministic IOTS whose traces are input/output conflict-free (thus delay-insensitive). For such a system, our solution is complete and efficient because it can find winning strategies or fair winning strategies in polynomial-time.

This paper complements our previous work [8] where q-coverage (not strong) is considered.

A byproduct of the paper is the generalized delay operator. To our knowledge, this is the first time that suspension traces of an IOTS are related to those of its composition with

queues. In existing works, only the relations between traces or quiescent traces are discussed ([22] and [8]).

In view of future work, we will try to find finer ways to estimate delay-insensitive traces. Also, we plan to generalize the results to include partial specifications. The major difficulty of this is that the relation between an IOTS and its composition with queues as formulated for fully specified systems in Section 3 has to be revised for partially specified ones.

References

[1] G. Antoniol, L.C. Briand, M.D. Penta, and Y. Labiche, "A Case Study Using the Round-Trip Strategy for State-Based Class Testing", In *Proc. 13th Intl. Symp. Software Reliability Engineering (ISSRE 2002)*, Annapolis, MD, USA, Nov. 17-20, 2002, IEEE Computer Society, pp. 269–279.

[2] S. Balemi, "Input/Output Processes and Communication Delays", *Discrete Event Dynamic Systems: Theory and Applications*, 4, 1994, pp. 41–85.

[3] R.V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley Object Technology Series, Addison-Wesley, 1999.

[4] C. Campbell, M. Veanes, J. Huo, and A. Petrenko, "Multiplexing of Partially Ordered Events", In F. Khendek and R. Dssouli, eds., *Proc. 17th IFIP TC6/WG 6.1 Intl. Conf. Testing of Communicating Systems (TestCom 2005)*, volume 3502 of *Lecture Notes in Computer Science*, Montreal, Canada, May 31 - June 2, 2005, Springer, pp. 97–110.

[5] M.A. Fecko, M.Ü. Uyar, A.S. Sethi, and P.D. Amer, "Issues in conformance testing: multiple semicontrollable interfaces", In S. Budkowski, A.R. Cavalli, and E. Najm, eds., *Proc. IFIP TC6/WG6.1 Joint Intl. Conf. Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE XI / PSTV XVIII)*, volume 135 of *IFIP Conference Proceedings*, Paris, France, Nov. 3-6, 1998, Kluwer, pp. 111–126.

[6] R. Groz, O. Charles, and J. Renévot, "Relating Conformance Test Coverage to Formal Specifications", In R. Gotzhein and J. Bredereke, eds., *Proc. IFIP TC6/WG6.1 Joint Intl. Conf. Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE IX / PSTV XVI)*, volume 69 of *IFIP Conference Proceedings*, Kaiserslautern, Germany, Oct. 8-11, 1996, Chapman & Hall, pp. 195–210.

[7] H.S. Hong, S.D. Cha, I. Lee, O. Sokolsky, and H. Ural, "Data Flow Testing as Model Checking", In *Proc. 25th Intl. Conf. Software Engineering*, Portland, OR, USA, May 3-10, 2003, IEEE Computer Society, pp. 232–243.

[8] J. Huo and A. Petrenko, "On Testing Partially Specified IOTS through Lossless Queues", In R. Groz and R.M. Hierons, eds., *Proc. 16th IFIP TC6/WG 6.1 Intl. Conf. Testing of Communicating Systems (TestCom 2004)*, volume 2978 of *Lecture Notes in Computer Science*, Oxford, UK, Mar. 17-19, 2004, Springer, pp. 76–94.

[9] I-Logix, "Rhapsody", http://www.ilogix.com/.

[10] C. Jard and T. Jéron, "TGV: Theory, Principles and Algorithms", *Intl. J. Software Tools for Technology Transfer*, 7(4), 2005, pp. 297–315.

[11] C. Jard, T. Jéron, L. Tanguy, and C. Viho, "Remote Testing Can Be as Powerful as Local Testing", In J. Wu, S.T. Chanson, and Q. Gao, eds., *Proc. IFIP TC6/WG6.1 Joint Intl. Conf. Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE XII / PSTV XIX)*, volume 156 of *IFIP Conference Proceedings*, Beijing, China, Oct. 5-8, 1999, Kluwer, pp. 25–40.

[12] D. Lee, K.K. Sabnani, D.M. Kristol, and S. Paul, “Conformance Testing of Protocols Specified as Communicating Finite State Machines - A Guided Random Walk Based Approach”, *IEEE Trans. Communications*, 44(5), May 1996, pp. 631–640.

[13] L.P. Lima, Jr. and A.R. Cavalli, “A Pragmatic Approach to Generating Test Sequences for Embedded Systems”, In *Proc. 10th IFIP TC6/WG6.1 Intl. Workshop on Testing of Communication Systems (IWTCS’97)*, Cheju Island, Korea, Sept. 8-10, 1997.

[14] C.-H. Liu, D.C. Kung, P. Hsia, and C.-T. Hsu, “Structural Testing of Web Applications”, In *Proc. 11th Intl. Symp. Software Reliability Engineering (ISSRE 2000)*, San Jose, CA, USA, Oct. 8-11, 2000, IEEE Computer Society, pp. 84–96.

[15] N. Lynch and M. Tuttle, “An Introduction to Input/Output Automata”, *CWI-Quarterly*, 2(3), Sept. 1989, pp. 219–246.

[16] C.E. Molnar, T.-P. Fang, and F.U. Rosenberger, “Synthesis of Delay-Insensitive Modules”, In *Proc. 1985 Chapel Hill Conf. Advanced Research in VLSI*, Chapel Hill, NC, USA, 1985, pp. 67–86.

[17] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp, “Optimal Strategies for Testing Nondeterministic Systems”, In G.S. Avrunin and G. Rothermel, eds., *Proc. ACM/SIGSOFT Intl. Symp. Software Testing and Analysis (ISSTA 2004)*, Boston, MA, USA, July 11-14, 2004, ACM, pp. 55–64.

[18] S. Naito and M. Tsunoyama, “Fault Detection for Sequential Machines by Transition-Tours”, In *Proc. 11th IEEE Ann. Intl. Symp. Fault-Tolerant Computing*, Los Alamitos, CA, USA, 1981, pp. 238–243.

[19] A.J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, “Generating Test Data from State-Based Specifications”, *Software Testing, Verification and Reliability*, 13(1), 2003, pp. 25–53.

[20] R.N. Taylor, D.L. Levine, and C.D. Kelly, “Structural Testing of Concurrent Programs”, *IEEE Trans. Softw. Eng.*, 18(3), 1992, pp. 206–215.

[21] J. Tretmans, “Test Generation with Inputs, Outputs and Repetitive Quiescence”, *Software - Concepts and Tools*, 17(3), 1996, pp. 103–120.

[22] J. Tretmans and L. Verhaard, “A Queue Model Relating Synchronous and Asynchronous Communication”, In R.J. Linn, Jr. and M.Ü. Uyar, eds., *Proc. 12th IFIP TC6/WG6.1 Intl. Symp. Protocol Specification, Testing and Verification (PSTV XII)*, volume C-8 of *IFIP Transactions*, Lake Buena Vista, FL, USA, June 22-25, 1992, North-Holland, pp. 131–145.

[23] F. Zhang and T.-Y. Cheung, “Optimal Transfer Trees and Distinguishing Trees for Testing Observable Nondeterministic Finite-State Machines”, *IEEE Trans. Softw. Eng.*, 29(1), 2003, pp. 1–14.

Appendix

Lemma 3 For IOTS L_δ, a DI trace strongly q-covers a transition if it strongly covers the transition.

Proof: For trace $u \in$ *DI-traces*(L_δ), we have $u \in$ *traces*$(L_\delta) \subseteq$ *traces*$(Q(L)_\delta)$, and *delay*$_\delta[I](uO^*\delta^*) \cap$ *traces*$(L_\delta) = uO^*\delta^* \cap$ *traces*(L_δ). Therefore, if u strongly covers the transition, all traces in *delay*$_\delta[I](uO^*\delta^*) \cap$ *traces*(L_δ) strongly covers the transition, which means that u strongly q-covers the transition according to Definition 2. ◊

Proof of Proposition 1 (outline): (if) Obvious.

(Only if) According to Definition 2 and Definition 6, for any $u \in TT(f_1) \cap$ *traces*$(Q($*Spec*$)_\delta)$ such that $f_1(u) =$ *stop*, all traces in *delay*$_\delta[I](uO^*\delta^*) \cap$ *traces*$($*Spec*$_\delta)$, including IOCF ones, strongly cover the transition. According to Lemma 3 and the fact that IOCF traces are DI traces, traces in *delay*$_\delta[I](uO^*\delta^*) \cap$ *IOCF-traces*$($*Spec*$_\delta)$ strongly q-cover the transition. Thus, strategy $f_2 = f$[*prune*(*delay*$_\delta[I](TT(f_1)O^*\delta^*) \cap$ *IOCF-traces*$($*Spec*$_\delta)$)] is the one that we are looking for. ◊

Proof of Proposition 2: (if) By Lemma 3.

(only if) For $L \in$ *IOTS*(I, O), we define a relation R on traces in *delay*$_\delta[I](u_1O^*\delta) \cap$ *traces*(L_δ): $(u, v) \in R$ if $u \in$ *delay*$_\delta[I](v)$. R is reflexive, as $u \in$ *delay*$_\delta[I](u)$. R is antisymmetric because $(u, v) \in R$ and $u \neq v$ implies that u is derived from v by shifting input actions to the right, which means $(v, u) \notin R$. R is transitive, since we can prove $u \in$ *delay*$_\delta[I](v)$ and $v \in$ *delay*$_\delta[I](w) \Rightarrow u \in$ *delay*$_\delta[I]($*delay*$_\delta[I](w)) =$ *delay*$_\delta[I](w)$ (delay a trace twice is the same as delaying it once). Therefore, R is a partial order. A minimal element of the partial order R, i.e., v s.t. $\forall u.(u, v) \in R \Rightarrow u = v$, is the candidate of the trace u_2 that we look for.

First, a minimal element v must exist because *delay*$_\delta[I](u_1O^*\delta) \cap$ *traces*(L_δ) is finite and non-empty (recall that L is input-progressive, so there are only finitely many output sequences leading to quiescence after u_1). Second, v is DI, because *delay*$[I](vO^*\delta^*) \cap$ *traces*$(L_\delta) =$ *delay*$[I](v)\delta^* \cap$ *traces*(L_δ) (v ends with δ, after which no output is possible) $= v\delta^* \cap$ *traces*(L_δ) (v is minimal) $= vO^*\delta^* \cap$ *traces*(L_δ) (v ends with δ). Third, v strongly covers (s_1, a, s_2) because, according to Definition 2, u_1 strongly q-covers the transition if all traces in *delay*$_\delta[I](u_1O^*\delta^*) \cap$ *traces*(L_δ) strongly cover the transition.

If $|v{\downarrow_\delta}| > 2^{3|S|}$, let $v = v_1v_2\ldots v_n$, $v_i \in (I \cup O)^*\delta$ for $0 < i \leq n$. We can construct a sequence of state set triples $(S_0, \varnothing, \varnothing)$, (S_1, T_1, Z_1), (S_2, T_2, Z_2),… $(S_{n-1}, T_{n-1}, Z_{n-1})$, $(\varnothing, T_n, \varnothing)$, where $T_n \neq \varnothing$ and (S_i, T_i, Z_i) is reached by trace $v_1v_2\ldots v_i$ for $0 < i \leq n$. S_i contains the states reached

by paths of trace $v_1v_2\ldots v_i$ that have not traversed the transition yet, T_i contains the states reached by paths of trace $v_1v_2\ldots v_i$ that have already traversed the transition and not in S_i (i.e., $S_i \cap T_i = \varnothing$), and Z_i contains the states reached by the paths of traces belong to delay$[I](v_1v_2\ldots v_i) \setminus \{v_1v_2\ldots v_i\}$.

If $n > 2^{3|S|}$, there must be some triples visited twice along v because there are at most $2^{3|S|}$ triples of the form (S_i, T_i, Z_i). By deleting strings between every two occurrences of the same triples, we obtain a trace that still reaches $(\varnothing, T_n, \varnothing)$. The obtained trace is DI (because $Z_n = \varnothing$), strongly covers the transition (because $S_n = \varnothing$ and $T_n \neq \varnothing$), and has no more than $2^{3|S|}$ δ actions (by deletion of redundant occurrences of the triples), so it is the trace u_2 that we look for.

u_2 strongly q-covers the transition by Lemma 3. ◊