On Testing Partially Specified IOTS through Lossless Queues

Jia Le Huo[1] and Alexandre Petrenko[2]

[1]Department of Electrical and Computer Engineering, McGill University
3480 University Street, Montreal, Quebec, H3A 2A7, Canada
Jiale@macs.ece.mcgill.ca
[2]CRIM, Centre de recherche informatique de Montréal
550 Sherbrooke Street West, Suite 100, Montreal, Quebec, H3A 1B9, Canada
Petrenko@crim.ca

Abstract. In this paper, we discuss how to test partially specified IOTS through lossless queues. A liberal assumption is made of the IOTS model by allowing both blocked and unspecified input actions. For testing IOTS through unbounded queues, we demonstrate that test cases can directly be derived from the specification when the transition coverage criterion is used, and we provide two test derivation algorithms, for fully specified and partially specified IOTS, respectively. Applying the derived tests to test IOTS through bounded queues is also discussed.

1 Introduction

Transition systems with concurrent input/output behavior are usually modeled by input/output transition systems (IOTS). Here, we explore how to test IOTS through queues with the following scenario of system communication in mind. As shown in Fig. 1, L_1 and L_2 are two message-passing systems. Output actions of L_1 are stored in the input queue of L_2, and, if the input queue is not empty, L_2 can read an input action and make a transition according to the action read. The communication from L_2 to L_1 is symmetrically configured. This scenario is common in communicating systems.

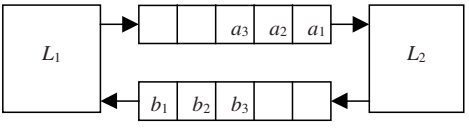

Fig. 1. The communication between two message-passing systems. L_1 and L_2 "read" input actions from their input queues and "write" output actions into the output queues

We notice that, in this scenario, either system can block its own input actions by not reading the input queue; the output actions of the other system, however, are not blocked, but stored in the queues. Therefore, the systems do not have to be receptive to input actions in every state, whereas their output actions are never blocked. Queues

R. Groz and R.M. Hierons (Eds.): TestCom 2004, LNCS 2978, pp. 76–94, 2004.

in this scenario store actions for a later consumption and, thus, are called *lossless queues.*

A system that can block input actions is usually rendered as an IOTS with input actions missing in some states. The system's behavior in these states is fully specified. In particular, the system decides not to read input actions from the input queue.

Input actions could also be missing due to underspecification. The missing actions are "don't cares", i.e., the consequence of the actions is not specified, so that any behavior of the IOTS's implementations after the actions is acceptable.

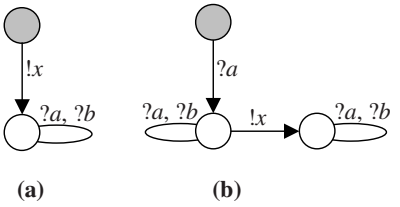

Fig. 2. Difference between two types of missing input actions in IOTS. For the IOTS in the figure, the input actions (decorated with ?) are *a* and *b*, the output action (decorated with !) is *x*, and the starting states are shaded by grey: (a) the missing input actions are blocked; (b) the missing input action *b* is unspecified

Fig. 2 illustrates the difference between these two types of missing input actions. In Fig. 2(a), the starting state has no transition on any input action. For a system with input queue, this means that, in this state, the IOTS does not read any input action from its input queue, so input actions are blocked. In Fig. 2(b), on the other hand, input action *a* is specified in the starting state, but input action *b* is not. Since it is hard to imagine that the IOTS can read only input action *a*, but not action *b*, without knowing which action is in the input queue, the missing action *b* is understood as unspecified. Implementations of this IOTS can behave arbitrarily after reading *b* in the starting state.

We say that an action is *enabled* in a state if the corresponding transition on the action starting from the state is defined. An IOTS is *input-enabled* if all input actions are enabled in every state. Moreover, an IOTS is *fully specified* if, in each state, either all input actions are enabled or no input action is enabled; otherwise, the IOTS is *partially specified*. Input-enabled IOTS are fully specified, but fully specified IOTS may have missing input actions and, therefore, need not be input-enabled.

In this paper, specifications can be partially specified, whereas implementations must be fully specified, but not necessarily input-enabled.

A model closely related to IOTS, called input/output automata (IOA), is first formalized in [7], among others. The difference between the two models is marginal, at least from the viewpoint of testing. In [7], it is stated that an IOA "generates output and internal actions autonomously". A system modeled by IOA, therefore, cannot have its output blocked by other systems. Non-blocking of output is ensured by requiring that IOA be input-enabled.

Although there is some research done for IOTS-based testing (for references, see, e.g., [3], [9], [10]), none of them, in our opinion, provides a satisfactory answer to testing IOTS in the aforementioned scenario.

Testing based on the IOTS model is explored in [12], among others, which establishes the so-called **ioco** testing framework. In that framework, a tester and an implementation under test (IUT) communicate synchronously: testers are modeled by labeled transition systems (LTS), and implementations are modeled by input-enabled IOTS. Because of synchronous composition, testers of the **ioco** framework can block the output actions of IUT, violating the assumption that "output actions can never be blocked by the environment" [p. 106, 12]. Also because of synchronous communication, when applying the **ioco** framework to test IOTS through queues, one has to compose IOTS with queues, which is not realistic if the queues have unbounded capacity.

The **ioco** testing framework is further elaborated for multi-channel IOTS (MIOTS) in [5]. In that paper, in each state of an implementation, either all input actions of a communication channel are enabled or they are all blocked. Similar to the approach in [12], output actions of IUT can be blocked. Moreover, testers are empowered with the ability to observe refusal of input or output by IUT. While it is indeed possible to detect absence of output (quiescence) by using a proper timer, it is unclear how refusal of input can be observed in an arbitrary system.

Testing IOA systems with synchronous communication is also studied in [11], where it is assumed that testers, instead of blocking output, can observe input/output conflict with IUT. In that paper, however, both specifications and implementations must be input-enabled.

Testing IOTS through queues with unbounded capacity is first explored in [13] and [14], where both specifications and implementations can block input actions. However, the proposed approach relies on explicitly composing IOTS with infinite state queue contexts, so it is not clear how this approach could be implemented in practice.

Testing IOTS through unbounded queues is also considered in [6], where a stamping mechanism is proposed to order the output actions with respect to the input actions, while quiescence is ignored. A stamping process observes and records local actions of IUT, so it is not always realistic to assume that such a process is available.

Testing IOTS through bounded queues is explored in [8], which only considers input-enabled specifications. However, the queue model used in that paper is different from the one used here. In [8], it is assumed that queues feed input actions to IOTS, which leads to the requirement that both testers and IUT must be input-enabled to avoid blocking the output of queues. In the communication scenario of Fig. 1, we assume that contents of queues are read by IOTS, so testers and IUT can block input actions. Examples of the queue model assumed in [8] are shift registers, whereas examples of the queue model assumed in this paper are message queues.

This paper studies how to test IOTS through lossless queues. Unlike previous work, we make a liberal assumption about the system model, namely, the specification of a system can have both blocked and unspecified input actions. We believe that such a model is closer to real system specifications than other models known to us.

A "naïve" test derivation algorithm would derive tests from the composition of such a specification IOTS and its input/output queues. Computing the composition is usually not viable, when the queues have unbounded capacity, or faces the state explosion problem, when the queues have bounded capacity. Moreover, there is no

guarantee that a test case derived by this approach will observe the queues' capacities because the information of capacities is lost in the composition.

Here, we derive tests directly from a specification, not from its composition with queues. The derived tests aim at covering transitions of the specification. For testing IOTS through unbounded queues, the resulting test cases traverse only specified transitions, whereas in the case of bounded queues, the test cases also obey the bound of queues.

The paper is organized as follows. We provide some preliminaries of the paper in Section 2. Section 3 introduces the testing architecture. Section 4 discusses how to test fully specified IOTS through unbounded queues, building the basis for testing partially specified IOTS through unbounded queues in Section 5. Sections 4 and 5 provide two test derivation algorithms with the transition coverage criterion in mind. Section 6 briefly discusses testing IOTS through bounded queues. Conclusions are provided in Section 7.

2 Preliminaries

Here, we use a definition of input/output transition systems (IOTS) that is similar to the one of input/output automata (IOA) in [7]. Formally, an *input/output transition system* is a 5-tuple $L = <S, I, O, \lambda, S_0>$, where

- S is a countable (not necessarily finite) set of states;
- I and O are finite sets of input and output action types, respectively, which satisfy the condition $I \cap O = \emptyset$;
- $\lambda \subseteq S \times (I \cup O \cup \{\tau\}) \times S$ is a transition relation, where $\tau \notin I \cup O$ is the internal action type;
- $S_0 \subseteq S$ is a non-empty, finite set of initial states.

After [12], we only consider strongly converging specifications and implementations, i.e., systems that contain no cycle of internal transitions. We use $IOTS(I, O)$ to represent the set of all IOTS with input set I and output set O.

For IOTS $L = <S, I, O, \lambda, S_0>$, we use $init(s)$ to denote the set of actions enabled in state $s \in S$, i.e., $init(s) = \{a \in (I \cup O \cup \{\tau\}) \mid \exists s_1 \in S \text{ s.t. } ((s, a, s_1) \in \lambda)\}$. L is *input-enabled* if all input actions are enabled in each state, i.e., $I \subseteq init(s)$ for each $s \in S$; 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 each $s \in S$. If L is not fully specified, it is *partially specified*. State $s \in S$ is called *stable* if no output or internal actions are enabled in s: $init(s) \cap (O \cup \{\tau\}) = \emptyset$. State $s \in S$ with no action enabled, i.e., $init(s) = \emptyset$, is called a *deadlock* state. L *deadlocks* if there is a deadlock state reachable from a starting state.

With multiple initial states, internal transitions, and a transition relation (not a function), the IOTS considered in this paper are non-deterministic and, thus, can model a wide range of systems on various levels of abstraction. On the other hand, we call an IOTS *deterministic* if it has a single initial state, contains no internal transitions, and the transition relation is a function, i.e., $(s, a, s_1), (s, a, s_2) \in \lambda$ for $a \in I \cup O$ implies $s_1 = s_2$.

The projection operator $\downarrow A$ projects action sequences onto the alphabet $A \subseteq I \cup O$. Let ε denote the empty sequence of actions, $\varepsilon_{\downarrow A} = \varepsilon$. For $u \in (I \cup O \cup \{\tau\})^*$ and $a \in I \cup O \cup \{\tau\}$, $(ua)_{\downarrow A} = u_{\downarrow A}a$ if $a \in A$; otherwise, $(ua)_{\downarrow A} = u_{\downarrow A}$. A sequence $u \in (I \cup O)^*$ is called a *trace* of IOTS L in state $s \in S$ if there exist actions $a_1, \ldots, a_k \in I \cup O \cup \{\tau\}$, such that $u = (a_1 \ldots a_k)_{\downarrow (I \cup O)}$, and states $s_1, \ldots, s_{k+1} \in S$, such that $(s_i, a_i, s_{i+1}) \in \lambda$ for all $i = 1, \ldots, k - 1$ and $s_1 = s$. L *executes* trace u if L makes a sequence of transitions from its starting state and the corresponding action sequence projected onto $I \cup O$ is u. We use *traces*(s) to denote the set of traces of L in state s, and sometimes, by using L to refer to the set of L's initial states, we use *traces*(L) to denote the union of traces in L's initial states. State $s \in S$ with a sequence $b_1 b_2 \ldots b_k \in O^*$ such that $(b_1 b_2 \ldots b_k)^* \subseteq$ *traces*(s) is called an *oscillating* state. L *oscillates* if there is an oscillating state reachable from a starting state.

IOTS L is called *input-progressive* if it neither oscillates nor deadlocks. If L is input-progressive, it must consume an input action to make a transition in less than $|S|$ steps, where $|S|$ is L's number of states.

Following [13] and [12], we refer to a trace that takes IOTS L from state $s \in S$ to a stable state as a *quiescent trace* in s, and we use *qtraces*(s) to denote the set of all quiescent traces in s. Similar to the case of traces, *qtraces*(L) denotes the union of the quiescent traces in L's initial states. Traces and quiescent traces can be used to distinguish non-deterministic systems, whereas traces alone are sufficient to distinguish deterministic systems.

We use a usual operator **after**. For IOTS L, L-**after**-U denotes the set of states that are reachable by L when it executes the traces in the set U.

We use suspension traces to refer to sequences of quiescent traces executable by an IOTS. As usual ([12]), the symbol δ indicates quiescence, i.e., the absence of output and internal actions in a system. To explicitly represent quiescence in IOTS L, we add self-looping δ transitions to the stable states of L, similar to [12]. The augmented IOTS is denoted as L_δ, where the δ actions are treated as output actions. For state s of L, we define a *suspension trace* in s to be a trace of L_δ in s. We use *straces*(L) to denote the set of all suspension traces of L in the initial states. *straces*(L) is a superset of traces and quiescent traces augmented with intermediate quiescence.

Composition of IOTS formalizes the interaction of several systems. Here, we use the traditional parallel composition $\|$ of labeled transition systems (LTS), i.e., transition systems that do not distinguish input from output.

Formally, for IOTS $L_1 = \langle S, I_1, O_1, \lambda_1, S_0 \rangle$ and $L_2 = \langle T, I_2, O_2, \lambda_2, T_0 \rangle$ such that $O_1 \cap O_2 = \varnothing$, the *parallel composition* $L_1 \| L_2$ is the IOTS $\langle R, (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, \lambda, S_0 \times T_0 \rangle$, where the set of states $R \subseteq S \times T$ and the transition relation λ are the smallest sets obtained by applying the following inference rules:

- $S_0 \times T_0 \subseteq R$;
- 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 $s_2 t_2 \in R$ 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 $s_2 t_1 \in R$ 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 $s_1 t_2 \in R$ and $(s_1 t_1, a, s_1 t_2) \in \lambda$.

Sometimes, we have to transform a partially specified IOTS to a fully specified one by completing the former's input transitions, so we use a completion operator similar to [6]. For IOTS $L = <S, I, O, \lambda, S_0>$, operator *Comp*: $IOTS(I, O) \rightarrow IOTS(I, O)$ is defined as $Comp(L) = <S \cup \{s^t_L\}, I, O, \lambda_c, S_0>$ where s^t_L is a trap state and λ_c is defined as $\lambda \cup \{(s, a, s^t_L) \mid init(s) \cap I \neq \emptyset, a \in I \setminus init(s)\} \cup \{(s^t_L, b, s^t_L) \mid b \in I \cup O\}$. Notice that, unlike the operator in [6], $Comp(L)$ is a fully specified IOTS, not input-enabled, because the states, where all input actions are blocked, have no input enabled in $Comp(L)$.

3 Testing Architecture with Lossless Queues

When testing a communicating system, we assume the closed system shown in Fig. 1. The tester and the implementation under test (IUT) are the end systems of the queues. Both systems, along with the queues, are modeled by IOTS. The interaction between the components in the closed system is described by their parallel composition.

The IUT *Imp* belongs to $IOTS(I, O)$ and the tester *Test* belongs to $IOTS(O \cup \{\delta\}, I)$, where symbol δ denotes the detection of quiescence, i.e., output queue of *Imp* is empty. The input actions of *Imp* correspond to the output actions of *Test*, and vice versa. When seen in the closed system, the action types of *Imp*, *Test*, the input queue Q_I, and the output queue $Q_{O'}$ are assigned according to Fig. 3, to avoid using the same action types at different interfaces.

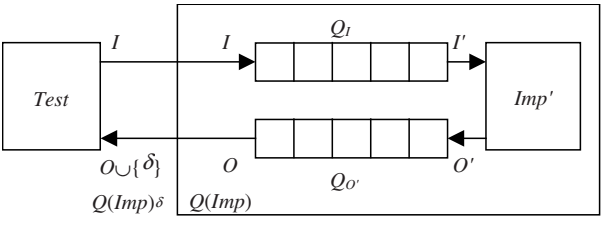

Fig. 3. The input and output action types of the components in the testing architecture: $Test \in IOTS(O \cup \{\delta\}, I)$, $Imp' \in IOTS(I', O')$, $Q_I \in IOTS(I, I')$, and $Q_{O'} \in IOTS(O', O)$

In the closed system, the actions of *Imp* are relabeled by $'$. 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' belongs to $IOTS(I', O')$ and is derived from L by relabeling each action $a \in I \cup O$ to a'. *Imp* in the closed system is relabeled by $'$.

For the queues, each input action a (or a') corresponds to an output action a' (or a). We define the queue model with the $'$ operator. 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 $\lambda_A = \{(u, a, ua) \mid u, ua \in S_A\} \cup \{(av, a', v) \mid av, v \in S_A\}$.

By definition, Q_A is input-enabled and has infinitely many states, so it is "unbounded". As an example, Fig. 4 shows an unbounded queue $Q_{\{a\}}$ with a single input action a.

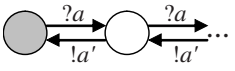

Fig. 4. An unbounded queue with a single input action a

As seen in Fig. 3, the input queue is Q_I, and the output queue is $Q_{O'}$ (notice the property of the $'$ operator: $a'' = a$). The behavior of the closed system shown in Fig. 3 can be described by the IOTS $Test \parallel (Q_I \parallel Imp' \parallel Q_{O'})_\delta$, where L_δ is IOTS L augmented by adding self-looping δ transitions (see Section 2).

Imp composed with its input/output queues can be described by the operator $Q(\ )$. For $L \in IOTS(I, O)$, $Q(L) = hide(I' \cup O')(Q_I \parallel L' \parallel Q_{O'})$, where operator $hide(A)(\ )$ relabels transitions of an IOTS with actions in A to internal action τ. $Q(L)$ is similar to the queue operator as in [13] and [14]. From the tester's point of view, the observable behavior of the closed system shown in Fig. 3 can be described by $Test \parallel Q(Imp)_\delta$, since the actions in $I' \cup O'$ are not observable.

Some properties are usually assumed for IOTS in a composition, e.g., input-enabledness [7] and full compatibility [8]. Here, we state a compatibility condition for each interface in the closed system shown in Fig. 3.

For *Test* and *Imp*, output actions are usually under their total control, so the output actions of *Test* and *Imp* have to be accepted immediately by $Q(Imp)_\delta$ and $Q_{O'}$, respectively.

Definition 1. Let $L_1 = <S, I_1, O_1, \lambda_1, S_0>$ and $L_2 = <T, I_2, O_2, \lambda_2, T_0>$, L_1 is *output compatible* with L_2 if, for any state st of $L_1 \parallel L_2$ and any action $a \in O_1 \cap I_2$, $a \in init(s)$ implies $a \in init(t)$.

If L_1 is output compatible with L_2, the output of L_1 is not blocked by L_2 in the composition. L_1 is output compatible with L_2 if L_2 is input-enabled, but the condition is not necessary. In Fig. 3, *Test* is output compatible with $Q(Imp)_\delta$, and *Imp* is output compatible with $Q_{O'}$, because the queues are input-enabled (and therefore, $Q(Imp)_\delta$ is input-enabled).

On the other hand, requiring the queues in the testing architecture (Fig. 3) to be output compatible with the tester or IUT (as in [8]) is too strong for the closed system with lossless queues. Here, output actions of queues can be stored for a later consumption. This means that an input queue can be composed with a system without being output compatible with the system. Such a queue is a storage media that does not lose data not requested immediately, such as message queues, so it is *lossless*. A queue that is not output compatible with the system must be lossless. In [8], on the contrary, a queue is a media that only transfers data, but does not keep it, such as shift registers, so that some data might be lost if the system at the end of the queue is not receptive to the queue's output.

We require that, before executing a test case, $Q(Imp)$ must be properly initialized, i.e., *Imp*, Q_I, and Q_O must be in (one of) their initial states, respectively. The requirement can be met with the following three assumptions. First, we can reset *Imp* reliably. Second, the input queue Q_I, which is usually under the control of *Imp*, is reset reliably to an empty queue when *Imp* is reset. Third, the tester only assigns the verdict **pass** after emptying the output queue Q_O, which is usually under the tester's control. On the other hand, if the verdict **fail** is assigned, *Imp* is immediately rejected, so there is no need to execute another test case.

Instead of assuming that we can clear the output queue before executing a test case, we make the last assumption to prevent the situation where an IUT can produce a wrong output after the tester reaches the verdict **pass**. This assumption immediately excludes specifications that oscillate from further consideration, but still leaves us with a wide class of specifications. Moreover, we require that specifications do not have deadlock states, which are usually used to model system breakdown triggered by unspecified behavior or implementation faults. Therefore, we only consider in the paper input-progressive specifications, which do not oscillate or deadlock by definition.

We find it convenient to use the delay operator as defined in [1] to describe the behavior of $Q(L)$. Delay operators are a subset of so-called semi-commutation functions (see [4]). Intuitively, an IUT's output actions can be delayed from the viewpoint of a tester; whereas the tester's output actions (i.e., the IUT's input actions) can be delayed from the viewpoint of the IUT. The delay operation expresses the effect of queues on traces.

For a sequence set (language) $E \subseteq A^*$, a subalphabet $A_1 \subseteq A$, operator $delay[A_1]$: $2^{A^*} \rightarrow 2^{A^*}$ calculates the smallest superlanguage of E such that for $u, v \in A^*$, any $a \in A \setminus A_1$ and $a_1 \in A_1$:

- $E \subseteq delay[A_1](E)$ and
- $ua_1av \in delay[A_1](E)$ implies $uaa_1v \in delay[A_1](E)$.

According to the definition, $delay[A_1](E)$ derives a language from E by shifting symbols in A_1 towards the end of each word in E while keeping the relative order of symbols in A_1 and $A \setminus A_1$, respectively.

From the viewpoint of a tester, the traces and quiescent traces of an input-progressive IOTS $L = \langle S, I, O, \lambda, S_0 \rangle$ in a queue context are $traces(Q(L)) = pref(delay[O](traces(L)))$, where $pref(U)$ is the prefix closure of trace set U, and $qtraces(Q(L)) = delay[O](qtraces(L))$, respectively.

On the other hand, after a trace u is executed by the tester, L can execute any trace in $delay[I](uO^*) \cap traces(L)$. Since L is input-progressive, each input action can cause at most $|S| - 1$ output actions, so u can cause at most $l(u) = |u_{\downarrow I}| \times (|S| - 1) - |u_{\downarrow O}|$ additional output actions in L. Therefore, the corresponding traces executable by L can be refined as $delay[I](uO^{l(u)}) \cap traces(L)$, where A^n is the sublanguage of A^* with at most n symbols in each word. Finally, when a quiescent trace u is executed by $Q(L)$, L executes any quiescent trace in $delay[I](u) \cap qtraces(L)$.

In the following, we use the variable *Spec* along with *Imp*. *Spec* and *Imp* represent the specification and implementation of a system, respectively, that belong to $IOTS(I, O)$ and have finite number of states.

4 Testing Fully Specified IOTS through Unbounded Queues

We assume in this section that *Spec* is fully specified and input-progressive, whereas *Imp* is fully specified. Neither *Spec* nor *Imp* has to be input-enabled. In Section 5, we will lax the restriction on fully specified *Spec*.

There are a couple of conformance relations that can be formulated between *Spec* and *Imp* in a context with unbounded queues, for example, see [13], [6], and [8]. We briefly introduce them as follows.

Definition 2. For $Spec, Imp \in IOTS(I, O)$,

- *Imp* is *queue-context trace included* into *Spec* if $traces(Q(Imp)) \subseteq traces(Q(Spec))$;
- *Imp* is *queue-context quiescent trace included* into *Spec* if $traces(Q(Imp)) \subseteq traces(Q(Spec))$ and $qtraces(Q(Imp)) \subseteq qtraces(Q(Spec))$;
- *Imp* is *queue-context suspension trace included* into *Spec* if $straces(Q(Imp)) \subseteq straces(Q(Spec))$.

Queue-context trace inclusion relation is similar to the $\leq_{trQ}$ relation in [13]. In [6], the trace inclusion relation is used. Queue-context quiescent trace inclusion relation is similar to the $\leq_{Q}$ relation in [13]. Finally, the suspension trace inclusion relation is used in the **ioco** testing framework [12], and the queued testing framework [8] uses the queued suspension trace inclusion relation.

Since $Q(L)$ usually has infinitely many states, its trace set may not be regular. In [13], an attempt is made to use so-called tracks to characterize the traces and quiescent traces of $Q(Spec)$ and $Q(Imp)$. However, [13] only proves that tracks are finite if the specification has finite behavior, i.e., with finitely many traces and quiescent traces. It is not known whether tracks are regular for (finite state) specifications with infinite behavior, so it is not clear how the track characterization can be applied to these specifications.

In the following, we restrict ourselves to the case of the queue-context quiescent trace inclusion relation and derive tests directly from the original specification alone according to the transition coverage criterion. The results for other relations can be similarly formulated.

We first define a test case with respect to the set of traces of $Q(Spec)$ that we want to verify in $Q(Imp)$ (test purposes) using the chosen conformance relation. As usual, we require that a test case has finite behavior and is deterministic. The latter means that in each state of the test case, either only one input action to *Imp* is enabled, or all output actions of *Imp* (including quiescence, which indicates that the output queue of *Imp* is empty) are enabled, except for deadlock states, where the test case terminates and the verdicts are assigned. Therefore, the structure of a test case should be a tree, which branches only when output actions of the IUT are read from the queue. The verdicts are assigned in the following way. As assumed in Section 3, **pass** verdicts are only assigned after the tester observes quiescence, i.e., only when the tester emptied the output queue of the IUT. On the other hand, verdict **fail** is assigned when wrong output or premature quiescence is observed. In particular, $delay[I](\beta O^{|(\beta)}) \cap traces(Spec)$ contains all traces that can be executed by *Spec* after the tester executes β and the input queue Q_I is empty. If the intersection is empty, verdict **fail** is assigned to the tester state corresponding to the observation of β. Similarly, $delay[I](\{\beta\}) \cap$

qtraces(*Spec*) contains all quiescent traces that can be executed by *Spec* after the tester executes β and observes quiescence of Q(*Spec*). Therefore, we assign **fail** if the intersection is empty or **pass** otherwise.

Definition 3. For $Spec \in IOTS(I, O)$,

1. An *output-branching trace tree* (OBTT) of *Spec* is a finite set of traces $U \subseteq traces(Q(Spec))$ that satisfies the following conditions: for $\forall\ u_1, u_2 \in U$, there exist $v, w_1, w_2 \in (I \cup O)^*$ and $b_1, b_2 \in O$ $(b_1 \neq b_2)$ such that $u_1 = vb_1w_1$ and $u_2 = vb_2w_2$.
2. For OBTT U of *Spec*, a *test case* $T(U)$ with respect to the queue-context quiescent trace inclusion relation is a deterministic IOTS $T(U) = \langle S_t \cup \{\textbf{pass}, \textbf{fail}\}, O \cup \{\delta\}, I, \lambda_t, \{\varepsilon\}\rangle$, where the state set S_t and the transition relation λ_t are the smallest sets derived by the following inference rules:
   - $pref(U) \subseteq S_t \subseteq traces(Q(Spec))$;
   - for $\beta, \beta a \in pref(U)$, where $a \in I$, $(\beta, a, \beta a) \in \lambda_t$;
   - for $\beta \in S_t$, where there is no $a \in I$ such that $\beta a \in pref(U)$,
     - for $b \in O$, $(\beta, b, \textbf{fail}) \in \lambda_t$ if $delay[I](\beta b O^{|(\beta b)}) \cap traces(Spec) = \varnothing$; otherwise, $\beta b \in S_t$ and $(\beta, b, \beta b) \in \lambda_t$;
     - $(\beta, \delta, \textbf{fail}) \in \lambda_t$ if $delay[I](\{\beta\}) \cap qtraces(Spec) = \varnothing$; otherwise, $(\beta, \delta, \textbf{pass}) \in \lambda_t$.
3. The *test length* of a test case $T(U)$ is the length of the longest input sequences that the tester has to apply, i.e., $\textbf{max}\{|u_{\downarrow I}| \mid u \in traces(T(U))\}$.
4. A *test suite* is a set of test cases.

The number of expected output actions is finite because *Spec* is input-progressive; moreover, test case has no cycles. Therefore, a test case has finite behavior.

We have the following proposition claiming the soundness of the test cases.

Proposition 1. For *Spec*, $Imp \in IOTS(I, O)$, if *Imp* is *queue-context quiescent trace included* into *Spec*, then for any test case $T(U)$ of *Spec* as in Definition 3, no state of $T(U) \parallel Q(Imp)_\delta$ contains **fail** as a substate.

One possible way to define an output-branching trace tree is covering a transition of *Spec*. Transition coverage is a widely used criterion in software testing. In protocol testing, covering a given transition is also a typical test purpose.

A trace $u \in traces(L)$ *covers* a transition (s_1, a, s_2) of $L \in IOTS(I, O)$ if there exist $\beta \in pref(u)$ and, if $a \in (I \cup O)$, $\beta a \in pref(u)$ such that $s_1 \in L$-**after**-β. A test case $T(U)$ *covers* a transition (s_1, a, s_2) of *Spec* if there is a trace u in the composition $T(U) \parallel (Q_I \parallel Spec' \parallel Q_O)_\delta$ such that $u_{\downarrow I \cup O'}$ covers (s_1, a', s_2) of *Spec'*. According to the discussion in Section 3, a test case $T(U)$ covers a transition of *Spec* if and only if there exists a trace $v \in \text{delay}[I](traces(T(U))O^*)$ covering the transition of *Spec*. A test suite is a *transition cover test* of *Spec* if each transition (s_1, a, s_2) of *Spec* is covered by at least one test case in the suite.

Notice that, due to limited control, a test case covering a transition cannot guarantee that the transition is actually executed in any test run. We can only assume that if the test case is executed a sufficient number of times, the transition will eventually be executed. This is a so-called fairness or *all-the-weather* assumption.

The following proposition states that it is sufficient to look into the traces of *Spec*, not the traces of $Q(Spec)$, to derive a transition cover test.

Proposition 2. For $Spec = <S, I, O, \lambda, S_0>$, a transition (s_1, a, s_2), and a test case $T(U_1)$, where $U_1 \subseteq traces(Q(Spec))$, that covers the transition, there exists a test case $T(U_2)$, where $U_2 \subseteq traces(Spec)$, that also covers the transition; moreover, the test length of $T(U_2)$ does not exceed that of $T(U_1)$.

Proof: According to the definition of a test case covering a transition of *Spec*, there is a trace u of $T(U_1) \parallel (Q_I \parallel Spec' \parallel Q_{O'})_\delta$ such that $u_{\downarrow I' \cup O'}$ covers transition (s_1, a', s_2) of $Spec'$. Therefore, $(u_{\downarrow I' \cup O'})'$ is a trace of *Spec* and covers (s_1, a, s_2). Let $U_2 = \{(u_{\downarrow I' \cup O'})'\}$ (a singleton) and $T(U_2)$ be the test case for the OBTT U_2. $T(U_2)$ is a test case covering (s_1, a, s_2) because $a_1a_1'a_2a_2'a_3a_3'\ldots$, where $a_1a_2a_3\ldots = (u_{\downarrow I' \cup O'})'$, is a trace of $T(U_1) \parallel (Q_I \parallel Spec' \parallel Q_{O'})_\delta$ and $(a_1a_1'a_2a_2'a_3a_3'\ldots)_{\downarrow I' \cup O'} = a_1'a_2'a_3'\ldots = u_{\downarrow I' \cup O'}$ covers (s_1, a', s_2).

Moreover, if $T(U_1)$ executes $u_{\downarrow I' \cup O}$, the traces executable by *Spec* is a subset of $pref(delay[I](u_{\downarrow I' \cup O}O^*))$, so $(u_{\downarrow I' \cup O'})' \in pref(delay[I](u_{\downarrow I' \cup O}O^*))$ ($(u_{\downarrow I' \cup O'})'$ is a trace of *Spec*). Thus, $T(U_2)$'s test length $|(u_{\downarrow I' \cup O'})'_{\downarrow I}|$ is equal to or less than $|(u_{\downarrow I' \cup O})_{\downarrow I}|$. Since $u_{\downarrow I' \cup O}$ is a trace executable by $T(U_1)$, the test length of $T(U_2)$ does not exceed that of $T(U_1)$. QED

Intuitively, if a tester $T(U)$ can execute a trace u of *Spec*, then it is possible that the trace u' is executed by $Spec'$ in the closed system $T(U) \parallel (Q_I \parallel Spec' \parallel Q_{O'})_\delta$. As a result, $T(U)$ covers all transitions of *Spec* that are covered by u in *Spec*.

According to Proposition 2, to derive a test suite that is a transition cover test of *Spec*, we only have to find a set of traces that cover every transition of *Spec*. Since the trace set is based on a regular language, i.e., $traces(Spec)$, there is an algorithm to derive such a set. Here, we propose an algorithm to derive a transition cover test with the shortest (in terms of the test length) test cases.

Procedure 1. To derive a transition cover test for *Spec*

Input: $Spec = <S, I, O, \lambda, S_0>$

Output: A transition cover test of *Spec* $\{T(U_1), T(U_2), \ldots\}$

Step 1: Let $U = \varnothing$, $V = O^{|S|-1} \cap traces(Spec)$. While *Spec* has a transition not covered by traces in U, do:

Step 1.1: for each v $\in V$:

add v to U if v covers a transition that is not covered by any trace in U;

Step 1.2: let $V = VIO^{|S|-1} \cap traces(Spec)$;

end of the while-loop in Step 1.

Step 2: For each $u \in U$

delete u from U if $u \in pref(U \setminus \{u\})$.

Step 3: Let $i = 1$. While $U \neq \varnothing$, do:

Step 3.1: let $U_i = U$;

Step 3.2: for each pair of traces $\beta, \beta a \in pref(U_i)$ such that $a \in I$, let $U_i = U_i \setminus \beta(I \cup O \setminus \{a\})(I \cup O)^*$;

Step 3.3: let $U = U \setminus U_i$ and $i = i + 1$;

end of the while-loop in Step 3.

Step 4: Use Definition 3 to build test cases $T(U_1)$, $T(U_2)$, …, and return the test suite $\{T(U_1), T(U_2), \ldots\}$.

Step 1 of Procedure 1 implements a breadth-first search for the traces covering transitions of *Spec* with the shortest input projections. The resulting traces are stored in the set *U*. Step 2 deletes the traces that are prefixes of other traces in *U*. Step 3 groups traces in *U* to derive the output-branching trace trees, since we need two different test cases for two traces in *U* that have a common prefix followed by two different actions, at least one of which is an input action. Step 4 builds test cases according to the trace trees.

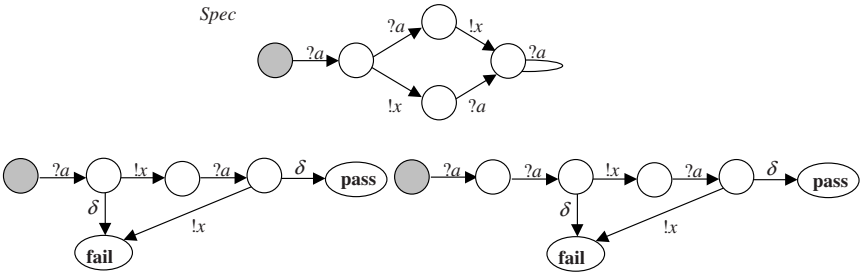

Fig. 5. A specification IOTS and a transition cover test for it

Example 1. Fig. 5 shows a specification and a transition cover test derived by Procedure 1. After Step 1 is applied, *U* could be {*a*, *ax*, *aa*, *aax*, *axa*, *aaxa*}. After Step 2, traces *a*, *ax*, *aa*, and *aax* are deleted from *U* because *a*, *ax* ∈ *axa* and *aa*, *aax* ∈ *aaxa*, respectively. Step 3 separates *axa* and *aaxa* into two different output branching trace trees, and Step 4 builds the test cases as shown in the figure.

Notice that, due to the fact that the order of traces examined in Step 1.1 is not fixed, other transition cover tests could also be derived by the procedure. For example, we could have *U* = {*a*, *ax*, *axa*, *aa*, *aax*, *axaa*} after Step 1 and, accordingly, *U* = {*aax*, *axaa*} after Step 2. The largest test length of the test cases (three) is not influenced, however, by this non-deterministic choice of traces.

In each step of Procedure 1, set *U* has at most $|\lambda|$ traces, each with the input projection of at most $|S|$ actions, where $|\lambda|$ and $|S|$ are the number of transitions and the number of states in *Spec*, respectively. The reason is that any transition of *Spec* can be covered by a trace of at most $|S|$ actions, and each trace in *U* covers at least one new transition. Therefore, Procedure 1 stops in finite steps for a finite state *Spec*, so the procedure is an algorithm that derives a test suite with at most $|\lambda|$ test cases, whose test lengths are at most $|S|$.

5 Testing Partially Specified IOTS through Unbounded Queues

In this section, we assume that *Spec* is partially specified and input-progressive, whereas *Imp* is fully specified.

For partially specified IOTS, we might have some unexpected results when the test cases by Definition 3 are executed.

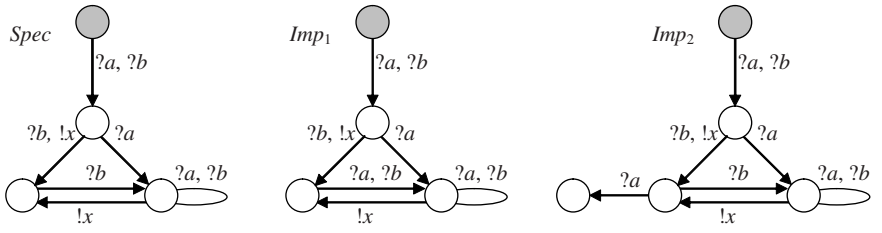

Fig. 6. A partially specified IOTS *Spec* and two implementations

Example 2. In Fig. 6, since the transition on input action *a* is not specified in *Spec* after sequence *ax*, Imp_1 should be considered a correct implementation for *Spec*. However, Imp_1 fails a test with trace *aax*. According to Definition 3, the test case will reach the verdict **fail** if trace *aaxx* is observed. However, $aaxx \in delay[O](axax)$, so the trace can be observed when testing Imp_1.

There are several solutions to this problem when testing partially specified IOTS, one of which is to complete the specification and let any implementation pass a test if unspecified input actions are executed ([6] and [2]). However, when the closed system of our testing architecture is considered (Fig. 3), this solution may cause some problems.

Example 3. In Fig. 6, Imp_2 should also be a correct implementation of *Spec*. When a test case with trace *aaxbx* is executed, Imp_2 may execute the trace *axa*. This trace leads Imp_2 to a deadlock state, where input action *b* is blocked. According to Definition 3, the tester observes premature quiescence: $aaxb\delta$.

The problem in Example 3 is inherent to the solution of completing the specification because the intuition behind the completion operation is that all input actions should be accepted by the IUT, which is not the case if we consider systems that can refuse to read their input actions from queues. Intuitively, the problem can be solved by restraining the tester from applying any input that might not be specified in *Spec*. As a result, some transitions may not be testable, in other words, covered by the resulting test cases.

To avoid executing unspecified input actions of *Spec*, traces covering the transitions to the trap state s^t_{Spec} of *Comp*(*Spec*) should be excluded from consideration. If s^t_{Spec} is reached, there is no need to further test *Spec* as its behavior might not be specified.

A state of $Q(Comp(L))$ is an *exception state* if it has the trap state s^t_L as a substate. We define the set of all exception states of $Q(Comp(L))$ as $S_E(L)$. The *exception trace set* of $Q(Comp(L))$ is defined as $etraces(L) = \{u \in traces(Q(Comp(L))) \mid Q(Comp(L))\text{-}\mathbf{after}\text{-}u \cap S_E(L) \neq \emptyset\}$. The *exception suspension trace set* of $Q(Comp(L))$ is defined as $estraces(L) = \{u \in (I \cup O \cup \{\delta\})^* \mid \exists v \in pref(u) \cap straces(Q(Comp(L)))$ s.t. $Q(Comp(L))\text{-}\mathbf{after}\text{-}v \cap S_E(L) \neq \emptyset\}$. Due to the definition of operator *Comp*, $estraces(L)$ contains sequences that are not in $straces(Q(Comp(L)))$, but their prefixes are in $straces(Q(Comp(L)))$ and lead $Q(Comp(L))$ to some exception states.

The conformance relations introduced in Section 4 are rewritten below to account for partially specified *Spec*:

Definition 4. For *Spec*, *Imp* ∈ *IOTS*(*I*, *O*),

- *Imp* is *queue-context trace included* into *Spec* if *traces*(*Q*(*Imp*)) \ *etraces*(*Spec*) ⊆ *traces*(*Q*(*Comp*(*Spec*))) \ *etraces*(*Spec*);
- *Imp* is *queue-context quiescent trace included* into *Spec* if *traces*(*Q*(*Imp*)) \ *etraces*(*Spec*) ⊆ *traces*(*Q*(*Comp*(*Spec*))) \ *etraces*(*Spec*) and *qtraces*(*Q*(*Imp*)) \ *etraces*(*Spec*) ⊆ *qtraces*(*Q*(*Comp*(*Spec*))) \ *etraces*(*Spec*);
- *Imp* is *queue-context suspension trace included* into *Spec* if *straces*(*Q*(*Imp*)) \ *estraces*(*Spec*) ⊆ *straces*(*Q*(*Comp*(*Spec*))) \ *estraces*(*Spec*).

When *Spec* is fully specified, Definition 4 reduces to Definition 2. In Definition 4, the exclusion of *Spec*'s exception sets eliminates all unspecified behavior, so that the rest of *Imp* should be specified by *Spec*. Based on this discussion, we refine the definition of output-branching trace trees.

Definition 5. For *Spec* ∈ *IOTS*(*I*, *O*), an *output-branching trace tree* of *Spec* is a finite set of traces $U \subseteq traces(Q(Spec)) \setminus etraces(Spec)$ that satisfies the following conditions: for $\forall\ u_1, u_2 \in U$, there exist $v, w_1, w_2 \in (I \cup O)^*$ and $b_1, b_2 \in O$ $(b_1 \neq b_2)$ such that $u_1 = vb_1w_1$ and $u_2 = vb_2w_2$.

At the same time, Definition 3 still applies to test cases for partially specified IOTS.

Similar to Section 4, we want to derive a transition cover test of *Spec* with respect to the queue-context quiescent trace inclusion relation. The definitions of a trace and test case covering a transition of *Spec* remain the same as in Section 4. The definition of transition cover test, on the other hand, should take into account exception traces. Formally, a transition of *Spec* is *coverable* if there is a test case that covers the transition. A test suite is a *transition cover test* of *Spec* if each coverable transition of *Spec* is covered by at least one test case in the suite.

The following statement is the generalization of Proposition 2 to the case of partially specified *Spec*.

Proposition 3. For $Spec = \langle S, I, O, \lambda, S_0 \rangle$, a coverable transition (s_1, a, s_2), and a test case $T(U_1)$, where $U_1 \subseteq traces(Q(Spec)) \setminus etraces(Spec)$, that covers the transition, there exists a test case $T(U_2)$, where $U_2 \subseteq traces(Spec)$, that also covers the transition; moreover, the test length of $T(U_2)$ does not exceed that of $T(U_1)$.

Proof: According to the definition of a test case covering a transition of *Spec*, there is a trace u of $T(U_1) \parallel (Q_I \parallel Spec' \parallel Q_{O'})_\delta$ such that $u_{\downarrow I' \cup O'}$ covers transition (s_1, a', s_2) of *Spec'*. Similar to the proof of Proposition 2, we let $U_2 = \{(u_{\downarrow I' \cup O'})'\}$, and prove that $T(U_2)$ is a test case covering (s_1, a, s_2), and the test length of $T(U_2)$ does not exceed that of $T(U_1)$. The only difference from that proof is that we now have to prove $(u_{\downarrow I' \cup O'})' \notin etraces(Spec)$ so that U_2 is an OBTT according to Definition 5.

The states reachable by *Spec*, after the tester executes a trace v and the input queue is empty, is *Comp*(*Spec*)-**after**-*delay*[*I*](*vO**) (see Section 3), so checking whether $v \in etraces(Spec)$ is equivalent to checking whether the trap state $s^t_{Spec} \in Comp(Spec)$-**after**-*delay*[*I*](*vO**).

Suppose $(u_{\downarrow I' \cup O'})' \in etraces(Spec)$, then $s^t_{Spec} \in Comp(Spec)$-**after**-*delay*[*I*]($(u_{\downarrow I' \cup O'})'O^*$), which implies that $s^t_{Spec} \in Comp(Spec)$-**after**-*delay*[*I*]($u_{\downarrow I \cup O}O^*$) (because $(u_{\downarrow I' \cup O'})' \in pref(delay[I](u_{\downarrow I \cup O}O^*))$), which in turn implies that $u_{\downarrow I \cup O} \in$

etraces(*Spec*). This result, however, contradicts the fact that $u_{\downarrow_{I\cup O}} \in pref(U_1) \subseteq traces(Spec) \setminus etraces(Spec)$. Therefore, $(u_{\downarrow_{I\cup O}})' \notin etraces(Spec)$. QED

Attempts to generalize Procedure 1 to the case of partially specified *Spec* faces the problem that it is unknown how to determine which transitions of *Spec* are coverable because *etraces*(*Spec*) is not regular.

Here, we take a pragmatic approach by restricting the test length of the test cases, which allows us to determine which transitions are coverable with the given constraint. Formally, a transition of *Spec* is *k-coverable* if there is a test case $T(U)$ with test length k that covers the transition. We can verify that if a transition is k-1 coverable, it is k-coverable. A test suite is a *transition k-cover test* if all *k-coverable* transitions are covered by at least one test case in the suite.

Given a bound k, we can now generalize Procedure 1 to derive a transition k-cover test for a partially specified *Spec*. Intuitively, this can be done by examining traces of *Spec* incrementally to derive the test cases whose test lengths are not larger than k. When examining trace $va \in traces(Spec)$, where $a \in I$, we have to verify whether va belongs to *etraces*(*Spec*). Similar to the proof of Proposition 3, checking whether $va \in etraces(Spec)$ is equivalent to checking whether $s^t_{Spec} \in Comp(Spec)$-**after**-$delay[I](vaO^{I(va)})$. Since $vaO^{I(va)}$ is a finite set, $delay[I](vaO^{I(va)})$ is a finite set, too. The problem of verifying whether $va \in etraces(Spec)$ is, therefore, decidable.

Based on the discussions above, we have the following algorithm.

Procedure 2. To derive a transition k-cover test for *Spec*

Input: $Spec \in IOTS(I, O)$ and bound k

Output: A transition k-cover test of *Spec* $\{T(U_1), T(U_2), \ldots\}$

Step 1: Let $U = \varnothing$, $V = O^{|S|-1} \cap traces(Spec)$, $i = 0$. While *Spec* has a transition not covered by traces in U and $i \le k$, do:

  Step 1.1: for each $v \in V$

    add v to U if v covers a transition that is not covered by any trace in U;

  Step 1.2: let $V = VIO^{|S|-1} \cap traces(Spec)$, for $v \in V$:

    delete v from V if $s^t_{Spec} \in Comp(Spec)$-**after**-$delay[I](vO^{I(v)})$;

  Step 1.3: $i = i + 1$;

end of the while-loop in Step 1.

Step 2: For each $u \in U$

  delete u from U if $u \in pref(U \setminus \{u\})$.

Step 3: Let $i = 1$. While $U \neq \varnothing$, do:

  Step 3.1: let $U_i = U$.

  Step 3.2: for each pair of traces $\beta, \beta a \in pref(U_i)$ such that $a \in I$, let $U_i = U_i \setminus \beta(I \cup O \setminus \{a\})(I \cup O)^*$;

  Step 3.3: let $U = U \setminus U_i$ and $i = i + 1$;

end of the while-loop in Step 3.

Step 4: Use Definition 3 to build test cases $T(U_1)$, $T(U_2)$, … , and return the test suite $\{T(U_1), T(U_2), \ldots\}$.

Step 1 of Procedure 2 is different from that of Procedure 1: it stops when the bound k is reached and checks whether a trace in set V belongs to *etraces*(*Spec*).

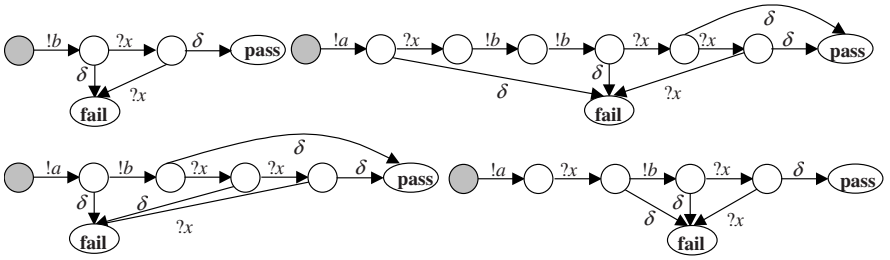

Fig. 7. A transition cover test for the *Spec* in Fig. 7

Example 5. Fig. 7 shows a transition 3-cover test derived by Procedure 2 for the *Spec* in Fig. 6. All transitions on action a, except the one from the starting state, are not covered by this suite. However, it can be proved that these transitions on a are, in fact, not coverable, so the obtained test suite is a transition cover test for the *Spec*.

To obtain the test suite in Fig. 7, $U = \{a, ax, b, ab, axb, axbx, axbb\}$ after Step 1, and $U = \{b, ab, axbx, axbb\}$ after Step 2.

6 Testing IOTS through Bounded Queues

The test suites derived by Procedures 1 and 2 have finitely many test cases, and each test case, by definition, has finite behavior. Therefore, once a test suite is constructed, we can calculate the capacity of the queues used when a conforming implementation of an input-progressive specification is tested. This suggests that bounded queues (i.e., queues with bounded capacity) can be used in the testing architecture (Fig. 3) for a given test suite. Intuitively, this can be done by requiring that the queue capacities be large enough so that the queues do not overflow when a conforming IUT is tested. Since we assume that *Spec* is input-progressive, there is a lower bound of the queue capacities to meet the condition. When the output queue overflows, on the other hand, we are sure that a non-conforming IUT is tested.

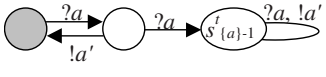

Fig. 8. A bounded queue with capacity 1 and a single input action a

We start with the definition of bounded queues. Formally, a *bounded queue with input set A and capacity n*, $Q_{A\text{-}n}$, is a deterministic IOTS $\langle S_{A\text{-}n} \cup \{s^t_{A\text{-}n}\}, A, A', \lambda_{A\text{-}n}, \{\varepsilon\}\rangle$, where the set $S_{A\text{-}n} = \{u \in A^* \mid |u| \le n\}$, and the transition relation $\lambda_{A\text{-}n} = \{(u, a, ua) \mid u, ua \in S_{A\text{-}n}\} \cup \{(av, a', v) \mid av, v \in S_{A\text{-}n}\} \cup \{(u, a, s^t_{A\text{-}n}) \mid u \in S_{A\text{-}n}, |u| = n, \text{ and } a \in A\} \cup \{(s^t_{A\text{-}n}, a, s^t_{A\text{-}n}) \mid a \in A \cup A'\}$. The trap state $s^t_{A\text{-}n}$ once reached indicates that the queue overflows. Fig. 8 shows an example of a bounded queue, $Q_{\{a\}\text{-}1}$.

According to the definition, bounded queues are input-enabled, so the compatibility conditions of the testing architecture (Section 3) are not violated when we replace unbounded queues with bounded ones.

When bounded queues are used, the execution of a test case *Test* can be described by $Test \parallel (Q_{I\text{-}n} \parallel Imp' \parallel Q_{O'\text{-}m})_{\delta}$. The queue capacities n and m, for input and output queues, respectively, should be large enough to ensure that there is no state of $Test \parallel (Q_{I\text{-}n} \parallel Spec' \parallel Q_{O'\text{-}m})_{\delta}$ that has $s^t_{I\text{-}n}$ or $s^t_{O'\text{-}m}$ as a substate. The value n can always be taken equal to the longest test length of the test cases, which is at most $|S|$ or k for fully or partially specified *Spec*, respectively, and m could be estimated similarly. Further reducing the estimated value of n and m is an optimization problem and is not discussed here due to the lack of space.

7 Conclusion

In this paper, we discussed how to derive tests when testing (not necessarily input-enabled) IOTS through lossless queues with the intention to cover each transition of a non-deterministic specification. We introduced the testing architecture, along with the compatibility condition at each interface. Although the traces of IOTS composed with a pair of unbounded queues are usually not regular, we demonstrated that the transition cover test can be derived from a regular language, namely, the trace set of the specification. We first considered testing fully specified IOTS and formalized the basic ideas of the test derivation algorithm. Treating partially specified IOTS, where coverability of transitions is still an open problem, we took a pragmatic approach by restricting the test length of test cases. Based on the test suites derived for testing IOTS through unbounded queues, we discussed how to use bounded queues in the testing architecture, which makes our method more practical.

Our work differs from previous work by distinguishing between blocked and unspecified input actions, which brings our model closer to the real-world designs of communicating systems. IOTS in this paper can have states that block input actions, where the system does not read its input queue. Unspecified input actions are treated as “don’t care” situation in the specification, so our test derivation method does not require fully specified designs.

An important contribution of this paper is that we use a widely used (transition) coverage criterion to derive a finite test suite directly from a specification. This allows us to avoid either explicitly composing the specification with unbounded queues (as in [14]) or devising IUT with a local observer (as in [6]).

Concerning future work, it is interesting to explore how to determine whether a transition of a partially specified IOTS is coverable, so that we could estimate the bound k for which a transition k-cover test is a transition cover test. Moreover, it is still unknown, when testing IOTS through unbounded queues, whether there is an algorithm to derive test cases based on some fault models. Also, we only demonstrated how to derive transition cover tests for the queue-context quiescent trace inclusion relation. It would be interesting to derive tests for more stringent conformance relations, e.g., queue-context suspension trace inclusion. A foreseeable difficulty of this work is that intermediate quiescence of a system is not always observable through the queue contexts. Some other conformance relations could be tested by exploiting the ability to observe overflow of bounded queues. Consider two

single-state IOTS, where both states are stable. One IOTS is input-enabled, whereas the other deadlocks. They are not distinguishable according to the conformance relations in this paper. In fact, they cannot be distinguished when tested through unbounded queues. On the other hand, if a bounded input queue is used with the IOTS that deadlocks, the queue could eventually overflow, so the deadlocking IOTS is distinguishable from the input-enabled one. Last, but not least, although the test cases derived by Procedures 1 and 2 are the shortest in terms of test lengths, the number of test cases is not the smallest. Some test cases are redundant because the transitions that they cover are covered by other test cases. Further reducing the number of test cases is an optimization problem still under investigation.

Acknowledgement. This work was in part supported by the NSERC discovery grant OGP0194381.

References

1. Balemi, S.: Control of Discrete Event Systems: Theory and Application. Ph.D. thesis, Swiss Federal Inst. of Technology, Zurich, Switzerland (1992)
2. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional Testing with IOCO. In: Proc. 3rd Intl. Workshop on Formal Approaches to Testing of Software, FATES 2003. Canada (2003)
3. Brinksma, E., Tretmans, J.: Testing Transition Systems: An Annotated Bibliography. In: Cassez, F., Jard, C., Rozoy, B., Ryan, M. (eds.): Modeling and Verification of Parallel Processes. Lecture Notes in Computer Science, Vol. 2067. Springer-Verlag, Berlin Heidelberg New York (2001)
4. Clerbout, M., Latteux, M., Roos, Y.: Semi-Commutations. In: Diekert, V., Rozenberg, G. (Eds.): The Book of Traces. World Scientific (1995)
5. Herrink, L., Tretmans, J.: Refusal Testing for Classes of Transition Systems with Inputs and Outputs. In: Mizuno, T., Shiratori, N., Higashino, T., Togashi, A. (Eds.): Formal Description Techniques and Protocol Specification, Testing and Verification. Chapman & Hill (1997)
6. Jard, C., Jéron, T., Tanguy, L., Viho, C.: Remote Testing Can be as Powerful as Local Testing. In: The Proceedings of the IFIP Joint International Conference, Methods for Protocol Engineering and Distributed Systems, FORTE XII/PSTV XIX. China (1999)
7. Lynch, N., Tuttle, M. R.: An Introduction to Input/Output Automata. In: CWI Quarterly, Vol. 2, No. 3 (1989)
8. Petrenko, A., Yevtushenko, N., Huo, J. L.: Testing Transition Systems with Input and Output Testers. In: Proc. IFIP 15th Int. Conf. Testing of Communicating Systems. TestCom'2003, France. Lecture Notes in Computer Science, Vol. 2644. Springer-Verlag, Berlin Heidelberg New York (2003)
9. Phalippou, M.: Executable Testers. In: The Proceedings of the IFIP Sixth International Workshop on Protocol Test Systems, IWPTS'93. France (1993)
10. Segala, R.: Quiescence, Fairness, Testing and the Notion of Implementation. In: The Proceedings of CONCUR'93. Lecture Notes in Computer Science, Vol. 715. Springer-Verlag, Berlin Heidelberg New York (1993)
11. Tan, Q. M., Petrenko, A.: Test Generation for Specifications Modeled by Input/Output Automata. In: The Proceedings of the IFIP 11th International Workshop on Testing of Communicating Systems, IWTCS'98. Russia (1998)

12. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. In: Software-Concepts and Tools, Vol. 17, Issue 3 (1996)
13. Tretmans, J., Verhaard, L.: A Queue Model Relating Synchronous and Asynchronous Communication. In Linn, R. J., Jr., Üyar, M. Ü. Eds.: Protocol Specification, Testing and Verification, XII. Elsevier Science Publishers B. V. (North-Holland) (1992)
14. Verhaard, L., Tretmans, J., Kim, P., Brinksma, E.: On Asynchronous Testing. In: The Proceedings of the IFIP 5th International Workshop on Protocol Test Systems, IWPTS'92. Canada (1992)