Testing of Partial Order Input/Output Automata

Gregor v. Bochmann[1], Guy-Vincent Jourdan[1], Stefan Haar[2]

[1] School of Information Technology and Engineering (SITE)
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada, K1N 6N5
{bochmann,gvj}@site.uottawa.ca

[2] IRISA/INRIA
Rennes, France
Stefan.Haar@irisa.fr

Abstract. An Input/Output Automaton is an automaton with a finite number of state where each transition is associated with a single input or output interaction. In this paper, we consider a generalization of this formalism, the Partial Order Input/Output Automata (POIOA), in which each transition is associated with a partially ordered set of inputs and outputs. This new formalism allows the specification of concurrency between inputs and outputs in a very general, direct and concise way. In this paper, we give a formal definition of this formalism, and define several conformance relations for comparing system specifications expressed in this formalism. Then we show how to derive a test suite that guarantees to detect faults defined by a POIOA-specific fault model: transfer faults, missing output faults, unspecified output faults, weaker precondition faults and stronger precondition faults.

Keywords: testing distributed systems, partial order, finite state automata, conformance relations, partial order automata

1 Introduction

Finite State Machines (FSM) are commonly used to model sequential systems. When modeling concurrent systems, other models have been used, such as multi-port automata [13], where several distributed ports are considered and an input at a given port can generate concurrent outputs at different ports. The multi-port automata model is however not really adapted for truly distributed systems, since input concurrency is not taken into account. In [1], a new model of automata is introduced, where each transition is equipped with a *bipartite partially ordered set*, consisting of a set of concurrent inputs and a set of causally related concurrent outputs. This new model provides the ability to directly and explicitly specify concurrency between inputs, and causal relationships between inputs and outputs. A testing method for this new model was also proposed. Even though the model is up to exponentially smaller than the equivalent multi-port model, in a case of an automata having an adaptive

distinguishing sequence, the testing method proposed is able to generate a checking sequence which is polynomial in the number of inputs, and thus up to exponentially shorter than a checking sequence generated for an equivalent specification written as a multi-port automaton.

This model still has the limitation that no order constraint can be defined for the concurrent inputs of a given transition. In this paper we present a more general model where order constraints can be defined for inputs as well as outputs for a given transition. This provides a more symmetrical framework which simplifies the composition of several automata. The order constraints for inputs defined for a given automaton can then be interpreted as assumptions that are made about the behavior of the automaton's environment. A transition is therefore characterized by a multi-set of input/output events, where certain input or output interactions may occur several times, and a partial order between these events. We assume, however, that a transition starts with inputs and that there is no conflict between the initial inputs of different transitions starting from the same automaton state.

We explain in this paper how the testing method that was defined for the previous model can be extended to our general case in an efficient manner. The basic idea is as follows: In order to test the order constraints imposed by a given input on the outputs of a given transition, first all inputs that may be applied (according to the partial order of the transition) before the given input, are applied and the resulting outputs are recorded. Then the given input is applied and the resulting outputs are recorded. A given output will occur in the first set of observed outputs if and only if it has no order constraint depending on the given input. The tests concerning the different inputs of a given transition can be combined into several test cases. However, several test cases are in general required to completely test the implemented partial order between inputs and outputs. Finally, the well-known methods for testing finite state machines can be used in our context for identifying the states of the automaton and to bring the implementation into the starting state from where a given transition can be tested.

In Section 2 of this paper, we first give an intuitive explanation of our model of Partial-Order Input/Output Automata (POIOA), and then give a formal definition. We also discuss different criteria for comparing the partial orders of different transitions, and based on this, how the behavior of different POIOA can be compared. In particular, we consider that the specification of a system component is given in the form of a POIOA M1, as well as an implementation of this component in the form of a POIOA M2. We define a conformance relation, called quasi-equivalence, which states that, if satisfied between M2 and M1, the implementation provides the outputs in an order satisfying the specification, provided that the environment of the component presents the inputs in an order satisfying the specification.

In Section 3, we present the testing methodology in detail and show that all faults defined by a given fault model are detected by the derived test sequence. We also indicate how the results observed during the tests can be used to diagnose specific faults within the implementation. Then, in Section 4, we provide some discussion of the assumptions we have to make about the implementation in order to assure the detection of all faults by our testing method. We also discuss how one can monitor the order between outputs in respect to one another, which may be defined by the specification or be implemented in a particular manner in the implementation. There

are no simple test cases for this purpose because the implementation may behave in a non-deterministic manner. Finally, we discuss the assumptions we have made for our POIOA specification model. We give some justification for these assumptions and also discuss why it may be interesting to remove some of these assumptions in future work.

2 Partial Order Input/Output Automata

2.1 Basic concepts

An **Input/Output Automaton (IOA)** is an automaton with a finite number of states where each transition is associated with a single input or output interaction [2]. In [1] we considered the testing of so-called "Input/Output Partial Automata" where each transition is associated with one or more, possibly concurrent inputs and zero, one or more outputs that are related to the inputs by a given dependency relation. For instance, an output o_2 for a transition t may only occur after the occurrence of two inputs i_1 and i_2, while output o_1 may occur as soon as input i_1 has occurred.

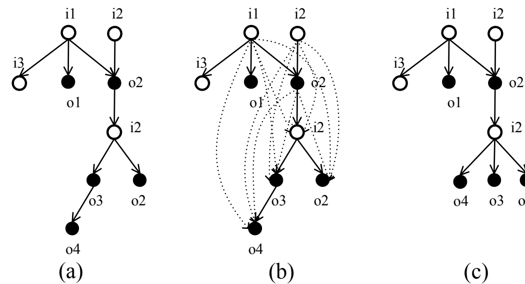

Figure 1: partially ordered multisets of input and output events

In this paper we consider a more general form of automata where each transition is associated with a partially ordered set of input and output events. For instance as shown in Figure 1(a), a given transition t may be associated with the following set of events: inputs i_1, i_2 (occurring twice), i_3, and outputs o_1, o_2 (occurring twice), o_3 and o_4 for which the following ordering relations must hold: $i_1 < o_1$; i_2 (first occurrence) $< o_2$ (first occurrence); $i_1 < o_2$ (first occurrence); o_2 (first occurrence) $< i_2$ (second occurrence); $i_1 < i_3$; i_2 (second occurrence) $< o_3$ and o_2 (second occurrences), and $o_3 < o_4$.

Here the notation *event-1* < *event-2* means that there is an order constraint between the occurrence of *event-1* and *event-2*, namely, that *event-2* occurs after *event-1*. The order between the events of a transition represent two aspects: (a) a safety properties, and (b) a liveness properties, sometimes called "progress properties". The order *event-*

1 < event-2 implies the safety property stating that event-2 will only happen after event-1 has already happened. If *Earlier-2* is the set of events *e* of the transitions for which *e < event-2* holds, then the order of events implies that *event-2* will eventually occur when all events in *Earlier-2* have occurred. We note that often, as shown in Figure 1(a), we are only interested in the basic order constraints from which the complete order relationship can be constructed by transitivity. For instance, Figure 1(b) shows the complete order relationship generated by the constraints shown in Figure 1(a).

It is clear that such specifications of partial order input/output automata (POIOA) allow for much concurrency between the inputs and outputs belonging to the same transition. We believe that this is an important feature for describing distributed systems. In fact, it is often difficult to determine, in a distributed system, in which order in time two particular events occur if they occur in different points in space. Therefore one may ask the question how it would be possible to check whether two interactions, for instance i_1 and i_3, occur in the order specified (for instance in the order $i_1 < i_3$ as specified above). One way to bring some rigor into this question is to introduce ports, sometimes called interaction points or service access points, and to associate each input/output event with a particular port of the distributed system. Then one may assume that the order of events at each port can be determined, while the order of events occurring at different ports can not be determined. The situation is similar in UML sequence diagrams, were vertical lines represent different system components and events belonging to the same component are normally executed in sequential order while the order of events at different components is only constrained by message transmissions. – In this paper we do not introduce ports nor system components. We simply assume that the order of execution of two events can be determined if a particular order of execution is specified for them.

For a **reactive** automaton, in the following called **input-guarded**, we assume that all initial events of each transition are inputs. We call an event of a transition initial if there is no other event that must precede it. In general, for so-called **active** automata, there may be transitions that can start with the production of an output. This corresponds to an output transition in a classical IOA. In the following, we normally assume that the considered automaton is input-guarded.

In order to allow for a straightforward determination of the next transition of a POIOA, we assume that the following condition is satisfied concerning the initial events of different transitions starting from the same state: The set of initial events of two transitions starting from the same state must be disjoint. We say that the automaton has **exclusive transitions**.

In addition, we assume that each state of the automaton is a "strong synchronization point", that is, the initial input for the next transition will only become available after all events of the previous transition have occurred. We note, however, that this assumption may not always be realistic in a distributed system; and one may consider a distributed model with several local components where the sequential order between transitions is weak sequencing, that is, events pertaining to the next transition may occur at a given component after all **local** events of the previous transitions have (locally) occurred. This weak sequencing semantics has been adopted for High-Level Message Sequence Charts (HMSC) [3], where the "local" events are those pertaining to a given system component, as for UML

sequence diagrams. In fact, the model of HMSC is similar to our model of POIOA: A HMSC is a kind of state diagram where each state represents the execution of a sequence diagram (MSC) and the transition from one state to another represents the sequential execution of the two associated MSCs with weak sequencing semantics. In the POIOA model a transition can be equated to the partial order defined by a sequence diagram. For simplifying the semantics of the POIOA, we have assumed that the sequential execution of two transitions has strong sequencing semantics. It is to be noted that weak sequencing leads to many difficulties for the implementation of the specified ordering in a distributed environment, as discussed in many research papers [4,5,6,7].

As explained by Adabi and Lamport [8], the specification of the requirements that a system component must satisfy in the context of a larger system normally consists of two parts: (a) the assumptions that can be made about the behavior of the environment in which the component will operate, and (b) the guarantees about the behavior of the component that the implementation must satisfy. In the context of IOA (see for instance [9]), the guarantees are related to the output produced by the specified component (in relation to the inputs received), while the environment should satisfy certain assumptions about the order in which input to the component will be provided (in relation with the produced output earlier). In the case of a partially defined, state-deterministic IOA (where the state is determined by the past sequence of input/output events), the fact that in a given state some given input is not specified is then interpreted as the assumption that the environment will not produce this input when the component is in that given state.

During the testing of an implementation for conformance to an IOA specification, two types of problems may occur: After a given execution trace, that is, a given sequence of input and output events, the specification will be in a particular state. If the next event that occurs does not corresponds to a transition of the IOA specification then we have encountered a problem: If the event is an output, an implementation fault has been detected; if it is an input, this is an unexpected input, also called "unspecified reception", which represents a wrong behavior of the environment.

A specification of a system component C in the form of a POIOA S_C, similarly, can be interpreted as defining assumptions about the environment of C and guarantees that the implementation of C must satisfy. The difference between an IOA and a POIOA is that a transition of the latter is characterized by a set of input/output events with a defined partial order instead of a single input or output event. Similarly as for an IOA, one can define a dual specification for a given POIOA which represents the most general behavior of the environment and can be used for testing an implementation of the given specification.

It is clear that the behavior of a *POIOA* S can be modeled by an *IOA* S' as follows: The states of S' include the states of S and a large number of intermediate states that correspond to the partial execution of a transition. For instance, the POIOA transition t shown in Figure 2(a) can be modeled by the IOA transitions shown in Figure 2(c). The conformance testing of an implementation in respect to a specification S may therefore be performed by a test suite that checks the performance in respect to S' and that is obtained by one of the known test development methods for finite state machines or IOA. However, this approach is not very efficient since the equivalent

IOA specification S' is in general much more complex than the original POIOA specification S.

We propose in this paper a method for deriving a test suite that guarantees to detect all faults defined by a POIOA-specific fault model. Specifically, we propose to test an implementation for the following faults:

- *Transfer fault:* A transition of the implementation leads to a state different from what is specified.
- *Unspecified output:* An output produced during a given transition is not in the set of outputs specified; or the number of occurrences of an output is larger than allowed by the specification.
- *Missing output*: An output foreseen by the transition is not produced after all possible inputs (those that could be applied before that output according to the specification) have been applied.
- *Weaker precondition:* An output foreseen by a transition is produced before all the events that are specified as precondition have occurred.
- *Stronger precondition:* An output foreseen by a transition is not produced after all its precondition events have occurred, but it is produced after certain other expected input events have been applied.

Transfer faults are tested by simply adapting the classical state recognition and transition verification methods from IOAs to POIOAs [1]. Testing for the other faults require new techniques, especially the weaker and stronger precondition faults. Our approach to catch these types of faults is to test each input event of a given transition separately. For each input event i of the tested transition, we proceed in two steps. First of all, we apply to the implementation all input events that are not *after* i in the partial order of the transition (that is, inputs that are either before i or concurrent to i), and we observe the produced outputs. The set of outputs that we observe are not pre-conditioned by i in the implementation, since they are generated even though i has not been input yet. If one of these observed outputs is *supposed* to have i as precondition according to the POIOA specification, then we have detected a case of weaker precondition fault in the implementation. During the second step input i is applied and the subsequent outputs are observed. These outputs are pre-conditioned by i in the implementation, since they were not produced until after i was input. If one of these observed inputs was *not supposed* to have i as precondition according to the POIOA specification, then we have detected a case of stronger precondition fault in the implementation. We repeat this process for all inputs of all transitions.

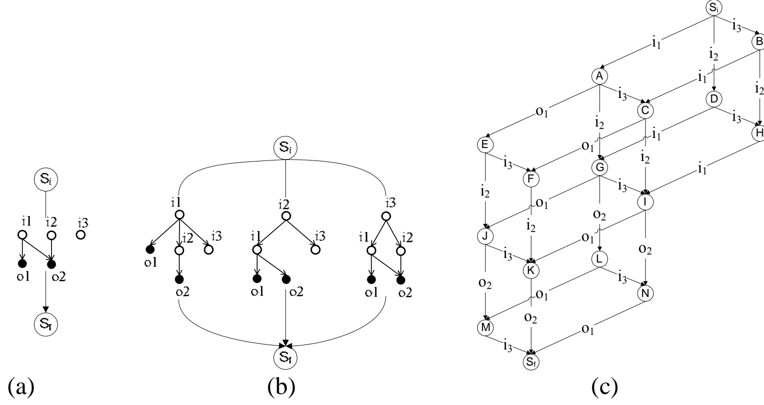

Figure 2: (a) specification of transition. (b) implementation of the specified transition in terms of three separate transitions. (c) An Input/Output Automata model equivalent to the POIOA transition shown in (a)

2.2 Formalization of the automata model

In the following, we suppose that two disjoint nonempty sets I and O (representing inputs and output) are given; set $V= I \cup O$. Recall that a *partial order* is a pair (E, $\leq$) where E is a set and $\leq$ *is* a transitive, antisymmetric and reflexive binary relation on E, and $<$ is the irreflexive part of $\leq$. The set of *minimal* elements of E is the set $min(E)=\{x\in E,$ for all $z\in E,\ z \leq x$ implies $z=x\}$.

Definition: *An **V-pomset** (partial order multi-set) is a tuple $\omega= (E, \leq,\mu)$ such that*

1. *$(E, \leq)$ is a partial order, and*
2. *μ: $E\rightarrow V$ a total mapping called the **labeling.***

A POIOA is a special kind of finite state transition machine. A POIOA has a finite number of states and transitions, and each transition is associated with a partial order of input and output events, as follows:

Definition: *A **Partial Order Input/Output Automaton** (or **POIO Automaton, POIOA**) is a tuple $M= (S, s^{in}, I, O,T)$, where*

1. *S is a finite set of **states** and $s^{in}\in S$ is the **initial state**; the number of states of M is denoted $n = |S|$;*
2. *I and O are disjoint and nonempty **input** and **output** sets, respectively ;*
3. *$T \subseteq S\times\Omega(I\cup O) \times S$ is the set of **transitions**, where $\Omega(I\cup O)$ is the set of all $(I\cup O)$-pomsets.*

*We say that an POIOA is **input-guarded** if for each transition the minimal events are all inputs.*

*We say that an POIOA has **exclusive transitions** if for any two transitions $t_1=(s, \omega_1,s_1)$ and $t_2=(s,\omega_2,s_2)$ starting from the same state s, we have*

$min(\omega_1) \cap min(\omega_2)= \varnothing$. Note: This implies that the next transition from a given state is determined by the first input that occurs.

*We say that a POIOA is **strongly synchronized** if all input/output events of one transition are performed before any event of the next transition of the automata occurs.*

We consider in this paper mainly strongly synchronized, input-guarded POIOA. The implications of these restrictions are discussed in Section 4.

In this paper we assume that a POIOA is the specification of a distributed system where the inputs and outputs may occur at different system interfaces. For the purpose of testing an implementation for conformance with a given POIOA specification, we also assume that we can model the implementation by a POIOA and that the implementation satisfies certain assumptions that can be modeled by restrictions on the form of the POIOA implementation model.

2.3 Comparing the behavior of different POIOA

In this section we first define several basic conformance relations that can be used for comparing the behavior of different POIOA, for instance the specification of a system and its implementation. It turns out that these basic relations correspond to the different types faults that an implementation may have. We then discuss the nature of these relations and define the quasi-equivalence relation that can be easily tested by the method described in Section 3.

We consider the following basic conformance relations between two pomsets ω_2 and ω_1 . We assume in the following discussion that ω_1 is associated with a transition in the system specification and ω_2 is associated with a corresponding transition in the model of its implementation.

- ω_2 has more **outputs constraints** (that is, has less outputs) than ω_1 (written $\omega_2 =>^{\neq}_{OC} \omega_1$). This corresponds to missing output faults. The opposite, $\omega_1 =>^{\neq}_{OC} \omega_2$, corresponds to unspecified output faults.
- ω_2 has more **input constraints** (that is, has less inputs) than ω_1 (written $\omega_2 =>^{\neq}_{IC} \omega_1$). This means that the implementation considers certain inputs that are foreseen by the specification to be unexpected, and the behavior of the implementation in response to such input is not defined by the POIOA model of the implementation. The opposite means that the implementation expects some input that is not foreseen by the specification; if some output depends on this input (according to the implementation model) this input will not occur unless the additional input is applied..
- ω_2 has more **output constraints depending on inputs** than ω_1 (written $\omega_2 =>^{\neq}_{OCI} \omega_1$). This corresponds to a stronger precondition fault. The opposite corresponds to weaker precondition faults.
- ω_2 has more **output constraints depending on outputs** than ω_1 (written $\omega_2 =>^{\neq}_{OCO} \omega_1$). This means that there is less concurrency between different outputs in the implementation as compared with the specification. The opposite corresponds to **output order faults**. This is a type of fault not mentioned in the earlier discussion in Section 2.1.

- ω_2 has more **input constraints depending on outputs** than ω_1 (written $\omega_2 =>^{\neq}_{ICO} \omega_1$). This means that the implementation makes additional assumptions about the time when inputs may occur, as compared with the specification. In the case of inputs not satisfying the constraints of the implementation model, its behavior is not defined.
- ω_2 has more **input constraints depending on inputs** than ω_1 (written $\omega_2 =>^{\neq}_{ICI} \omega_1$). This means that the implementation makes additional assumptions about the allowed concurrency of inputs, as compared with the specification.

In a similar fashion we write $\omega_2 =>_{XXX} \omega_1$ for the non-strict version of $=>^{\neq}_{XXX}$, meaning that $\omega_2 =>^{\neq}_{XXX} \omega_1$ or $\omega_2 = \omega_1$.

Formally, these relations can be defined as follows: If $\omega_1 = (E_1, \leq_1, \mu_1)$ and $\omega_2 = (E_2, \leq_2, \mu_2)$ are two pomsets labeled over an input/output alphabet $I \cup O$, we define the above relations as follows:

- $\omega_2 =>_{OC} \omega_1$ if the bag $\mu_1(E_1) \cap O$ is included in the bag $\mu_2(E_2) \cap O$.
- $\omega_2 =>_{IC} \omega_1$ … similarly
- $\omega_2 =>_{OCI} \omega_1$ if $E_2 = E_1$, $\mu_2(E_2) = \mu_1(E_1)$ and $\leq_2$ can be obtained from $\leq_1$ by repeating the following process:
  - Step 1: Add some additional order relationship instances, each indicating that some output event should wait for some input event.
  - Step 2: Form the new pomset by taking the transitive closure of the order relationship obtained in Step 1.
- $\omega_2 =>_{OCO} \omega_1$ if similarly $\leq_2$ can be obtained from $\leq_1$ by a process as above, where in Step 1 some relationship instances are added, each indicating that some output event should wait for some other output event.
- $\omega_2 =>_{ICO} \omega_1$ if … similarly.
- $\omega_2 =>_{ICI} \omega_1$ if … similarly.

Definition*: We say that a pomset* $\omega_2 = (E_2, \leq_2, \mu_2)$ *is quasi-equivalent to a pomset* $\omega_1 = (E_1, \leq_1, \mu_1)$ *, written* $\omega_2 =>_{qe} \omega_1$ *, if there is a finite sequence of pomsets* $\omega^{(i)}$ *for i = 1, 2, … , n such that* $\omega^{(1)} = \omega_1$, $\omega^{(n)} = \omega_2$ *and for all i = 2, 3, …, n either*

- $\omega^{(i)} =>_{OCO} \omega^{(i-1)}$ *or*
- $\omega^{(i-1)} =>_{ICI} \omega^{(i)}$ *or*
- $\omega^{(i-1)} =>_{ICO} \omega^{(i)}$

The means that ω_2 is quasi-equivalent to ω_1 if either $\omega_2 = \omega_1$ or it is obtained from ω_2 by reducing the input order constraints (input-input or input-output), and/or by increasing the output-output constraints. We note that the reduced input order constraints of an implementation will not be tested in the case of conformance testing where one wants to verify whether the implementation satisfies the requirements of the specification. During the testing, the test case (the environment of the system under test) will have to fulfill the assumptions that the specification makes about the environment, which includes the input-input constraints of the specification.

It would be interesting to test whether the output-output constraints of the implementation are stronger than those defined in the specification. However, this is difficult to test, as discussed in Section 4.2.

Definition: *Let* $M = (S, s^{in}, I, O, T)$ *be a POIOA. A (finite)* ***transition trace*** *of M is a word* $w = \omega_1 \omega_2 \omega_3 \ldots \omega_n$ *such that there exist* $t_1 t_2 t_3 \ldots t_n \in T^*$ *such that the* $t_i = (s_i, \omega_i, s_i^+)$ *satisfy*

1. $s_1 = s^{in}$
2. $s^+_i = s_{i+1}$ *for all i.*

We denote the set of transition traces of M as Tr(M).

Definition: *Let* $M = (S, s^{in}, I, O, T)$ *and* $M' = (S', s^{in'}, I, O, T')$ *be two POIOA, and let* $w = \omega_1\omega_2\omega_{3\ldots}\omega_n$ *be a transition trace of M and* $w' = \omega'_1\omega'_2\omega'_{3\ldots}\omega'_n$ *be a transition trace of M'. We say that w is quasi-equivalent to w' (written* $w =>_{qe} w'$*) iff (by induction)*

- *if n=1 (*$w = \omega_1$ *and* $w' = \omega'_1$ *)* $\omega_1 =>_{qe} \omega'_1$
- *else (*$w = w_1\ \omega_2$ *and* $w' = w_1'\ \omega'_2$*.)* $w_1 =>_{qe} w_1$ *and* $\omega_2 =>_{qe} \omega'_2$.

We now define the notion of *trace quasi-equivalence* between two POIOA as being the fact that the traces of one POIOA are by quasi-equivalence included in the traces of the other one. Note: This notion has some similarity with the notion of quasi-equivalence as defined for partially defined finite state machines.

Definition: *Let* $M = (S, s^{in}, I, O, T)$ *and* $M' = (S', s^{in'}, I', O', T')$ *be two POIOA. M is **trace quasi-equivalent to** M' if*

1. $I' \subseteq I$ *and* $O' \subseteq O$

For each t' in Tr(M'), there is a t in Tr(M) such that $t =>_{qe} t'$

In summary, the trace quasi-equivalence of a POIOA M with a POIOA M' (where M may be the implementation of M') means that (a) M may have a different number of states than M', (b) M may have additional transitions (for which there are no corresponding transitions in M') which must have exclusive inputs with the transitions defined in M', and these additional transitions may involve inputs and outputs that are not defined for M'. However, the transitions of M that correspond to transitions in M' must be quite similar: (c) they must involve the same input and output events, and (d) they must have the same order constraints for outputs depending on inputs, whereas (e) the concurrency between outputs may be reduced.

The interesting property of trace quasi-equivalence is the following: If an implementation M is trace quasi-equivalent to the specification M', then this implementation will exhibit the (safeness and liveness) properties to be guaranteed by the outputs according to the specification, if the assumptions concerning the applied inputs, as specified by the specification, is satisfied by the real environment in which the implementation evolves.

3 Transition Testing

In this section, we explain a method for generating test cases for POIOA and outline the diagnostics for each of the possible implementation faults outlines in Section 2.1. We then show how to combine the elementary test cases for specific input events into longer sequential test cases, and conclude with a method for testing full conformance of implementations with reliable resets. We concentrate on the testing of individual transitions and their input/output behavior, related to unspecified and missing outputs

and weaker or stronger preconditions. For the testing of transfer faults, the known methods developed for finite state machines can be directly applied to POIOA [11].

3.1 Single input event testing

Let $M= (S, s^{in}, I, O, T)$ be a POIOA, and let $t=(s,\omega,s') \in T$ be a transition of T. Our goal is to generate a set of test cases, in the following called test suite, to verify that an implementation of M has *correctly implemented* a corresponding transition. By *correctly implemented* we mean that the corresponding transition does not include any of the faults mentioned in Section 2.1. We therefore have to test that the implemented transition has the same input and output events as specified for ω and that the output constraints depending on inputs are the same. We note that the here described test cases do not check the output constraints depending on outputs, nor any stronger order constraints for inputs (that are not assumed by the specification).

Notation: Given a pomset $\omega = (E_\omega, \leq_\omega, \mu)$ and an element $x \in E_\omega$, we write $\downarrow x = \{y \in E_\omega, \neg(x \leq_\omega y)\}$ for the set of elements of E_ω that are neither greater than nor equal to x.

For each input event i of the transition $t=(s,\omega,s')$, the test suite should include the following test case, where we assume that the implementation is already in the starting state of the corresponding transition:

1. Enter all the input events in $\downarrow i$, respecting the ordering constraints, and observe the multiset $S1$ of produced output events.
2. Enter input event i and observe the multiset $S2$ of produced output events.
3. Enter the input events of ω that have not been input yet, respecting the ordering constraints, and observe the multiset $S3$ of produced output events.

Proposition: The tested transition of the implementation is quasi-equivalent to the corresponding transition in the specification if and only if, for all inputs of the transition, the observed output multisets S1, S2 and S3 have the value predicted by the specification.

The following diagnostics can be issued depending on the outputs observed within the three multisets S1, S2 and S3:

- *Unspecified output fault:* the number of occurrences of an output o in $S1 \cup S2 \cup S3$ is larger than the number of occurrence of that output as specified in ω. In this case, at least one occurrence of o is produced by the implementation while not being specified by the POIOA specification.
- *Missing output*: the number of occurrence of an output o in $S1 \cup S2 \cup S3$ is smaller than the number of occurrence of that output as specified in ω. In this case, at least one occurrence of o is not produced by the implementation while being specified by the POIOA specification.

If there is a single fault for a given output label, we can diagnose the following faults:

- *Weaker precondition:* the number of occurrences of the output o in $S1$ is larger than the number predicted by the specification. In this case, at least one occurrence of o is produced by the implementation prior to the input i while i is specified by the POIOA specification as a precondition of o.

- *Stronger precondition:* the number of occurrences of the output o in $S2$ is larger than the predicted by the specification. In this case, at least one occurrence of o is produced by the implementation only after the input i (that is, i is a precondition for o in the implementation) while i is not specified by the POIOA specification as a precondition of o.

3.2 Testing several input events

The test protocol described in the Section 3.1 cannot be used to test stronger or weaker preconditions for more than one input event per transition. It is possible to extend this protocol to several input events, as long as these events are all ordered in the partial order of the corresponding pomset.

Let $i_1, i_2,..., i_k$ be a set of input events of the pomset $\omega = (E_\omega, \leq_\omega, \mu)$ of a transition of the IOPOA going from state s to state s' such that $i_1 \leq_\omega i_2 \leq_\omega ... \leq_\omega i_k$. Then the test cases for these inputs, as described in the subsection above, can be combined into a single test case that tests all k inputs sequentially as follows, assuming that the implementation is in the starting state of the transition:

1. For $m=1$ to k
   a. Enter all the input events in $\downarrow i_m$ that have not been input yet, respecting the ordering constraints, and observe the multiset $S1_m$ of produced output events.
   b. Enter input event i_m and observe the multiset $S2_m$ of produced output events.
2. Enter the input events of ω that have not yet been input, respecting the ordering constraints, and observe the multiset $S3$ of produced output events.

The diagnostics described in Section 3.1 can be adapted in the following way:

- For missing and unspecified outputs faults, use the multiset $S1_1 \cup S1_2 \cup ... \cup S1_k \cup S2_1 \cup S2_2 \cup ... \cup S2_k \cup S3$, in place of $S1 \cup S2 \cup S3$.
- For weaker and stronger precondition faults, apply the diagnostics described in Section 3.1 for each input i_m separately ($m=1$ to k), using the multiset $S1_1 \cup S1_2 \cup ... \cup S1_m \cup S2_1 \cup S2_2 \cup ... \cup S2_{m-1}$ in place of S_1, and $S2_m$ in place of S2.

Moreover, it can be easily shown that the test cases of two concurrent inputs (as defined in Section 3.1) cannot be combined into a single pass through the transition. Indeed, if i_1 and i_2 are two concurrent inputs, then by definition $i_2 \in \downarrow i_1$ and $i_1 \in \downarrow i_2$, so whichever input is tested first, the other one will have to be entered during Step 2 of the test case and thus will not be testable anymore during the same pass through the transition.

Since we have shown that we can test any number of input events sequentially in the same pass over a given transition if and only if these events are all mutually ordered in the pomset of that transition (they define a *chain* in the order), we can deduce that an optimal strategy (in terms of number of passes) for testing a given transition for all inputs consists of finding the minimum number of chains of the order that would include all input events. This is a classical property of ordered set theory called the *minimal chain decomposition* of an order [10] (and the number of chains in the minimal chain decomposition is equal to the largest number of mutually incomparable elements of the order). Thus, in order to test a transition associated with the pomset $\omega = (E_\omega, \leq_\omega, \mu)$ in an optimal way, we must create $\omega_I=(E_I, \leq_I, \mu_I)$, the

projection of ω onto the input events, then create a minimal chain decomposition of $(E_I, \leq_I)$, and finally, for each chain of the decomposition, bring the implementation to the starting state of the transition and apply the above combined test case for the input events of that chain.

4 Discussion

In this section, we come back to the assumptions we have made about the POIOA representing the system specification and about the properties of the implementation, and discuss their nature and the possibility of avoiding them. We also discuss how to test constraints on output concurrency and input order assumptions.

4.1 Assumptions about the implementation

When we apply our testing algorithm, we make the following assumptions on the specification and the implementation:

1. As explained in Section 3.3, we must make some assumptions on the number of states of the implementation.
2. **Bounded response time:** This means that when all precondition events of a given output event have occurred, then the implementation will produce the output within a bounded time delay. Therefore, if the output did not occur within this delay, it can be concluded that it will not be produced without any additional input being applied. This appears to be a reasonable assumption. It is also made for testing finite state machines, where after each input one has to wait for the resulting outputs before the next input is applied.
3. One-to-one correspondence of transitions in the specification and implementation: We have implicitly assumed that the implementation of a given transition of the specification can be modeled as a single transition in the implementation POIOA. As an example, we consider the case of the specification transition shown in Figure 2(a). It has three concurrent initial inputs. A valid implementation may foresee three different transitions, depending on which of the three inputs occurs first, as shown in Figure 2(b). Assume now that the implementation of the third transition has a missing output (o_1 does not occur). Then this fault will not be detected by the test cases derived from the specification according to our method. In order to detect all such faults, one may adapt our test generation procedure by applying it not to the specification, but to a refined model of split transitions, as shown in Figure 2(b) . However, this leads to very long test suites, since the number of these interleaving can be exponentially larger than the number of order constraints. - The assumption of one-to-one correspondence of transitions also implies that the implementation exhibits a certain form of determinism in its behavior. For instance, it is not allowed that the implementation, non-deterministically, sometimes realizes the specified ordering constraints, and sometimes does not, possibly depending on some parameter outside the scope of the specification (for instance, depending on

the temperature, or the total execution time since the start-up of the implementation.

4.2. Testing other conformance relations

As mentioned earlier, the proposed test cases are not able to check the allowed concurrency between outputs, that is, the output constraints depending on outputs. For instance, how could one test that the implementation has not implemented the transition shown in Figure 1(c), instead of the specified transition of Figure 1(a) which specifies that output o4 should come after output o3. The transition shown in Figure 1(c) does not satisfy this requirement. The model of the latter transition allows for non-determinism (the three outputs after the second input i2 may be generated in any order). If we assume that the implementation may be faulty and actually be modeled by the transition of Figure 1(c), then we may also assume that the implementation really presents such non-determinism, possibly due to different possible interleavings of concurrent activities. If we want to distinguish between the transitions of Figure 1(a) and 1(c) through testing, we have to execute an appropriate test case several times: If in one of the observed execution scenarios, output o4 occurs before o3, then we know that the implementation does not realize the transition of Figure 1(a). For a positive verdict, we need to execute the test a large number of times in order to obtain a sufficiently high statistically assurance that the right order to not obtained by chance. We have the same difficulty of non-determinism when we want to test whether the implementation has stronger output constraints depending on outputs than the specification.

We have not discussed above how to test whether a transition of the implementation model has less inputs or has stronger input order constraints than the corresponding transition of the specification. These are cases where the implementation makes stronger assumptions about its environment that defined in the specification. This means that during the execution of a complete test suite, there will be test cases that apply inputs to the implementation that are not expected, or that apply inputs in an order that is not foreseen by the implementation model. Since the POIOA model of the implementation does not define the behavior of the implementation in these cases, we can only assume that the implementation will detect these situations and provide some kind of error message or unusual behavior pattern that can distinguished by the test harness. Under this assumption, it can therefore be expected that such problems will be detected by a test suite derived as explained in Section 3.

6 Conclusion

In this paper, we introduce a new type of automata, called Partial-Order Input/Output Automata, for which each transition is associated with a partially ordered set of input and output events. This new model is a generalization of a previous version where only bipartite orders where permitted. Relaxing the bipartite constraint allows to specify any combination of inputs and outputs, with any concurrency or ordering

constraints between these events. We provide a formal model for these automata, and give formal definitions of the comparison of behavior between two automata. We then provide a testing methodology which can be used to detect some of the faults that have been formally specified. Finally, we explain the assumptions we have made about the implementations under test and about the POIOA model, provide some justifications and discuss why it would be interesting to remove some of these assumptions in future work.

Acknowledgments. This work has been supported in part by grants from the Natural Sciences and Engineering Research Council of Canada, as well as by the INRIA sabbatical support for the third author.

References

[1] Haar, S., Jard C. Jourdan GV. Testing Input/Output Partial Order Automata. *Proc. TestCom 2007, LNCS 4581*, pages 171-185, Springer 2007.

[2] N. A. Lynch and M. R. Tuttle, An introduction to input/output automata, *CWI Quarterly,* 2(3), 1989, pp. 219-246.

[3] Mauw S., Reniers, M. , *High-level Message Sequence Charts*, Proceedingsof the Eight SDL Forum, SDL'97: Time for Testing - SDL MSC and Trends, pp 291-306, A. Cavalli and A. Sarma, editors, Evry, France, 23-26 September, 1997.

[4] Alur, R., Etessami, K. and Yannakakis, M.. Realizability and verification of MSC graphs. *Theor. Comput. Sci.*, 331(1):97–114, 2005.

[5] A. Mooij, J. Romijn, and W. Wesselink. Realizability criteria for compositional MSC. In *11th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'06)*, volume 4019 of *LNCS*. Springer, 2006.

[6] A. J. Mooij, N. Goga, and J. Romijn. Non-local choice and beyond: Intricacies of MSC choice nodes. In *Fundamental Approaches to Soft. Eng. (FASE'05)*, 2005.

[7] Castejón, H. N., Bræk, R., Bochmann, G.v., Realizability of Collaboration-based Service Specification. *In APSEC conference*, Nov. 2007

[8] M. Abadi and L. Lamport, Conjoining specifications, *ACM Transactions on Programming Languages & Systems*, vol.17, no.3, May 1995, pp. 507-34.

[9] G. v. Bochmann, Submodule construction for specifications with input assumptions and output guarantees, *in Proc. FORTE'02 (22st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems)*, Chapman&Hall, 2002, pp.

[10] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, (51):161–166, 1950.

[11] Lee D, Yannakakis M (1996) Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE*, 84(8):1089–1123.

[12] M. G. Gouda and Y.-T. Yu, Synthesis of communicating Finite State Machines with guaranteed progress, *IEEE Trans on Communications*, vol. Com-32, No. 7, July 1984, pp. 779-788.

[13] G. Luo, R. Dssouli, G. v. Bochmann, P. Ventakaram and A. Ghedamsi, Generating synchronizable test sequences based on finite state machines with distributed ports, *Proceedings of the IFIP Sixth International Workshop on Protocol Test Systems,* Pau, France, September 1993, pp. 53-68.