Testing Systems Specified as Partial Order Input/Output Automata

Gregor v. Bochmann[1], Stefan Haar[2], Claude Jard[3], Guy-Vincent Jourdan[1]

[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

[3] Université Européenne de Bretagne
ENS Cachan, IRISA, Campus de Ker-Lann, 35170 Bruz, France
Claude.Jard@bretagne.ens-cachan.fr

Abstract. An Input/Output Automaton is an automaton with a finite number of states where each transition is associated with a single input or output interaction. In [1], we introduced a new formalism, in which each transition is associated with a bipartite partially ordered set made of concurrent inputs followed by concurrent outputs. In this paper, we generalize this model to Partial Order Input/Output Automata (POIOA), in which each transition is associated with an almost arbitrary partially ordered set of inputs and outputs. This formalism can be seen as High-Level Messages Sequence Charts with inputs and outputs and allows for the specification of concurrency between inputs and outputs in a very general, direct and concise way. We give a formal definition of this framework, 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: missing output faults, unspecified output faults, weaker precondition faults, stronger precondition faults and transfer faults.

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

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 automaton 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. This model is very close to High-level Messages Charts (HMSC), a standard already used to specify in a global manner dynamic behaviors of distributed systems [3].

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 the 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 any deviation from quasi-equivalence will be 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 the assumptions we have made for our POIOA specification model. We give some justification for these assumptions and 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].

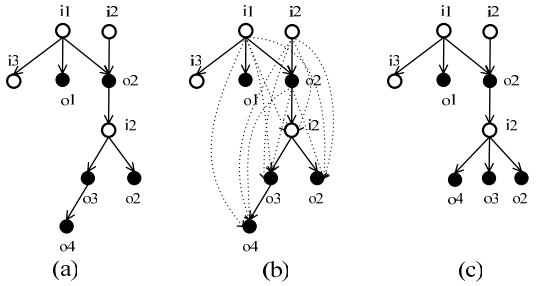

Figure 1: partially ordered multisets of input and output events

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 i_2 (second occurrence) $< o_2$ (second occurrence), 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 property, and (b) a liveness property, sometimes called "progress property". 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 HMSC or in UML sequence diagrams, where 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 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 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. 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.

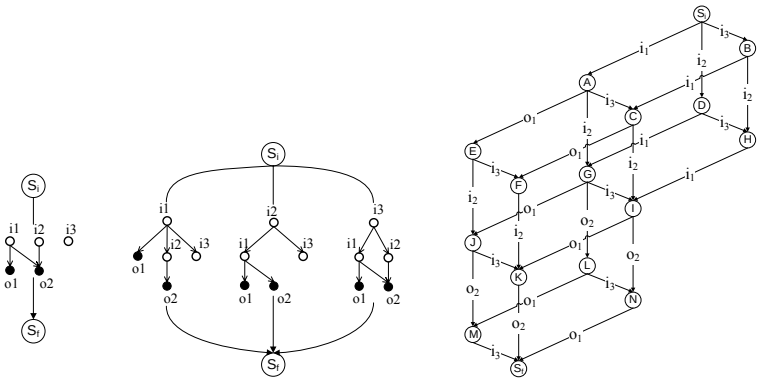

Figure 2: (a) specification of transition. (b) implementation of the specified transition in terms of three separate transitions. (c) An Input/Output Automata model equivalent to the POIOA transition shown in (a)

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:

- *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.
- *Unsupported input*: The implementation does not support the reception of all input events foreseen in the specification.
- *Missing output constraint:* An output foreseen by a transition is produced before all the events that are specified as precondition have occurred.
- *Additional output constraint depending on input:* 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.
- *Additional input constraints:* The implementation does not support the reception of certain inputs in some order allowed by the specification.
- *Transfer fault:* A transition of the implementation leads to a state different from what is specified.

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 missing and additional ordering constraints. Our approach to catch these types of faults is to test each input event of a given transition separately, as explained in Section 3. When combined with transfer fault detection, our new tests might have to be applied a number of times per transition according to the classical fault detection methods.

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 $z=x\}$. For two elements x and y of E, we note $x||y$ if neither $x \leq y,$ nor $y \leq x$.

Definition 1: *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. *$\mu: E \rightarrow V$ 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 2: *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 finite 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 $t1=(s, \omega_1, s_1)$ and $t_2=(s, \omega_2, s_2)$ starting from the same state s, we have $min(\omega_1) \cap min(\omega_2) = \emptyset$. 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 strongly synchronized, input-guarded POIOA with exclusive transitions. 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 of 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.

Definition 3*: We say that a (I∪ O)-pomset* $\omega_I = (E_I, \leq_I, \mu_I)$ *is quasi-equivalent to a (I∪ O)-pomset* $\omega_S = (E_S, \leq_S, \mu_S)$ *, written* $\omega_I =>_{qe} \omega_S$ *, if and only if:*

1. *The input and output events are the same, that is,* $E_I = E_S = E$*, and,* $\mu_I = \mu_S = \mu$
2. *The following conditions are satisfied concerning the order relations between events: for all e, o, i in E such that* $\mu(o) \in O$ *and* $\mu(i) \in I$
   a. $e <_S o$ *implies* $e <_I o$
   b. $e <_S i$ *implies* $e <_I i$ *or* $e ||_I i$
   c. $e ||_S i$ *implies* $e ||_I i$

The means that ω_I (corresponding to an implementation) is quasi-equivalent to ω_S (corresponding to a specification) if the sets of input events and output events are the same and certain conditions are satisfied for the order relations of the two *(I∪ O)*-pomsets. We talk about a **missing output fault** if some output event of ω_S is not included in ω_I. We talk about an **unexpected output fault** if, on the contrary, some output event of ω_I is not included in ω_S. We say that the implementation has an **unsupported input fault** if input event of ω_S is not included in ω_I. If, on the contrary, there is an input event in ω_I that is not included in ω_S, then this means that the implementation is ready to accept additional input which is not a fault; this condition would normally not be tested.

The conditions concerning the order relations have the following significance. Condition 2(a) states that if an output event has an order constraint in ω_S then it must have the same constraint in ω_I . If that is not the case, we say that the implementation has a **missing output constraint fault**. Condition 2(c) implies (when e is an output) that an output event that has no order constraint depending on input in the specification should not have such a constraint in the implementation. If this is not true, we talk about an **additional output constraint depending on input fault**. We note that if two output events have no ordering relation in the specification, there is the possibility that a quasi-equivalent implementation introduces such a relation.

The ordering constraints on inputs are described by Conditions 2(b) and 2(c). They imply that the implementation may not introduce any **additional input constraints**. However, an implementation may not rely on the assumption that an ordering constraint for input defined in the specification is actually observed by the environment; such a more powerful implementation is allowed.

Quasi-equivalence captures the notion of “compatible implementation”, an implementation that accepts any inputs compatible with the specification (and may

accept more) and which will produce outputs defined by the specification in an order compatible with the specification.

Definition 4: *Let M= (S, s^{in}, I, O, T) be a POIOA. A (finite)* ***transition trace*** *of M is a word w = $\omega_1\omega_2\omega_3...\omega_n$ such that there exist $t_1t_2t_3... t_n \in T^*$ 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 5: *Let M= (S, s^{in}, I, O, T) and M'= (S', $s^{in'}$, I, O, T') be two POIOA, and let w = $\omega_1\omega_2\omega_3...\omega_n$ be a transition trace of M and w' = $\omega'_1\omega'_2\omega'_3...\omega'_n$ 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 = ω_1 and w' = ω'_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 6: *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 $O' \subseteq O$*
2. *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 initial 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 very similar order relations, as defined by points 2(a), (b), and (c) in Definition 3.

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, are 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, unsupported inputs, missing output constraints, additional output constraints on input and additional input constraints. 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 is quasi-equivalent to the transition of the specification (according to Definition 3). We therefore have to test that the implemented transition has the same input and output events as specified for ω and that the constraints between events are compatible with the definition of quasi equivalence.

We make the following assumptions regarding the testing environment:

- Assumption 1: we can detect when the implementation receives "unspecified input" in the case that the implementation has additional input constraints and receives an input event that it did not expect. In such a situation, the implementation may for instance return some error message to the tester.
- Assumption 2: we can observe the order between two outputs if an order relation is defined in the specification.

Notation: Given a $(I \cup O)$-pomset $\omega = (E_\omega, \leq_\omega, \mu)$ and an element $x \in E_\omega$, we write $\mathcal{P}(x) = \{y \in E_\omega, y <_\omega x)\}$ for the set of elements that are before x (the "Past" of x). We write $\mathcal{NF}(x) = \mathcal{P}(x) \cup \{y \in E_\omega, x \|_\omega y\}$ for the set of elements of E_ω that are neither greater than nor equal to x (element Not in the "Future" of x).

3.1.1. Basic test suite

We say that a set of inputs is serialized if it respects the ordering constraint of the specification and the inputs are sent one at a time, waiting for all possible output to be produced before the next input is sent.

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 $\mathcal{NF}(i)$ in a serialized way, 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 in a serialized way, and observe the multiset *S3* of produced output events.

We say that an implementation *I* is *almost quasi equivalent* to a specification *S* if it is quasi equivalent except for possible missing output constraints on outputs, and possible added output constraints on input.

Proposition: The tested transition of the implementation is almost 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 values 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.
- *Unsupported input*: Based on Assumption 1, this fault will be detected through an error message received from the implementation.

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

- *Missing output constraint depending on input:* 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*.
- *Additional output constraint depending on input:* 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*.
- *Additional input constraint on input:* Since inputs will be applied in all allowed relative orders, such faults will be detected through an error message received from the implementation, based on Assumption 1.

3.1.2. Testing output-output ordering constraints

The basic test suite detailed above will not necessarily detect a fault of missing output constraint depending on another output. For instance, the order between o_3 and o_4 in Figure 1(a) is not realized in the implementation shown in Figure 1(c). In the case of such a fault, two outputs that are ordered in the specification could be concurrent in the implementation, or implemented in the opposite order. The case of concurrent implementation would allow for nondeterminism, which means that the implementation may produce these outputs sometimes in the order defined by the specification and sometimes in the opposite order. If we want to detect these faults, we have to observe not only the sets S1, S2 and S3, but also the order in which the outputs in each set occur. And in the case that the behavior of the implementation

may be non-deterministic, we have to execute an appropriate test case several times: if in one of the observed execution scenarios, the ordering of outputs is reversed, then we know that the implementation does not realize the specification. For a positive verdict, we need to execute the test a large number of times in order to obtain a sufficiently high statistical assurance that the right order was not obtained by chance.

3.1.3 Added input constraints on output

The test cases presented so far will not necessarily catch the case of an additional input constraint depending on output. In this case, an output and an input are specified as concurrent but the implementation expects the output to be produced before accepting the input. This is an error since it is legitimate, according to the specification, to enter the input before the output is produced, but the implementation would not accept this behavior.

We note that such an additional input constraint may introduce, by transitivity, an additional constraint between inputs. If this is the case, the latter constraint will be detected by the basic test suite. Otherwise, specific test, as described below, must be applied in order to detect the additional input constraints depending on output.

For each input event i of the transition $t=(s,\omega,s')$, let $\omega' = (E_\omega \leq_\omega, \mu) \setminus \{\mathcal{P}(i) \cup i\}$ be the sub-pomset of ω restricted to the events that do not belong to $\mathcal{P}(i) \cup i$, and let $O_{min} = \{x \in \min(E_{\omega'}), x \in \mu(E) \cap O\}$ be the set of minimal elements of ω' that are output events.

If $O_{min} \neq \varnothing$, then the 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 $\mathcal{P}(i) \cup i$, respecting the ordering constraints, and (if possible) before any element of O_{min} is produced by the implementation.
2. Observe that the implementation outputs all elements of O_{min}.

Note that the tester may fail to perform the first step, in that elements of O_{min} are produced before all of $\mathcal{P}(i) \cup i$ can be input. In this case, the test is *INCONCLUSIVE* and should be done again.

Proposition: *If a transition successfully passes the basic test suite and the above test case for each input that has a non-empty O_{min}, then the implementation has no additional input constraints depending on outputs.*

Proof: Assume that a constraint $o > i$ has been added by the implementation. By definition, it means that in the specification, $i \|_\omega o$. When testing i, if $o \in O_{min}$ then the test above will exhibit the problem (input i will put the implementation in unspecified behavior mode, which will be detected based on Assumption 1). If $o \notin O_{min}$ then either there exists $o' \in O_{min}$ such that $o' >_\omega o$, or there exists $i' \in \omega'$ such that $i' >_\omega o$ and $i' \|_\omega i$. In the former case, the constraint $o' > i$ has been added (by transitivity) and we are back to the first case. In the second case, the constraint $i' > i$ has been added (by transitivity) and this will be seen when applying the basic test suite to i'.

3.2 Testing several input events

The tests described in the Section 3.1 are designed to test the behavior of the implementation in respect to the behavior in relation with a particular input event of a particular transition. We show in the following how several of these elementary test cases can be combined into a larger test case that covers several input events of a given transition. This can be done as long as the corresponding input events are all ordered in the partial order of the transition $(I \cup O)$-pomset.

Let $i_1, i_2, \ldots, i_k$ be a set of input events of the $(I \cup O)$-pomset $\omega = (E_\omega, \leq_\omega, \mu)$ of a transition of the IOPOA going from state s to state s' such that $i_1 \leq_\omega i_2 \leq_\omega \ldots \leq_\omega i_k$. Then the basic test cases for these inputs, as described in Section 3.1.1, 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 $\mathcal{NF}(i_m)$ 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 3.1.1 can be adapted in the following way:

- For missing and unspecified outputs faults, use the multiset $S1_1 \cup S1_2 \cup \ldots \cup S1_k \cup S2_1 \cup S2_2 \cup \ldots \cup S2_k \cup S3$, in place of $S1 \cup S2 \cup S3$.
- For missing and additional output constraints, apply the diagnostics described in Section 3.1.1 for each input i_m separately ($m=1$ to k), using the multiset $S1_1 \cup S1_2 \cup \ldots \cup S1_m \cup S2_1 \cup S2_2 \cup \ldots \cup S2_{m-1}$ 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 $i_2 \in \mathcal{NF}(i_1)$ and $i_1 \in \mathcal{NF}(i_2)$, 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 $(I \cup O)$-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 $(I \cup O)$-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.

3.3 Discussion

We now come back to the assumptions we have made about the POIOA that represent the system specification and about the implementation that are also modeled as POIOAs. When we apply our testing method, we make the following assumptions about the specification and the implementation:

1. **Assumption 1 – unspecified input are detectable**: We think that this assumption (see Section 3.1) is reasonable, since the behavior of an implementation in response to unexpected input is not well defined. Therefore it can be assumed that the behavior observed in such a situation would be different from what the specification prescribes.
2. **Assumption 2 – specified ordering of outputs can be observed**: As explained in Section 2, we assume that the ordering of events can be observed if the specification prescribes a specific order. Therefore this assumption (see Section 3.1) is reasonable.
3. **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.
4. **No transition splitting in the 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 specification model that has split transitions, as shown in Figure 2(b). However, this leads to very long test suites, since the number of the split interleavings could be exponentially larger than the number of order constraints.

4 Conclusion

In this paper, we present a generalization of a previous test method 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. So, we get closer to the standard modeling languages like HMSC or UML sequence diagrams. We provide a formal model for these automata, and give the formal definition of the quasi-equivalence relationship which can be used

for the comparison between the behaviors of two automata. We then provide a testing methodology which can be used to determine whether any implementation is quasi-equivalent to a given specification. 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 second 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*, Proceedings of 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.

1. [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.