Conformance Testing with Respect to Partial-Order Specifications

Gregor Bochmann

Conformance Testing with Respect to Partial-Order Specifications

Gregor v. Bochmann

School of Electrical Engineering and Computer Science
University of Ottawa, Canada
bochmann@uottawa.ca

Abstract: This paper deals with the testing of distributed systems. An implementation under test is checked for conformance with the properties defined by a reference specification. Since distributed systems usually have multiple interfaces, the reference specification will not define the order of all pairs of interactions taking place at different interfaces. Therefore a specification formalism supporting the definition of partial orders is required. Different such formalisms are compared in this paper, including MSC-Charts (or Interaction Overview Diagrams). A variation of this formalism, called Partial-Order-Charts (PO-Charts) is proposed which makes abstraction from the exchange of messages. It concentrates on the specification of partial orders between local actions in different system components. It is shown that the partial-order testing approach introduced for a single partial order specification can be adapted to testing PO-Charts which define various combinations of different partial orders which are sequenced by strict or week sequencing, including loops. Various examples are given to compare this testing approach with state machine testing methods which can be applied for **bounded** PO-Charts for which one can derive an equivalent state machine. The testing complexities and fault model assumptions of these two approaches are compared.

1 Introduction

Conformance testing is an activity where an implementation under test (IUT) is checked for conformance to a specification. For this purpose, input interactions are applied by testers at the different interfaces of the IUT and the outputs provided by the IUT are observed by the testers and are compared with what is expected according to the requirements defined by the specification. For distributed systems, the order of interactions taking place at different interfaces are often irrelevant for the defined behavior, furthermore, it is sometimes difficult to control the order of inputs at different interfaces, and to observe the order of outputs at different interfaces. For this reason, state machine models for the specification are not appropriate, since they precisely define a total order for all interactions. As a consequence, partial-order specifications have been proposed for describing the required behavior of distributed systems. A well-known example of a partial-order notation is Message Sequence Charts (MSC, or UML Interaction diagrams).

In order to test partial-order specifications, [Haar] proposed the concept of Partial-Order Input-Output Automata (POIOA) and discussed how to derive conformance test suites from such specifications. A POIOA is a state machine where each transition involves in general a set of input and output interactions for which a partial order is defined for their execution. However, each state of the POIOA represents a global synchronization point involving all the distributed interfaces. This enforces strict sequencing between the execution of subsequent transitions, that is, an interaction of the next transition can only occur after all interactions of the preceding transition have been completed. In real distributed systems, one often rather wants to impose weak sequencing which means that sequencing is enforced locally at each interface (or each component of the distributed system), but not globally.

Concepts for specifying control flow in distributed systems with partial orders including strict AND weak sequencing was proposed in [Castejon]. These concepts are quite similar to the more formal definition of MSC-Charts given by Alur and Yannakakis [Alur]. In a few words, an MSC-Chart is a state machine in which each state is associated with an MSC to be executed and the transitions between states are spontaneous. We modify this concept as follows and call it Partial-Order Chart (PO-Chart) by specifying for each transition whether it represents strict or weak sequencing, and by associating with each state a partial order of actions (including inputs, outputs and local actions) where each action is placed on a vertical "swim-lane" ("process" in MSC, or "role" in [Castejon]). Such a partial order is very similar to an MSC, but the arrows represent a partial-order dependency, and not necessarily exchanges of messages (as in MSCs and MSC-Charts).

We discuss in this paper how a test suite can be derived from a given PO-Chart specification. The main point is the fact that the partial-order test derivation from [Haar] can be applied to execution paths involving several PO-Chart states (corresponding to several transitions in the POIOA model). For limiting the length of the test suite in the case of loops in the PO-Chart, we adopt the approach that is common in software testing: assuming regularity of the IUT (as explained in [Bouge]), which means that one assumes that there exists an integer k such that, if a loop has been executed k times, then no further fault would be found if one executed the loop more than k times.

The paper contains many examples to illustrate the discussion. For the testing of an IUT in respect to a PO-Chart the partial-order testing of [Haar] is compared with state machine testing methods based on an equivalent state machine model. However, often the PO-Chart specifications are not **bounded** [Alur], which means that no equivalent finite state model exists.

The paper is structured as follows: In Section 2, an introduction to POIOA testing is given, as well as a formal definition of partial orders. In Section 3, we discuss the different notations for defining the reference specification for testing. In Section 4 we show how the partial-order testing of [Haar] and state machine testing can be applied to PO-Charts. In Section 5, we provide some comments comparing the specification formalisms of POIOA, collaboration ordering, MSC-Charts and PO-Charts. We also compare the complexity measures for partial-order and state machine testing, as well as the underlying fault models. Section 6 contains the conclusions.

2 Preliminaries

2.1 Testing POIOA

The testing of POIOA was introduced in [Haar]. A POIOA is a state machine where each state transition involves possibly several input and output interactions for which a partial order is specified for execution. When all interactions of a transition have been performed, the machine enters the next state and is ready to execute another transition. One normally assumes that each transition starts with a single or several (concurrent) input interaction(s). An example of such a transition is shown in Figure 1(a). This transition starts with the single input i1 which is followed by two concurrent outputs o1a and o1b, each followed by a sequence of input and output, i2 followed by o2 and i3 followed by o3, respectively.

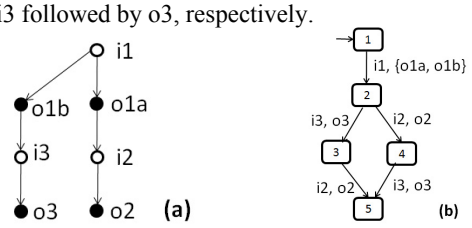

Fig. 1: (a) A partial order with two roles. (b) An equivalent state machine

When testing an implementation for conformance with a POIOA specification, one has to verify the following two aspects:

1. The partial order of interactions specified for each transition is implemented as specified.
2. Each transition leads to the correct next state.

For the second aspect, traditional state machine testing approaches can be used, such as Distinguishing Sequences [Bochmann a] or the HIS method [Bochmann c]. For this purpose one needs state identification sequences for each state which are applied after the execution of a transition, and which should be checked for validity on the implementation. We do not discuss these issues further in this paper.

For the testing of the partial order of input and output interactions defined for a given transition t, the following **partial-order test** has been proposed (see for instance [Bochmann a]). For each input i of t, perform the following test (where it is assumed that the implementation is already in the starting state of the transition):

1. Apply all inputs (different from i) that are not after i in the partial order of t (in an order satisfying the partial order), and observe the set of output interactions, called O1.
2. Apply i, and observe the set of subsequent output interactions, called O2.
3. Apply all other inputs of t (in an order satisfying the partial order), and observe the set of output interactions, called O3.

If one of the output sets is different than what is expected from the specified partial order, we have detected a fault in the implementation. We have a guarantee of fault detection under the assumption that the transition t is realized in the implementation

as a single transition and in the form of a partial order. For the example transition shown in Figure 1(a), we obtain the following test suite (where the tested input is written in bold, and the expected output sets are given in {}):

- For testing i1: <{}, **i1** {o1a, o1b}, i2 {o2}, i3 {o3}>
- For testing i2: <{}, i1 {o1a, o1b}, i3 {o3}, **i2** {o2}>
- For testing i3: same test case as for i1.

In [Bochmann a], it was also explained that the tests for several inputs can be combined into a single test case if one of the input comes after another one. For the example transition, we are left with the two test cases given above.

In this paper, we limit our attention to quiescent states of the IUT, that is, states in which no further outputs are produced by the IUT unless further input is applied. The above partial-order test goes only through quiescent states, since the next input is only applied after some time-out period to ensure that no additional output is expected. An interaction sequence is called a quiescent trace [Simao] if each input is applied when the IUT is in a quiescent state. For example, the sequence < i1, o1a, i3, etc. > is allowed by the partial order of Figure 1(a), but it is not quiescent. The testing of non-quiescent traces is discussed in [Bochmann a].

It was noted that the length of the resulting test suite for testing a single transition using this method is much shorter in the presence of many concurrent inputs as compared with traditional state machine testing. For the example transition shown in Figure 1(a), the corresponding state machine (showing only quiescent states) is shown in Figure 1(b). State machine testing (without state identification) yields for this state machine the same two test cases above. However, if there are more concurrent inputs, the number of states of the corresponding state machine will blow up exponentially (see also Section 5).

The notion of POIOA has been criticized because it assumes that there is global synchronization (involving all interaction points) in each state of the automaton. It was argued that this is not realistic if the behavior of the POIOA is supposed to represent the behavior of a distributed system where interactions take place at different interaction points distributed over several system components. To avoid this criticism, we consider in this paper the concepts explained Section 3, which allow for strict sequencing of transitions (as in the case of POIOA), as well as for weak sequencing (which is more natural in distributed environments).

2.2 Formal definition "partial order"

Given a set E of events, a partial order on E is a binary relation < of events which is transitive, antisymmetric and irreflective. If <e1, e2> is in <, we say that e1 is before e2. Often we characterize an order by the event pairs that generate all pairs in the order by transitivity closure. We call these the generating event pairs of the order. For instance, the arrows in Figure 1(a) correspond to the **generating event pairs**, for instance the pair <i1, o1a>. However, the partial order defined by this figure also includes pairs such as <o1a, o2> which are obtained by transitivity.

In order to deal with a situation where the same type of event occurs several times, one usually considers a Partially Ordered Multi-Set (Pomset). Given a partial order

(E, >) where some events in E may be of the same type, a Pomset on (E, >) is obtained by adding a labeling function L: E -> V, where V is a set of labels. For a given event e ϵ E, L(e) represents the type of event e. In fact, the names given to events in our figures represent the type of the event shown. For instance, the first event in Figure 1(a) is of type i1.

We call **initiating** event any minimum event of the order, that is, event e ϵ E is minimum if there is no event e' ϵ E such that e' < e. Similarly, we call **terminating** event any maximum event of the order. In the remainder of this paper, when we talk about a partial order, we always mean a Pomset where the set of labels V is often partitioned into two subsets: the set I of inputs and the set O of outputs.

3 The concept of PO-Charts

3.1 Collaborations

Concepts for describing the behavior of distributed systems in a global view have been proposed in [Castejon]. First, the UML concept of collaborations is used. A collaboration identifies the different roles that the components of the distributed system may play in a given application. However, this UML concept does not talk about the dynamic aspect of the behavior. For describing the dynamic aspect of the behavior, it is proposed to decompose a given collaboration into several sub-collaborations (each involving possibly a subset of roles) and indicating in which order these sub-collaborations are performed. Using the sequencing primitives of UML Activity diagrams (sequence, alternative and concurrency) an Activity-like notation is proposed, however, with the following modifications to the semantics: (a) a single Activity – called a "collaboration" – would normally involve several parties (roles – or swim-lanes); and (b) sequencing between successive collaborations may be in strict sequence (as in UML Activity diagrams, where any sub-activity of the second collaboration can only start when all sub-activities of the first have been completed), or in weak sequence (where a role may start with its activities of the second collaboration when it has completed its own sub-activities for the first).

A simple example is shown in Figure 2 (this is Figure 3 from [Castejon]). This is a simplified model of the execution of a medical test at the patient's premises in the context of tele-medicine. There are three roles in the system, as shown by the UML collaboration diagram of Figure 2(a): **dt** (doctor terminal), **tu** (test unit), and **dl** (data logger). Figure 2(b) shows the dynamic behavior of the Test collaboration: A *Test* starts with the *DoTest* sub-collaboration which is followed by the *LogValues* sub-collaboration. This may be repeated several times until the *GetValues* sub-collaboration is performed. The whole may be repeated several times.

The *Test* collaboration shown in the figure can, in turn, be used as a sub-collaboration in a larger context of tele-medicine, as discussed in [Castejon]. This notation, therefore, allows for writing hierarchically structured behavior specifications. At the most detailed level, the behavior of a collaboration can be defined in the form of a Message Sequence Chart (MSC, also called UML Interaction diagram). A

very simple example is shown in Figure 3(a) for the behavior of the *GetValues* collaboration included in Figure 2(b).

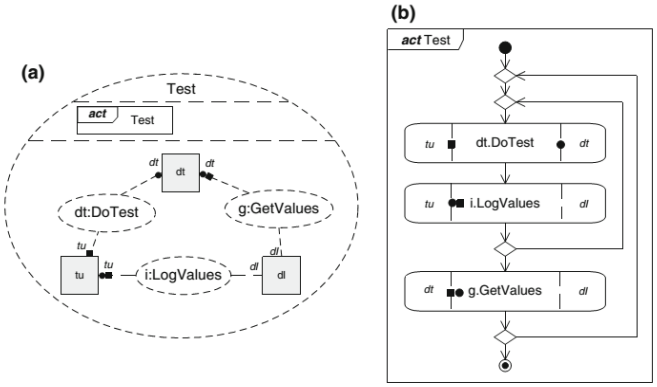

Fig.2: The Test collaboration: (a) UML Collaboration diagram, (b) behavior definition containing three sub-collaborations

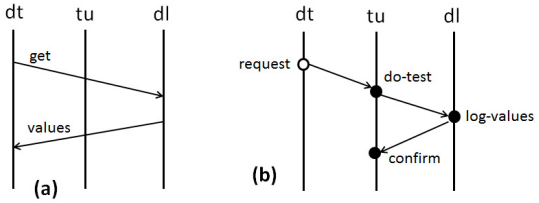

Fig. 3: (a) MSC defining the *GetValues* collaboration. (b) Partial order with roles defining *DoTest* and *LogValues* combined

However, in the context of this paper, we prefer to define the behavior of a basic, unstructured collaborations in the form of what we call a **partial order with roles**. This is a notion very similar to an MSC. Like in MSCs, the roles involved in the behavior are explicitly shown as vertical line. Actions performed by a role are indicated by dots (events) with their names (event labels) and the partial order between these events is indicated by arrows. However, these arrows do not necessarily represent messages, as in MSCs. An example is given in Figure 3(b) for the behavior of the two collaborations *DoTest* and *LogValues* combined (see Figure 2(b)).

The semantics of the sequencing primitives that define the order in which sub-collaborations are executed are defined in [Castejon] informally, based on the semantics of Activity diagrams (with modifications). A formal definition, using partial orders of events, is given by Israr [Israr] where, in addition, performance aspects are considered.

3.2 MSC-Graphs

In their article of 1999 [Alur], Alur and Yannakakis consider model checking of MSCs. This paper contains several discussions that are useful for our purpose:

1. The paper formally defines the semantics of an MSC (basic features only) based on partial orders.
2. The paper formally defines the notation of **MSC-Graphs** which correspond to the UML notation of Interaction Overview Diagram (see for instance figure 17.27 in [UML]). An MSC-Graph is an oriented graph where each node represents an MSC and each edge represents the sequential execution of the pointed MSC after the initial MSC. It is assumed in [Alur] that all edges either represent strict sequencing (called synchronous concatenation) or weak sequencing (called asynchronous concatenation). However, in this paper we assume that for each edge the type of sequencing can be specified separately (similar as in the collaboration notation discussed in Section 3.1).
3. The paper formally defines **Hierarchical MSCs** which is an extension of MSC-Graphs were a node may also represent another MSC-Graph or Hierarchical MSC. However, it is assume that there is no recursion in this dependency. It is shown how a Hierarchical MSC can be flattened in order to obtain an equivalent (more complex) MSC-Graph. As this notation does not introduce any additional power of description, we do not further discuss this notation in this paper.
4. The paper defines a subset of MSC-Graphs, called **bounded MSC-Graphs** which have the important property that the defined behavior is regular, that is, it can be represented by a finite state machine. Therefore, such MSC-Graphs can be model-checked (which is further discussed in [Alur]), and also, for such MSC-Graphs state machines testing methods can be applied. – An algorithm for determining whether a given MSC-Graph is bounded is also given. Essentially, it proceeds as follows: (a) The **communication graph** of an MSC has nodes corresponding to the roles (processes) of the MSC and an arc from p1 to p2 if role p1 sends a message to p2 in the MSC. (b) Given a subset S of nodes of an MSC-Graph, the communication graph of S is the union of the communication graphs of all the MSCs in the nodes of S. In such a graph, the roles that receive or send a message in some MSC of the graph are called the active processes of the graph. (c) An MSC-Graph is bounded if for each cycle c in the graph, the communication graph of the nodes on this cycle (after eliminating all non-active roles) is strongly connected.

3.3 PO-Charts

Inspired by the definition of MSC-Charts, we use in this paper the notion of PO-Charts. These charts are defined like MSC-Charts, except that each node, instead of containing an MSC, contains a **partial order with roles**, as defined in Section 3.1. **Hierarchical PO-Charts** and **bounded PO-Charts** can be defined as described for MSC-Charts in [Alur].

The main difference with MSC-Charts is the fact that for each edge representing the sequential execution between two partial orders with roles, it is indicated whether

sequencing is weak or strict. Weak sequencing (abbreviated "ws") means that sequencing is enforced for each role separately. Strong sequencing (abbreviated "ss") means that the initiating events of the second partial order may only occur after all terminating events of the first partial order have occurred. This means that a strong synchronization point is introduced at this point during the execution.

We note that Hierarchical PO-Charts are an alternative notation for defining the behavior of collaborations as discussed in Section 3.1. We prefer this notation because it has a formally defined semantics, however, it does not support directly concurrency. An example of a PO-Chart is shown in Figure 4(left): This chart defines the *Test* collaboration already shown in Figure 2(b) – with a small change of control flow.

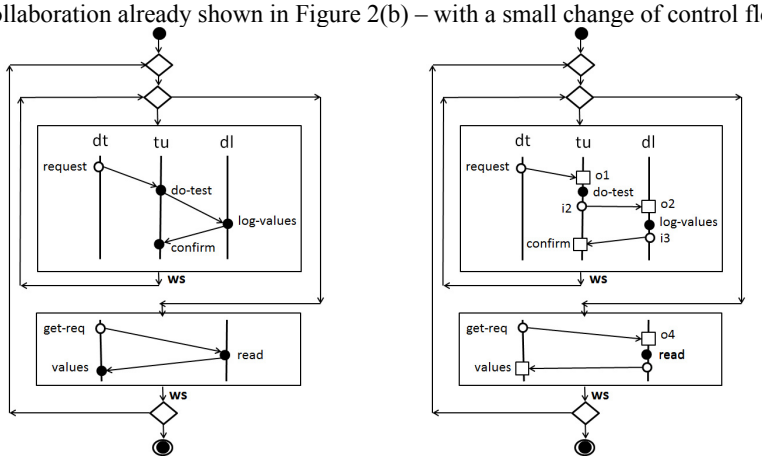

Fig. 4: (left) PO-Chart representing the *Test* collaboration. (right) The same behavior with additional input-output interactions for testing.

4 Conformance testing with respect to PO-Charts

4.1 General testing assumptions

Test architecture

For distributed systems, the test architecture has a big impact on testing. For testing a system that has several interfaces, one often uses the distributed test architecture where a local tester is associated with each interface of the system. If the local testers cannot communicate with one another, there are usually synchronization problems for controlling the order of inputs to be applied to the system and difficulties to observe the order of outputs at different interfaces. Therefore an architecture with local testers without mutual communication provides usually incomplete testing power. – In the following we assume a test architecture with local testers that can communicate with one another by message passing.

Architectures for testing distributed systems in respect to specifications in the form of MSC were described in a recent paper [Dan]. In this context, it was assumed that the processes identified in the MSC can be classified as system or user processes.

Then the user processes are replaced by local testers (that may, or may not communicate with one another). These testers exchange messages as specified by the given MSC.

We take a slightly different approach for testing distributed implementations in respect to PO-Chart specifications. We assume that each role of the PO-Chart may have a local interface to which a local tester can be attached. We assume in this paper that these local testers can communicate with one another by message passing. We assume that, at each local interface, the local tester communicates with the implementation of the role behavior through input and output interactions. These are synchronous interactions between the tester and the role implementation, without queuing. This is similar to the interactions of POIOA, although the interactions of PO-Charts are associated with a particular role.

Let us consider the example PO-Chart of Figure 4 (left). In order to define a suitable test architecture, we have to determine which of the given actions are input or output, or whether they are local actions that cannot be observed by the local tester. It may also be necessary to introduce additional input or output interactions in order to increase the power of testing. For this example system, we propose the enhanced PO-Chart of Figure 4 (right) which contains a few additional interactions with the local testers. Non-observable local actions are represented by dark dots, inputs by white dots, and outputs by white rectangles.

Test suites

Since usually no finite test suite provides the guarantee that all possible implementation faults would be detected, Bougé et al. [Bouge] suggest to consider a sequence of test suites TS_i (i= 1, 2, …) with increasing complexity, such that all faults detected by TS_i would also be detected by TS_{i+1} . Then one can talk about **validity** of such a set of test suites, which means that for any possible implementation fault, there is a test suite TS_i (for some i) that would detect this fault.

Fault models

[Bouge] also stresses the point that there are always some assumptions that are made about the tested implementations. These assumptions are often called **fault model**. The fault model defines the range of faults that should be detected by the given test suite. And at the same time these assumptions also state what properties of the implementation are assumed to be correctly implemented (and therefore need not be tested).

In this context, [Bouge] mentions the following types of assumptions that are important for justifying the selection of particular test suites:

- **Regularity hypothesis:** This is an assumption about the structure of the implementation. Assuming that we have some measure of the complexity of each test case, the regularity hypothesis states that there is a value of complexity k such that, if the implementation behaves correctly for all test cases with complexity less than k then it behaves correctly for all test cases. – In program testing, for example, one typically executes loops only once or twice and assumes that if no fault was detected then further executions of the loop will not lead to undetected faults.

- **Uniformity hypothesis:** This assumption justifies the practice in program testing where the domain of input parameters is partitioned into sub-domains and some random values are selected in each sub-domain for testing. It is assumed that, if the implementation behaves correctly for some value in a sub-domain, then it will behave correctly for all values in that domain.

How do these considerations apply to the testing of PO specifications? – The uniformity hypothesis applies to the variation of parameters of inputs to the implementation. In this paper we assume a finite set of distinct inputs where parameters can be ignored. – The regularity hypothesis takes on a particular form in the context of FSM testing methodology. The typical fault model of state machine testing assumes that the number of states of the implementation is not much larger (if not smaller or equal) to the number of states of the reference specification. Under this assumption, the test suites with test cases of bounded length can provide the guarantee of fault coverage.

For testing PO-Charts, we propose to use a regularity hypothesis similar to what is used for program testing. A PO-Chart defines possible control flows from the initial MSC-node to the final MSC-node. This is like the control flow in a program. As in program testing, we propose to test a PO-Chart by executing the different control paths through the chart, possibly using the well-known All-Branches or All-Paths criteria and executing loops typically once or twice.

4.2 Testing PO-Charts

For a bounded PO-Chart, there are essentially two approaches to test suite selection:

- The partial-order testing method proposed in this paper: The test designer should select a number of control flow paths through the PO-Chart, concatenate the partial orders of the nodes in the order of the path, using weak or strong sequencing as defined by the PO-Chart. The resulting partial order of inputs and outputs is then tested using the **partial-order test** described in Section 2.1.
- The state machine testing approach (with all its different test selection methods): Derive the state machine that has the same behavior as the PO-Chart, and then use one of the state machine testing methods.

If the PO-Chart is unbounded, the second approach is not applicable. As an example, we first discuss in the following the testing of a very simple bounded PO-Chart, and then consider an examples of unbounded PO-Charts.

4.3 Example of a bounded PO-Chart

We consider the PO-Chart shown in Figure 5(a) which contains a loop with node A containing the partial order of Figure 1(a) with an additional ordering constraint indicated by the dotted arrow. All sequencing between nodes are week sequences, alt-

hough for this example this is equivalent to strong sequencing. Figure 5(b) shows the corresponding state machine.

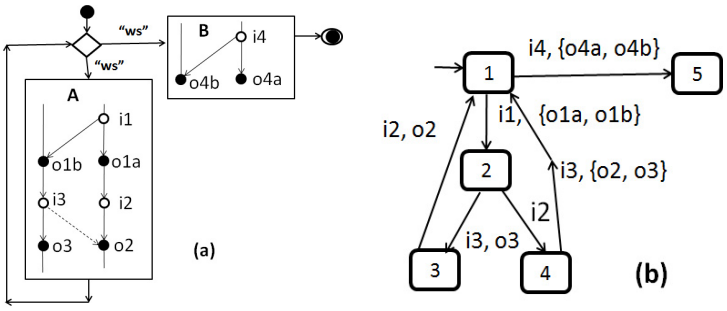

Fig. 5: (a) Example of a PO-Chart. (b) An equivalent state machine

State machine testing

The state machine of Figure 5(b) is a partially defined machine. We recall that many state machine testing methods assume that the machine is fully defined (that is, in all states for all inputs). Let us assume that input messages received by the system are stored in a message pool until the system gets into a state where the input can be consumed (this is called "full reordering of messages" in [Castejon], Section 3.1). This is advantageous in distributed systems for avoiding race conditions [Bochmann b]. If we assume this for the state machine of Figure 5(b), then in all states all inputs can be accepted – inputs for which no transition is defined in the current state will provide no output, however, the state of the system changes since the content of the pool changes.

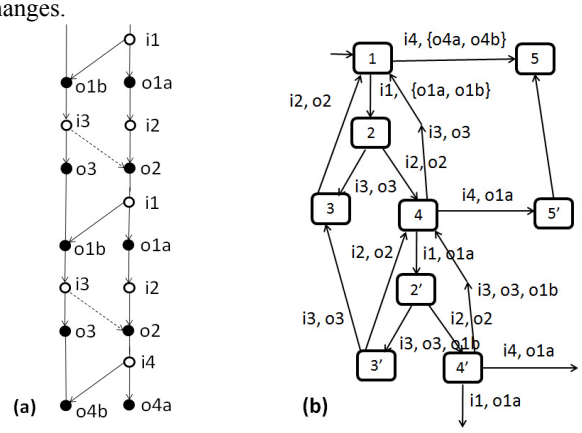

Fig. 6: (a) Partial order of path p(2) based on the PO-Chart of Fig. 5(a). (b) Part of the infinite state machine equivalent to the PO-Chart of Fig. 5(a) without the dashed dependency.

Under this assumption, the state machine of Figure 5(b) has unique input-output (UIO) sequences of length one for all states, except for state 5. Let us assume that we ignore the possible implementation fault that the i4-transition leads to one of the states 1, 2, 3 or 4. A test suite with fault detection guarantee for implementations built as state machines with not more than 5 states would first include the test cases for validating the UIO sequences. These tests would already include the test cases <i1, i2, i3, i4> and <i1, i3, i2, i4> which cover all transitions of the state machine.

Partial-order testing

This PO-Chart allows for an infinite number of execution paths p(n) – for n = 0, 1, 2, … where p(n) consists of n repetitions of the partial order of node A followed by one execution of node B. If we only test the path p(1), using the **partial order test** described in Section 2.1, we obtain the two test cases <**i1**, i2, **i3**, **i4**> and <i1, i3, **i2**, i4>. For the path p(2) – shown in Figure 6(a) - we obtain the following test cases (where the dependencies of the bold inputs are determined): <**i1**, i3, **i2**, **i1**, i3, **i2**, **i4**> and <i1, i2, **i3**, i1, i2, **i3**, i4>.

4.4 Examples of unbounded PO-Charts

Let us consider the PO-Chart of Figure 5(a) again, but now without the dashed dependency. This chart is not bounded because there is no dependency where the right process has to wait for the left process. In this case, the right process may execute a second i1 input before an i3 input is applied to the left process, and this may repeat after a second i2 input is applied to the right process.

With partial-order testing, we obtain for testing the path p(2) the following test cases: <**i1**, i3, **i2**, **i1**, i3, **i2**, **i4**> and <i1, i2, i1, i2, i4, **i3**, **i3**>. For state machine testing one may want to test an initial part of the corresponding infinite state machine. Such an initial part is shown in Figure 6(b).

Another example of an unbounded PO-Chart is shown in Figure 4(right). The shortest execution path goes through the partial orders of the two nodes only once. In this case, the **partial-order test** gives rise to the following test cases (again, the inputs for which the dependencies are tested are written in bold): <**request**, get-req, **i2**, **i3**, **i5**> and <request, i2, i3, **get-req**, i5>. The realization of a correct implementation for the behavior defined by this PO-Chart is not straightforward, as discussed in [Faleh]. An implementation using messages for the order dependencies shown in the PO-Chart does not work because in the case that the doctor terminal (dt) sends a *get-req* message immediately after the request message to the test unit (tu), the former message may arrive at the *data-logger* (dl) before the *o2* message arrives, which means that the data value returned does not include the last test measurement. Such a fault would be detected by the first test case. A well-known solution for a correct implementation is to count the number of times that the first node of the chart is executed, and include this information in the messages sent to the *data-logger* [Faleh].

We note that this implementation fault would possibly not be detectable if there was no testing interface at the *test unit* system component. On the other hand, the observation of the *confirm* output at the *test unit* is not useful for fault detection. We

note that the first test case requires some coordination between the local testers at the doctor terminal and the *test unit* in order to make sure that the *i2* input is not applied before the *get-req* input at the doctor terminal and all resulting outputs (in this case none – a timeout is assumed after each input in the test case) are observed.

5 Discussion

Specification formalism

The POIOA specification formalism has been criticized for assuming strong synchronization points in each state of the machine. This corresponds to PO-Charts in which all sequential edges between different nodes (partial orders) have strict sequencing. In contrast to the partial orders associated with POIOA transitions, the partial orders associated with a node of a PO-Chart indicates for each interaction the role (process or interface) where the interaction takes place. This additional information allows us to define weak sequencing between different nodes of a PO-Chart.

One could possibly introduce an extension to POIOA where the sequencing of incoming and outgoing transitions at each state could be explicitly specified by indicating for each initiating event of an outgoing transition what are the terminating events of an incoming transition for which this initial event must wait before proceeding (if the state was reached by that incoming transition). If it has to wait for all terminating events, then we have the situation of normal POIOA where the incoming transition must be completely terminated before an outgoing transition may start. Weak sequencing could also be specified if the roles of the events are known. However, we are not convinced that such a generality of defining the sequence of transitions is useful – we prefer the addition of the role information (as defined for PO-Charts) which automatically defines the semantics of weak sequencing (if this is the desired form of sequencing).

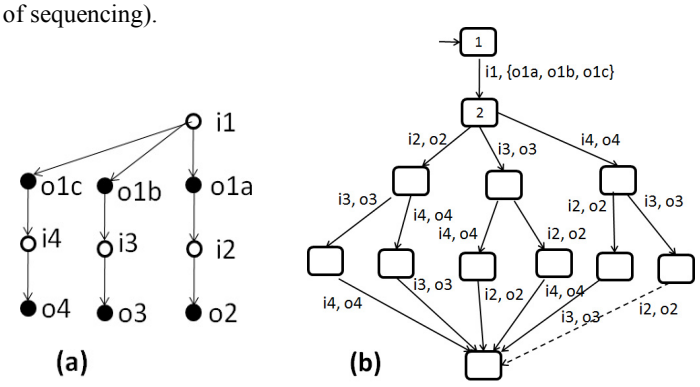

Fig. 7: (a) A partial order with roles. (b) An equivalent state machine.

Testing complexity

It was argued in [Bochmann a] that the **partial order test**, as discussed in Section 2.1, is of much lower complexity than state machine testing when applied to systems

with concurrency. This does not show up in the simple examples discussed above. But it is clear that the number of states of a state machine that is equivalent to a partial order grows exponentially if the degree of concurrency in the partial order increases. With the number of states of the reference specification, also the length of the test suite will grow accordingly.

As an example, we consider here a variation of the partial order of Figure 1(a) where we assume that three inputs i2, i3 and i4 are enabled after the input i1 (see Figure 7(a)). The corresponding state machine is shown in Figure 7(b). The state machine testing approach requires at least 6 test cases to cover all the branches of the state machine. With the partial order approach, we obtain the three test cases <**i1**, i2, i3, **i4**>, <i1, i3, i4, **i2**>, and <i1, i2, i4, **i3**>.

Different fault models

One may suspect that the lower complexity of the partial-order tests implicitly implies that their fault coverage is lower. This is in fact true. The fault model for which the partial-order tests provide fault coverage makes stronger assumptions about the tested implementation than the fault model used with state machine testing.

It was shown in [Bochman a] that the partial-order test method provides complete fault coverage under the assumption that the tested implementation has the behavior that can be defined by a single partial order (without alternatives). That is, a transition of the POIOA model is implemented as a single transition in the implementation POIOA. An example of an implementation that does not satisfy this assumption would be an implementation that has the behavior of Figure 7(b) with a single output fault in the dashed transition. Such a fault is not detected by the test suite given above, and this faulty implementation cannot be described by a single partial order.

6 Conclusions

For describing the behavior of distributed systems with multiple interfaces, one needs the notion of partial order for the interactions at the different interfaces, since there is no total order defined for all the interactions of the system. We have compared different notions of partial-order specifications, including POIOA, ordering of collaborations, and MSC-Charts (also called Interaction Overview Diagrams). We propose the use of a variant of the latter, called Partial-Order-Charts (PO-Charts). We have shown that for the testing of distributed systems in respect to such behavior specifications, the partial-order tests of [Haar] can be used.

In the case that the PO-Chart is bounded, one can also derive an equivalent state machine and use FSM testing methods. We provided in this paper a preliminary comparison of testing complexities and fault models for these two testing approaches (partial-order tests and FSM testing methods). For systems with much concurrency, the partial-order tests are advantageous if one can assume that the fault model of partial-order testing is satisfied. This is presumably the case when the implementation uses message passing between the different system components to implement the order dependencies defined in the specification.

Acknowledgements

I would like to thank Guy Vincent Jourdan for many fruitful discussions on testing POIOA.

References

1. [Alur] R. Alur and M. Yannakakis, Model checking of Message Sequence Charts, Proc. CONCUR<99, Springer LNCS 1664, pp. 114-129, 1999.
2. [Bochmann a] G. v. Bochmann, S. Haar and G. V. Jourdan, Testing systems specified as partial-order inputoutput automata, Proc. IFIP Testcom/FATES Workshop, Tokyo, June 2008, LNCS.
3. [Bochmann b] G. v. Bochmann, Deriving component designs from global requirement, In: Baelen, S.V., Graf, S., Filali, M., Weigert, T., Gerard, S. (eds.) Proceedings of the First International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB 2008), Toulouse. CEUR Workshop Proceedings, vol. 503, pp. 55-69 (2008).
4. [Bochmann c] G.v. Bochmann and G.V. Jourdan, Partial Order Input/Output Automata: Model and Test, unpublished document.
5. [Bouge] L Bougé, N. Choquet, L. Fribourg and M.C. Gaudel, Test sets generation from algebraic specifications using logic programming, Journal of Systems and Software 6, 3pp. 343-360, 1986.
6. [Castejon] (H. N. Castejòn, G. v. Bochmann and R. Braek, On the realizability of collaborative services, Journal of Software and Systems Modeling, Vol. 10 (12 October 2011), pp. 1-21.
7. [Dan] H. Dan and R. M. Hierons, Conformance testing from Message Sequence Charts, Proc. 4th Intern Conf. on Software Testing, Verification and Validation (IEEE), pp 279-288, 2011.
8. [Faleh] M. N. M. Faleh and G. v. Bochmann, Transforming dynamic behavior specifications from Activity Diagrams to BPEL, Proc. IEEE 6th Intern. Symp. on Service-Oriented System Engineering, Irvine, Calif., Dec. 2011, pp. 305-311.
9. [Haar] S. Haar, C. Jard, and G.-V. Jourdan, "Testing input/output partial order automata," Proc. TestCom '07 / FATES '07: Springer LNCS 4581, 2007, pp. 171–185.
10. [Israr] T. Israr, G.v. Bochmann, Performance modeling of distributed collaboration services with independent inputs-outputs, Proc. of 5th Intern. Workshop on Non-functional Properties in Modeling: Analysis, Languages and Processes co-located with 16th Intern. Conf. on Model Driven Engineering Languages and Systems, Miami, USA, September 29, 2013.
11. [Simao] A. Simao and A. Petrenko, "Generating asynchronous test cases from test purposes," Information and Software Technology, vol. 53, p. 12521262, 2011.
12. [UML] OMG Unified Modeling Language, Version 2.5, March 2015, http://www.omg.org/spec/UML/2.5/PDF/ (accessed June 2016)