Implementing MSC Tests with Quiescence Observation

Sergiy Boroday[1], Alexandre Petrenko[1], Andreas Ulrich[2]

[1] Centre de recherche informatique de Montreal (CRIM), 550 Sherbrooke West, Suite 100
Montreal, Quebec, Canada
Sergiy.Boroday@crim.ca, Alexandre.Petrenko@crim.ca
[2] Siemens AG, Corporate Technology, CT SE 1, 80200 Munich, Germany
Andreas.Ulrich@siemens.com

Abstract. Given a test scenario as a Message Sequence Chart (MSC), a method for implementing an MSC test in a distributed asynchronous environment is suggested. Appropriate test coordination is achieved using coordinating messages and observed quiescence of a system under test. A formal definition and a classification of faults with respect to the test scenario are introduced. It is shown that the use of quiescence observation improves the fault detection and allows implementing sound tests for a wider class of test scenarios than before.

Keywords: Distributed testing, Message Sequence Charts, sound tests, test implementations, fault detection power.

1 Introduction

With the recent trend towards multi-core processing, multithreading, and system-on-chip development as well as the emergence of Web services and service oriented architecture, asynchronous communication paradigms, concurrency, and distribution become mainstream in software development. This trend poses high strains on the testing process that needs to deal with the concurrent and distributed nature of such systems.

Various proprietary and open-source platforms for distributed testing have emerged in diverse application areas. While quite a variety of programming and scripting languages are used in the testing domain, including TTCN-3, an internationally standardized testing language, a common choice for designing and visualizing tests is the use of Message Sequence Charts (MSC) and derivatives, such as UML2 Sequence Diagrams [5, 6, 9].

Development of supporting tools for distributed testing faces a number of technical challenges. One of them is the fact that test components of the tester have often to be deployed in different environments. However, in our opinion, the major obstacle limiting the success of distributed testing is the lack of a solid theoretical foundation and efficient distributed test generation algorithms and tools.

In this paper, we study the problem of implementing a test derived from a test scenario specification described by an MSC. We focus on a distributed asynchronous environment to run tests, where several independent test components communicate with the system under test (SUT) and between themselves via FIFO channels. The contributions in this paper are twofold. First, a classification of faults that characterize the faulty functional behavior of a distributed system at its external interfaces is proposed. Second, we elaborate a way of observing the SUT quiescence in order to increase the fault detection power of the implemented tests, thus improving previous work in this respect. Based on the suggested classification, the fault detection power of the presented test implementation algorithms is determined.

Section 2 introduces the problem of distributed testing and discusses briefly the related work. Afterwards, the theoretical foundations of the proposed approach, which is based on the poset theory, are laid down in Section 3. In Section 4, the problem of soundness of test implementations is addressed and an algorithm for obtaining a sound test implementation from a given test scenario is elaborated. Section 5 introduces a systematic analysis of distributed system faults resulting in an improved algorithm with a higher and defined degree of fault detection power. Section 6 concludes the paper.

2 The Distributed Testing Problem

Given a distributed system, hereafter called SUT, we assume that a test system is also distributed, executing tests by several cooperating test components, which access different ports/channels of the SUT. All the communications are assumed to be bidirectional, asynchronous and use unbounded FIFO channels. We consider that a test specification is available as an MSC describing the expected order of test and SUT events in a particular test scenario. Thus, test specifications contain no explicit verdicts, which are produced by a designated Master Test Component (MTC). The verdict *pass* should indicate that no error in the SUT has been detected when a given test is executed, namely, that all the expected test events (and only them) have occurred in a correct order. Errors in the SUT manifest themselves as violations of these conditions and should trigger the verdict *fail*. Note that sometimes other verdicts, such as *inconclusive*, are also considered, e.g., within the *ptk* tool [3].

The main challenge in implementing a test specification is to ensure that all the test events are executed in the right order, especially if test components run concurrently and in a distributed way. To address this problem, a proper coordination between test components has to be added to obtain a test implementation. The test engineer can be relieved from solving this task manually by tools that automate the transformation of a test specification into a test implementation with a required coordination. With a proper coordination of test components a test implementation becomes sound, i.e., it avoids false positives (false alarms), when a correct SUT is considered to be faulty by a flawed test implementation; moreover, coordination can improve fault detection, i.e., reduce the number of false negatives. The problem of soundness is important because false positives produced due to ill-conceived coordination can confuse the test engineer and discourage her from further using the test tool. The fault detection power of a test is equally important; although detection of all the possible faults is hardly feasible in distributed testing as shown later. Devising coordination mechanisms in test implementations such that they are sound and have a guaranteed fault detection capability especially when communication delays may mask SUT faults is the main subject of this paper.

Major work on the generation of distributed tests that has been reported previously is given in [1, 2, 3]. The proposed algorithms can deal only with a subclass of distributed test specifications. While the problem of guaranteeing the detection of deviations of the observed behavior from the expected behavior is resolved with the assumption of negligible latency of coordinating messages compared to SUT messages [2], we propose a different solution based on the assumption of the availability of a *null* event (a timeout which suffices for the SUT or test component to send and receive messages over FIFO channels). This event occurrence is interpreted as the observation of the SUT quiescence and is used to resolve races. A designated quiescence observation output was previously used in automata-theoretical testing theories, see, e.g., [15]. An MSC adaptation of the automata based testing approach with quiescence is suggested in the context of centralized testing [17].

The problems of soundness of test implementation and fault detection, formalized in this paper resemble the controllability and observability problems encountered in distributed FSM-based testing [16].

Fault detection in distributed systems is discussed in [7], where the expected behavior is modeled by a partial order input/output automaton which can be viewed as an HMSC. While the work [7] focuses on the generation of several tests to discover faults in implementations of a given transition, we aim to obtain a single test implementation for a test scenario given as an MSC and target the necessary coordination among test components. If several test scenarios are given then we assume that they are mutually output (or verdict) consistent [18].

The work in [10] discusses the synthesis of concurrent state machines from an MSC. The authors consider an MSC scenario description as a closed system (complete MSC). Thus, their contribution is more relevant to the system design process than to the test process. The main difference here is that from the provided scenario a part that represents the test specification (incomplete MSC) has to be extracted before an implementation can be synthesized. It turns out that the test synthesis algorithms are quite different from the algorithms that synthesize the complete, i.e., closed, system. A similar work that considers the behavior model synthesis from scenarios as closed systems can be found in [13].

The authors of [12] present another method to generate tests from UML models using the TGV tool. The obtained tests are represented by input/output transition systems, which can then be transformed into sequence diagrams. In our work, we avoid such an intermediate representation of tests and are more concerned with races and fault detection in the resulting test implementations.

3 A Poset Theoretical Framework for Analyzing Distributed Tests

3.1 Posets

A *binary relation* ρ over a ground set E is a subset of $E \times E$. A transitive closure of ρ is the smallest transitive relation $[\rho]$ over E that contains ρ. A (strict) *partial order* over a set E is an irreflexive transitive and anti-symmetric binary relation over E. A *poset* is a pair of a set E and a partial order over this set.

Two elements e_1 and e_2 of a poset $(E, <)$ are *incomparable* if neither $e_1 < e_2$ nor $e_2 < e_1$. In the context of distributed systems, incomparable elements represent concurrent events. A poset element e_1 *immediately* precedes (follows) another element e_2 if $e_1 < e_2$ ($e_2 < e_1$) and there exists no $e \in P$ such that $e_1 < e < e_2$ ($e_2 < e < e_1$).

A *cover relation* of a partial order is the minimal relation, the transitive closure of which equals the partial order.

A poset $P_1 = (E_1, <_1)$ is called the *restriction* of the poset $P_2 = (E_2, <_2)$ onto E, and denoted $P_2 \downarrow E$, if E_1 is the restriction of E_2 onto E, i.e., $E_1 = E_2 \cap E$ and $<_1$ is the restriction of $<_2$ onto $E_1 \times E_1$, i.e., $<_1 = <_2 \cap E \times E$.

A poset $P_1 = (E, <_1)$ is *finer* than a poset $P_2 = (E, <_2)$ if $<_1 \supseteq <_2$, that is the former partial order contains the latter partial order. In this case, we also say that P_2 is *coarser* than P_1. Note that one poset can be finer or coarser than another poset only if both share the same ground set. A total order that is finer than a given poset is a *linearization* of the given poset. Given a set E, a poset $P_1 = (E_1, <_1)$ is *finer* than a poset $P_2 = (E_2, <_2)$ *on E* if the restriction of P_1 onto E is finer than the restriction of P_2 onto E.

3.2 Messages, Events, and MSCs

An MSC describes message exchange in terms of message send and receive events. Beside these communication events, events not related to the message exchange are sometimes considered too, and are called *local* events.

Given a set of all possible messages, we associate each message m with two distinct events, send of the message $!m$ and the corresponding receive $?m$. Broadcast is not allowed, that is one send event is always matched with exactly one receive event. Thus, send-receive matching is a bijective mapping of sends into receives. This matching is also understood as a send-receive precedence partial order, *match,* where each send precedes the matching receive.

An *MSC* is a collection of pairwise disjoint event posets, called here *local* posets, such that no matching send and receive belong to the same local poset. Each local poset represents the behavior of an MSC instance/liveline. The partial orders of the local posets are referred to as *local* orders.

Thus, the MSC considered in this paper can describe complex concurrent behavior usually visualized with parallel expressions or co-regions, but not branching (alternatives), cycling, or timed behavior.

Here we introduce a designated local event, called *null*. The null event models observation of quiescence, which is usually implemented in practice with a sufficiently long timer. The left-hand part of Fig. 1 shows an MSC, where a timer start and timeout indicate the absence of events at T_1 during 10 time units. With a dotted arrow we show that if time progresses at the same rate for all the instances, the occurrence of a timeout on one instance can have implications on the timing of events on another component.

In this paper, we in fact assume that time does progress at the same rate for all the instances and rely on this assumption in interpreting null events. We represent in

MSC timer start and timeout events as a single local null event, as shown in Fig. 1 on the right. The formal meaning of the null event is explained later.

We allow an MSC to be incomplete, namely, some sends or receives do not necessarily have counterparts. In this work, MSCs are used in the context of testing. Thus, for simplicity, we assume that in a complete MSC, there is one distinguished instance SUT that represents the behavior of the system under test, while all the other instances represent test components. The messages which neither originate from nor arrive to an SUT are called *coordinating* messages.

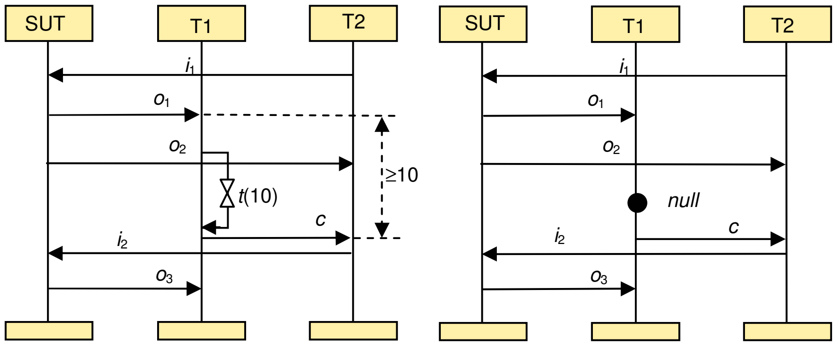

Fig. 1. An MSC with a null event

3.3 Causal Order, Enforceable Order, and Races of MSCs

We recall the notions of a causal order and a race of an MSC. Let E be a set of all the events of an MSC M. The MSC M is called *consistent* (in asynchronous semantics) if there exists a partial order which is finer than the send-receive precedence partial order *match* on the set E as well as the local orders, and which respects FIFO ordering condition, meaning that if two send events of an instance are ordered in this partial order, then the matching receive events of the same instance are similarly ordered. The coarsest partial order on the set E, which satisfies the above conditions, is called the *causal* order $<_M$, forming with E the *causal* poset of the MSC. If the above partial order does not exist, MSC M is *inconsistent.*

A race occurs between events which are ordered in the MSC, but this order is not enforceable. The *enforceable* partial order of a consistent MSC M is the coarsest partial order which is finer than the send-receive precedence partial order *match* and which respects FIFO ordering condition, ordering of an event and a later send or local (i.e., null) event of the same instance.

Unlike the causal order, the enforceable order describes only the ordering which could be guaranteed in a distributed system [4]. For instance, it is impossible to enforce the order of consecutive receives of the messages sent by different instances, since channel delays are a property of channels and not the receiving instance. Thus, an MSC M contains a *race* when its enforceable partial order differs from the causal order $<_M$, otherwise it is *race-free* [4].

3.4 Causal Order, Enforceable Order, and Races of MSCs with Null Events

Now we consider MSCs which can have designated null events, which do not occur on the SUT instance. Informally, a null event models a delay sufficient for the SUT to become quiescent (stable) and all messages in transit to arrive. As we show later, assuming that all the SUT messages arrive before the null event occurs allows us to build simpler test implementations than before with weaker assumptions. Our assumption implies that while the null event is technically treated as a local event, it affects ordering of events on all the other instances. Consider the example in Fig. 1. The MSC on the right-hand side of Fig. 1 is not race-free since events $?o_2$ and $?c$ are causally ordered and this ordering is not enforceable. This means that if the null event is just a usual local event, which does not satisfy the above assumption, the message o_2 may in fact arrive after c. It may happen when the latency of message o_2 exceeds a delay imposed by the null event, and the latency of the coordinating message is negligible. However, if the delay modeled by the null event at T_1 allows for all SUT messages, including o_2 to arrive, c cannot be sent before the arrival of o_2. As one can see from the diagram on the left-hand side of Fig. 1, the time interval between $!i_1$ and $?c$ is not smaller than the delay represented by the null event, unless time passes with different speed on different components, which is excluded by our assumptions. Therefore, message o_2 must arrive prior to c. Thus a null event on one instance does affect orderings of the events on other instances. This effect is formally defined as follows.

Let $<_\delta$ be the coarsest partial order over the events of an MSC M, called the *null-enforcing* order, such that

- for each send event of a message i to the SUT and each null event e_{null}, such that $!i <_M e_{\text{null}}$, it holds for the matching receive event $?i$ that $?i <_\delta e_{\text{null}}$;
- for each send event $!o$ of the SUT and each null event e_{null}, if for each $!i$, $!i <_M !o$ implies $!i <_M e_{\text{null}}$ then $?o <_\delta e_{\text{null}}$;

where $<_M$ is the causal order of the MSC M.

Based on this notion, we formally introduce the causal and enforceable orders, as well as the notions of race and consistency for MSCs with null events.

The causal order of an MSC with null events is defined as for an MSC without them taking additionally into account the ordering imposed by null events. An MSC with null events is called δ-*consistent* if there exists a partial order which is finer than both the causal order $<_M$ and the null-enforcing order $<_\delta$. The coarsest partial order, which satisfies these condition is called the δ-*causal* order and denoted $<_M^\delta$. Along with the set of all the events of the MSC (including null events) it forms the δ-*causal* poset.

For a δ-consistent MSC, the δ-*enforceable* order is the coarsest partial order which is finer than both the enforceable order and null-enforcing order. An MSC with null events has a δ-*race* if its δ-enforceable order differs from the δ-causal order, otherwise the MSC is δ-*race-free*. Note that a race-free MSC is also δ-*race-free*. Indeed, δ-enforceable and δ-causal orders are both transitive closures of the union of the null-enforcing order with the enforceable and causal orders, respectively. Thus in a race-free MSC, not only enforceable and causal, but also δ-enforceable and δ-causal orders coincide.

3.5 Test Scenario, Specification, and Implementation

We consider that the process of test implementation begins when the test designer defines a test scenario, which describes the expected order of message receives and sends by the SUT and test components of a test system. We define a *test scenario* $test_{\mathrm{scen}}$ as a complete and consistent MSC, where only one non-empty instance, called the SUT (system under test), communicates with all the other instances, constituting a *test specification* $test_{\mathrm{spec}}$ which specifies the behavior of the test components. In other words, the test scenario MSC has neither local events nor communications between test components. In a given test scenario, $(E^{\mathrm{SUT}}_{\mathrm{spec}}, <^{\mathrm{SUT}}_{\mathrm{spec}})$ denotes the SUT instance, and $test_{\mathrm{spec}} = \{(E^{1}_{\mathrm{spec}}, <^{1}_{\mathrm{spec}}), (E^{2}_{\mathrm{spec}}, <^{2}_{\mathrm{spec}}), \ldots\}$ is a test specification, where $(E^{i}_{\mathrm{spec}}, <^{i}_{\mathrm{spec}})$ is the specification of the i-th test component. Similarly to the preceding work, we require that each test component specification coincides with the restriction of the casual order of the test scenario onto the event set of this component.

It has to be noted that numerous testing techniques use the notion of test purpose rather than test scenario, e.g., in [1], [8]. We define a *test purpose tp* of a test scenario as a poset of events of test components $E^{1}_{\mathrm{spec}} \cup E^{2}_{\mathrm{spec}} \cup \ldots$ and the partial order $<_{tp}$ such that one event e_1 immediately precedes another event e_2 in $<_{tp}$ whenever the matching event of e_1 precedes the matching event of e_2 in $<^{\mathrm{SUT}}_{\mathrm{spec}}$ and either both events belong to the same test component or at least one of them is a send event. Save for consecutive receive events of different test components, the order of which is relaxed in *tp*, the test purpose mirrors $<^{\mathrm{SUT}}_{\mathrm{spec}}$: two events are ordered in $<_{tp}$ if and only if their matching events are similarly ordered in $<^{\mathrm{SUT}}_{\mathrm{spec}}$. The order of the receives of different test components is relaxed to reflect the fact that in an asynchronous distributed system, messages consecutively sent by the SUT via different channels could arrive in any order due to variable communication delays. However, other receive events of different test components still can be ordered in the test purpose by transitivity.

Thus, in our framework, a test scenario refers to the behavior of a closed system, while a test purpose refers to that of an open system, which excludes the SUT. Unlike the test specification, the test purpose describes also the order of events that belong to different test components.

It is known [1, 2, 3] that distributed tests may have races, thus in a distributed environment special implementation efforts are needed to enforce a specified order of events. Races are usually resolved by introducing additional coordinating messages. However, here we also rely on null events to construct implementations of the test specifications resolving races related to the SUT. Now we define a test implementation formally.

Let $test_{\mathrm{spec}}$ be a test specification. A test *implementation* $test_{\mathrm{imp}}$ of $test_{\mathrm{spec}}$ is an MSC each instance $(E^{i}, <^{i})$ of which represents a test component T_i such that $(E^{i}, <^{i}) \downarrow E^{i}_{\mathrm{spec}} = (E^{i}_{\mathrm{spec}}, <^{i}_{\mathrm{spec}})$. Thus, besides the events of the test specification, the test implementation can have other events which are send and receive events of coordinating messages or null events. We use $<_{\mathrm{imp}}^{\delta}$ to denote the δ-causal order of a test implementation. Note that since a test scenario has no null events, the causal order of a test scenario coincides with its δ-causal order.

In the rest of the paper, we suggest two algorithms for constructing test implementations and discuss their fault detection power. While the first suggested algorithm misses some faults, it has a lighter coordination mechanism and thus employs fewer coordinating messages and null events. The second algorithm extends the first one with additional coordination mechanisms in order to detect more faults (up to the maximum possible in our framework).

4 Sound Test Implementations

4.1 Soundness

Soundness is an important property of a test implementation, which, informally, means that the test implementation composes with a correct SUT without deadlocks. Thus, we introduce the following definition of MSC composition.

A (*horizontal*) *composition* $M_1 \cup M_2$ of two MSCs M_1 and M_2 with the disjoint sets of events is an MSC M that contains all the local posets of M_1 and M_2 and only them.

A test implementation $test_{\mathrm{imp}}$ of $test_{\mathrm{spec}}$ is *sound* with respect to the test scenario $test_{\mathrm{scen}}$ if $test_{\mathrm{imp}} \cup \{(E^{\mathrm{SUT}}_{\mathrm{spec}}, <^{\mathrm{SUT}}_{\mathrm{spec}})\}$ is δ-race-free.

If a δ-race occurs on the SUT, i.e., when certain SUT events are ordered causally but their order is not enforceable, then some messages can be consumed by the SUT in an order inconsistent with $test_{\mathrm{scen}}$. If δ-race occurs on the test implementation, the actual order of events on test components could diverge from the expected one even if the SUT is correct.

When a test specification includes concurrent messages to the SUT, its implementation may either obey or eliminate such concurrency. This motivates the following definition. A test implementation $test_{\mathrm{imp}}$ is *concurrency preserving* if whenever two SUT receive events, or an SUT receive event and an SUT send event are unordered in $<^{\mathrm{SUT}}_{\mathrm{spec}}$, the matching events of the test implementation are also unordered in the δ-causal order of the test implementation.

Concurrency preserving test implementations do not order concurrent send events and thus the matching SUT receive events. In other words, they do not exclude any particular linearization of SUT receive events, on which faults may occur. Moreover, such test implementations preserve their soundness during the refinement of test scenarios. We say that a test scenario $test_{\mathrm{scen}}'$ is a *refinement* of another test scenario $test_{\mathrm{scen}}$ if their instances share the same set of events, $<^{\mathrm{SUT}}_{\mathrm{spec}}{}'$ is finer than $<^{\mathrm{SUT}}_{\mathrm{spec}}$, and whenever an event immediately follows another event in $<^{\mathrm{SUT}}_{\mathrm{spec}}{}'$ but not in $<^{\mathrm{SUT}}_{\mathrm{spec}}$, the former event is a send event. This means that a refinement of a test scenario includes an added dependency of an SUT send event to previously concurrent send or receive events.

Proposition 1. If $test_{\mathrm{scen}}'$ is a refinement of $test_{\mathrm{scen}}$ then a concurrency preserving test implementation sound with respect to $test_{\mathrm{scen}}$ is also sound with respect to $test_{\mathrm{scen}}'$.

Hereafter, we consider test implementations which are both concurrency-preserving and sound. Obviously, if in a given test scenario, every SUT receive is ordered in $<^{\mathrm{SUT}}_{\mathrm{spec}}$ with respect to each other SUT event then each sound test implementation is trivially concurrency preserving. Hereafter, such a test scenario is called *input-sequential*. Most of the previous work is restricted to input-sequential test scenarios. Here, we construct sound test implementations for arbitrary test scenarios; however we claim the highest fault detection power only for test implementations derived from input-sequential test scenarios.

4.2 Generating Sound Test Implementations

To construct a test implementation of a given test specification, it is possible to use the approach proposed in [1] for input-sequential test scenarios. According to this approach, coordinating messages and co-regions are added to the test specification MSC to obtain a test implementation. Concurrency is introduced in the MSC test implementation as a co-region which contains receive events of coordinating and SUT messages. The approach effectively resolves races among these messages. The synchronization ensures that the next message to the SUT is not sent until all the expected messages are received.

The resulting test implementation is not concurrency preserving, since it always sends messages to the SUT sequentially. There are in fact some enhancements of this approach in [2, 3]. They allow concurrent events in the test scenario, but order in fact concurrent SUT receive events prior to the construction of a test implementation. Moreover, races between coordinating messages are resolved due to an additional assumption on negligible latency of coordinating messages and the use of the *inconclusive* verdict.

The work in [1] does not claim that the resulting test implementations are always sound. The reported tool relies more on the test engineer to define synchronization than on a systematic procedure. A more recent technique [2] yields sound test implementations only for a restricted class of test specifications without consecutive SUT receive events. In this case, races, called irresolvable blocking conditions in [14], cannot be resolved. Here, we use the SUT quiescence observation to solve this problem.

Consider the test scenario example with consecutive receive events in Fig. 2. The coordinating message enforces the desired order of send events in the test implementation, however, the race between the two SUT receive events remains, and thus, the test implementation is unsound. Since the test scenario only describes the behavior of the SUT when i_1 arrives before i_2, in the case when i_2 wins the race the SUT behavior is not specified. Assume that there is another scenario where the SUT in response to $?i_2$ followed by $?i_1$ produces $!o_2$ and then $!o_1$. Then the test execution with a correct SUT may proceed as shown in the right part of Fig. 2. This behavior leads to the *fail* verdict since the messages sent by the SUT arrive in the order different from what is expected by T_2. This contradicts the intuition behind soundness. Formally speaking, the composition of $(E^{\mathrm{SUT}}_{\mathrm{spec}}, <^{\mathrm{SUT}}_{\mathrm{spec}})$ and the test implementation on the left-hand part of Fig. 2 is not race-free, since, according to the definition of the causal and enforceable orders, the receive events $?i_1$ and $?i_2$ are causally ordered, but this order is not enforceable. To resolve the race it suffices to add a null event (delay) between $?c$ and $!i_2$ in T_1. This delay allows message i_1 to arrive prior to sending message i_2, or, formally, the null event follows $?i_1$ in the null-enforcing order, which contributes both to the enforceable and causal orders, thus resolving the race.

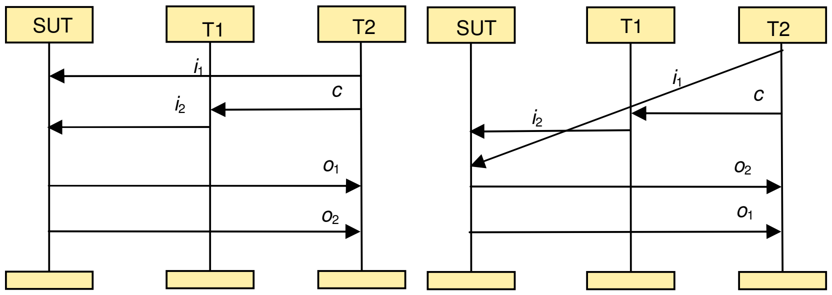

Fig. 2. Unsound test implementation and its possible execution

To construct concurrency preserving sound test implementations we extend the approach in [1] using null events as follows. Unlike [2, 3], to obtain concurrency preserving test implementations, concurrent SUT receive events are kept unordered. The idea is that when several messages need to be sent to the SUT concurrently by several test components, all of them have to be notified by a coordinating message by each test component as soon as it receives all the expected concurrent messages from the SUT. The notified test components send messages to the SUT only upon receiving all coordinating messages, which are treated as concurrent events. Thus ordering of coordinating messages is avoided. Whenever one send event $!i$ immediately follows a send event of another test component in the test purpose tp, one test component notifies the other with a coordinating message, so the second send event occurs only after the first one. Having received the coordinating message the last test component executes the null event. This delay allows the SUT to receive all the messages just sent. If the send event $!i$ immediately follows several send events and not only one, the null event is executed only after the receive events of all the coordinating messages, preceding $!i$ in the constructed test implementation. The above discussion leads to the following algorithm.

Algorithm 1.
Input: An MSC test scenario $test_{\text{scen}}$ with the SUT specification $(E^{\text{SUT}}_{\text{spec}}, <^{\text{SUT}}_{\text{spec}})$ and test components specifications $(E^{1}_{\text{spec}}, <^{1}_{\text{spec}})$, $(E^{2}_{\text{spec}}, <^{2}_{\text{spec}})$, ….
Output: A test implementation $test_{\text{imp}}$.

1. The initial relations $R_1, R_2, \ldots$ defining the test components $T_1, T_2, \ldots$ are the cover relations of $<^{1}_{\text{spec}}, <^{2}_{\text{spec}}, \ldots$.
2. $k := 1$.
3. Add coordinating messages c_k as follows:
   For each j and each event e of E^{j}_{spec}, and each l, $j \neq l$, such that some send $!i$, $!i \in E^{l}_{\text{spec}}$, immediately follows e in the test purpose tp do
   - $R_l := \{(?c_k, !i)\} \cup R_l$ for each send $!i$ of E^{l}_{spec}, which immediately follows e in tp;
   - $R_j := \{(e, !c_k)\} \cup R_j$;
   - $k := k + 1$.
4. $m := 1$.

5. Add null events as follows:
   For each test component j and for each send $!i$ of E^j_{spec} which immediately follows a send of another test component in tp do
   - $R_j := R_j \cup (e_{\text{null}}{}^m, !i);$
   - for each non-null event e such that $(e, !i) \in R_j$, $R_j := R_j \cup (e, e_{\text{null}}{}^m)$;
   - $m := m + 1$.
6. Determine a transitive closure of each R_j, the resulting partial orders define the test components of the test implementation $test_{\text{imp}}$.

Proposition 2. The test implementation obtained by Algorithm 1 is sound and concurrency preserving.

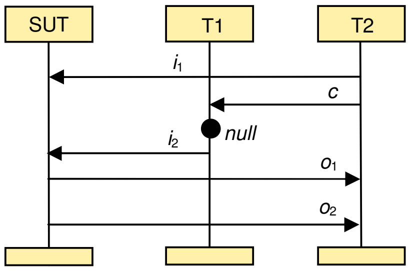

Fig. 3. A sound test implementation composed with the SUT

The corrected, sound version of the test implementation in Fig. 2 constructed by Algorithm 1 is $\{T_1 = (\{?c, e_{\text{null}}, !i_2\}, \{(?c, !i_2), (?c, e_{\text{null}}), (e_{\text{null}}, !i_2)\}), T_2 = (\{!i_1, !c, ?o_1, ?o_2\}, \{(!i_1, !c), (!i_1, ?o_1), (!i_1, ?o_2), (!c, ?o_1), (!c, ?o_2), (?o_1, ?o_2)\})\}$. The composition of this test implementation with the SUT specification is shown in Fig. 3. Note that the event $!c$ precedes the event $?o_1$ in the causal relation of the composition of the test implementation with a correct SUT.

Concluding this section, we stress that the proposed algorithm for building test implementations ensures soundness, which is different from the related work in [1], and is applicable to a wider class of test specifications allowing consecutive SUT receive events compared to [2].

5 Fault Detection

5.1 Detectable and Undetectable Faults

While soundness is an important characteristic of a test implementation, another essential feature is its ability to detect faults. The behavior of an SUT on a sound test implementation becomes erroneous once it produces a trace of events that cannot be produced by the SUT specification $(E^{\text{SUT}}{}_{\text{spec}}, <^{\text{SUT}}{}_{\text{spec}})$ of a test scenario. Such observed errors are typically caused by faults within the SUT. Aiming at test implementations that detect as many faults as possible we formally define and classify faults as follows.

An erroneous trace contains an *output* fault if the set of events in the trace does not coincide with the set $E^{\mathrm{SUT}}_{\mathrm{spec}}$ due to missing or superfluous SUT send events, which are not in $E^{\mathrm{SUT}}_{\mathrm{spec}}$. Output faults are detected by any sound test implementation since they always lead to a deadlock. To detect output faults a Master Test Component (MTC) uses a timer to identify deadlock of a test component which has not received an expected message and thus has not informed the MTC about its successful termination. Faults which occur when the SUT executes the expected send events, but in a wrong order, are more difficult to detect, since communication delays may mask such faults. Consider the example in Fig. 4, where message o_2 occurs out-of-order (left-hand: the test scenario, right-hand: the erroneous test execution). The premature SUT send event of message o_2 goes undetected with a sufficiently long delay in the channel from the SUT to the second test component.

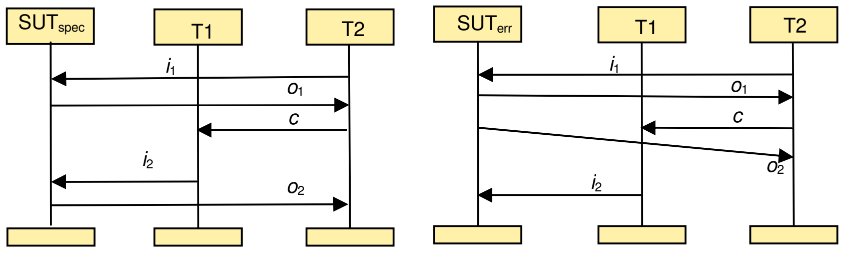

Fig. 4. Premature send event masked by a communication delay

An SUT trace $t = a_1 \ldots a_n$ over the set $E^{\mathrm{SUT}}_{\mathrm{spec}}$, where $n = |E^{\mathrm{SUT}}_{\mathrm{spec}}|$, has an *order* fault $a_1 \ldots a_k$, $k \leq n$, with respect to the test scenario $test_{\mathrm{scen}}$, if

- $a_1 \ldots a_{k\text{-}1}$ is a prefix of a linearization of $(E^{\mathrm{SUT}}_{\mathrm{spec}}, <^{\mathrm{SUT}}_{\mathrm{spec}})$, but $a_1 \ldots a_k$ is not; and
- $a_1 \ldots a_k \downarrow I$ is a prefix of a linearization of $(E^{\mathrm{SUT}}_{\mathrm{spec}}, <^{\mathrm{SUT}}_{\mathrm{spec}}) \downarrow I$, where I is the set of receive events in $E^{\mathrm{SUT}}_{\mathrm{spec}}$.

The event a_k of the trace is the first premature SUT event, after which a correct ordering of SUT receive events can no longer be guaranteed by a test implementation. The behavior of the SUT may become unspecified afterwards. Considering the type of the premature event in an order fault, we have the following possible violations of the order $<^{\mathrm{SUT}}_{\mathrm{spec}}$. An SUT send event can permute with another SUT send event or an SUT receive event, while an SUT receive event can only permute with an SUT send event, but not with an SUT receive event. The reasoning behind this is that a sound test implementation enforces the ordering of SUT receive events in $<^{\mathrm{SUT}}_{\mathrm{spec}}$ by ordering the matching send events accordingly. This observation leads to the following definition of three types of the order faults.

An order fault $a_1 \ldots a_k$ is

- a *swapped send* fault if a_k is a send event that follows another send event in $<^{\mathrm{SUT}}_{\mathrm{spec}}$;
- a *premature send* fault if a_k is a send event that follows a receive event in $<^{\mathrm{SUT}}_{\mathrm{spec}}$;
- a *delayed send* fault if a_k is a receive event that follows a send event in $<^{\mathrm{SUT}}_{\mathrm{spec}}$.

To simplify our discussion, we further assume that a trace contains just a single order fault which involves two events, a send and a receive or two send events. Depending on the number of test components executing the matching send and receive events, the order faults can occur either locally or in a distributed way. A fault is *local* if the SUT exchanges the two messages related to the events in the order fault with the same test component or *distributed* if it does so with two test components.

An order fault cannot always be detected within a trace t by a test implementation $test_{\mathrm{imp}}$ if $test_{\mathrm{imp}} \cup \{t\}$ has a race, since the race may be resolved in such a way that masks the fault. Recall that the soundness of $test_{\mathrm{imp}}$ implies that $test_{\mathrm{imp}} \cup \{(E^{\mathrm{SUT}}_{\mathrm{spec}}, <^{\mathrm{SUT}}_{\mathrm{spec}})\}$ is δ-race-free. Therefore, to guarantee that an order fault is discovered irrespectively of communication delays, we say that a test implementation $test_{\mathrm{imp}}$ *detects* an order fault in the trace t if the composition $test_{\mathrm{imp}} \cup \{t\}$ is not δ-consistent. Such inconsistency leads to a deadlock in one of the test components, yielding a *fail* verdict (via a timeout in the MTC).

Detection of a swapped send fault depends on its type. A local swapped send fault can directly be detected by the involved test component as an output fault since it receives the SUT messages via a FIFO channel. However, as already discussed above, messages consecutively sent by the SUT via different channels could arrive in any order due to variable communication delays. This implies that distributed swapped send faults cannot be detected at all.

A local delayed send fault involving the events $?i$ and $!o$ such that $!o <^{\mathrm{SUT}}_{\mathrm{spec}} ?i$ could be detected simply by the test component that restrains sending i until it receives o. To detect a distributed delayed send fault, a coordinating message whose send event follows the SUT receive event $?o$ and whose receive event precedes the SUT receive event $!i$ is additionally required.

A premature send fault is the only fault, whose detection requires null events. The example in Fig. 4 illustrates that an SUT with such a fault composed with a sound test implementation but without null events is a consistent MSC, and thus the fault is not detected. In an input-sequential test scenario, a local premature send fault involving the SUT events $?i$ and $!o$ such that $?i <^{\mathrm{SUT}}_{\mathrm{spec}} !o$ can be detected if the test component waits for any potential SUT output before actually sending event i. In other words, the test component should observe SUT quiescence by executing a null event immediately prior to the send of message i. Thus, the test component makes sure that message o is sent only in response to message i.

When a premature send fault is distributed, the matching events $!i$ and $?o$ occur in two test components, the test component which receives message o uses a null event and, additionally, coordinating messages to synchronize with other test components including one that sends message i. The coordinating messages ensure that the null event immediately precedes the send of message i. We elaborate an algorithm for implementing tests ensuring the detection of these faults in the next section.

In the case when a test scenario is not input-sequential different concurrent SUT receive events can be involved in a premature send fault. Therefore to determine which particular SUT input triggers the premature SUT send event, one needs to order all the concurrent messages to be sent by the test components. In other words, a premature send fault is not detectable by a single concurrency-preserving test implementation. Construction of several test implementations from a given test scenario is not discussed in this paper, having the goal to build a single test implementation for a given test scenario. Note that considering all potential interleavings of concurrent SUT receive events may lead to the combinatorial explosion.

5.2 Increasing Fault Detection Power of Test Implementations

Test implementations produced by Algorithm 1 clearly detect local swapped send faults. Moreover, they detect delayed send faults. Indeed, if a delayed send fault is local, it is directly observed; if distributed, Step 3 of Algorithm 1 inserts a coordinating message to detect it. We summarize these observations in the following statement.

Proposition 3. A test implementation, generated by Algorithm 1 detects local swapped send and delayed send faults.

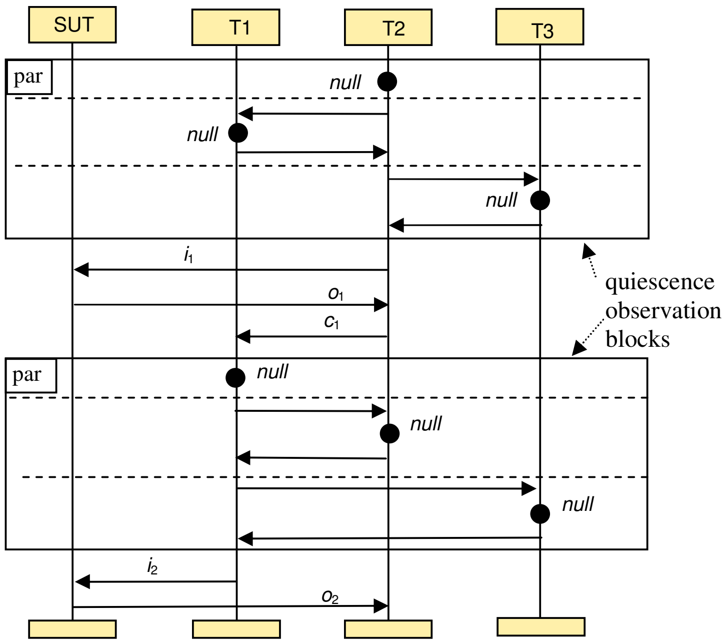

Fig. 5. A straightforward way of observing global quiescence

The detection of premature send faults cannot be guaranteed using coordinating messages only, but Algorithm 1 can be enhanced by using null events as outlined above. The idea is to check quiescence of the SUT at all test components after all expected SUT messages have been received just before sending a next message to the SUT. Such a quiescence checking ensures that the SUT is in a stable state where no further message can be produced and no message to test components remains in transit. This approach eliminates the possibility of masking an expedited send fault by communication delays. A "global" SUT quiescence should be established using null events accompanied by a proper coordination among test components. A straightforward solution consists in placing null events in all test components prior to sending a message to the SUT. The test component that sends the next message to the SUT, in this case, notifies all partner test components and collects the confirmations that the null events have occurred prior to sending the message, as shown in Fig. 5.

This synchronization mechanism allows detection of all the premature send faults when the test scenario is input-sequential.

Such a straightforward approach may result in a high number of coordinating messages and null events, which can slow down test execution. The number of coordinating messages can be reduced if, prior to sending a message to the SUT, only the test components expecting to receive messages from the SUT are notified. Moreover, to reduce the number of null events, we suggest using a null event prior to this notification. In this case, there is no need to use the null events in all the test components. The suggested procedure is summarized as follows.

Algorithm 2.
Input: An MSC test scenario $test_{\text{scen}}$ with the SUT specification ($E^{\text{SUT}}_{\text{spec}}$, $<^{\text{SUT}}_{\text{spec}}$) and the test components specifications (E^1_{spec}, $<^1_{\text{spec}}$), (E^2_{spec}, $<^2_{\text{spec}}$), ….
Output: A test implementation $test_{\text{imp}}$.

1. Apply Algorithm 1.
2. For each send event $!i$ of each test component T,
   - Add a null event prior to $!i$, unless it was already added in Step 3 of Algorithm 1.
   - For each test component T', $T' \neq T$ with receive event(s) of SUT message(s) triggered by the message i, insert two coordinating messages, one sent from T to T' after the null event in T and another one from T' to T before the abovementioned receive event(s) of SUT message(s).

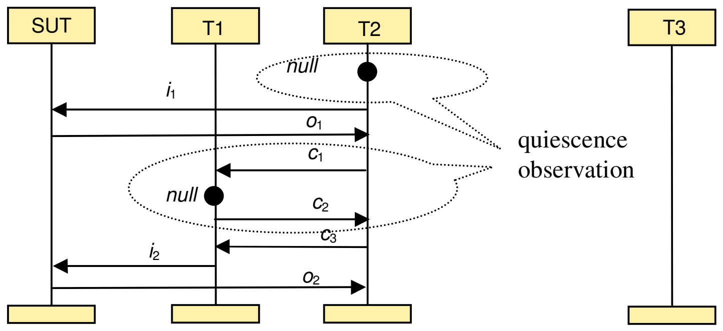

Fig. 6. A test implementation according to Algorithm 2

Compared to the test implementation obtained by Algorithm 1, the test implementation obtained by Algorithm 2 in Fig. 6 has an additional null event, followed by two coordinating messages c_2 and c_3. This allows the detection of the premature send fault due to the reordering of the events $!i_2$ and $?o_2$. The sole purpose of the coordinating message c_3 is to avoid the race between c_2 and o_2. The suggested approach is usually more economical than the straightforward one in the number of null events and coordinating messages (compare the number of null events and coordinating messages in Fig. 5 and 6). However, there is a price to pay. First, in the straightforward approach, a message sent to one test component can arrive concurrently together with the causally independent null event of another test component. In other words, null events can be viewed as a local quiescence observation. Second, the straightforward approach allows for a better diagnosis of faults. Indeed, if a test component of the test specification has no events at all, and an unexpected SUT message is detected, it is impossible to diagnose, which SUT receive event has triggered the unexpected message.

Proposition 4. Given an arbitrary test scenario, the test implementation obtained by Algorithm 2 is sound, concurrency preserving, and detects delayed send and local swapped send faults. Moreover, for an input-sequential test scenario, it also detects premature send faults.

Proposition 5. For an input-sequential test scenario the test implementation returned by Algorithm 2 has the highest fault detection power.

The last proposition holds because the only undetectable type of order faults in Algorithm 2 is a distributed swapped send fault. Table 1 summarizes the fault detection capabilities of both algorithms.

Table 1. Fault detection power of Algorithms 1 and 2.

Fault classes		**Algorithm 1**	**Algorithm 2**
Output fault		✓	✓
Swapped send fault	local	✓	✓
	distributed	✗*	✗*
Premature send fault	local	✗	✓**
	distributed	✗	✓**
Delayed send fault	local	✓	✓
	distributed	✓	✓

*) theoretically undetectable in an asynchronous environment
**) for input-sequential test scenarios

6 Conclusions

Based on the analysis of faults in distributed systems, a novel method for distributed testing that involves the use of observable quiescence to implement MSC test specifications is proposed. Unlike the existing methods it does not eliminate concurrency in a test specification and delivers the highest possible fault detection power (according to the suggested classification). The proposed notions of soundness and faults can be also applied to evaluate other test implementations.

A prototype tool implementing the algorithms 1 and 2 is currently in preparation. The tool development concerns representing test implementations in Promela to allow for model checking and simulation as well as mapping the Promela test implementation into various test execution languages. On the theoretical side, we currently consider generalization of the framework to use pomsets in defining test specifications and investigate other algorithms for distributed testing with a known fault detection power. For example, it is known that races can be compensated by input queues [4, 11]. The explicit use of such queues could result in simplified test implementations with a similar fault detection power compared to the algorithms presented here. This area however is subject of further research.

References

1. Grabowski, J., Koch, B., Schmitt, M., Hogrefe, D.: SDL and MSC Based Test Generation for Distributed Test Architectures. In: SDL Forum'99, pp. 389-404 (1999)
2. Baker, P., Bristow, P., Jervis, C., King, D., Mitchell, W.: Automatic Generation of Conformance Tests from Message Sequence Charts. In: SAM 2002: SDL and MSC Fourth International Workshop. LNCS, vol. 2599, pp. 170-198 (2003)
3. Mitchell. W.: Characterizing Concurrent Tests Based on Message Sequence Chart Requirements. In: Applied Telecommunication Workshop, pp. 135-140 (2001).
4. Holzmann, G., Peled, D., Redberg, M.: Design Tools for Requirement Engineering. Bell Labs Technical J. 2(1): 86-95 (1997)
5. OMG UML Specification http://www.omg.org/spec/UML/2.1.2/
6. Haugen, O.: Comparing UML 2.0 Interactions and MSC-2000. In: SAM 2004: SDL and MSC Fourth International Workshop. LNCS, vol. 3319, pp. 69-84 (2004)
7. Bochmann, G., Haar, S., Jard, C., Jourdan, G.-V.: Testing Systems Specified as Partial Order Input/Output Automata. In: TestCom/FATES 2008. LNCS, vol. 5047, pp. 169-183 (2008)
8. Deussen, P. H., Tobies, S.: Formal Test Purposes and the Validity of Test Cases. In: FORTE 2002, pp. 114-129 (2002)
9. Pickin, S., Jézéquel, J.-M.: Using UML Sequence Diagrams as the Basis for a Formal Test Description Language. In: 4th Int. Conf. on Integrated Formal Methods (IFM 2004). LNCS, vol. 2999, pp. 481-500 (2004)
10. Alur, R., Etessami, K., Yannakakis, M.: Inference of Message Sequence Charts. IEEE Trans. on Soft. Eng. 29 (7): 623-633 (2003)
11. Mitchell, B.: Resolving Race Conditions in Asynchronous Partial Order Scenarios. IEEE Trans. on Soft. Eng. 31(9): 767-784 (2005)
12. Pickin, S., Jard, C., Jeron, T., Jézéquel, J.-M., Le Traon, Y.: Test Synthesis from UML Models of Distributed Software. IEEE Trans. on Soft. Eng. 33 (4): 252-268 (2007)
13. Uchitel, S., Brunet, G., Chechik, M.: Behaviour Model Synthesis from Properties and Scenarios. In: 29th IEEE/ACM International Conf. on Soft. Eng., pp. 34-43 (2007)
14. Baker, P., Bristow P., P., King, D., Thomson, R., Burton, S., Bristow, P.: Detecting and Resolving Semantic Pathologies in UML Sequence Diagrams. In: ESEC/SIGSOFT FSE, pp. 50-59 (2005)
15. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software - Concepts and Tools. 17(3): 103-120 (1996)
16. Cacciari, L., Rafiq, O.: Controllability and Observability in Distributed Testing. Information and Soft. Technology. 41(11-12): 767-780 (1999)
17. Lund, M.S., Stolen, K.: Deriving Tests from UML. 2.0 Sequence Diagrams with neg and assert. In: 1st International Workshop on Automation of Soft. Testing, pp. 22-28 (2006)
18. Boroday, S., Petrenko, A., Ulrich, A.: Test Suite Consistency Verification. In: 6th IEEE East-West Design & Test Symposium (EWDTS 2008), pp. 235-238 (2008)