Towards Decentralized Conformance Checking in Model-Based Testing of Distributed Systems

Bruno Lima*† and João Pascoal Faria*†
*INESC TEC,
FEUP campus, Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
†Faculty of Engineering, University of Porto,
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal

{bruno.lima, jpf}@fe.up.pt

***Abstract*—In a growing number of domains, the provisioning of end-to-end services to the users depends on the proper interoperation of multiple products, forming a new distributed system. To ensure interoperability and the integrity of this new distributed system, it is important to conduct integration tests that verify not only the interactions with the environment but also the interactions between the system components. Integration test scenarios for that purpose may be conveniently specified by means of UML sequence diagrams, possibly allowing multiple execution paths. The automation of such integration tests requires that test components are also distributed, with a local tester deployed close to each system component, and a central tester coordinating the local testers. In such a test architecture, it is important to minimize the communication overhead during test execution. Hence, in this paper we investigate conditions upon which conformance errors can be detected locally (local observability) and test inputs can be decided locally (local controllability) by the local testers, without the need for exchanging coordination messages between the test components during test execution. The conditions are specified in a formal specification language that allows executing and validating the specification. Examples of test scenarios are also presented, illustrating local observability and controllability problems associated with optional messages without corresponding acknowledgment messages, races and non-local choices.**

***Index Terms*—Model-Based Testing; Conformance Checking; Integration Testing; Distributed Systems; UML.**

I. Introduction

In a growing number of domains, the provisioning of end-to-end services to the users depends on the proper interoperation of multiple products (devices, applications, etc.), possibly from different vendors, forming a new distributed system. Examples of such systems can range from simple sports monitoring applications [1] to fall detection systems for seniors [2] that in the event of an emergency notify formal or informal caregivers.

To ensure interoperability and the integrity of such distributed systems or systems of systems, it is important to conduct integration tests that verify not only the end-to-end communications but also the interactions between the system components. The verification of interactions between components is especially important for improving fault detection and localization.

Integration test scenarios for that purpose may be conveniently specified by means of UML Sequence Diagrams [3] (SDs), because they are an industry standard well suited for describing and visualizing the interactions that occur between the components and actors of a distributed system.

In previous work [2] we proposed an approach and toolset architecture for automating the testing of end-to-end services in distributed and heterogeneous systems, comprising a visual modeling environment and a distributed test generation and execution engine. In that approach, the only manual activity required is the description of the participants and behavior of the services under test with UML SDs, which are automatically translated to a formal notation for efficient test input generation and conformance checking at runtime.

In our approach, in order to be able to check not only the interactions with the environment but also the interactions between the system components, a *local tester* is deployed close to each system component, and a *central tester* coordinates the local testers. In such a test architecture, it is important to minimize the communication overhead during test execution, namely in the presence of time constraints. It is equally important to detect errors as early as possible and closely as possible to the offending component, to provide more helpful test reports and facilitate fault localization.

Hence, in this paper we investigate conditions upon which observed events can be checked locally (*local observability*) and the decision of when and what inputs to inject can be decided locally (*local controllability*) by the local testers, without the need for exchanging coordination messages between the test components during test execution. The conditions are specified in the VDM formal specification language [4] [5], which allows executing and validating the specification. Illustrative examples of test scenarios that exhibit different combinations of the above properties are also presented.

The rest of the paper is organized as follows: section II presents an introduction to the integration testing of distributed systems based on UML SDs; Section III presents the problem of decentralized conformance checking and the conditions for local observability and controllability; examples of integration test scenarios that exhibit different combinations of those properties are presented in section IV; related work is presented in section V; conclusions and future work are presented in section VI.

II. Model-based integration testing of distributed systems

In our previous work [2] we proposed a scenario-based approach for automating the integration testing of end-to-end services in distributed and heterogeneous systems. Our approach is based on four main ideas:

- different front-end and back-end modeling notations;
- online (adaptive) model-based testing (MBT) strategy;
- distributed test architecture and algorithms;
- automatic mapping of test results.

Test scenarios are specified using an accessible front-end notation such as UML SDs, which are automatically translated to a back-end formal notation suitable for efficient test input generation and conformance checking at runtime. At the end of test execution, test results (conformance errors and coverage information) are mapped back to the front-end notation (see Figure 1).

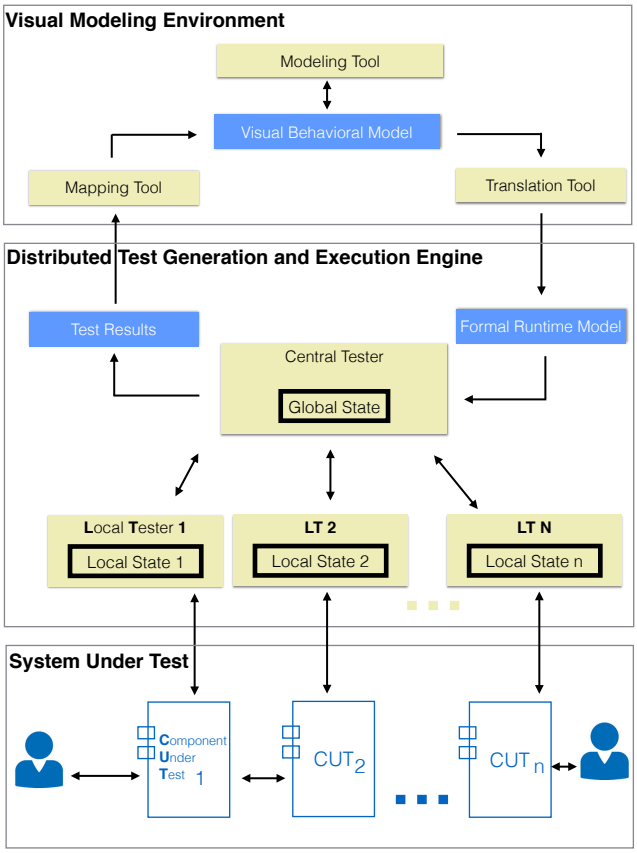


Fig. 1. Architecture for model-based integration testing of distributed systems

We follow an *online* MBT strategy (also called *adaptive* or *on-the-fly*), to cope with non-determinism common in distributed systems. In an online strategy, test generation and execution are performed together, allowing test inputs to be decided based on the outputs observed so far [6].

In order to be able to check not only the interactions with the environment but also the interactions between the components of the system under test (SUT), we follow a hybrid test architecture, in which a *local tester* is deployed close to each system component, and a *central tester* coordinates the local testers (see Figure 1). This is more effective than a purely centralized approach or a purely distributed approach, meaning that more conformance errors can be detected [7].

As illustrated in Figure 2, in the case of a component under test (CUT) that interacts with actors in the environment (users or external applications), the local tester is responsible for simulating the actors, injecting the inputs from the actors to the CUT (acting as *a test driver*) and monitoring and checking the outputs from the CUT to the actors (acting as a *test monitor*). Besides that, it is also responsible for monitoring all the messages exchanged between that CUT and the rest of the SUT.

In the case of a CUT that does not interact with the environment, the local tester is responsible for monitoring and checking all the messages exchanged between that CUT and the rest of the SUT, acting as a test monitor. However, a local tester may also simulate the behavior of a CUT, acting as a *test stub*; in that case, it injects the outgoing messages from that CUT to the rest of the SUT, and monitors and checks the incoming messages from the rest of the SUT to that CUT.

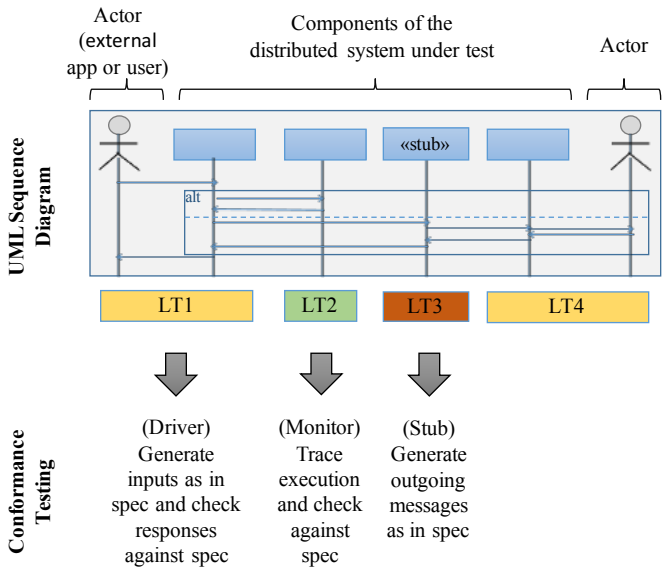


Fig. 2. Roles of local testers in the integration testing of distributed systems based on UML SDs

In such a test architecture, it is important to minimize the communication overhead during test execution, namely in the presence of time constraints. It is equally important to detect errors as early as possible and closely as possible to the offending component, to provide more helpful test reports and facilitate fault localization. To that end, besides the test monitoring and control activities, model execution and conformance checking activities need also be distributed. Nevertheless, some coordination messages may still need to be exchanged between the local testers during test execution. That will be addressed in the next section.

III. Conditions for decentralized conformance checking

As previously explained, it is important to minimize the communication overhead during distributed test execution. Ideally, after the central tester sets up and initiates the local testers for a given test scenario (described by a UML SD), no communication between the test components should occur during test execution, and the central tester would only need to receive a verdict from each local tester at the end of successful test execution or as soon as an error is detected. However, depending on the test scenario under consideration, some errors may not be detected locally.

Hence, in this section, we investigate, from a theoretical point of view, conditions upon which conformance checking of observed execution traces against the expectations set by an SD under consideration can be performed by the local testers alone based on the events observed locally, without the need to communicate those events to the central tester to ensure that the final test verdict is correct (*local observability*). We also investigate, from a theoretical point of view, conditions upon which test inputs can be decided locally, without communication among test components during test execution (*local controllability*).

For determining local observability, we do not care how test inputs are injected, just assume that they are injected by some mechanism (by the local testers, users or other means) and that the test input events are observed by the local testers.

Given an SD (describing a test scenario), we determine local observability in three steps:

- Calculate the valid global traces defined by the SD;
- Determine the valid local traces in each lifeline;
- Determine if there are and which are the invalid SUT traces that are not locally detectable.

We formalize the conditions for local observability and controllability with the VDM formal specification language [4] [5]. This allows executing and validating the specification with a support tool, such as Overture (http://overturetool.org/). We use the mathematical notation of VDM, instead of the ASCII notation. As a preliminary step, we formalize the structure and semantics of UML SDs. VDM specifications have been used as an oracle before [8], but not in distributed cases.

A. Valid global traces defined by a UML Sequence Diagram

In UML, an SD is a variant of an Interaction [3]. Figure 3 describes the structure of Interactions in VDM. For simplicity, we omit the applicable integrity constraints and the definition of some basic types. An interaction comprises a set of lifelines (representing in our case CUTs or actors), messages (restricted in our case to asynchronous messages, although synchronous messages could easily be handled), and combined fragments (restricted in this paper to the most common ones). A combined fragment covers a subset of lifelines and is divided in a sequence of one or more interaction operands; the semantics is determined by the interaction operator. Co-regions may be represented by `par` combined fragments covering the desired message ends, so we allow messages to cross boundaries of combined fragments. In each lifeline, the ordering of message ends and start/finish boundaries of combined fragments and interaction operands is represented by assigning sequential natural numbers to their locations. Combined fragments with the `opt` and `loop` operator have only one operand. Operands of combined fragments with the `opt`, `loop` and `alt` operator can have a guard.

```
types

Interaction ::
  lifelines         : Lifeline-set
  messages          : Message-set
  combinedFragments : CombinedFragment-set;

Message ::
  id           : MessageId         -- unique
  sendEvent    : LifelineLocation  -- unique
  receiveEvent : LifelineLocation  -- unique
  signature    : MessageSignature; -- not necessarily unique

LifelineLocation = Lifeline × Location;

Lifeline ::
  name : String;

Location = ℕ;

CombinedFragment ::
  interactionOperator : InteractionOperatorKind
  operands            : InteractionOperand+ -- seq of 1 or more
  lifelines           : Lifeline-set;

InteractionOperatorKind = <seq> | <alt> | <opt> | <par> |
                          <strict> | <loop>;

InteractionOperand ::
  guard            : [InteractionConstraint]
  startLocations   : LifelineLocation-set
  finishLocations  : LifelineLocation-set;

InteractionConstraint ::
  minint       : [ValueSpecification]  -- used in loops
  maxint       : [ValueSpecification]  -- used in loops
  specification: [ValueSpecification];
```

Fig. 3. Interactions

In general, the semantics of an Interaction is expressed in terms of two sets of valid and invalid traces [3]. In this paper, we don't handle the rarely used constructs for defining invalid traces (such as the `neg` interaction operator), so only the valid traces are relevant here. A *trace* is a sequence of event occurrences [3], corresponding, in this context, to the sending or receiving of messages at lifelines (Figure 4). We assume that there is no global clock that allows ordering event occurrences from different lifelines, so timestamps are not stored in the event occurrences.

For computing the set of valid traces defined by an Interaction, we start by computing traces with extended information (see `TraceExt` in Figure 4). For distinguishing events that refer to different messages in the SD with the same signature, we store the unique message identifier together with the event. To distinguish events that refer to different repetitions of the same message in loops (possibly nested), we keep an iteration counter; for example, a message occurring in the third iteration of the second iteration of a nested loop (inside a top-level loop) is numbered with the sequence [2, 3].

```
Trace = Event*; -- seq of 0 or more

Event ::
  type       : EventType
  signature  : MessageSignature
  lifeline   : Lifeline;

EventType = <Send> | <Receive>;

TraceExt = EventExt*;

EventExt ::
  type       : EventType
  signature  : MessageSignature
  lifeline   : Lifeline
  location   : Location
  messageId  : ℕ
  itercounter: ℕ*; -- used in loops, possibly nested
```

Fig. 4. Traces

We formalize the semantics of an Interaction by the function `validTraces` in Figure 5. Some trivial auxiliary functions are omitted for simplifying the presentation.

The valid traces with extended information are computed recursively, following the nested structure of combined fragments. First, the top level events (message ends) that are not contained inside combined fragments are determined (function `topLevelEvents`), and each event generates a trace containing only that event (function `validTracesExt`). The top level combined fragments are also determined, and the set of valid traces of each combined fragment is recursively computed (see call to `expandCombinedFragment` in `validTracesExt`). Then, the different sets of top-level traces are combined (function `freeComb`), generating the set of valid global traces; the traces are combined preserving the order of events per trace, lifeline (according to the locations and iteration counters) and message (send event before the receive event). We don't assume FIFO channels, that is message overtaking may occur (consistently with the UML standard), but it would be easy to add the FIFO assumption.

The semantics of each type of combined fragment is defined in Figure 6. The expansion of each operand is similar to the expansion of the top-level diagram.

In `seq`, `strict` and `par` combined fragments, all operands represent mandatory behaviors. Traces from consecutive operands are combined according to the semantics of each operator (see `seqComb`, `strictComb` and `parComb`).

The operands of `alt` and `opt` combined fragments represent behaviors that are selected for execution non-deterministically and/or according to the values of guard conditions. In this paper, we don't take into account possible guard conditions defined, because they shouldn't limit the set of possible traces but only the conditions upon which each trace may be selected. Hence, the sets of valid traces are given simply by the `expandAlt` and `expandOpt` functions.

To keep the model executable, in this paper we restrict our attention to loops that have defined minimum and maximum numbers of iterations, so the operand of a loop represents a behavior that may be repeated a number of iterations chosen non-deterministically and/or according to a guard condition

```
functions

validTraces: Interaction → Trace-set
validTraces(sd) ≜  removeExtraTraceInfo(validTracesExt(sd));

removeExtraTraceInfo: TraceExt-set → Trace-set
removeExtraTraceInfo(s) ≜
 {[mk_Event(e.type, e.signature, e.lifeline) | e ∈ t] | t ∈ s};

validTracesExt: Interaction → TraceExt-set
validTracesExt(sd) ≜
 freeComb({{[e]} | e ∈ topLevelEvents(sd)}
   ∪{expandCombinedFragment(sd, c) | c ∈ topLevelCombFrag(sd)}));

topLevelEvents: Interaction → EventExt-set
topLevelEvents(sd) ≜
  {mk_EventExt(<Send>, m.signature, m.sendEvent.#1,
     m.sendEvent.#2, m.id, []) |  m ∈ sd.messages •
       ∄ c ∈ sd.combinedFragments • contains(c, m.sendEvent)}
∪ {mk_EventExt(<Receive>, m.signature, m.receiveEvent.#1,
     m.receiveEvent.#2, m.id, []) | m ∈ sd.messages •
       ∄ c ∈ sd.combinedFragments • contains(c, m.receiveEvent)};

-- Given several sets of traces, gives all interleavings of
-- traces, one from each set, preserving the order of events per
-- trace, lifeline and message.
freeComb: TraceExt-set-set → TraceExt-set
freeComb(ss) ≜
  if ss = {} then {[]}
  else let s ∈ ss in
       ⋃{freeComb(t1, t2) | t1 ∈ s, t2 ∈ freeComb(ss \ {s})};

-- Gives all interleavings of two traces that preserve the
-- order of events per trace, lifeline and message.
freeComb: TraceExt × TraceExt → TraceExt-set
freeComb(t1, t2) ≜
  if t1 = [] ∨ t2 = [] then {t1 ⌢ t2}
  else (if ∃ e ∈ t2 • precedes(e, hd t1) then {}
        else {[hd t1] ⌢ r | r ∈ freeComb(tl t1, t2)})
     ∪ (if ∃ e ∈ t1 • precedes(e, hd t2) then {}
         else {[hd t2] ⌢ r | r ∈ freeComb(t1, tl t2)});

precedes: EventExt × EventExt → 𝔹
precedes(e1, e2) ≜
 (e1.messageId = e2.messageId ∧ e1.itercounter = e2.itercounter ∧
  e1.type = <Send> ∧e2.type = <Receive>)
  ∨ (e1.lifeline = e2.lifeline
     ∧ (e1.location < e2.location
        ∨ e1.location = e2.location
          ∧ precedesIter(e1.itercounter, e2.itercounter)));
```

Fig. 5. Valid traces defined by an Interaction

between the specified limits (see `expandLoop`). Consecutive iterations are combined in a weak sequencing way as specified in the UML standard (see `iterate`). However, the theoretical approach is also applicable for unbounded loops.

B. Interactions locally observable

Given a test scenario described by a UML SD (interaction), we next derive the conditions upon which conformance checking of observed execution traces against the expectations set by the SD can be performed by the local testers alone based on the events observed locally, without the need to communicate those events to the central tester to ensure that the final test verdict is correct. We call interactions with that characteristic *locally observable* or *locally checkable*.

To that end, function `uncheckableLocally` in Figure 7 determines the global traces that are locally valid but not globally valid. Conformance checking can be performed locally if such traces don't exist (see `isLocallyObservable`).

```
expandCombinedFragment: Interaction × CombinedFragment →
TraceExt-set
expandCombinedFragment(sd, c) ≜
 cases c.interactionOperator:
   <seq>     → expandNary(sd, c.operands, seqComb),
   <strict>  → expandNary(sd, c.operands, strictComb),
   <par>     → expandNary(sd, c.operands, parComb),
   <alt>     → expandAlt(sd, c.operands),
   <opt>     → expandOpt(sd, c.operands₁),
   <loop>    → expandLoop(sd, c.operands₁)
 end;

expandNary: Interaction × InteractionOperand* × (TraceExt
× TraceExt → TraceExt-set) → TraceExt-set
expandNary(sd, args, comb) ≜
 if args = []  then {[]}
 else ⋃{comb(t1,t2) | t1 ∈ expandOperand(sd, hd args),
                      t2 ∈ expandNary(sd, tl args)};

-- Weak sequencing of two traces, given by the interleavings that
-- preserve the order of events per trace and lifeline
seqComb: TraceExt × TraceExt → TraceExt-set
seqComb(t1, t2) ≜
 if t1 = [] ∨ t2 = [] then {t1 ↷ t2}
 else {[hd t1] ↷ r | r ∈ seqComb(tl t1, t2)} ∪
      if ∃ e ∈ t1 • (hd t2).lifeline =e.lifeline then {}
      else {[hd t2] ↷ r | r ∈ seqComb(t1, tl t2)};

-- Strict sequencing of two traces, given by their concatenation.
strictComb: TraceExt × TraceExt → TraceExt-set
strictComb(t1, t2) ≜ {t1 ↷ t2};

-- Parallel combination of two traces, given by the interleavings
-- that preserve the order of events per trace.
parComb: TraceExt × TraceExt → TraceExt-set
parComb(t1, t2) ≜
 if t1 = [] ∨ t2 = [] then {t1 ↷ t2}
 else {[hd t1]↷r | r ∈parComb(tl t1, t2)} ∪
      {[hd t2]↷r | r ∈parComb(t1, tl t2)};

expandAlt: Interaction × InteractionOperand* → TraceExt-set
expandAlt(sd, args) ≜ ⋃ {expandOperand(sd,arg) | arg ∈ args};

expandOpt: Interaction × InteractionOperand → TraceExt-set
expandOpt(sd, arg) ≜ expandOperand(sd, arg) ∪ {[]};

expandLoop: Interaction × InteractionOperand → TraceExt-set
expandLoop(sd, arg) ≜
 ⋃ {iterate(expandOperand(sd, arg), n) |
     n ∈ {arg.guard.minint,..., arg.guard.maxint}};

iterate: TraceExt-set × ℕ → TraceExt-set
iterate(s, n) ≜
 if n = 0 then {[]}
 else ⋃{seqComb(t1,addIterNum(t2,n))|t1 ∈ iterate(s,n-1),t2 ∈ s};

addIterNum: TraceExt × ℕ → TraceExt
addIterNum(t, iter) ≜
 [μ(e, itercounter ↦ [iter] ↷ e.itercounter) | e ∈ t];

expandOperand: Interaction × InteractionOperand → TraceExt-set
expandOperand(sd, o) ≜
 freeComb({{[e]} | e ∈ nestedEvents(sd, o)}
   ∪ {expandCombinedFragment(sd, c) | c ∈ nestedCombFrag(sd,o)});
```

Fig. 6. Valid traces defined by combined fragments

By *global traces globally valid*, we mean the set of valid traces defined by the SD, as given by the function `validTraces` previously derived. By projecting the set of traces globally valid onto each lifeline, we get the set of subtraces valid locally in each lifeline (see function `projectTraces`).

By *global traces locally valid*, we mean all the feasible global traces which projections onto the lifelines yield sub-

```
-- Determines if conformance checking can be performally locally.
isLocallyObservable: Interaction → 𝔹
isLocallyObservable(sd) ≜ uncheckablyLocally(sd) = {};

-- Gives global traces that are locally but not globally valid.
uncheckablyLocally: Interaction → Trace-set
uncheckablyLocally(sd) ≜ let V = validTraces(sd) in
  joinTraces([], projectTraces(V, sd.lifelines)) \ V;

-- Projects a set of traces (T) onto a set of lifelines (L).
projectTraces: Trace-set × Lifeline-set → (Lifeline ⇸ᵐ Trace-set)
projectTraces(T, L) ≜{l ↦ {projectTrace(t, l) | t ∈ T} | l ∈ L};

-- Projects a trace (t) onto a lifeline (l).
projectTrace: Trace × Lifeline → Trace
projectTrace(t, l) ≜ [e | e ∈ t • e.lifeline = l];

-- Gives the feasible joins of traces from different lifelines,
-- respecting the order of events per trace and message. The
-- first argument is an accumulator for already processed events.
joinTraces: Trace × (Lifeline ⇸ᵐ Trace-set) → Trace-set
joinTraces(left, m) ≜
  if m = {↦} then {left}
  else ⋃{ ⋃{if t = [] then joinTraces(left, {l}⩤m)
            else joinTraces(left ↷ [hd t], m ++ {l ↦{tl t}})
            | t ∈ m(l) • t = [] ∨ isFeasible(left ↷ [hd t])}
         | l ∈ dom m};

-- Checks if a trace or subtrace is feasible, that is, respects
-- the fact that messages can only be received after being sent.
isFeasible: Trace → 𝔹
isFeasible(t) ≜
 ∀ i ∈ inds t • tᵢ.type = <Receive> ⇒
  #{j|j∈{1,...,i}• tⱼ.type=<Send> ∧ tⱼ.signature=tᵢ.signature} ≥
  #{j|j∈{1,...,i}• tⱼ.type=<Receive> ∧tⱼ.signature= tᵢ.signature};
```

Fig. 7. Determining if interactions are locally observable

traces that are locally valid. By *feasible global trace*, we mean any trace (involving events on the SD lifelines) that respects the fact that messages can be received only after being sent. We assume that messages may be lost in the transmission channel, that is, it is possible that a message is sent and not received, but the opposite is not possible. Since we assume that only the message signature is guaranteed to be observable, but not the sender/receiver of the message (in case of receive/send events, respectively) nor a unique message instance identifier, in the presence of multiple message occurrences with the same signature, we can only make sure that, at any given point in the trace under analysis, the number of 'send' occurrences is greater or equal to the number of 'receive' occurrences (see function `isFeasible`).

To promote model executability, we compute (in function `joinTraces`) the global traces locally valid from the subtraces locally valid. In fact, the global traces locally valid are given, equivalently, by all the possible combinations (joins) of valid subtraces, with one subtrace from each lifeline, restricted to the combinations that yield feasible global traces (in the sense described previously) and preserve the order of events per lifeline.

Examples of interactions that can and cannot be checked locally are shown in section IV.

C. Primitives for local and global conformance checking

For the sake of completeness, we present in Figure 8 the primitives needed to perform incremental conformance

checking locally in each lifeline by each local tester (function `checkNextEvent`), and perform a final conformance check globally, in case needed, by the central tester (function `finalConformanceCheck`).

```
types
Verdict = <Valid> | <Invalid> | <Inconclusive>;

functions
-- Checks if the next observed event in a lifeline is valid,
-- given a (valid) sequence of previously observed events in the
-- lifeline, and the set of valid traces for the lifeline.
checkNextEvent: Trace × Event × Trace-set → 𝔹
checkNextEvent(prevEvents, event, validLocalTraces) ≜
  ∃(prevEvents) ⌢[e] ⌢- ∈ validLocalTraces • e = event;

-- Final conformance checking, given the observed local traces.
finalConformanceCheck: Interaction ×(Lifeline →(m) Trace)→ Verdict
finalConformanceCheck(sd, localTraces) ≜
  let V = validTraces(sd),
      J = joinTraces([], {l↦{localTraces(l)} | l ∈ sd.lifelines})
  in if J ∩ V = {} then <Invalid>
     else if J ⊆ V then <Valid>
     else <Inconclusive>;
```

Fig. 8. Incremental and global conformance checking primitives

For performing incremental conformance checking, we assume that each local tester receives a specification of the traces to be accepted locally at the begin of test execution. For performing final conformance checking in case needed, we assume that the global tester receives from the local testers the traces observed locally at the end of test execution.

The decision procedure `finalConformanceCheck` produces the verdict *Invalid* if there is no feasible global trace corresponding to the observed local traces (computed by `joinTraces`) that is also a globally valid trace. The verdict is *Valid* if all the feasible global traces corresponding to the observed local traces are also globally valid traces. Otherwise, the verdict is *Inconclusive*.

Based on the definitions given, it is easy to conclude that if a test scenario is locally observable, inconclusive verdicts cannot occur (the reverse is not necessarily true). Hence, the addition of coordination messages between test components to ensure local observability will also discard inconclusive verdicts.

D. Interactions locally controllable

Given a test scenario described by a UML SD (interaction), we next derive sufficient conditions upon which the decision of when and what test inputs to inject can be performed by the local testers alone, based on the events previously observed locally, without the need for communication between the test components during test execution. We say that a test scenario is *locally controllable* in case such decisions can be performed locally, without generating, directly or indirectly, invalid traces.

We assume that all lifelines behave according to the following causality rules:

1) the set of messages that can be sent by a lifeline at a given point is a function only of the sequence of previous event occurrences in the lifeline, being the time for deciding to send a message non-deterministic;
2) messages can only be received after being sent, being the transmission time non-deterministic and the lifelines input-enabled (that is, always able to receive inputs, even if they are internally stored for later processing);
3) the possibility of a lifeline remaining quiescent (i.e., not sending output without first receiving input [9]) is also a function only of the sequence of previous event occurrences in the lifeline.

```
isLocallyControllable: Interaction → 𝔹
isLocallyControllable(sd) ≜ unintendedTraces(sd) = {};

-- Gives subtraces that can be generated according to causality
-- rules, but end in an unintended send, receive or termination.
unintendedTraces: Interaction → Trace-set
unintendedTraces(sd) ≜ let V = validTraces(sd), T = prefixes(V),
 L = sd.lifelines, P = projectTraces(V,L),
 us={q⌢[e] | q ∈ T, p⌢[e] ∈ T • e.type = <Send>
     ∧ projectTrace(q,e.lifeline)=projectTrace(p,e.lifeline)}\T,
 ur=⋃{{q⌢[e] | q ∈ prefixes({p}) • isFeasible(q⌢[e])} |
        p⌢[e] ∈ T • e.type = <Receive>} \ T,
 ut={p | p ∈ T • allMsgsReceived(p) ∧
     ∀ l∈L • mayRemainQuiescent(projectTrace(p, l), P(l))} \ V
in ur ∪ us ∪ ut;

prefixes: Trace-set → Trace-set
prefixes(T) ≜ {[]} ∪ ⋃{{t₁,…,ᵢ | i ∈ inds t} | t ∈ T};

-- Determines if a lifeline may remain quiescent after a valid
-- local subtrace (t), given the set of valid local traces (S)
mayRemainQuiescent: Trace × Trace-set → 𝔹
mayRemainQuiescent(t,S) ≜ t∈S ∨ ∀(t)⌢[e]⌢-∈S • e.type=<Receive>;

allMsgsReceived: Trace → 𝔹
allMsgsReceived(t) ≜
 |[e|e ∈ t • e.type=<Send>]| = |[e|e ∈ t • e.type=<Receive>]|;
```

Fig. 9. Checking if interactions are locally controllable

The first two parts of function `unintendedTraces` in Figure 9 determines erroneous global subtraces, with all events valid except the last one, that can be generated if each lifeline, knowing the set of traces valid locally, behaves according to the causality rules 1 and 2 above. The function computes all global subtraces that can be generated, and then subtracts the valid ones.

According to rule 1, if two global subtraces p and q have identical projections onto a lifeline l, and a send event e can occur immediately after p in lifeline l (i.e., the concatenation of p and e is also a valid global subtrace), then it may also occur immediately after q. This gives part `us` of the computation.

According to rule 2, if a receive event e can occur immediately after a valid global subtrace p (i.e., the concatenation of p and e is also a valid global subtrace), then e may also occur earlier, as long it is appears after the send event (the receive event may also occur later, but that would be a redundant check). This gives part `ur` of the computation.

The third part of function `unintendedTraces` (variable `ut`) determines erroneous global traces that can be generated, with all events valid, but missing additional events. The function computes all subtraces with all messages received, at the end of which all lifelines may decide to remain quiescent according to rule 3, and then subtracts the valid global traces.

Examples of interactions that can and cannot be controlled locally are shown in section IV. As illustrated in the examples,

races and non-local choices compromise local controllability, and hence require that extra coordination messages are communicated among the local testers.

E. Primitive for test input generation

For the sake of completeness, we present in Figure 10 a primitive to determine the next messages that can be sent by a lifeline, based on the sequence of events observed previously in the lifeline and the set of traces valid locally in the lifeline, consistently with the first causality rule previously presented. This is useful for local testers that simulate the behavior of actors or CUTs.

```
-- Gives the next messages that can be sent by a lifeline, given
-- the previous events observed and the traces valid locally.
nextSendEvents: Trace × Trace-set → Event-set
nextSendEvents(prevEvents, validLocalTraces) ≜
 {e | (prevEvents)↷[e]↷- ∈ validLocalTraces • e.type = <Send>};
```

Fig. 10. Determining the next messages that can be sent from a lifeline

IV. Examples

A. Locally observable and controllable scenario

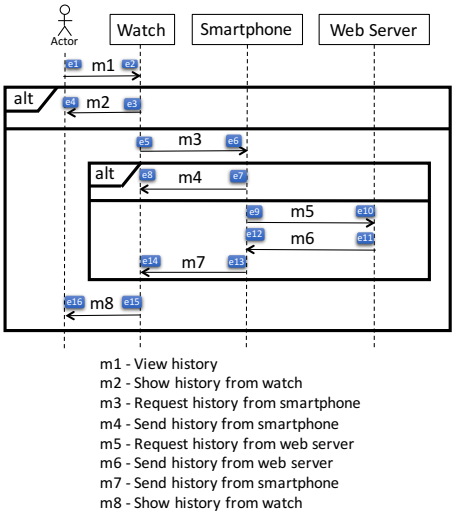


Fig. 11. Example of locally observable and controllable scenario

The SD that describes the scenario of visualizing the history of the last ten workouts in a smartwatch application is shown in Figure 11. At this point a user tries to access his running training history on his watch; if the last ten workouts are in the watch memory they are immediately shown to the user, otherwise the watch needs to communicate with the application installed on the smartphone asking for the last ten workouts. Again, if the mobile application has this information it sends immediately to the watch that presents this information to the user; however, if this information is not in the application memory, the application still have to communicate with the web server that will send these data, which are then sent to the watch and finally presented to the user.

```
operations
testAltNested() ≜ (
let
  l1 = mk_Lifeline("User"),
  l2 = mk_Lifeline("Watch"),
  l3 = mk_Lifeline("Smartphone"),
  l4 = mk_Lifeline("WebServer"),
  o11 = mk_InteractionOperand(nil,{mk_(l1,2),mk_(l2,2),mk_(l3,1),
mk_(l4,1)}, {mk_(l1,4), mk_(l2,4), mk_(l3,2), mk_(l4,2)}),
  o12 = mk_InteractionOperand(nil,{mk_(l1,4),mk_(l2,4),mk_(l3,2),
mk_(l4,2)}, {mk_(l1,6),mk_(l2,12),mk_(l3,11),mk_(l4,8)}),
   f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3, l4}),
   o21 = mk_InteractionOperand(nil, {mk_(l2, 6), mk_(l3, 4),
mk_(l4, 3)}, {mk_(l2, 8), mk_(l3, 6), mk_(l4, 4)}),
   o22 = mk_InteractionOperand(nil, {mk_(l2, 8), mk_(l3, 6),
mk_(l4, 4)}, {mk_(l2, 10), mk_(l3, 10), mk_(l4, 7)}),
   f2 = mk_CombinedFragment(<alt>, [o21, o22], {l2, l3, l4}),
   m1 = mk_Message(1, mk_(l1, 1), mk_(l2, 1), "m1"),
   m2 = mk_Message(2, mk_(l2, 3), mk_(l1, 3), "m2"),
   m3 = mk_Message(3, mk_(l2, 5), mk_(l3, 3), "m3"),
   m4 = mk_Message(4, mk_(l3, 5), mk_(l2, 7), "m4"),
   m5 = mk_Message(5, mk_(l3, 7), mk_(l4, 5), "m5"),
   m6 = mk_Message(6, mk_(l4, 6), mk_(l3, 8), "m6"),
   m7 = mk_Message(7, mk_(l3, 9), mk_(l2, 9), "m7"),
   m8 = mk_Message(8, mk_(l2, 11), mk_(l1, 5), "m8"),
   sd1 = mk_Interaction({l1, l2, l3, l4}, {m1, m2, m3, m4, m5,
m6, m7, m8}, {f1, f2}),
   e1 = mk_Event(<Send>, "m1", l1),
   e2 = mk_Event(<Receive>, "m1", l2),
   …
   e15 = mk_Event(<Send>, "m8", l2),
   e16 = mk_Event(<Receive>, "m8", l1)
in (
    assertEqual({[e1,e2,e3,e4], [e1,e2,e5,e6,e7,e8,e15, e16],
[e1,e2,e5,e6,e9,e10,e11,e12,e13,e14,e15,e16]}, validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertTrue(isLocallyControllable(sd1));
 )
);
```

Fig. 12. VDM encoding of the SD presented in Figure 11

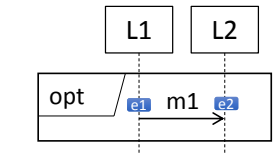
The VDM encoding of the SD is shown in Figure 12. As we can formally verify, this example has three valid global execution traces ($[e1, e2, e3, e4]$, $[e1, e2, e5, e6, e7, e8, e15, e16]$ and $[e1, e2, e5, e6, e9, e10, e11, e12, e13, e14, e15, e16]$) and is locally observable. This means that is not necessary any communication with the central tester to detect any fault during the distributed testing of the system.

B. Non-locally observable but locally controllable scenarios

Non-locally observable scenarios are scenarios where local observation is not enough, that is, local observations may be valid, but globally the sequence of events does not correspond to a valid trace. Examples of this type of scenarios are shown in Figure 13. In Figure 13 (a) it is presented an SD containing an 'opt' fragment and its valid traces. In this case, if the event `e1` occurs and for some reason the message `m1` gets lost, it is not possible to check locally that an error has occurred. This happens because the empty trace $[]$ is a valid execution trace, which makes the observation $[e1]$ on `L1` and the observation $[]$ on `L2` both valid locally. However, globally they do not correspond to a valid execution trace.

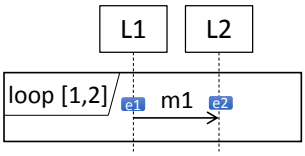
In Figure 13 (b) it is presented an SD containing a 'loop' fragment and its valid traces. In this case, and like in the previous example, if the second iteration occurs and the second message `m1` was sent ($[e1]$) but never received, looking only locally at `L1` and `L2`, it will not be possible to detect that a

problem has occurred since $[e1, e1]$ is a valid execution trace for L1 and $[e2]$ is also a valid execution trace for L2. However, when we look globally at the combination of these two traces, we find that none of their combinations gives a valid trace.



Valid Traces: [e1, e2], []
Uncheckably Locally: [e1]

(a)



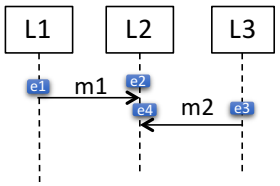
Valid Traces: [e1, e2], [e1, e2, e1, e2], [e1, e1, e2, e2]
Uncheckably Locally: [e1, e1, e2], [e1, e2, e1]

(b)

Fig. 13. Examples of non-locally observable (but locally controllable) scenarios because of the presence of optional messages without corresponding acknowledgment messages

C. Locally observable but not locally controllable scenario

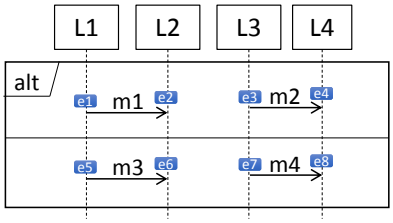
Locally observable but not locally controllable scenarios are scenarios where errors can be detected locally, but decision of when and what inputs to inject cannot be done locally, since it is necessary information from other components.



Valid Traces: [e1,e2,e3,e4], [e1,e3,e2,e4], [e3,e1,e2,e4]
Unintended Traces: [e1, e3, e4], [e3, e1, e4], [e3,e4]

Fig. 14. Example of locally observable but not locally controllable scenario because of a race condition

An example of such scenarios is shown in Figure 14. In this case, the system is composed of three components that are represented by the different lifelines. In this scenario the observation of the execution traces according to L2 has to be $[e2, e4]$, however without additional information the component represented by L3 does not know when it can send m2 in order to guarantee that it should reach L2 after m1. Without that information, the unintended traces shown in Figure 14 may occur.



Valid Traces:
[e1, e2, e3, e4], [e1, e3, e2, e4], [e1, e3, e4, e2],
[e3, e1, e2, e4], [e3, e1, e4, e2], [e3, e4, e1, e2],
[e5, e6, e7, e8], [e5, e7, e6, e8], [e5, e7, e8, e6],
[e7, e5, e6, e8], [e7, e5, e8, e6], [e7, e8, e5, e6]
Unintended Traces:
[e1, e2, e7], [e1, e7], [e7, e1], [e7, e8, e1],
[e5, e6, e3], [e5, e3], [e3, e5], [e3, e4, e5]

Fig. 15. Example of non-locally observable nor locally controllable scenario because of a non-local choice

D. Non-locally observable nor locally controllable scenario

Non-locally observable nor locally controllable scenario are scenarios where decisions regarding conformance checking and input injection are not possible locally.

An example of such a scenario is shown in Figure 15. In this scenario the system to be tested involves four lifelines. According to the 'alt' fragment, if the component represented by L1 sends the message m1, L3 should send the message m2; if L1 sends m3, L3 must send m4. In this situation, and without communication between the lifelines, sending erroneous messages is not locally detectable. For example if m1 is sent by L1 and m4 is sent by L3, locally the observations are valid. Similarly when L1 sends m1, L3 locally does not know if it needs to send m2 or m4, which makes the system not locally controllable.

V. Related Work

A. Model-based testing

MBT techniques and tools have attracted increasing interest from academia and industry, because of their potential to increase the effectiveness and efficiency of the test process, by means of the automatic generation of test cases (test sequences, input test data, and expected outputs) from behavioral models of the SUT [6].

However, MBT approaches found in the literature suffer from several limitations [10], such as the lack of integrated support for the whole test process, the difficulty to bridge the gap between the model and the implementation, the difficulty to control the test case explosion problem, and the lack of support for the integration testing of distributed systems.

MBT can be done *offline* or *online*. Offline testing means that test cases are first generated and subsequently executed [11], while in online testing test generation and execution are performed together so that the test generator can react to how the SUT behaves [12]. The use of online testing is necessary if the SUT is non-deterministic, because the test

generator can see which path the SUT has taken, and follow the same path in the model [13]. We focus our research work on the *online* testing of distributed and heterogeneous systems (instead of offline testing), to cope with non-determinism. We also propose a toolset architecture to support the whole test process in an integrated fashion.

MBT approaches use a high variety of models. In general, one can distinguish state based and scenario based approaches [14]. State based approaches use abstract state machines [15], UML state machines [16], input-output automata [17] or similar behavioral models for describing *all possible behaviors* of the system or its components. Scenario-based approaches use UML SDs, message sequence charts (MSC) [18] or similar behavioral models for describing interactions between components of the system or interactions between the system and the environment that occur in specific contexts, representing *key system behaviors*. State-based models are best suited for capturing system design decisions and are usually more detailed, whilst scenario-based models are best suited for capturing system requirements [14] and are usually less detailed.

We focus our research work on a *scenario-based* approach (instead of state-based testing), because scenario-based models are more convenient for describing and visualizing the interactions that occur between the components and actors of a distributed system in key scenarios [2] [19]. Scenario-based models also help partially avoiding the test case explosion problem. To facilitate industrial adoption, we opted for using UML SDs [3] [20] as the input behavioral models.

B. Distributed testing architectures

One difficulty in testing distributed systems is that their distributed nature imposes theoretical limitations on the conformance faults that can be detected by the test components, depending on the test architecture used [7], [21]. Two basic test architectures have been proposed in the past to test distributed systems: a purely distributed test architecture with independent local testers communicating synchronously with the components of the SUT [22]; a purely centralized test architecture, in which a single centralized tester interacts asynchronously with the components of the SUT. More recently, Hierons [7] proposed a hybrid framework that combines local testers and a centralized tester. He proved that this architecture is more powerful than the distributed and centralized approaches, i.e., it has a higher fault detection capability. However, his work is only concerned with conformance relations for system testing of distributed systems based on input-output transition systems, and does not consider integration testing.

Given its advantages, we base our research work on the *hybrid test architecture* proposed by Hierons [7], but with the additional objectives of minimizing the communication overhead, avoiding, whenever possible, communications with the centralized tester or between the local testers, and supporting integration testing.

Apart from the fact that we take into account not only the interactions with the environment but also the interactions between the SUT components, our conformance checking procedure (`finalConformanceCheck`) is more restrictive than the **dioco** (distributed input-output) conformance relation described in [7] for a purely distributed test architecture, because some traces accepted as valid in **dioco** may be correctly deemed inconclusive in our case. The addition of coordination messages between the test components to ensure observability leads to a hybrid test architecture similar to the one proposed in [7] in terms of fault detection capability, but in general with a smaller communication overhead; after the addition of those messages, our conformance checking procedure becomes essentially equivalent to the $\mathbf{dioco_s}$ conformance relation defined in [7] for the hybrid test architecture.

C. Observability and controllability in distributed testing

There are some works in the literature that analyze observability and controllability in the context of distributed system testing. One of them is the work from Hierons [9], where he investigates the use of coordination messages to overcome controllability problems when testing from an input output transition system and gives an algorithm for introducing sufficient messages. From his work, we adapted the first condition of the *unintendedTraces* function. However, Hierons approach is focused on system testing (focusing only on the interactions with the environment), so the system is represented using a single lifeline and each message is represented by a single event (the system sending or receiving a message). In our work, we also analyze the interactions between the system components, and hence need to distinguish the send and receive events. This leads to the need to add a second condition related to the reception of messages (whose transmission time may be arbitrary) in the *unintendedTraces* function.

Another related work is Mitchell's work [23]; in this paper, he discusses the problems related to race conditions in scenarios described through MSCs or UML SDs. The author presents solutions to these problems but only to basic scenarios, without control flow variants, so can not be directly applied to more complex scenarios, such as those that are usually found in the description of distributed system interactions.

Boroday et al. [24] propose algorithms to extend test scenarios for distributed systems represented by MSCs or UML SDs, in order to obtain race-free scenarios suitable for test implementation, by inserting coordination messages between test components and quiescence observation events in each test component. However, in their work, only the interactions with the environment are modeled, whilst we also analyze the interactions between the system components and assume that a test component is deployed close to each component of the SUT, which modifies the determination of race-free scenarios. Nevertheless, part of the approach for insertion of coordination messages may be adapted to our context.

VI. Conclusions and Future Directions

Given the growing importance of distributed systems testing and the need to minimize the communication overhead during test execution, we address in this article the problem of

decentralized conformance checking in MBT of distributed systems.

In the case of integration testing of distributed systems based on test scenarios described by UML SDs, we established the conditions upon which coordination messages need not be exchanged between the local testers that monitor and control each component of the distributed system.

The first condition refers to local observability - the ability to detect conformance errors locally by the local testers based on the events observed locally, without the need to communicate with other local testers or a central tester.

The second condition refers to local controllability - the ability to decide when and what test inputs to inject by each local tester based on the events observed locally, without the need to communicate with other local testers or a central tester.

Local controllability and observability are formally established by analyzing the set of valid traces defined by a UML SD under consideration. We also present examples of local observability and local controllability problems associated with the presence of optional messages without corresponding acknowledgment messages, races and non-local choices.

As future work, we intend to refine the local observability and local controllability conditions and the primitives for conformance checking and test input generation, taking into account interaction parameters, message parameters, guard conditions and time constraints in UML SDs. Equivalent conditions and primitives that do not require the computation of the set of valid traces will also be investigated, because this set can be very large or even infinite. We will also investigate how to generate the coordination messages needed in the absence of local observability and/or local controllability.

ACKNOWLEDGMENT

This research work was performed in scope of the project NanoSTIMA. Project "NanoSTIMA: Macro-to-Nano Human Sensing: Towards Integrated Multimodal Health Monitoring and Analytics/NORTE-01-0145-FEDER-000016" is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). The authors would also like to acknowledge the anonymous reviewers for their helpful comments.

REFERENCES

[1] A. G. Taylor, "Keeping active with the activity app," in *Get Fit with Apple Watch*. Springer, 2015, pp. 63–69.

[2] B. Lima and J. P. Faria, *Software Technologies: 10th International Joint Conference, ICSOFT 2015, Colmar, France, July 20-22, 2015, Revised Selected Papers*. Cham: Springer International Publishing, 2016, ch. Automated Testing of Distributed and Heterogeneous Systems Based on UML Sequence Diagrams, pp. 380–396. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-30142-6_21

[3] OMG, "OMG Unified Modeling Language TM (OMG UML) Version 2.5," Object Management Group, Tech. Rep., 2015.

[4] J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef, *Validated Designs For Object-oriented Systems*. Santa Clara, CA, USA: Springer-Verlag TELOS, 2005.

[5] P. G. Larsen, K. Lausdahl, N. Battle, J. Fitzgeral, S. Wolff, S. Sahra, M. Verhoef, P. Tran-Jørgensen, T. Oda, and P. Chisholm, "VDM-10 Language Manual," Tech. Rep., 2016.

[6] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[7] R. M. Hierons, "Combining Centralised and Distributed Testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 5:1–5:29, Oct. 2014. [Online]. Available: http://doi.acm.org/10.1145/2661296

[8] B. K. Aichernig, *Automated Black-Box Testing with Abstract VDM Oracle*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 250–259. [Online]. Available: http://dx.doi.org/10.1007/3-540-48249-0_22

[9] R. M. Hierons, "Overcoming controllability problems in distributed testing from an input output transition system," *Distributed Computing*, vol. 25, no. 1, pp. 63–81, 2012. [Online]. Available: http://dx.doi.org/10.1007/s00446-011-0153-5

[10] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22Nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, ser. WEASELTech '07. New York, NY, USA: ACM, 2007, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/1353673.1353681

[11] S. Schulz, J. Honkola, and A. Huima, "Towards model-based testing with architecture models," in *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*. IEEE, 2007, pp. 495–502.

[12] M. Mikucionis, K. G. Larsen, and B. Nielsen, "T-uppaal: Online model-based testing of real-time systems," in *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*. IEEE, 2004, pp. 396–397.

[13] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012. [Online]. Available: http://dx.doi.org/10.1002/stvr.456

[14] W. Grieskamp, *Formal Approaches to Software Testing and Runtime Verification: First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ch. Multi-paradigmatic Model-Based Testing, pp. 1–19. [Online]. Available: http://dx.doi.org/10.1007/11940197_1

[15] Microsoft, "Spec Explorer," http://research.microsoft.com/en-us/projects/specexplorer/, May 2016.

[16] A. Huima, "Implementing Conformiq Qtronic," in *Testing of Software and Communicating Systems*. Springer, 2007, pp. 1–12.

[17] Q. Tani and A. Petrenko, "Input/output automata'," in *Testing of Communicating Systems: Proceedings of the IFIP TC6 11th International Workshop on Testing of Communicating Systems (IWTCS'98) August 31-September 2, 1998, Tomsk, Russia*, vol. 3. Springer, 2013, p. 83.

[18] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *Formal methods in system design*, vol. 19, no. 1, pp. 45–80, 2001.

[19] A. Ulrich, E.-H. Alikacem, H. H. Hallal, and S. Boroday, *From Scenarios to Test Implementations Via Promela*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 236–249. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16573-3_17

[20] H.-G. Gross, *Component-Based Software Testing with UML*. Springer Berlin Heidelberg, 2005.

[21] R. M. Hierons, M. G. Merayo, and M. Núñez, "Scenarios-based testing of systems with distributed ports," *Software: Practice and Experience*, vol. 41, no. 10, pp. 999–1026, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1062

[22] A. Ulrich and H. König, "Architectures for Testing Distributed Systems," in *Testing of Communicating Systems*, ser. IFIP - The International Federation for Information Processing, G. Csopaki, S. Dibuz, and K. Tarnay, Eds. Springer US, 1999, vol. 21, pp. 93–108. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-35567-2_7

[23] B. Mitchell, "Resolving race conditions in asynchronous partial order scenarios," *IEEE Transactions on Software Engineering*, vol. 31, no. 9, pp. 767–784, Sept 2005.

[24] S. Boroday, A. Petrenko, and A. Ulrich, "Implementing MSC Tests with Quiescence Observation," in *Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, ser. TESTCOM '09/FATES '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 49–65. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-05031-2_4