IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 13, 2020, accepted August 17, 2020, date of publication September 4, 2020, date of current version September 23, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.3021858*

# Local Observability and Controllability Analysis and Enforcement in Distributed Testing With Time Constraints

**BRUNO LIMA[1,2], (Student Member, IEEE), JOÃO PASCOAL FARIA[1,2], (Member, IEEE), AND ROBERT HIERONS[3], (Senior Member, IEEE)**

[1]Department of Informatics Engineering, Faculty of Engineering, University of Porto, 4200-465 Porto, Portugal
[2]INESC TEC, FEUP, 4200-465 Porto, Portugal
[3]Department of Computer Science, The University of Sheffield, Sheffield S10 2TN, U.K.

Corresponding author: Bruno Lima (bruno.lima@fe.up.pt)

This work was financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, under research grant SFRH/BD/115358/2016 and within the project UIDB/50014/2020.

**ABSTRACT** Evermore end-to-end digital services depend on the proper interoperation of multiple products, forming a distributed system, often subject to timing requirements. To ensure interoperability and the timely behavior of such systems, it is important to conduct integration tests that verify the interactions with the environment and between the system components in key scenarios. The automation of such integration tests requires that test components are also distributed, with local testers deployed close to the system components, coordinated by a central tester. Test coordination in such a test architecture is a big challenge. To address it, in this article we propose an approach based on the pre-processing of the test scenarios. We first analyze the test scenarios in order to check if conformance errors can be detected locally (*local observability*) and test inputs can be decided locally (*local controllability*) by the local testers for the test scenario under consideration, without the need for exchanging coordination messages between the test components during test execution. If such properties do not hold, we next try to determine a minimum set of coordination messages or time constraints to be attached to the given test scenario to enforce those properties and effectively solve the test coordination problem with minimal overhead. The analysis and enforcement procedures were implemented in the DCO Analyzer tool for test scenarios described by means of UML sequence diagrams. Since many local observability and controllability problems may be caused by design flaws or incomplete specifications, and multiple ways may exist to enforce local observability and controllability, the tool was designed as a static analysis assistant to be used before test execution. DCO Analyzer was able to correctly identify local observability and controllability problems in real-world scenarios and help the users fix the detected problems.

**INDEX TERMS** Test scenarios, observability, controllability, distributed systems, time constraints.

## I. INTRODUCTION

Due to the increasing ubiquity, complexity and need for assurance of software-based systems [1], testing is a fundamental but challenging lifecycle activity, with a huge economic impact if not performed adequately [2]. This is particularly true for the end-to-end services that are being proposed in several domains (e-health, smart cities, etc.), taking advantage of recent advances in cloud, mobile computing, and Internet of Things (IoT) [3]–[5]. Such services depend on the proper interoperation of multiple devices and applications from different vendors, forming a distributed and heterogeneous system or system of systems, often subject to timing requirements. To ensure interoperability and the correct, secure and timely behavior of such systems, it is important to conduct integration tests that verify not only the interactions with the environment but also between the system components in key scenarios. However, test automation in this type of systems is a huge challenge [6].

Integration test scenarios may be conveniently specified by means of UML Sequence Diagrams [7] (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, and may be enriched with

The associate editor coordinating the review of this manuscript and approving it for publication was Jianquan Lu.

control flow variants and time constraints, as illustrated by the example of Fig. 1.

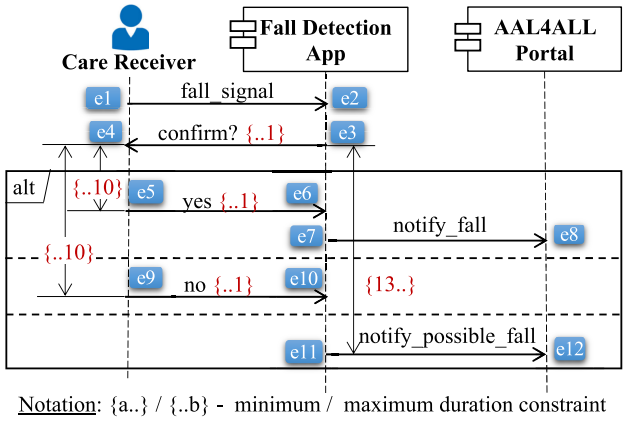



**FIGURE 1. Example of a fall detection scenario (simplified) from an ambient assisted living ecosystem (AAL4ALL).**

In this scenario, a care receiver has a smartphone that has installed a fall detection application. When this person falls, the application detects the fall and provides the user with a message which indicates that it has detected a drop giving the possibility for the user to confirm whether he/she needs help. If the user responds that he/she does not need help, the application does not perform any action; however, if the user confirms that he/she needs help or does not respond within 10 seconds, the application sends an alert to a web application called AAL4ALL Portal. In addition to the maximum duration constraint of 10 seconds for the user response, other time restrictions are also represented, namely 1 second as the maximum delivery time of the messages and 13 seconds between the sending of the notification to the user and the sending of the alert message in case of no response.

In order to be able to check the interactions with the environment (actors) and between the system components, and simulate inputs from the environment at multiple locations, *local testers* have to be deployed close to the system components, coordinated by a *central tester*, as depicted in Fig. 2. The local testers may act as *test monitors* (observing the messages sent and received by each component), *test drivers* (simulating inputs from the environment), or even *test stubs* (simulating responses from emulated system components).

To cope with non-determinism (multiple system outputs being possible for the same input sequence) and response time constraints, test inputs may have to be selected at runtime in an adaptive and responsive way, based on the observed execution events and the behavioral specification (UML SD), suggesting an *adaptive and distributed test input generation* approach. To facilitate fault localization, conformance errors (i.e., deviations from the behavioral specification) should be detected as early as possible and as close as possible to the offending components, suggesting an *incremental and distributed conformance checking* approach. Hence, the test components (central and local testers) in the middle layer

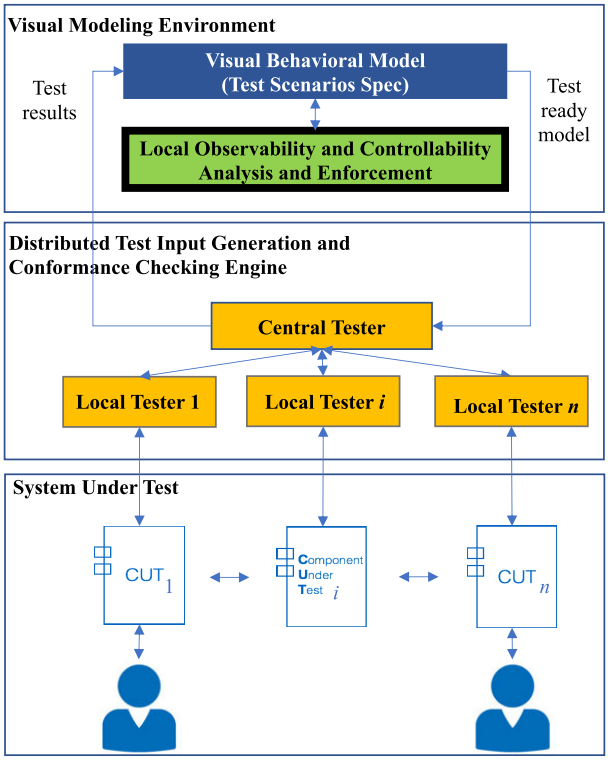



**FIGURE 2. Test architecture for the model-based integration testing of distributed systems.**

of Fig. 2 work as a *Distributed Test Input Generation and Conformance Checking Engine*.

Test coordination in such a test architecture is a big challenge.

To address this, we first check, in a pre-processing step (performed by the *Local Observability and Controllability Analysis and Enforcement* component in Fig. 2, closely integrated with the *Visual Modeling Environment*), if conformance errors can be detected locally (*local observability*) and test inputs can be decided locally (*local controllability*) by the local testers for the test scenario under consideration, without the need for exchanging coordination messages between the test components during test execution (which could delay test input selection and conformance checking and impose a communication overhead). In that case, a purely distributed testing approach can be followed: after the central tester initiates the local testers, no communication between test components occurs during test execution; the central tester only needs to receive a verdict from each local tester at the end of successful execution or as soon as an error is detected.

If the properties of local (distributed) observability and controllability do not hold for the test scenario under consideration, we next try to determine a minimum set of coordination messages or coordination time constraints to be attached to the given test scenario to enforce those properties, whilst preserving the semantics of the test scenario. Then the refined test scenario (test ready model in Fig. 2) is executed as in the purely distributed approach. If only coordination time

constraints are added, the whole testing approach is still purely distributed. But if coordination messages are added, the whole testing approach becomes a hybrid one (with some coordination messages exchanged during test execution, with minimal overhead and delays).

The analysis and enforcement procedures were implemented in the DCO Analyzer tool for test scenarios described by means of UML sequence diagrams. DCO Analyzer was able to correctly identify local observability and controllability problems in real-world scenarios and help the users fix the detected problems. Since many local observability and controllability problems may be caused by design flaws or incomplete specifications, and multiple ways may exist to enforce local observability and controllability, the tool was designed as a static analysis assistant to be used before test execution.

To our knowledge, although observability and controllability have been addressed by other authors in the context of distributed systems testing, they were not analyzed before in the context of integration testing with control flow variants and time constraints. The ability to recommend fixes in such a context is also absent in other works, to our knowledge.

The main contributions of this article are:

- examples of test scenarios that exhibit different combinations of local observability and local controllability properties, illustrating common causes of those problems and ways to overcome them when appropriate;
- a set of procedures and a tool to automatically analyze test scenarios with control flow variants and time constraints, and check their local (distributed) observability and controllability, pinpointing any violations found;
- a set of procedures and a tool to automatically suggest coordination messages and/or coordination time constraints to be added to test scenarios to enforce local observability and/or local controllability;
- description of a real-world case study showing the usefulness of local observability and controllability analysis.

The rest of the paper is organized as follows: Section II provides some insight about the problem addressed based on a few examples; Section III presents some concepts, assumptions and definitions; procedures for checking local observability and controllability are presented in Sections IV and V; procedures for enforcing local observability and controllability are presented in Section VI; implementation and evaluation (case study) are discussed in Sections VII and VIII; related work is presented in Section IX; conclusions and future work are presented in Section X.

## II. MOTIVATING EXAMPLES

Figures 3 and 4 show examples of simple scenarios to illustrate local observability and controllability problems and ways to overcome them.

Scenario a) illustrates a local controllability problem caused by a race condition. Based on local knowledge only,

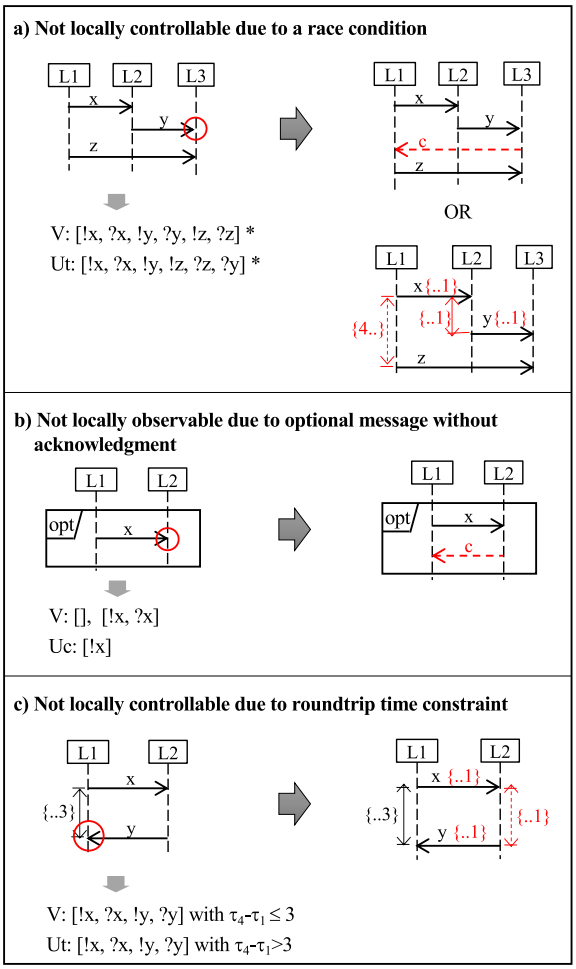



**FIGURE 3. Interaction fragments with local observability and controllability problems and possible refinements.**

lifeline $L1$ doesn't know when to send $z$ to ensure that it arrives at $L3$ after $y$, so it may generate invalid (*unintended*) traces with $?z$ before $?y$. On the right, are illustrated two ways to overcome this problem. In the first solution, a *coordination message* is transmitted from $L3$ to $L1$, so that $L1$ knows when to safely send $z$. From a testing perspective, assuming that $L1$ is simulated by a local tester (test driver) and $L3$ is monitored by another local tester, the coordination message would be exchanged between the local testers (without affecting the SUT). The communication overhead of this solution (1 message) is much smaller than the overhead incurred by a centralized testing approach, in which the events observed by the local testers are constantly communicated to the central tester (4 messages from the local testers at $L2$ and $L3$ to the central tester), that decides and communicates back to the local testers the next test inputs (2 messages from the central

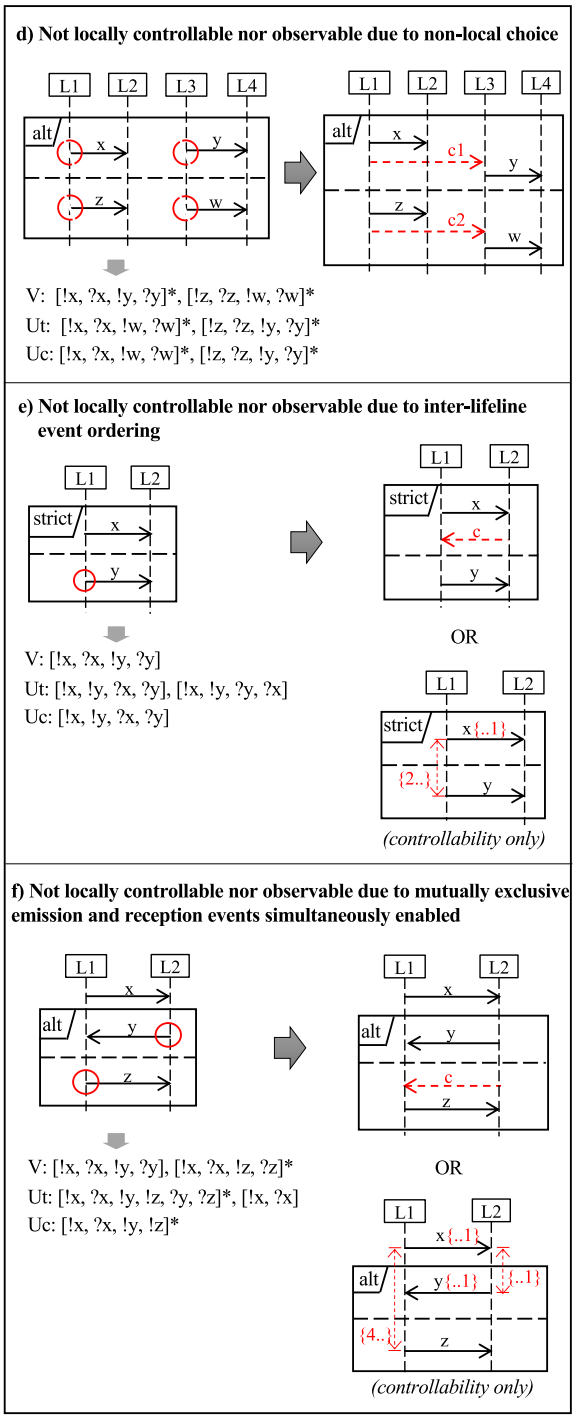



**FIGURE 4.** **Interaction fragments with local observability and controllability problems and possible refinements (continued).**

tester to the local tester at $L1$). The second solution relies on *coordination time constraints*. From a testing perspective, the maximum duration constraints could represent assumptions about the SUT behavior (lifelines and communication channels), and the minimum duration constraint could represent a constraint to be followed by the test driver at $L1$. If such assumptions can be made, this approach has the advantage of not implying any communication overhead during test execution (possibly at the cost of a pessimistic wait time at $L1$).

Scenario b) illustrates a local observability problem caused by an optional message without a corresponding acknowledgment message. If message $x$ is lost (i.e., is sent by $L1$ but does not arrive at $L2$), the problem will go unnoticed at $L2$, because not receiving any message is also a valid behavior. In other words, the invalid trace $[!x]$ is *locally uncheckable*. This problem may be overcome by adding a coordination (acknowledgment) message $c$, as illustrated on the right; now, if $x$ is lost, that will be noticed at $L1$. The coordination message need only be exchanged between the local testers. Again, the communication overhead of this solution (1 message) is smaller than the overhead of a centralized testing approach, in which the events observed by the local testers are constantly communicated to the central tester for conformance checking (2 messages from the local testers at $L1$ and $L2$ to the central tester).

In scenario c), a roundtrip time constraint causes a local controllability problem. Since there are no limits on the transmission times of $x$ and $y$, nor on the reaction time of $L2$, there is no guarantee that the roundtrip constraint will be met, so invalid (unintended) traces may be generated violating it. The problem may be solved by setting appropriate limits on the transmission and reaction times, as illustrated on the right. This example also illustrates a tension between local controllability and local observability, because the scenario on the left is locally observable, contrarily to the scenario on the right (inter-lifeline time constraints can only be checked after merging the traces observed at each lifeline).

Scenario d) (Fig. 4) illustrates a local observability and local controllability problem due to a non-local choice. In this case, and based only on local information, $L3$ does not know in which situations it should send $y$ or $w$, leading to invalid (*unintended*) traces with combinations of $x$ & $w$ or $z$ & $y$. Locally this error is also not detectable, since for $L2$ and $L4$, reception of $x$ or $z$ and $y$ or $w$ is always locally valid. In order to solve this problem (as shown on the right), two coordination messages ($c1$ and $c2$) are required between $L1$ and $L2$. With these coordination messages, $L3$ becomes able to know locally which message to send in order to ensure correct execution. Once again, the communication overhead of this solution (2 messages) is smaller than the overhead of a centralized testing approach.

Scenario e) illustrates a local observability and local controllability problem caused by an inter-lifeline event ordering constraint. Based on local knowledge only, lifeline $L1$ does not know when to send $y$ to ensure that this is done only after $x$ has reached $L2$ (the `strict` interaction operator requires that all events in one interaction operand occur before all the events in the next operand). The early emission of $y$ can then lead to invalid (*unintended*) traces with $!y$ before $?x$. On the other hand, the above error may not be locally observable, since, based on local knowledge only, the invalid execution trace $[!x, !y, ?x, ?y]$ is *locally uncheckable*. This problem may be overcome by adding a coordinating message $c$ between

$L2$ and $L1$, so that $L1$ knows when it can send message $y$. The communication overhead of this solution (1 message) is again smaller than the overhead of a centralized testing approach. Alternatively, the problem may be overcome by adding *coordination time constraints* in a way similar to scenario a) (with the difference that, in this case, the ordering we want to enforce is between events in different lifelines).

Scenario f) illustrates a local observability and local controllability problem caused by mutually exclusive emission and reception events simultaneously enabled. In this case, $L1$ and $L2$ do not have local information that allows them to determine which alternative should be executed; this can lead to invalid (*unintended*) traces in which both $y$ and $z$ are sent or none is sent. The scenario is also *locally uncheckable*, since the loss of both messages $y$ and $z$ will not be detected by $L1$ and $L2$. This problem may be overcome by adding a coordinating message $c$ between $L1$ and $L2$. Alternatively, the controllability problem may be overcome by adding *coordination time constraints* so that emission and reception events are not enabled at the same time from the perspective of any of the lifelines. In this case, the minimum duration constraint may be seen as a timeout after which $L1$ may send $z$.

In all cases, the scenarios on the right are *refinements* of the scenario on the left, in the sense that execution traces valid for the latter are also valid for the former (with coordination messages removed), although the opposite may not be true (that is, the semantics is narrowed for the sake of implementability and testability).

In the rest of the paper we show how to automatically check if an integration test scenario is locally observable and locally controllable, pinpointing any violations (locally uncheckable and unintended traces, respectively), and automatically suggest coordination messages and/or time constraints to enforce those properties.

The results of local observability and controllability analysis can be used by a user or a tool to refine the scenario or decide about the test approach in several ways:

- if the analysis shows that a test scenario is locally observable and controllable, then it can be executed safely in a decentralized way, without any communication overhead during test execution; this is particularly important when the local testers have to inject time-constrained inputs (as in Fig 1);
- if the analysis shows that a test scenario is locally controllable but not locally observable (as in scenario *b* above), then it can still be executed safely in a decentralized way, requiring only that events observed by the local testers are communicated to the central tester at the end of test execution to arrive at a final verdict (at the cost of delayed error detection, complicated by non-synchronized clocks);
- in many cases, local observability and controllability problems are associated with incomplete specifications or design flaws [8], so the analysis helps to identify the needed refinements;
- in other cases, the analysis helps identifying timing constraints or coordination messages to insert manually or automatically to enforce local observability or, at least, local controllability, with a lower communication overhead than a centralized testing approach.

## III. PRELIMINARIES

Before investigating the procedures for local observability and controllability checking of time-constrained SDs, it is important to formalize their syntax and semantics.

### A. TIME-CONSTRAINED SEQUENCE DIAGRAMS

In UML, an SD is a variant of an *Interaction* [7]. SDs may be annotated with *time constraints* [7], as illustrated by the SD of Fig. 1. Although the UML standard allows the specification of more complex constraints, in this article we restrict our attention to the types of time constraints that are commonly addressed in the literature and are most relevant in practice: constraints that specify the minimum and maximum duration between two events (message sending or receiving) in the same lifeline, or between the sending and receiving of a message between two lifelines.

### B. TIMED TRACES

In UML, the semantics of an Interaction is expressed in terms of sets of *valid* and *invalid traces* [7]. In this article, we do not handle the rarely used constructs for defining invalid traces (such as the *neg* interaction operator), so only the valid traces are relevant here.

In general, a trace is a sequence of *event occurrences* [7], corresponding to the sending or receiving of messages at lifelines. We represent an *event* by a tuple $\langle m, l, k\rangle$, where $m$ is the message, $l$ is the lifeline where the event occurs and $k$ is the event kind (*Send* or *Receive*). For example, the event $e1$ shown in Fig. 1 may be represented by the tuple $\langle$*"fall_signal", "Care Receiver", Send*$\rangle$.

In the presence of time constraints, it is important to store time information associated with the event occurrences. We use the term *timed traces* (or *t-traces*, for short) for traces that convey the time instants of the event occurrences, and represent them by a sequence of pairs of events and associated time instants, in some integer time scale (seconds, milliseconds, etc.) as in $[\langle e1, 1\rangle, \langle e2, 5\rangle, \langle e3, 8\rangle]$.

### C. TIME-CONSTRAINED TRACES

Since the set of valid timed traces defined by an SD is usually infinite, we need a finite representation by means of a set of *time-constrained traces* (or *tc-traces*, for short).

A tc-trace is a pair of a trace and an associated Boolean expression on *time constraints between pairs of event occurrences*. In those constraints, the time instance of the $i$-th event occurrence is represented by the *time variable* $\tau_i$. The time constraints are normalized as a conjunction of *difference constraints* [9] of the form $\tau_i - \tau_j \leq d$, where $d$ is a time duration literal (positive or negative integer).

For example, the SD of Fig. 1 defines the following valid tc-traces:

- $\langle [e1, e2, e3, e4, e5, e6, e7, e8], \tau_4 - \tau_3 \leq 1 \wedge \tau_5 - \tau_4 \leq 10 \wedge \tau_6 - \tau_5 \leq 1 \rangle$
- $\langle [e1, e2, e3, e4, e9, e10], \tau_4 - \tau_3 \leq 1 \wedge \tau_5 - \tau_4 \leq 10 \wedge \tau_6 - \tau_5 \leq 1 \rangle$
- $\langle [e1, e2, e3, e4, e11, e12], \tau_4 - \tau_3 \leq 1 \wedge \tau_3 - \tau_5 \leq -13 \rangle$

### D. VALID TRACES AND SATISFIABILITY CHECKING

We express the semantics of a time-constrained SD by a set of valid tc-traces. In this article, we assume that loops have (or are explored up to) a bounded number of iterations, so such a set is finite.

*Procedure 3.1 (Valid Time-Constrained Traces):* We compute the set $\mathcal{V}(\iota)$ of valid tc-traces defined by an interaction $\iota$ in 3 steps:

1) Compute the set $\mathcal{U}(\iota)$ of valid (untimed) traces defined by $\iota$ ignoring time constraints, following the procedure described in [10] (this set gives all the possible event combinations and total orderings defined by $\iota$);
2) Obtain the set $\mathcal{D}(t, \iota)$ of time constraints applicable to each trace $t$ in $\mathcal{U}(\iota)$ (see Proc. 3.2);
3) Determine the satisfiability of those constraints (*sat*), and select the traces with satisfiable constraints (see Proc. 3.3).

Formally,

$$\mathcal{V}(\iota) \triangleq \{(t, \mathcal{D}(t, \iota)) | t \in \mathcal{U}(\iota) \wedge sat(\mathcal{D}(t, \iota))\}$$

The procedure for obtaining the applicable time constraints is presented next. The last condition is important for SDs with loops, to make sure that time constraints are applied to event occurrences in the same loop iteration. To this end, the untimed traces calculated by $\mathcal{U}(\iota)$ include the iteration counter of each event occurrence.

*Procedure 3.2 (Applicable Time Constraints):* Generates a conjunctive expression with time constraints between time instants of event occurrences in a (untimed) trace $t$ of an interaction $\iota$, based on the constraints defined between pairs of events in $\iota$.

$$D(t, \iota) \triangleq \bigwedge \{\tau_i + min \leq \tau_j \leq \tau_i + max | 1 \leq i < j \leq |t| \\ \wedge \langle t_i, t_j, min, max \rangle \in timeConstr(\iota) \\ \wedge itercounter(t_i) = itercounter(t_j)\}$$

A set of time constraints $c$ is satisfiable for a trace $t$ if there is an assignment of non-decreasing time instants to the event occurrences in $t$ that satisfies all the constraints in $c$. Due to the special nature of the time constraints involved (conjunction of difference constraints), satisfiability can be checked in polynomial time, following the procedure summarized below (partly based on [9]) and illustrated in Fig. 5. The example refers to a trace derived from the SD of Fig. 1 that is valid when the time constraints are ignored but is invalid otherwise. In the case of a more general Boolean expression on difference constraints, as we will need later, we reduce the expression to disjunctive normal form (DNF), and apply the same procedure to each conjunctive term.

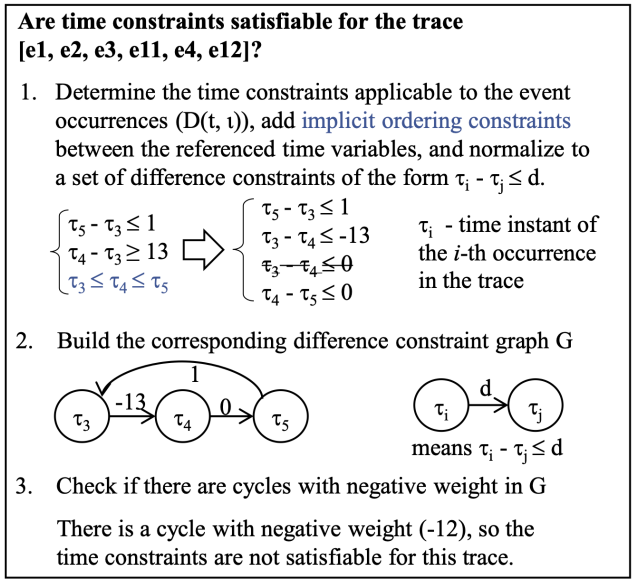



**FIGURE 5.** Satisfiability checking example (trace from Fig. 1).

*Procedure 3.3 (Satisfiability Checking):* Checks if a conjunctive expression $E$ on time constraints is satisfiable ($sat(E)$), i.e., there is an assignment of non-decreasing values to the time variables referenced in $E$ that makes the expression *true*, as follows:

1) Add to $E$ implicit ordering constraints $\tau_i \leq \tau_j$ between consecutive variables referenced in $E$ (ordered by their indices).
2) Normalize $E$ into a conjunction $E'$ of difference constraints of the form $\tau_i - \tau_j \leq d$, where $\tau_i$ and $\tau_j$ are integer (time) variables and $d$ is a literal integer.
3) Build the corresponding difference constraint graph $G$, with an edge $(i, j)$ of weight $d$ for each difference constraint $\tau_i - \tau_j \leq d$ in $E'$.
4) $E$ is satisfiable iff $G$ has no cycles of negative weight.

### E. OPERATORS ON TIMED TRACES AND TIME-CONSTRAINED TRACES

The definitions and procedures for local observability and controllability analysis use the operators defined in Fig. 6. Due to space limitations, implicit (instead of explicit) definitions are given for some operators, resorting to a function (*ext*) that gives the (possibly infinite) set of timed traces defined by a set of tc-traces. We also apply the *ext* function to complex structures (such as maps), in order to convert all occurrences of sets of tc-traces to corresponding sets of timed traces. By a *feasible* timed trace (see the join operator), we mean a timed trace with non-decreasing time instants that respects the fact that messages can be received only after being sent. Application examples can be found in Fig. 7.

| Name | Notation | Description | Formal definition |
|---|---|---|---|
| Project | $\pi_l t$ | Project a t-trace $t$ onto a lifeline $l$ | $\pi_l t = [(e,\tau) \in t \mid \text{lifeline}(e)=l]$ |
| | $\pi_l T$ | Project a set $T$ of t-traces onto a lifeline $l$ | $\pi_l T = \{\pi_l t \mid t \in T\}$ |
| | $\pi_L X$ | Project t-trace(s) $X$ onto a set $L$ of lifelines | $\pi_L X = \{l \mapsto \pi_l X \mid l \in L\}$ |
| | $\pi_Z Y$ | Project tc-trace(s) $Y$ onto lifeline(s) $Z$ | $\text{ext}(\pi_Z Y) = \pi_Z \text{ext}(Y)$ |
| Join | $\bowtie M$ | Feasible joins of the sets of t-traces in a map $M$ from lifelines to sets of t-traces | $t \in \bowtie M \Leftrightarrow \text{isFeasible}(t) \wedge (\forall l \in L \bullet \pi_L t \in M(l))$ |
| | $\bowtie N$ | Feasible joins of the sets of tc-traces in a map $N$ from lifelines to sets of tc-traces | $\text{ext}(\bowtie N) = \bowtie \text{ext}(N)$ |
| Difference | $U_1 \setminus U_2$ | Difference between two sets of tc-traces | $\text{ext}(U_1 \setminus U_2) = \text{ext}(U_1) \setminus \text{ext}(U_2)$ |
| Equality | $U_1 = U_2$ | Equality of sets of tc-traces | $U_1 = U_2 \Leftrightarrow \text{ext}(U_1) = \text{ext}(U_2)$ |

**FIGURE 6.** Operators on timed traces and time-constrained traces.

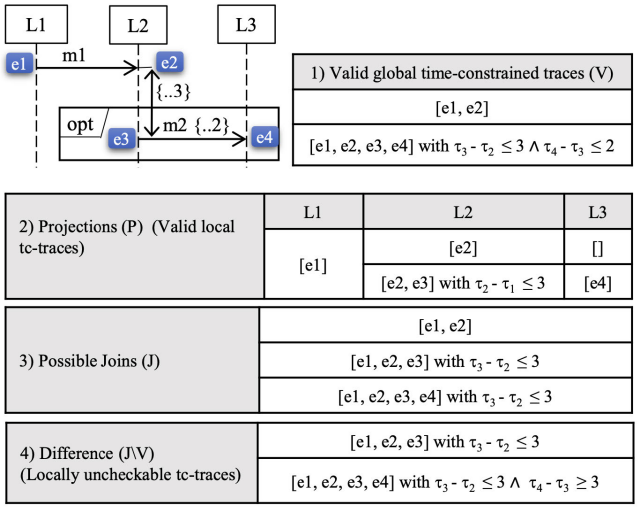



**FIGURE 7.** Example of local observability checking.

## IV. LOCAL OBSERVABILITY ANALYSIS

In this section, we present procedures to check if conformance checking of observed execution traces against the expectations set by a time-constrained 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*). The procedures presented in this article extend, for the case of time-constrained SDs, the procedures presented in [10] for SDs without time constraints.

Local observability is best defined in terms of timed traces, but, since the set of valid timed traces is usually (almost) infinite, it is best checked in terms of tc-traces.

### A. DEFINITIONS

In this article we assume a strict notion of conformance, i.e., we say that an observed time trace $t$ *conforms to* the specification (described by a time-constrained interaction $\iota$), or is *globally valid*, when $t \in ext(\mathcal{V}(\iota))$. However, in distributed testing, global traces are not directly observed, but only the local traces observed at each lifeline. We say that a timed trace $t$ is *locally valid* when $\forall_{l \in \mathcal{L}(\iota)}, \pi_l t \in ext(\pi_l \mathcal{V}(\iota))$, where $\mathcal{L}(\iota)$ denotes the lifelines in $\iota$.

We next define local observability based on the concepts of global and local validity.

*Definition 4.1 (Local Observability):* We say that a test scenario specified by a time-constrained interaction $\iota$ is *locally observable* iff there are no feasible timed traces that are locally valid but are not globally valid (also called *locally uncheckable* traces).

### B. LOCAL OBSERVABILITY CHECKING

*Procedure 4.1 (Local Observability Checking):* We check the *local observability* of a test scenario described by a time-constrained interaction $\iota$ in a constructive way (pinpointing violations), as follows:

1) Calculate the set $\mathcal{V}(\iota)$ of valid tc-traces defined by $\iota$;
2) Compute the valid local tc-traces in each lifeline, i.e., the projection $P$ of $\mathcal{V}(\iota)$ onto $\mathcal{L}(\iota)$;
3) Compute the set $J$ of all possible feasible joins of traces in $P$;
4) Compute the global tc-traces that are not locally checkable, by subtracting from $J$ the valid traces $\mathcal{V}(\iota)$.
5) The given scenario $\iota$ is locally observable iff the previous result is empty.

Formally,

$$isLocallyObservable(\iota) \triangleq (\bowtie (\pi_{\mathcal{L}(\iota)} \mathcal{V}(\iota))) \setminus \mathcal{V}(\iota) = \emptyset$$

*Theorem 4.1 (Correctness of Procedure 4.1):* Procedure 4.1 correctly checks if an interaction $\iota$ is locally observable.

*Proof:* Follows from Definition 4.1 and from the definitions of the operators involved in Procedure 4.1. Based on the meaning of the difference operator (see Fig. 5), the right-hand side of the formula in Procedure 4.1 can be rewritten:

$$\{t \mid t \in ext(\bowtie (\pi_{\mathcal{L}(\iota)} \mathcal{V}(\iota))) \wedge t \notin ext(\mathcal{V}(\iota))\} = \emptyset$$

Based on the definitions of the join operator (see Fig. 6), the first term $(t \in ext(\bowtie (\pi_{\mathcal{L}(\iota)} \mathcal{V}(\iota))))$ can be rewritten:

$$\forall_{l \in \mathcal{L}(\iota)}, \pi_l t \in ext(\pi_l \mathcal{V}(\iota))$$

This corresponds to the definition of local validity in Definition 4.1, whilst the second term $(t \notin ext(\mathcal{V}(\iota)))$ corresponds to the negation of global validity. Hence, we conclude that Procedure 4.1 correctly checks local observability.

*Example 4.1:* Procedure 4.1 is illustrated in Fig. 7. In this case, there are two tc-traces that are not locally checkable, so the scenario is not locally observable. The first one is due to an optional message without a corresponding acknowledgment message. The second one is due to an inter-lifeline time constraint (transmission constraint) that is not present in the projections onto the lifelines.

### C. IMPACT OF NON-SYNCHRONIZED CLOCKS

Next we show that imperfect clock synchronization in a distributed SUT does not affect local observability. In distributed testing, the time instant of each observed event occurrence is measured with the local clock of the respective lifeline. Although it is impossible to ensure perfect clock synchronization between lifelines, in practice the difference between the readings of any two clocks (*clock skew*) may be limited to a small value of the order of 10ms over Internet and below 1ms over LAN [11]. This clock skew might not have a practical impact for coarser time scales used in testing (e.g., seconds), but could be relevant for finer time scales (e.g., milliseconds). In any case, for test cases that run for short time spans, one can assume that there is no noticeable *clock drift* during test execution (i.e., clocks run at the same rate). Under this assumption, we next prove our proposition.

*Theorem 4.2 (Local Observability and Clock Synchronization):* If an interaction $\iota$ is locally observable with perfectly synchronized clocks, then it is also locally observable if clocks are not perfectly synchronized but run at the same rate.

*Proof:* Let us assume that $\iota$ is locally observable, i.e., all invalid feasible global traces are also locally invalid, in case the clocks are perfectly synchronized. Let us pick one arbitrary of those invalid feasible global traces $t$, and let us denote by $t_k$ an invalid local trace observed at a lifeline $k$ (based on our assumption, such lifeline and trace must exist). In case clocks are not perfectly synchronized but run at the same rate, the time instants of the corresponding observed local trace $t'_k$ in lifeline $k$ will differ from the time instants in $t_k$ by a constant amount (the clock skew $\delta_k$ of lifeline $k$). Since all the local time constraints we are considering are difference constraints, shifting time instants by the same amount will not affect the validity of those constraints. So $t'_k$ will also be checked as invalid by lifeline $k$. Hence we conclude that invalid traces will also be detected locally.

This result might look surprising, but is in reality consistent with the fact that scenarios with inter-lifeline time constraints not implied by other constraints are not locally observable.

## V. LOCAL CONTROLLABILITY ANALYSIS

### A. DEFINITIONS AND EXAMPLE

*Definition 5.1 (Local Controllability):* We say that a time-constrained interaction $\iota$ is locally controllable if no invalid timed traces are generated (i.e., there are no *unintended traces*) and all valid timed traces can be generated (i.e., there are no *missing traces*) when the lifelines and the communication channels behave in a locally correct way, using local knowledge only. Formally, denoting by $\mathcal{S}(\iota)$ the set of feasible timed traces that can be generated when the lifelines and the communication channels behave in a locally correct way, $\iota$ is locally controllable iff $\mathcal{S}(\iota) = ext(\mathcal{V}(\iota))$. Unintended traces are given by $\mathcal{S}(\iota) \setminus ext(\mathcal{V}(\iota))$. Missing traces are given by $ext(\mathcal{V}(\iota)) \setminus \mathcal{S}(\iota)$.

In a locally controllable interaction, local correctness of actions implies global correctness. Local controllability ensures that the decision of when and what inputs to inject can be taken locally by the local testers (simulating lifelines that represent external actors or mocked components) using local knowledge only, without the need to exchange coordination messages between the test components during test execution.

*Example 5.1:* The scenario of Fig. 1 is locally controllable. In fact, the projection of the defined time constraints onto the "Fall Detection App" lifeline generates the derived local constraints $time(e6) \leq time(e3) + 12$ and $time(e10) \leq time(e3) + 12$. So, the lifeline knows that, after requesting confirmation from the user (event $e3$), it should wait for a response (events $e6$ or $e10$) of up to 12 time units, and only send "notify_possible_fall" after at least one more time unit. Without the specified constraints, the scenario would not be locally controllable, because the lifeline would not know how much time to wait before sending "notify_possible_fall". This could result in the generation of invalid traces such as:

- $[e1, e2, e3, e4, e11, e12, e5, e6]$ (and other permutations with $e11$ before $e6$)
- $[e1, e2, e3, e4, e11, e12, e9, e10]$ (and other permutations with $e11$ before $e9$)

We next clarify and formalize the notion of a locally correct behavior of lifelines, in Definition 5.2, and communication channels, in Definition 5.3. The set $\mathcal{S}(\iota)$ contains all feasible timed traces that satisfy the conditions of 5.2 and 5.3.

*Definition 5.2 (Locally Correct Behavior of Lifelines):* A global timed trace $t$ in an interaction $\iota$ demonstrates a locally correct (and complete) behavior of a lifeline $l \in \mathcal{L}(\iota)$ iff the local timed trace $p = \pi_l t$ observed at $l$ satisfies the following conditions:

(a) all outputs (emissions) are locally valid, i.e.,

$$\forall i \in inds(p) \cdot isSend(p_i) \implies p_{1,\ldots,i} \in P_l$$

where $P_l = prefixes(V_l)$ and $V_l = \pi_l ext(\mathcal{V}(\iota))$;

(b) $l$ may remain in a *quiescent* state after $p$ (i.e., not send any output, at least without first receiving an input [12]), because one of the following holds (denoted $Q_l(p)$):

(i) $p$ is a locally valid trace, i.e., $p \in V_l$;

(ii) in case there are valid outputs that can be sent after $p$, there are also valid inputs that can be received with a deadline greater or equal than the deadline for the outputs (in this case, $l$ may decide to wait for input, and, if it does not arrive up to the deadline, will no longer be able to send any output); formally,

$$\forall p \curvearrowright [s] \in P_l \cdot isSend(s) \implies$$
$$\exists p \curvearrowright [r] \in P_l \cdot isRecv(r) \wedge time(r) \geq time(s);$$

(c) there are no missing intermediate outputs, i.e., for each input event $p_i$ in $p$, not send any output between $p_{i-1}$ and $p_i$ is a valid behavior of $l$ (because of a quiescent state or because possible outputs have not expired); formally,

$$\forall i \in inds(p) \cdot isRecv(p_i) \implies Q_l(p_{1,\ldots,i-1}) \vee$$
$$\exists p_{1,\ldots,i-1} \curvearrowright [s] \in P_l \cdot time(s) \geq time(p_i).$$

*Definition 5.3 (Correct Behavior of Communication Channels):* A global timed trace $t$ in an interaction $\iota$ demonstrates a correct (and complete) behavior of the communication channels iff the following conditions hold:

(a) messages are delivered within the specified transmission duration constraints (between pairs of related emission and reception events in $t$);
(b) all messages are delivered (i.e., for each emission event in $t$ there is a corresponding reception event).

### B. SYMBOLIC SIMULATION

Because $\mathcal{S}(\iota)$ may be infinite or almost infinite, we calculate a finite set $\mathcal{S}'(\iota)$ of tc-traces (instead of timed traces), equivalent to $\mathcal{S}(\iota)$ in the sense that $ext(\mathcal{S}'(\iota)) = \mathcal{S}(\iota)$.

$\mathcal{S}'(\iota)$ is calculated incrementally by symbolic simulation, starting from the empty tc-trace, as outlined in Procedure 5.1.

*Procedure 5.1 (Symbolic Simulation):* Computes a set $\mathcal{S}'(\iota)$ of tc-traces describing the timed traces that can be generated by the execution of an interaction $\iota$ when the lifelines and communication channels behave in a locally correct way, as follows:

$$\mathcal{S}'(\iota) \triangleq \{\langle u, c \wedge Q_\iota(\langle u, c\rangle)\rangle | \langle u, c\rangle \in \mathcal{T}_\iota^*(\langle [], true\rangle) \wedge sat(Q_\iota(\langle u, c\rangle))\}$$

where

- $\mathcal{T}_\iota(\langle u, c\rangle)$ is a *transition function* that gives the successors of tc-trace $\langle u, c\rangle$ (a pair of a trace $u$ and a constraint $c$) in the symbolic execution tree of $\iota$, by appending time-constrained emission or reception events generated according to conditions 5.2.a) or 5.3.a), in a proper temporal ordering. This ordering is determined by computing the earliest deadline $D$ among all emission deadlines, for lifelines that are not in a quiescent state, and delivery deadlines, for messages in transit. When working with tc-traces, $D$ is in fact a constraint on the time instants of the next event and previous events. For each candidate time-constrained event $\langle e, c'\rangle$ to append to $\langle u, c\rangle$, if the conjunction $c \wedge c' \wedge D$ is satisfiable, then the event is selected, generating the tc-trace $\langle u \curvearrowright [e], c \wedge c' \wedge D\rangle$.
- $\mathcal{T}_\iota^*(\langle [], true\rangle)$ denotes the set of tc-traces reachable from the empty tc-trace $\langle [], true\rangle$ by 0 or more applications of $\mathcal{T}_\iota$ (reflexive transitive closure);
- $Q_\iota(\langle u, c\rangle)$ denotes the condition (on the time variables of events in $u$) upon which the system may remain quiescent after the occurrence of $\langle u, c\rangle$, as set by conditions 5.2.b) and 5.3.b). If $Q_\iota(\langle u, c\rangle)$ is satisfiable, $\langle u, c\rangle$ is added to the result, further restricted by $Q_\iota(\langle u, c\rangle)$.

*Theorem 5.1 (Correctness of Procedure 5.1):* Procedure 5.1 correctly computes $\mathcal{S}'(\iota)$.

*Proof Sketch:* Conditions 5.2.a) and 5.3.a) are satisfied for any tc-trace in the generated execution tree, because they trivially hold for the initial empty state, and are explicitly considered in the transition function $\mathcal{T}_\iota$ that generates next states. Conditions 5.2.b) and 5.3.b) are also guaranteed, because they are explicitly considered in the quiescence condition $Q_\iota$ used to select tc-traces to include in $\mathcal{S}'(\iota)$. Condition 5.2.c) is also satisfied for any tc-trace in the generated execution tree, because it trivially holds for the initial empty state, and the temporal ordering constraint ($D$) considered in $\mathcal{T}_\iota$ guarantees that a message is not delivered in a timing after the expiration of any existent emission deadline of the target lifeline. The temporal ordering also guarantees that a quiescent state is reachable from any execution state generated (i.e., infeasible states are not generated). Procedure 5.1 is also complete, in the sense that it generates all feasible tc-traces that satisfy Definitions 5.2 and 5.3, due to the fact that all candidate events are considered in $\mathcal{T}_\iota$.

An example of an execution tree and possible quiescent tc-traces generated by the application of Procedure 5.1 is shown in Fig. 8.

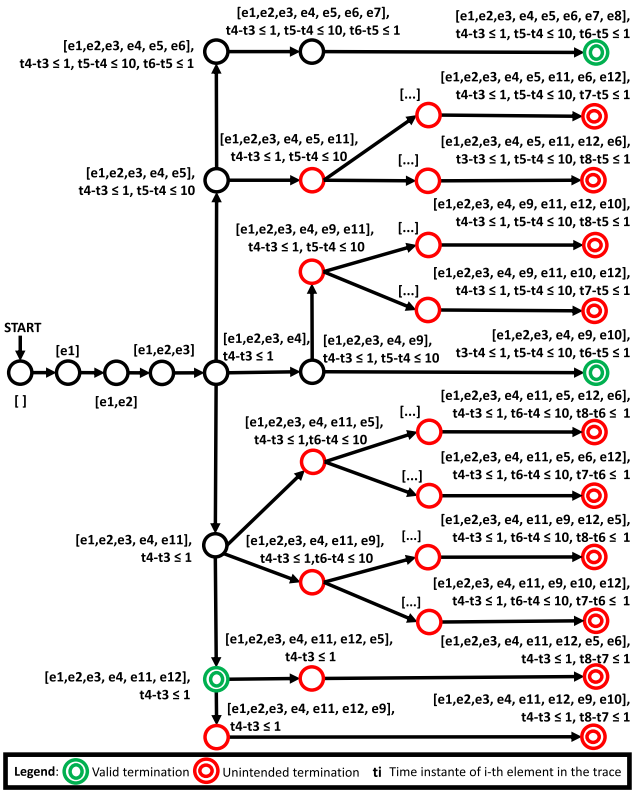



**FIGURE 8. Example of symbolic execution for the SD of Fig. 1 without the "{13..}" time constraint.**

## VI. LOCAL OBSERVABILITY AND CONTROLLABILITY ENFORCEMENT

As illustrated by the examples in Section II, many observability and controllability problems can be solved by the addition of coordination messages or coordination time constraints. Hence, in this section, we present algorithms to search for coordination messages or coordination time constraints to enforce local observability and/or local controllability of an interaction $\iota$, whilst preserving the traces valid locally at each lifeline (apart possibly from timing constraints).

To guide the search, we use the following heuristic: the locations of local observability or controllability problems (locations where the locally uncheckable or unintended traces deviate from valid traces) suggest points where coordination messages or time constraints need to be inserted.

Therefore, our main algorithm (Procedure 6.1) comprises four main steps, starting by determining the error locations.

*Procedure 6.1 (Local Observability and Controllability Enforcement):*

**Input:** test scenario described by a time-constrained interaction $\iota$ with local observability or controllability problems.

**Output:** a failure indication or a set of coordination messages and/or time constraints to enforce local observability and/or controllability.

1) Determine error locations (where the locally uncheckable or unintended traces deviate from valid traces);
2) Generate candidate coordination messages;
3) Generate candidate coordination time constraints;
4) Apply and evaluate candidate fixes (coordination messages or time constraints).

In the next subsections, we describe each of these steps.

### A. DETERMINATION OF ERROR LOCATIONS

*Procedure 6.2 (Determination of Error Locations):*

**Input:** test scenario described by a time-constrained interaction $\iota$.

**Output:** set $\mathcal{E}$ of missing or erroneous events in the unintended and/or locally uncheckable tc-traces of $\iota$; set $\mathcal{S}$ of lifeline locations in $\iota$ where the events in $\mathcal{E}$ occur (error locations).

1) Determine the set $\mathcal{V}$ of valid tc-traces defined by $\iota$;
2) Determine the set $\mathcal{U}$ of unintended and/or locally uncheckable tc-traces of $\iota$ (problematic traces);
3) Determine the set $\mathcal{E}$ of missing or erroneous events in the traces in $\mathcal{U}$, doing as follows for each trace $t \in \mathcal{U}$:
   a) if $t$ is a valid partial trace (i.e., $\exists v \in \mathcal{V} \cdot u \in prefixes(v)$), select all the valid next events, formally $\{e | t \frown [e] \in prefixes(\mathcal{V})\}$ (missing events);
   b) otherwise, select the first event $e$ in $t$ such that the prefix of $t$ up to $e$ is not a valid partial trace (erroneous event);
4) Determine the set $\mathcal{S}$ of lifeline locations in $\iota$ where the events in $\mathcal{E}$ occur (error locations).

### B. GENERATION OF COORDINATION MESSAGES

*Procedure 6.3 (Generation of Coordination Messages):*

**Input:** test scenario described by a time-constrained interaction $\iota$; set $\mathcal{S}$ of error locations computed by Procedure 6.2.

**Output:** sorted set of candidate coordination messages.

1) Determine a set $\mathcal{C}$ of candidate coordination messages, according to the following criteria:
   - they can start in any location in any lifeline (before/after any event or boundary);
   - they cannot cross boundaries of interaction operands, and, inside an interaction operand, can only be exchanged between participating lifelines;
   - they can terminate in any lifeline, different from the start lifeline, in the earliest possible location;
2) Rank the candidate messages in $\mathcal{C}$ based on their proximity to the error locations in $\mathcal{S}$, in order to obtain a sorted set $\mathcal{C}'$ of candidates.
3) Filter out candidates below a certain ranking threshold (e.g., to exclude candidates that do not touch any suspicious lifeline).

### C. GENERATION OF COORDINATION TIME CONSTRAINTS

As illustrated in Section II, several controllability problems (such as race conditions and inter-lifeline event ordering constraints) may be solved by adding coordination time constraints that impose an ordering between pairs of events. In fact, lifelines may coordinate their actions by dynamically exchanging coordination messages or by statically 'agreeing' on an adequate timing for their actions.

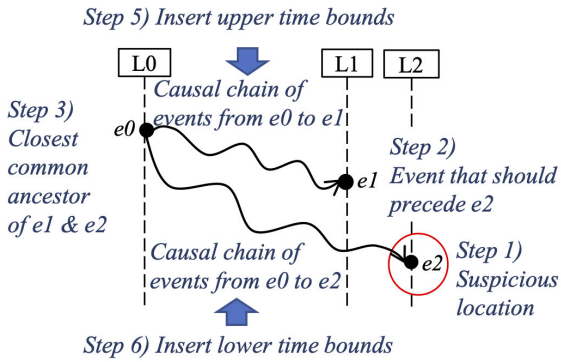
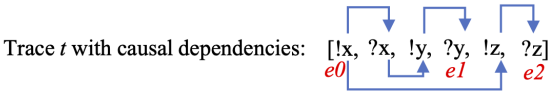



**FIGURE 9.** Fixing race conditions with coordination time constraints.

The pattern of race conditions that our heuristic algorithm looks for and the fix strategy used are illustrated in Fig. 9.




A. backward-slice(t, e1) = [!x, ?x, !y, ?y]
B. backward-slice(t, e2) = [!x, !z, ?z]
C. closest-common-ancestor(t, e1, e2) = last(intersect(A, B)) = !x
D. forward-slice(t, e0) = [!x, ?x, !y, ?y, !z, ?z]
E. bidirectional-slice(t, e0, e1) = intersect(A, D) = [!x, ?x, !y, ?y]
F. bidirectional-slice(t, e0, e2) = intersect(B, D) = [!x, !z, ?z]

**FIGURE 10.** Causal dependencies and slicing operations (trace of Figure 3.a).

We use several trace slicing operations illustrated in Fig. 10, based on the causal dependencies that exist between pairs of emission and reception events, and between all the events that precede an emission event in a lifeline and the

emission event itself (assuming the emission decision is taken based on the events previously observed in the lifeline).

*Procedure 6.4 (Generation of Coordination Time Constraints to Fix Race Conditions):*

**Input:** test scenario described by a time-constrained interaction $\iota$; set $\mathcal{E}$ of missing or erroneous events computed by Procedure 6.2.

**Output:** set of candidate fixes found, where each candidate fix is a set of time constraints to enforce the ordering between a particular pair of events.

1) Take as candidate instances for $e2$ the events in $\mathcal{E}$.
2) Take as candidate instances for $e1$ the events that immediately precede $e2$ (without intermediate events from the respective lifelines) in at least one valid trace $t \in \mathcal{V}$, and do not occur after $e2$ in any valid trace.
3) Take as candidate instances for $e0$ the closest common ancestors of $e1$ and $e2$ in the valid traces $t \in \mathcal{V}$ in which both occur (calculated as illustrated in Fig. 10).
4) Discard triples $(e0, e1, e2)$ where $e0 = e1$ or the maximum duration from $e0$ to $e1$ is less than the minimum duration from $e0$ to $e2$ (cases where $e1$ is guaranteed to precede $e2$).
5) Inject upper time bounds between pairs of events in the causal chain of events from $e0$ to $e1$ (bidirectional slice), based on default values for the maximum transmission time (between emission and reception events) and maximum reaction time (between an event in a lifeline and a subsequent emission event).
6) Determine the maximum duration $\tau$ from $e0$ to $e1$ that results from step 5, and inject a lower time bound $\tau$+1 (wait time) in the chain of events from $e0$ to $e2$, between an event in a lifeline and a subsequent emission event (giving priority to emissions performed by actors as close as possible to $e0$). If a time bound cannot be injected, the triple $(e0, e1, e2)$ is discarded.
7) Return the set of candidate fixes found, where each candidate fix is a set of time constraints to enforce the ordering between a pair $(e1, e2)$ of events.

Regarding controllability problems caused by pairs of mutually exclusive emission and reception events simultaneously enabled in a lifeline, we use a similar fix strategy: we inject coordination time constraints that impose an ordering between those events, based on their physical vertical location in the sequence diagram (although such physical location does not have a semantic meaning inside `alt` fragments, it usually has an intuitive meaning for the user).

*Procedure 6.5 (Generation of Coordination Time Constraints to Fix Pairs of Mutually Exclusive Reception and Emission Events Simultaneously Enabled):*

**Input:** test scenario described by a time-constrained interaction $\iota$.

**Output:** set of candidate fixes found, where each candidate fix is a set of time constraints to enforce the ordering between a particular pair of events.

1) Find pairs of events $e1$ and $e2$ that: (i) occur in the same lifeline, with $e1$ located before $e2$; (ii) are of different types (send and receive); (iii) are mutually exclusive (i.e., there is no valid trace in which both occur); and (iv) may be simultaneously enabled (from the perspective of their lifeline).
2) Perform step 3 as in Procedure 6.4, with the difference that distinct traces $t1$ and $t2$ have to be considered for $e1$ and $e2$, instead of a common trace $t$.
3) Perform steps 4, 5, 6 and 7 as in Procedure 6.4.

### D. APPLICATION AND EVALUATION OF CANDIDATE FIXES

*Procedure 6.6 (Application and Evaluation of Candidate Fixes):*

**Input**: test scenario described by a time-constrained interaction $\iota$; set $\mathcal{F}$ of candidate fixes, where each candidate fix is a single coordination message or a set of coordination time constraints, as computed by Procedures 6.3, 6.4 and 6.5.

**Output**: a failure indication or a set of coordination messages and/or time constraints to enforce local controllability and/or observability.

1) Search for single fix solutions, doing as follows for each candidate fix $f$ (message or constraint-set) in $\mathcal{F}$:
   a) apply the fix $f$ (i.e., insert the message or constraint-set in $\iota$), obtaining a new interaction $\iota'$;
   b) determine the set $\mathcal{V}'$ of valid traces defined by $\iota'$;
   c) if the projections of $\mathcal{V}$ and $\mathcal{V}'$ onto the lifelines of $\iota$ do not coincide (apart from coordination events and time constraints), discard $f$;
   d) if the set $\mathcal{U}'$ of unintended and/or locally uncheckable traces of $\iota'$ is empty, return $f$;
   e) otherwise, if $\#\mathcal{U}'$ (with coordination events removed) is not smaller than $\#\mathcal{U}$, discard $f$;
2) If a single fix solution was not found, search for multiple fix solutions using a greedy heuristic as follows:
   a) pick the candidate fixes in $\mathcal{F}$ that were not discarded, and rank them by increasing values of $\#\mathcal{U}'$ (with coordination events removed), obtaining a new ordered set $\mathcal{F}'$ of candidate fixes;
   b) for each candidate fix $f \in \mathcal{F}'$, by the defined order, insert $f$ onto $\iota$ and execute recursively Procedure 6.6; if a solution is found, return the inserted messages and/or time constraints.
3) If no single or multiple fix solution was found, fail.

Because of being based on several heuristics, the presented algorithm has several limitations. Although it was able to find a solution of minimum size in a few seconds or tenths of a second in all test cases and case studies we experimented with, it might be unable to find a solution when a solution exists, or might produce a solution more complex than needed.

## VII. IMPLEMENTATION

The algorithms described in this article were implemented in the DCO Analyzer tool [13]. DCO Analyzer is an application developed in Java [14] and VDM++ [15] to analyze

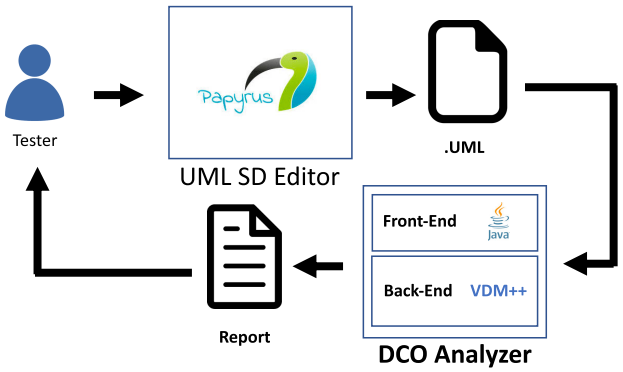



**FIGURE 11.** DCO analyzer overview.

UML SDs representing distributed systems test scenarios. As depicted in Fig. 11, the user can use any visual editor of UML SDs (e.g. Papyrus[1]) and then upload the created diagrams to DCO Analyzer.

Internally, DCO Analyzer comprises a front-end, developed in Java, and a back-end, developed in VDM++. The front-end is responsible for receiving and parsing.uml files describing UML SDs, verifying their conformance with the UML metamodel [16], and converting them into the formal representation expected by the back-end (VDM++ data structures). It is also possible to directly provide a.vdmpp file containing the VDM++ data structures; this may be useful to overcome limitations of modeling tools.

The DCO Analyzer back-end is capable of analyzing the following properties:

- **Valid Traces:** Set of valid global traces defined by the given SD;
- **Unintended Traces:** Set of invalid global traces caused by locally valid decisions (representing violations of local controllability);
- **Locally Uncheckable Traces:** Set of invalid global traces that cannot be verified locally (representing violations of local observability);
- **Local Controllability:** The diagram is locally controllable if there are no unintended traces;
- **Local Observability:** The diagram is locally observable if there are no locally uncheckable traces;
- **Coordination Messages and Coordination Time Constraints:** Set of coordination messages and/or time constraints to enforce local controllability and/or local observability.

In the back-end, we implemented all the algorithms and auxiliary operations needed for local observability and controllability analysis and enforcement in the VDM++ formal specification language [15]. Specifications in VDM++ can be directly executed with the Overture tool and translated to Java code ready for execution and integration with other code.

[1]https://papyrusuml.wordpress.com

To test the implemented algorithms regarding correctness and performance, we used several test scenarios coming from a nation-wide project in the ambient-assisted living (AAL) domain, plus additional test scenarios to maximize coverage. In total, 30 test scenarios (test cases) were defined, covering a variety of causes for locally observability and controllability (see Section II). The complete test suite ran in approximately 10 seconds in an Intel Core i7 machine running Windows 10 Professional at 2.20GHz with 16GB RAM.

Fig. 12 shows an example of an input test scenario, drawn with the Papyrus tool, for an online driving license renewal system (greatly simplified for illustration purposes). Fig. 13 shows the output produced by DCO Analyzer for this scenario when all the analysis options are selected.

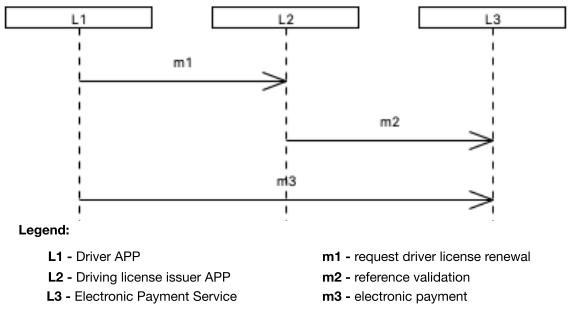



**FIGURE 12.** Initial SD in Papyrus.

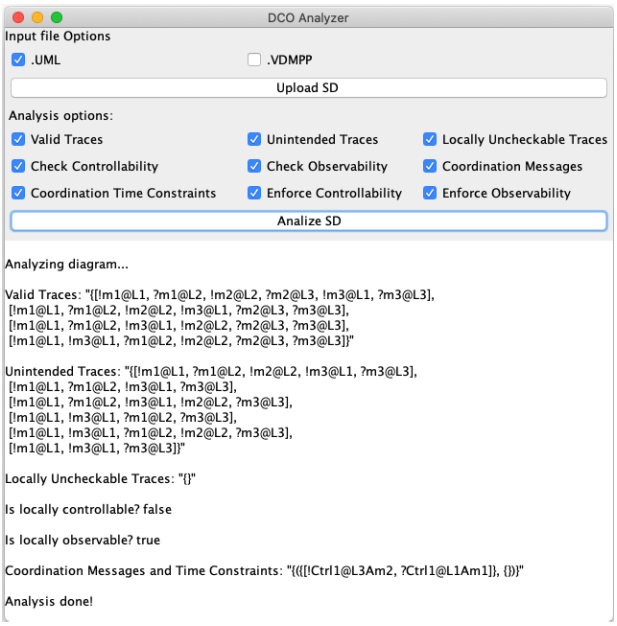



**FIGURE 13.** DCO Analyzer output.

In the output, a set of traces is represented between {...}, a trace (sequence of events) is represented between [...], the emission of a message $m$ by a lifeline $L$ is represented as $!m@L$, and the reception of a message $m$ at a lifeline $L$ is represented as $?m@L$.

In this example, our tool was able to detect that the given diagram is not locally controllable, indicating six unintended traces. These unintended traces are related to the possibility

of the electronic payment message ($m3$) being received by the electronic payment service ($L3$) before the reference validation message ($m2$).

In order to help the user to make this diagram locally controllable, our tool suggests adding a coordination message ($Ctrl1$) between the Electronic Payment Service ($L3$) and the Driver APP ($L1$), after $m2$ (suffix "Am2") and $m1$ (suffix "Am1"), respectively. In practice, such message might represent a payment authorization confirmation message, thereby ensuring that payment can only be made after the payment reference has been validated.

With this suggestion, the user can then refine the SD as shown in Fig. 14.

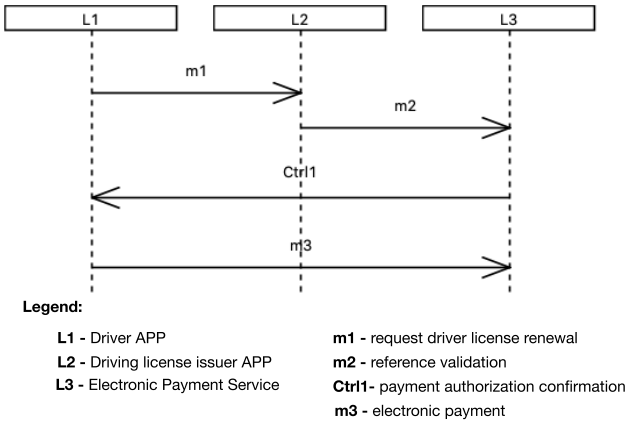



**FIGURE 14.** Refined SD in Papyrus.

The suggestion given by our tool can be used in several ways:

- the suggested message is actually implemented in the SUT, so the SD is just modified to include it (*incomplete specification*);
- the SUT is redesigned to incorporate the suggested message, and the SD is updated accordingly (*design flaw*);
- the system design is not changed, so the suggested message is marked as a *test coordination message* to be exchanged between the test components during test execution (e.g., between a test monitor co-located with $L3$ and a test driver co-located with $L1$).

Another example, illustrating a local observability problem, is shown in Fig. 15. This diagram represents the login scenario of a mobile application, where the user, after login, can receive pending notifications since the last time the application was connected to the server. By analyzing the diagram with our tool it is possible to determine that local testers are unable to locally detect the execution trace $[!m1@L1, ?m1@L2, !m2@L2]$, which corresponds to the case where the message $m2$ is sent but lost. Such loss will not be detected as an error at $L1$ because not receiving $m2$ is also a valid behavior at $L1$. The solution to this problem recommended by the DCO Analyzer is to place a coordination message between $L1$ and $L2$ upon receipt of $m2$ in $L1$.

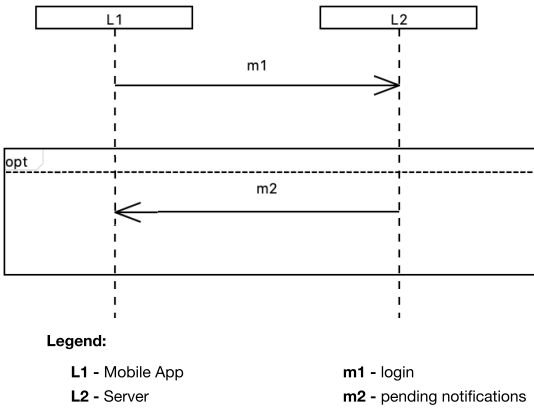



**FIGURE 15.** Example of a scenario not locally observable.

Such message can be interpreted as an acknowledgment message; if $m2$ is lost (or the acknowledgment message is lost), then a problem will be detected at $L2$.

More complex SDs are also supported, namely SDs with other control flow variants (`alt` and `loop` combined fragments) and time constraints.

DCO Analyzer executable files, algorithms (described in previous sections) implemented in VDM, and some test scenarios in UML can be found at https://brunolima.info/DCOANALYZER/.

## VIII. CASE STUDY

In order to validate the algorithms in industrial scenarios we conducted an evaluation experiment with real-world test scenarios from an industrial partner who is currently developing a solution for automatic incident detection on motorways. The goals of the evaluation are:

1) to check if our analysis tool is able to correctly identify local controllability and/or local observability issues in real-world test scenarios;
2) to check if the analysis is performed in an adequate time;
3) to check if the output results produced by the tool help the users to understand the root causes of the detected problems and refine the input test scenarios accordingly.

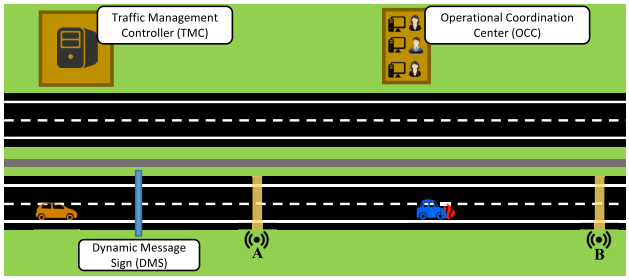



**FIGURE 16.** Traffic control system.

### A. MOTORWAY INCIDENT DETECTION PROJECT

The project of our industrial partner (here described in a simplified way for privacy reasons), illustrated in Fig. 16,

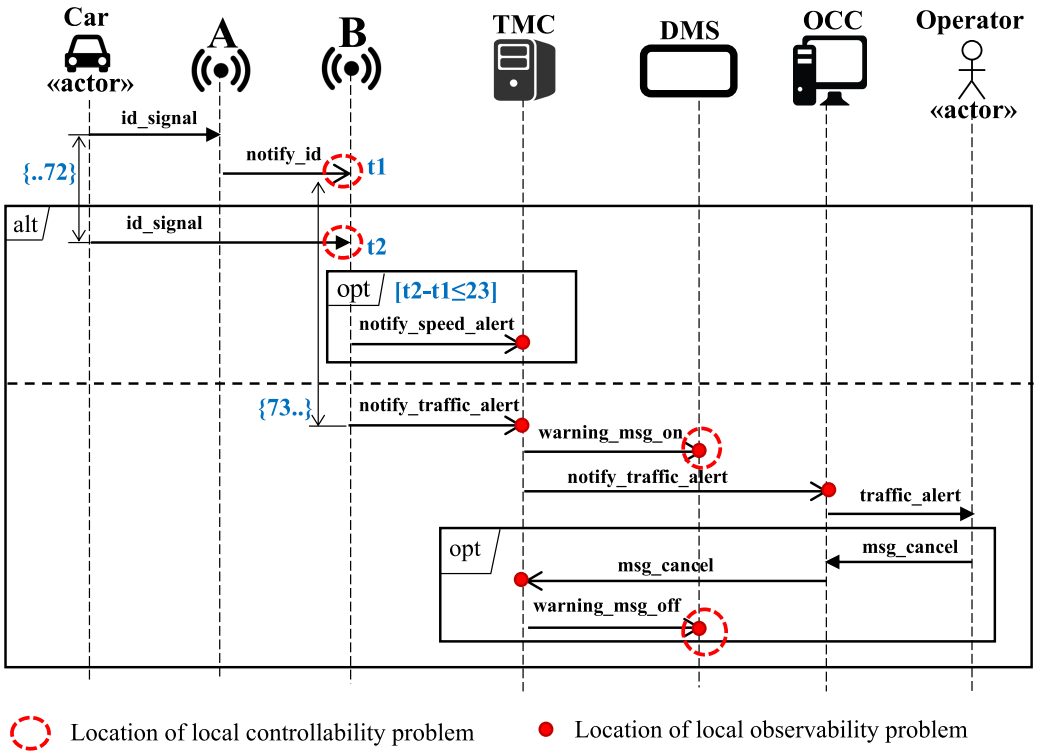



**FIGURE 17.** Initial scenario and problem locations.

consists of the placement of sensors on the motorways that interact with each other and are able (among other features) to detect incidents automatically.

When the system detects a possible incident, a message is automatically presented to the drivers through the Dynamic Message Sign (DMS), so that they can reduce the speed and thus reduce the possibility of a chain collision. On the other hand, the system also automatically informs the Operational Coordination Center (OCC) operators so that they can validate the occurrence and trigger the help assistance if necessary.

### B. TEST SCENARIO

We asked our partner to describe the system interactions (including temporal constraints) using UML SDs. Fig. 17 shows one of the scenarios that was provided.

The scenario involves 3 alternatives. In the first case, a vehicle circulating on the motorway is detected by sensors A and B, situated 1 km apart, in a time interval between 24 s and 72 s (indicating that the vehicle circulates at a speed between 50 and 150 km h$^{-1}$). In this case, the system does not need to take any action. In the second case, the vehicle is detected by the sensors A and B in a time interval less then 23 s, which corresponds to a speed above 150 km h$^{-1}$. In this case, the system sends a speed alert to the Traffic Management Controller (TMC). In the last case, a vehicle is detected by sensor A but is not detected by sensor B in the next 72 s, meaning that something may have occurred with the vehicle and it may be immobilized on the road. In this case, the system informs the TMC that automatically sends a message to be presented to the other drivers through the DMS and informs the OCC. In the OCC the operator visualizes the alert and can optionally cancel the alert which is done through the TMC that removes the message from the DMS.

### C. SCENARIO ANALYSIS - LOCAL CONTROLLABILITY

We analyzed the local controllability of the previous test scenario (Fig. 17) with our tool, which took 1.1 s to run in the machine previously described and reported 3 unintended tc-traces (with lifeline indicated only when needed to disambiguate):

1) [!*id_signal*, ?*id_signal*@*A*, !*notify_id*, !*id_signal*, ?*id_signal*@*B*, ?*notify_id*, ...], with $\tau_4 - \tau_1 \leq 72$;
2) [!*id_signal*, ?*id_signal*@*A*, !*id_signal*, ?*id_signal*@*B*, !*notify_id*, ?*notify_id*, ...], with $\tau_3 - \tau_1 \leq 72$;
3) [!*id_signal*, ?*id_signal*@*A*, !*notify_id*, ?*notify_id*, !*notify_traffic_alert*, ..., ?*warning_msg_off*, ?*warning_msg_on*], with $\tau_5 - \tau_1 \geq 73$.

These 3 tc-traces correspond to the following 2 problems, both related with race conditions:

1) Unexpected reception of *id_signal* at sensor B before reception of *notify_id* (unintended traces 1 and 2). As delays can occur in the transmission of the *notify_id* message between sensor A and sensor B, the message *notify_id* may arrive at sensor B before the message *id_signal*. As a consequence, the system may be unable

to use the sensor data or may incorrectly conclude that a vehicle is moving against the flow of traffic; this suggests a design flaw or an incomplete specification.
2) Unexpected late reception of *warning_msg_on* at DMS after reception of *warning_msg_off* (unintended trace 3). As delays can occur in the transmission of the *warning_msg_on* message between TMC and DMS, the message *warning_message_off* may arrive at DMS before the message *warning_message_on*. This case shows that in some situations an alert message can remain visible in the DMS even after it has been removed by the operator, thereby transmitting erroneous information to the drivers.

### D. SCENARIO ANALYSIS - LOCAL OBSERVABILITY

We also analyzed the local observability of the previous test scenario with our tool, which took 1.3 s to run in the machine previously described and reported 22 locally uncheckable tc-traces:

1) [!*id_signal*, ?*id_signal*@*A*, !*notify_id*, ?*notify_id*, !*id_signal*, ?*id_signal*@*B*, !*notify_speed_alert*], with $\tau_5 - \tau_1 \leq 72 \wedge \tau_6 - \tau_4 \leq 23$ (message *notify_speed_alert* lost);
...
22) [!*id_signal*, ?*id_signal*@*A*, !*notify_id*, ?*notify_id*, !*notify_traffic_alert*, ..., !*warning_msg_off*], with $\tau_5 - \tau_4 \geq 73$ (message *warning_message_off* lost).

After close inspection, we conclude that all the uncheckable tc-traces are due to the presence of the following 6 optional asynchronous messages without corresponding acknowledgment messages:

- *notify_speed_alert*;
- *notify_traffic_alert* from *SensorB* to *TMC*;
- *warning_message_on*;
- *notify_traffic_alert* from *TMC* to *OCC*;
- *message_cancel*;
- *warning_message_off*.

As explained in Section II, if any of these messages is lost, the problem will go undetected by the target lifeline, because not receiving a message is also a locally valid behavior. The solution recommended by our tool to enforce local observability consists of the addition of 6 corresponding acknowledgment (coordination) messages.

However, in discussion with our partner, considering the solution architecture and technologies, the possibility of such messages being lost was deemed negligible, and the insertion of acknowledgment messages was not considered a priority, so we focused only on fixing the local controllability issues as explained in the next section.

### E. SCENARIO REFINEMENT

In discussion with our industrial partner, we concluded that a maximum delay of 1 s could be assumed for all internal actions in the system (message emission after some observed events, and message transmission between lifelines). Hence, we ran our tool again asking for recommendations of coordination time constraints and/or coordination messages to enforce controllability, using the 1s upper bound for system transmission and reaction time where needed (these bound are currently configured in a configuration file).

The tool recommended the addition of 3 upper time bounds and 2 lower time bounds as indicated in red in Fig. 18, solving both controllability problems. The analysis took 1.8 s to run in the machine previously described.

Our partner accepted the suggestions, but opted to further refine the test scenario as indicated by the solid arrows in Fig. 18. Considering that a maximum car speed of 450 km $h^{-1}$ could be safely assumed, the minimum time for a car to travel between sensors A and B was changed from 3 to 8 s. Our partner also decided to redesign the operator user interface, so that traffic alert messages can only be canceled after 5 s; hence, the minimum operator response time was changed from 2 to 5 s.

Other test scenarios from the same project were also analyzed and refined successfully using the same procedure.

Those scenarios are related to other traffic anomalies that can be detected and notified using the same road infrastructure (see Fig. 16), namely:

- cars that reverse direction after passing the first sensor (A), causing the sensor activation sequence A-A;
- cars that move against the flow of traffic, causing an activation of sensor B without a prior activation of sensor A.

Those scenarios differ from the scenario in Fig. 17 in the initial sensor activation sequence, but share a similar traffic alert notification sequence, and present similar types of observability and controllability problems.

### F. DISCUSSION

Regarding the goals of the experiment, we conclude that:

1) our tool was able to correctly identify relevant local controllability issues in real-world test scenarios, including issues that escaped manual inspection;
2) the analysis was performed quickly by the tool (in a few seconds);
3) the outputs produced by the tool helped in understanding and fixing the root causes of the detected problems (in this case, incomplete specifications or system design flaws).

### G. THREATS TO VALIDITY

Our experiments have several validity threats. First, our validation examples may not cover all possible real-world scenarios. In order to reduce this possibility, in addition to the scenarios provided by our industrial partner, we also tested fictitious scenarios with all UML combined fragments. Second, the manual interpretation of the error messages produced by our solution can only mean that people with some experience in modeling can understand the errors in more complex scenarios. In order to better understand this phenomenon we asked our industrial partner to analyze the results produced by

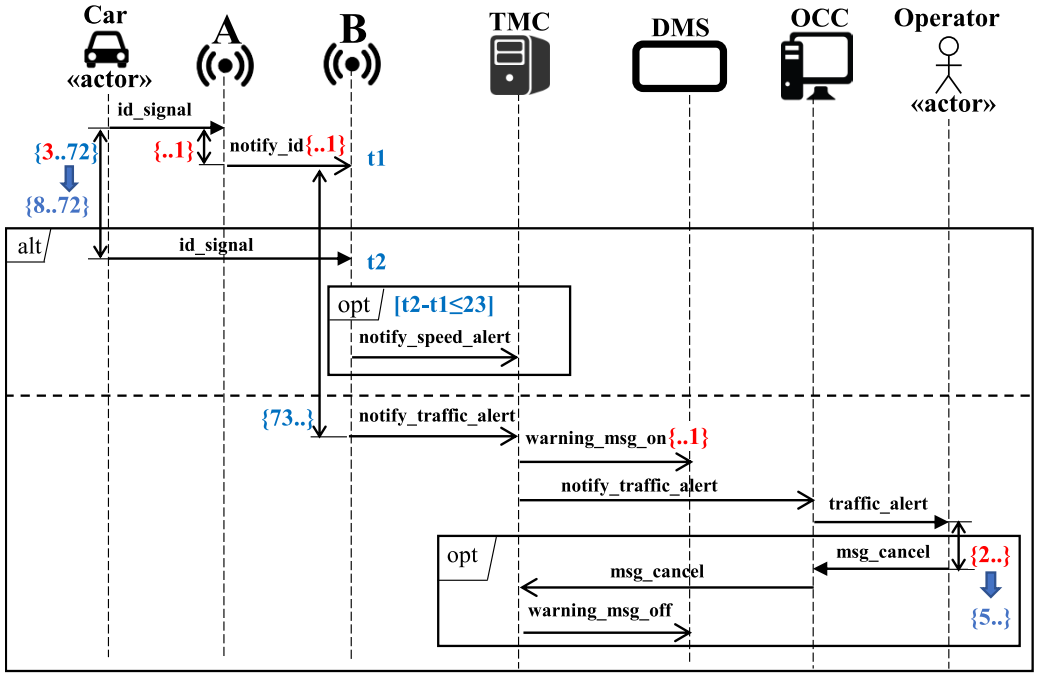



**FIGURE 18.** Refined locally controllable scenario (automatic refinement in red, followed by manual refinement indicated with solid arrows).

our tool; this analysis was performed by people with different modeling experiences. The results showed that although there is a better perception from more experienced professionals, the less experienced ones can also understand the problems that have been detected.

## IX. RELATED WORK

### A. MODEL-BASED TESTING

Model-based testing (MBT) techniques and tools promote the effectiveness and efficiency of the test process, by means of the automatic generation of executable test cases from behavioral models of the system under test (SUT) [17].

MBT can be performed offline, with separate test generation and execution phases [18], or online, with intermixed phases [17], [19], [20]. The latter is the preferred approach 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 [21].

Regarding the input models, one can distinguish state-based approaches, in which UML state machines [22] or similar models [23], [24] are used for describing all possible behaviors of the SUT or its components, and scenario-based approaches, in which UML SDs [25], message sequence charts (MSC) [26] or similar models [27] are used for describing interactions between the system components or with the environment in key scenarios, minimizing test case explosion [28].

However, few works address the challenges of MBT for distributed systems, and the works found are mostly focused on system testing and not integration testing.

### B. OBSERVABILITY AND CONTROLLABILITY IN DISTRIBUTED SYSTEMS TESTING

One difficulty in distributed systems testing is observability, because communication delays and the lack of a global clock limit the conformance faults detectable. Three test architectures have been proposed, with different conformance relations and fault detection capabilities: a purely distributed test architecture with independent local testers communicating synchronously with the SUT components [29]; a purely centralized test architecture, in which a single central tester interacts asynchronously with the SUT components [30]; a hybrid test architecture that combines local testers and a central tester to achieve a higher fault detection capability [30].

Under the hybrid approach of [30], the central tester is responsible for deciding and sending test inputs to the SUT components, and local testers are responsible for observing the events (inputs and outputs) at each location; the SUT outputs are observed by the local testers and sent to the central tester. This way, the local testers are able to detect conformance faults associated with an incorrect combination or an incorrect ordering of events occurring in the same location, whilst the central tester is able to detect conformance faults associated with an incorrect combination of events or an incorrect ordering of pairs of input and output events occurring at different locations (e.g., an SUT output that is prematurely produced at one location before an input is injected at another location). In our approach, we further decentralize test input generation and injection, minimizing the messages exchanged between the test components during test execution and increasing the responsiveness of the test harness, whilst

keeping the same fault detection capability. Another difference in our work is that we check not only the interactions with the environment (system testing perspective), as in those works, but also the interactions between the system components (integration testing perspective), as well as timing constraints.

Another difficulty in distributed systems testing is controllability, i.e., the difficulty for the local testers to decide when and what test inputs to inject, without causing global conformance faults (e.g., in the presence of race conditions or non-local choices). Solutions proposed in the literature are based on the insertion of coordination messages between test components [8], [12], [31], but they do not handle timing constraints and, in most of the cases, they address only the ''when'' and not the ''what'' aspect (i.e., they don't consider control flow variants).

In [8], the author discusses the problems related to race conditions in scenarios described through MSCs or UML SDs, and presents solutions to these problems. The focus of their work is on analyzing scenario-based requirements specifications, but such scenarios can also be used for testing purposes. However, only basic scenarios are considered, without control flow variants and timing constraints.

In [12], the author investigates the use of coordination messages to overcome controllability problems when testing from an input/output transition system (IOTS) and give an algorithm for introducing sufficient messages. The algorithm operates by identifying all of the controllability problems, and then resolving these one at a time. The author also characterizes the types of controllability problems that cannot be solved this way, and introduces the notion of *strongly uncontrollable* test cases. The author also proves that the problem of minimizing the number of coordination messages used is NP-hard. However, the approach is focused on system testing only and not integration testing, i.e., the messages exchanged between the system components are not considered (the observation of these messages by the local testers may reduce the need for introducing coordination messages). Other differences with our work are that they do not consider timing constraints, and assume that test inputs are deterministic (which we do not require).

In [31], the authors 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 (based on timeout events) in each test component. However, in their work, only the interactions with the environment are modeled, and they do not consider control flow variants and time constraints.

A common limitation of the above works (except [8]) is that they only consider the messages exchanged with the environment (system testing perspective), represented by a single input or output event, and not the messages exchanged between the system components (integration testing perspective), that need to be represented by pairs of send and receive events.

More recently, observability and controllability in the context of integration testing of distributed systems based on UML SDs were analyzed in [10]. In order to be able to check not only the interactions with the environment but also the interactions between the system components during integration testing, local testers are deployed close to the system components, coordinated by a central tester. They introduce the notions of local observability and local controllability, and present procedures to check if a given test scenario (represented by a UML SD) is locally observable and/or locally controllable. However, they did not take time constraints into consideration and do not provide procedures to enforce local observability and local controllability, as we do here. The handling of time constraints greatly complicates the analysis and enforcement procedures, because of the need to work with time-constrained traces instead of plain (untimed) traces. Even if not specified explicitly in the provided test scenarios, time constraints play an important role for test coordination in distributed testing, as shown in this article.

### C. OTHER TESTABILITY ISSUES IN DISTRIBUTED SYSTEMS TESTING

When testing a distributed system, it is sometimes necessary to test a running/deployed system (*runtime validation*), the additional challenge being that testing should not interfere with system use. In runtime validation, a component of a system is said to be *testable* if it has a separate test interface whose use reduces the potential for interference. *Isolation methods* have been proposed for components that are not testable. There is a line of work in which approaches to runtime validation have been developed using Testing and Test Control Notation Language Version 3 (TTCN-3) in order to enhance applicability [32]. The proposed approach (TT4RT) includes a test management layer and a test isolation layer. A further development aimed to optimise the placement of test components that interact with system components, with this taking into account resource availability and network connectivity [33]. It has also been noted that a system might have some components that are testable and some that are not, with a procedure being proposed to choose the appropriate test isolation approaches [34]. The focus of this line of work is on the execution of abstract test cases that have already been provided, and it does not address coordination issues. However, there is potential for runtime validation approaches to be integrated with techniques, such as those described in this article, that analyse test case and address coordination problems where they exist.

### D. TIME CONSTRAINTS IN DISTRIBUTED SYSTEMS TESTING

The temporal dimension is addressed in several works, but very few refer to distributed systems testing.

**TABLE 1. Summary of comparison of related works in distributed systems testing and analysis and our work (*).**

| | [29] | [30] | [8] | [12] | [31] | [10] | [32]-[34] | [35] | [36] | [37] | [38] | [39] | [40] | [41] | [42] | * |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Scenario-based input model | - | - | X | - | X | X | X | X | X | X | - | - | X | - | - | X |
| Distributed test architecture | X | X | - | X | X | X | X | - | - | - | X | X | X | X | X | X |
| Internal interaction checking | X | - | X | - | - | X | X | X | X | X | - | X | X | - | - | X |
| Control-flow variants | X | X | - | X | - | X | X | X | X | X | X | X | X | X | X | X |
| Timing constraints | - | - | - | - | - | - | X | X | X | X | - | X | X | - | - | X |
| Non-determinism | - | X | - | X | - | X | X | X | X | X | X | X | X | - | - | X |
| Conformance checking | - | - | - | X | - | X | - | - | - | X | X | X | X | - | - | - |
| Observability analysis | - | - | - | - | - | X | - | - | - | - | - | - | - | X | X | X |
| Controllability analysis | - | - | X | X | X | X | - | - | - | - | - | - | - | X | X | X |
| Observability enforcement | - | - | - | - | - | - | - | - | - | - | - | - | - | X | X | X |
| Controllability enforcement | - | - | X | X | X | - | - | - | - | - | - | - | - | X | X | X |
| Tool implementation | - | - | - | - | - | - | X | - | - | - | - | - | - | - | - | X |
| Real-world case studies | - | - | X | - | - | - | X | - | - | - | - | - | - | - | - | X |

In [35], the authors derive the valid traces for Timed Message Sequence Charts (T-MSCs), similar to UML SDs, but do not address the problem of conformance checking based on distributed observations. Timed traces are represented by incorporating special time events between normal events.

In [36], the authors present a timed model of communicating finite-state machines, which communicate by exchanging messages through channels and use event clocks to generate collections of T-MSCs. In a more recent work [37], the authors address model checking message-passing systems with real-time requirements. As behavioral specifications, they use TC-MSCs (time-constrained MSCs), in which lower and upper bounds on the time interval between certain pairs of events are added to plain MSCs. As system model, they use a network of communicating finite state machines with local clocks, whose global behavior can be regarded as a timed automaton. Their goal is to verify (by model checking) that all timed behaviors exhibited by the system model conform to the timing constraints imposed by the specification, and not to check the conformity of the implementation with the specification or system model.

In [38], the authors derive conformance relations taking into account the event timestamps obtained with the local clocks present at each system port (point of interaction with the environment), assumed to differ up to a maximum clock skew, but only for system testing.

In [39], the authors show that conformance checking in the presence of time constraints, within a distributed test architecture without a global clock, can be done in two phases: in the first phase, each local tester checks local conformance according to the *tioco* conformance relation; in the second phase, the local traces are brought together and it is checked if events are exchanged following some communication rules. Their results do not apply directly to UML SDs [7], since they assume internal multicast communications, among other differences.

In [40], the authors present criteria and decision procedures to check the conformance of observed execution traces (based on distributed observations) against the specification, in the context of integration testing of distributed systems based on UML SDs enriched with time constraints.

However, none of the above works address the observability and controllability properties, as we do in this article.

The only previous work we found that relates the issue of observability and controllability to time constraints is [41]. In this article, the author demonstrates how to solve the problems of observability and controllability using coordination messages and time constraints. However, they do not support timing constraints or non-determinism in the input models, only consider interactions with the environment, and restrict their attention to SUT behaviors consisting of alternating sequences of inputs from the environment and outputs to the environment. In a more recent work [42], the authors propose to solve controllability problems using so called synchronization messages, for the same type of input models, but do not support timing constraints either in the input model.

### E. SUMMARY

Table 1 summarizes the main characteristics and features covered by the related works previously analyzed, in comparison with our work. Although some works address observability and controllability problems in distributed systems testing and design, none addresses the problem of observability and controllability analysis and enforcement for time-constrained distributed systems, as we do in this article. We believe this is a key contribution to help solving the test coordination problem in distributed testing with time constraints.

## X. CONCLUSION AND FUTURE WORK

Given the growing importance of distributed systems testing, and the benefits of distributed conformance checking and test input selection in the scenario-based integration testing of distributed systems, particularly in the presence of time constraints and non-determinism, we presented in this article an approach to assess if test scenarios are ready for distributed execution, and, if not, refine them to become test ready with minimal overhead.

Our approach is based on the notions of local (or distributed) observability and controllability, that is, the ability to perform conformance checking (observability) and test input selection (controllability) in a purely distributed way, without exchanging coordination messages between the test

components during test execution or overlooking conformance faults or causing incorrect test inputs.

Local observability and controllability are checked in a constructive way (pinpointing violations) by analyzing the set of valid time-constrained traces defined by a time-constrained test scenario under consideration. Local observability is determined based on operators introduced in the paper for manipulating time-constrained traces (join, project and difference). Local controllability is determined based on a symbolic simulated execution algorithm. If needed, local observability and/or local controllability are enforced by the addition of coordination messages and/or coordination time constraints, that are determined based on heuristic search.

All the algorithms were implemented in the DCO Analyzer tool, for test scenarios specified by means of UML sequence diagrams. To validate the algorithms and the tool in an industrial setting, we conducted an evaluation experiment with real-world test scenarios from an industrial partner. In that experiment, our tool was able to correctly identify local observability and controllability issues and recommend possible fixes; the outputs reported helped the users to understand and fix the root causes of the detected problems.

As future work, we intend to integrate DCO Analyzer as a static analysis tool in a full-fledged toolset for model-based distributed systems testing, and conduct further experiments in industrial settings.

## REFERENCES

[1] B. Boehm, "Some future software engineering opportunities and challenges," in *The Future of Software Engineering*. Cham, Switzerland: Springer, 2011, pp. 1–32.

[2] G. Tassey, *The Economic Impacts of Inadequate Infrastructure for Software Testing*, vol. 7007, Nat. Inst. Standards Technol., RTI Project 011, 2002.

[3] F.-Z. Moutai, S. Hsaini, S. Azzouzi, and M. E. Hassan Charaf, "Testing distributed cloud: A case study," in *Proc. Int. Symp. Adv. Electr. Commun. Technol. (ISAECT)*, Nov. 2019, pp. 1–5.

[4] H. Kim, A. Ahmad, J. Hwang, H. Baqa, F. Le Gall, M. A. R. Ortega, and J. Song, "IoT-TaaS: Towards a prospective IoT testing framework," *IEEE Access*, vol. 6, pp. 15480–15493, 2018.

[5] J. Hwang, A. Aziz, N. Sung, A. Ahmad, F. Le Gall, and J. Song, "AUTOCON-IoT: Automated and scalable online conformance testing for IoT applications," *IEEE Access*, vol. 8, pp. 43111–43121, 2020.

[6] B. Lima and J. P. Faria, "Automated testing of distributed and heterogeneous systems based on uml sequence diagrams," in *Proc. 10th Int. Joint Conf. Softw. Technol. (ICSOFT)*. Cham, Switzerland: Springer, Jul. 2015, pp. 380–396.

[7] *OMG Unified Modeling Language TM (OMG UML) Version 2.5*, Object Management Group, Needham, MA, USA, 2015.

[8] B. Mitchell, "Resolving race conditions in asynchronous partial order scenarios," *IEEE Trans. Softw. Eng.*, vol. 31, no. 9, pp. 767–784, Sep. 2005.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[10] B. M. C. Lima and J. C. P. Faria, "Towards decentralized conformance checking in model-based testing of distributed systems," in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation Workshops (ICSTW)*, Mar. 2017, pp. 356–365.

[11] D. L. Mills, "Internet time synchronization: The network time protocol," *IEEE Trans. Commun.*, vol. 39, no. 10, pp. 1482–1493, Oct. 1991.

[12] R. M. Hierons, "Overcoming controllability problems in distributed testing from an input output transition system," *Distrib. Comput.*, vol. 25, no. 1, pp. 63–81, Mar. 2012.

[13] B. Lima and J. P. Faria, "DCO Analyzer: Local controllability and observability analysis and enforcement of distributed test scenarios," in *Proc. 42nd Int. Conf. Softw. Eng. Companion (ICSE)*. New York, NY, USA: ACM, 2020, pp. 1–4.

[14] Oracle. (Dec. 2019) *Java SE 12*. [Online]. Available: https://www.oracle.com/technetwork/java/javase/overview/index.html

[15] E. Durr and J. van Katwijk, "VDM++, a formal specification language for object-oriented designs," in *Proc. Comput. Syst. Softw. Eng.*, 1992, pp. 214–219.

[16] *OMG Unified Modeling Language TM (OMG UML) Version 2.5.1*, Object Management Group, Needham, MA, USA, 2017.

[17] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Mateo, CA, USA: Morgan Kaufmann, 2010.

[18] S. Schulz, J. Honkola, and A. Huima, "Towards model-based testing with architecture models," in *Proc. 14th Annu. IEEE Int. Conf. Workshops Eng. Comput.-Based Syst. (ECBS)*, Mar. 2007, pp. 495–502.

[19] M. Mikucionis, K. G. Larsen, and B. Nielsen, "T-uppaal: Online model-based testing of real-time systems," in *Proc. 19th Int. Conf. Automated Softw. Eng.*, 2004, pp. 396–397.

[20] K. Chen, J. Lv, J. Huang, H. Guo, S. Su, and T. Tang, "Online conformance testing of CBTC on-board ATO functions based on UPPAAL-TRON framework," in *Proc. IEEE Intell. Transp. Syst. Conf. (ITSC)*, Oct. 2019, pp. 3334–3339.

[21] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Softw. Test., Verification Rel.*, vol. 22, no. 5, pp. 297–312, Aug. 2012.

[22] J. Lilius and I. P. Paltor, "Formalising UML state machines for model checking," in *Proc. Int. Conf. Unified Modeling Lang.* Cham, Switzerland: Springer, 1999, pp. 430–444.

[23] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with spec explorer," in *Formal Methods and Testing*. Cham, Switzerland: Springer, 2008, pp. 39–76.

[24] Q. Tani and A. Petrenko, "Input/output automata" in *Proc. Test. Commun. Syst. IFIP TC6 11th Int. Workshop Test. Commun. Syst. (IWTCS)*, vol. 3. Tomsk, Russia: Springer, Aug./Sep. 1998, p. 83.

[25] A. Z. Javed, P. A. Strooper, and G. N. Watson, "Automated generation of test cases using model-driven architecture," in *Proc. 2nd Int. Workshop Autom. Softw. Test (AST)*, May 2007, p. 3.

[26] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *Formal Methods Syst. Des.*, vol. 19, no. 1, pp. 45–80, 2001.

[27] W. Grieskamp, "Multi-paradigmatic model-based testing," in *Formal Approaches to Software Testing and Runtime Verification*. Cham, Switzerland: Springer, 2006, pp. 1–19.

[28] W. Grieskamp, "Multi-paradigmatic model-based testing," in *Proc. 1st Combined Int. Workshops Formal Approaches Softw. Test. Runtime Verification (FATES)*. Berlin, Germany: Springer, Aug. 2006, pp. 1–19.

[29] A. Ulrich and H. König, "Architectures for testing distributed systems," in *Testing of Communicating Systems* (The International Federation for Information Processing), vol. 21, G. Csopaki, S. Dibuz, and K. Tarnay, Eds. Cham, Switzerland: Springer, 1999, pp. 93–108.

[30] R. M. Hierons, "Combining centralised and distributed testing," *ACM Trans. Softw. Eng. Methodology*, vol. 24, no. 1, pp. 5:1–5:29, Oct. 2014.

[31] S. Boroday, A. Petrenko, and A. Ulrich, "Implementing MSC tests with quiescence observation," in *Proc. 21st IFIP WG 6.1 Int. Conf. Test. Softw. Commun. Syst. 9th Int. FATES Workshop*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 49–65.

[32] M. Lahami, F. Fakhfakh, M. Krichen, and M. Jmaiel, "Towards a TTCN-3 test system for runtime testing of adaptable and distributed systems," in *Proc. 24th IFIP WG 6.1 Int. Conf. Test. Softw. Syst. (ICTSS)*, in Lecture Notes in Computer Science, vol. 7641, B. Nielsen and C. Weise, Eds. Cham, Switzerland: Springer, 2012, pp. 71–86.

[33] M. Lahami, M. Krichen, M. Bouchakwa, and M. Jmaiel, "Using knapsack problem model to design a resource aware test architecture for adaptable and distributed systems," in *24th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2012)*, ser. Lecture Notes in Computer Science, B. Nielsen and C. Weise, Eds., vol. 7641. Springer, 2012, pp. 103–118.

[34] M. Lahami and M. Krichen, "Test isolation policy for safe runtime validation of evolvable software systems," in *Proc. Workshops Enabling Technol., Infrastruct. Collaborative Enterprises*, S. Reddy and M. Jmaiel, Eds., Jun. 2013, pp. 377–382.

[35] T. Zheng, F. Khendek, and L. Helouët, "A semantics for timed MSC," *Electron. Notes Theor. Comput. Sci.*, vol. 65, no. 7, pp. 85–99, May 2002.

[36] S. Akshay, B. Bollig, and P. Gastin, "Automata and logics for timed message sequence charts," in *Int. Conf. Found. Softw. Technol. Theor. Comput. Sci.* Cham, Switzerland: Springer, 2007, pp. 290–302.
[37] S. Akshay, P. Gastin, M. Mukund, and K. Narayan Kumar, "Checking conformance for time-constrained scenario-based specifications," *Theor. Comput. Sci.*, vol. 594, pp. 24–43, Aug. 2015.
[38] R. M. Hierons, M. G. Merayo, and M. Núnez, "Using time to add order to distributed testing," in *Proc. Int. Symp. Formal Methods*. Cham, Switzerland: Springer, 2012, pp. 232–246.
[39] C. Gaston, R. M. Hierons, and P. Le Gall, "An implementation relation and test framework for timed distributed systems," in *Proc. IFIP Int. Conf. Test. Softw. Syst.* Cham, Switzerland: Springer, 2013, pp. 82–97.
[40] B. Lima and J. Faria, "Conformance checking in integration testing of time-constrained distributed systems based on UML sequence diagrams," in *Proc. 12th Int. Conf. Softw. Technol.*, 2017, pp. 459–466.
[41] A. Khoumsi, "A temporal approach for testing distributed systems," *IEEE Trans. Softw. Eng.*, vol. 28, no. 11, pp. 1085–1103, Nov. 2002.
[42] S. Azzouzi, S. Hsaini, and M. E. H. Charaf, "A synchronized test control execution model of distributed systems," *Int. J. Grid High Perform. Comput.*, vol. 12, no. 1, pp. 1–17, Jan. 2020.

**BRUNO LIMA** (Student Member, IEEE) received the master's degree in informatics and computing engineering from the Faculty of Engineering of the University of Porto (FEUP), in 2014, where he is currently pursuing the Ph.D. degree with the Department of Informatics Engineering. He conducted a master's thesis on component testing and certification for an ambient assisted living ecosystem, as a member of the AAL4ALL Research Team, INESC TEC. He is currently an Assistant Lecturer with the Department of Informatics Engineering, FEUP. He is also a Researcher with INESC TEC, where he participates in research projects in the area of e-health and software engineering. His research interests include software engineering, certification, and software testing, particularly in the scope of e-health and ambient assisted living systems.

**JOÃO PASCOAL FARIA** (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from the Faculty of Engineering of the University of Porto (FEUP), in 1999. He is currently an Associate Professor with FEUP, a Senior Researcher with the Institute for Systems and Computer Engineering, Technology and Science (INESC TEC), and the President of the Sectorial Commission for Information and Communications Technology (CS/03) in the scope of the Portuguese Quality Institute (IPQ). He has more than 25 years of research and development experience in software engineering, having published more than 60 papers in several journals and conferences, and obtained four Best Paper Awards. His current research interests include model-based testing, software process improvement, and model-driven development. He is a member of ACM.

**ROBERT HIERONS** (Senior Member, IEEE) received the B.A. degree in mathematics from the Trinity College, Cambridge, and the Ph.D. degree in computer science from Brunel University. He then joined the Department of Mathematical and Computing Sciences, Goldsmiths College, University of London, before returning to Brunel University, in 2000. He was promoted to a Full Professor in 2003 and joined The University of Sheffield, in 2018.