Conformance Checking in Integration Testing of Time-constrained Distributed Systems based on UML Sequence Diagrams

Bruno Lima and João Pascoal Faria
Faculty of Engineering, University of Porto, Porto, Portugal
INESC TEC, Porto, Portugal

Keywords: Model-based Integration Testing, Conformance Checking, Distributed Systems, Time Constraints, UML.

Abstract: The provisioning of a growing number of services depends on the proper interoperation of multiple products, forming a new distributed system, often subject to timing requirements. To ensure the interoperability and timely behavior of this new distributed system, it is important to conduct integration tests that verify the interactions with the environment and between the system components. Integration test scenarios for that purpose may be conveniently specified by means of UML sequence diagrams (SDs) enriched with time constraints. The automation of such integration tests requires that test components are also distributed, with a local tester deployed close to each system component, coordinated by a central tester. The distributed observation of execution events, combined with the impossibility to ensure clock synchronization in a distributed system, poses special challenges for checking the conformance of the observed execution traces against the specification, possibly yielding inconclusive verdicts. Hence, in this paper we investigate decision procedures and criteria to check the conformance of observed execution traces against a specification set by a UML SD enriched with time constraints. The procedures and criteria are specified in a formal language that allows executing and validating the specification. Examples are presented to illustrate the approach.

1 INTRODUCTION

In a growing number of domains, the provisioning of end-to-end services to the users depends on the proper interoperation of multiple products (devices, applications, etc.) from different vendors, forming a new distributed system, often subject to timing requirements. Examples can range from sports monitoring applications (Taylor, 2015) to fall detection systems for seniors (Lima and Faria, 2016) or IoT systems.

To ensure interoperability and the timely behavior of such distributed systems, it is important to conduct integration tests that verify the interactions between the system components and with the environment. Integration test scenarios for that purpose may be conveniently specified by means of UML Sequence Diagrams (OMG, 2015) (SDs), because they are an industry standard well suited for describing and visualizing the interactions that occur between the components and actors of a distributed system.

In previous work (Lima and Faria, 2016) we proposed an overall approach to automate the integration testing in distributed and heterogeneous systems. The only manual activity required is the description of the participants and behavior of the services under test with UML SDs. In the proposed approach, a *local tester* is deployed close to each system component, coordinated by a *central tester*. However, the distributed observation of execution events by the local testers, combined with the impossibility to ensure clock synchronization in a distributed system, poses special challenges for checking the conformance of observed execution traces against the specification.

The main contributions of this paper are novel decision procedures and criteria to check the conformance of distributed execution traces against a specification set by a UML SD enriched with time constraints. The procedures and criteria are specified in the VDM formal language (Fitzgerald et al., 2005)(Larsen et al., 2016), which allows executing and validating the specification. Examples are presented to illustrate the approach.

Section 2 presents background on our approach for the integration testing of distributed systems; section 3 presents the decision procedures and criteria for conformance checking in the presence of time constraints; related work is presented in section 4; section 5 concludes the paper.

Lima, B. and Faria, J.
Conformance Checking in Integration Testing of Time-constrained Distributed Systems based on UML Sequence Diagrams.
DOI: 10.5220/0006474004590466
In *Proceedings of the 12th International Conference on Software Technologies (ICSOFT 2017)*, pages 459-466
ISBN: 978-989-758-262-2

2 BACKGROUND

In previous work (Lima and Faria, 2016) we proposed a scenario-based approach for automating the integration testing of end-to-end services in distributed and heterogeneous systems.

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

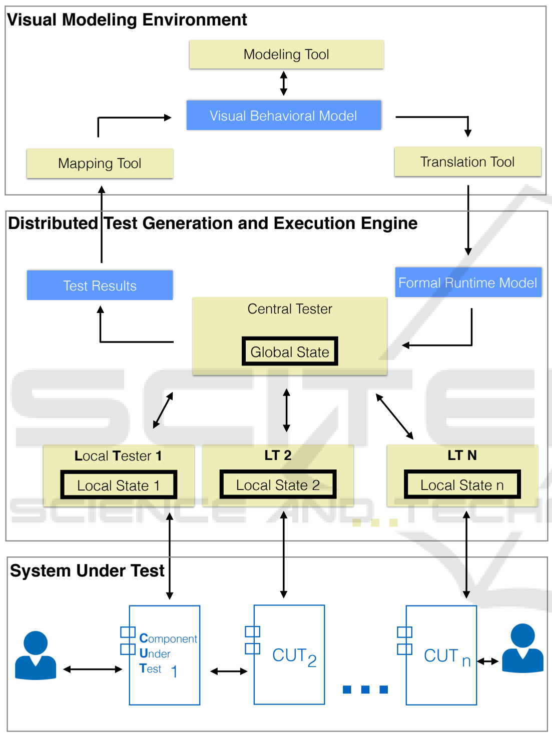


Figure 1: Architecture for model-based integration testing of distributed systems (Lima and Faria, 2016).

We follow an *online* model-based testing (MBT) strategy (or *adaptive*), to cope with non-determinism common in distributed systems. In an online strategy, test generation and execution are performed together, allowing test inputs to be decided based on the outputs observed so far (Utting and Legeard, 2007).

In order to be able to check not only the interactions with the environment but also the interactions between the components of the system under test (SUT), we follow a hybrid test architecture, in which a *local tester* is deployed close to each system component, and a *central tester* coordinates the local testers (see Figure 1). This is more effective than a purely centralized approach or a purely distributed approach, meaning that more conformance errors can be detected (Hierons, 2014).

In the case of a component under test (CUT) that interacts with actors (users or external applications) in the environment (see Local Tester 1 in Figure 1), the local tester is responsible for simulating the actors, injecting the inputs from the actors to the CUT (acting as *a test driver*) and monitoring and checking the outputs from the CUT to the actors (acting as a *test monitor*). Besides that, it is also responsible for monitoring all the messages exchanged between that CUT and the rest of the SUT. In the case of a CUT that does not interact with the environment (see Local Tester 2 in Figure 1), the local tester is responsible for monitoring and checking all the messages exchanged between that CUT and the rest of the SUT, acting as a test monitor.

In such a test architecture, it is important to minimize the communication overhead during test execution, namely in the presence of time constraints. It is equally important to detect errors as early as possible and closely as possible to the offending component, to provide more helpful test reports and facilitate fault localization. To that end, besides the test monitoring and control activities, model execution and conformance checking activities need also be distributed.

In more recent work (Lima and Faria, 2017) we investigated distributed conformance checking but without taking time observation and time constraints into consideration.

3 CONFORMANCE CHECKING WITH TIME CONSTRAINTS

As said before, conformance checking in the presence of time constraints in a distributed system is specially challenging. Hence, in this section, we investigate the decision procedures and criteria to check the conformance of observed execution traces against a specification set by a UML SD enriched with time constraints.

As in our previous work, the decision procedures and criteria are specified with the VDM formal specification language (Fitzgerald et al., 2005)(Larsen et al., 2016), to enable us to execute and validate the specification with a support tool, such as Overture (http://overturetool.org/).

Before investigating the conformance procedures and criteria, we need to formalize the structure and semantics of time-constrained SDs.

3.1 Time-constrained SDs

UML SDs may be annotated with time constraints (OMG, 2015). Although the UML standard allows the specification of more complex constraints, in this paper we restrict our attention to the types of time constraints that are commonly addressed in the literature of time-constrained distributed systems (see Related Work) and are most relevant in practice: constraints that specify the minimum and maximum delays (time elapsed) between two events (message sending or receiving) in the same lifeline, or between the sending and receiving of a message between two lifelines. We assume that the minimum and maximum delays are constant values (or expressions that can be evaluated before the scenario execution).

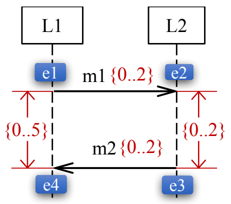


Figure 2: Example of a time-constrained SD.

Figure 2 presents a scenario with that type of time constraints in UML. Each interval is denoted $\{min..max\}$, where *min* or *max* can be omitted. Denoting by *ti* the time instant of event *ei*, the only valid execution trace defined by this SD is the *time-constrained trace* $[(e1,t1),(e2,t2),(e3,e3),(e4,t4)]$ with $t2 \in [t1,t1+2]$, $t3 \in [t2,t2+2]$, $t4 \in [t3,t3+2]$ and $t4 \in [t1,t1+5]$. An example of a valid *timed trace* (with the time instant of each event) is $[(e1,1),(e2,3),(e3,4),(e4,5)]$. An example of an invalid timed trace is $[(e1,1),(e2,3),(e3,5),(e4,7)]$.

To check if an observed trace is valid, besides verifying if the observed events occurred by a certain order, it is also necessary to verify if their time instants satisfy the time constraints defined in the SD, as will be formalized in the next two subsections. We start by computing the valid time-constrained traces defined by a SD.

In UML, an SD is a variant of an Interaction (OMG, 2015). Figure 3 presents an excerpt of the formalization of the structure of Interactions in VDM, highlighting the elements added to support time constraints as compared to our previous work (Lima and Faria, 2017). For simplicity, we omit the definition of combined fragments (which can be found in our previous work), integrity constraints, and some basic types. An Interaction comprises a set of lifelines (representing in our case CUTs or actors), messages (restricted in our case to asynchronous messages, although synchronous messages could easily be handled), combined fragments and time constraints. For each message, we define the locations of the send and receive events (`sendEvent` and `receiveEvent`), and the (optional) variables that represent the instants of time of occurrence of those events (`sendTimestamp` and `rcvTimestamp`).

A `TimeConstraint` imposes minimum and maximum values on the time that can elapse between two events (identified by the timestamp variables). Either the minimum or the maximum may be omitted.

3.2 Valid Traces defined by a Time-constrained SD

In general, the semantics of an Interaction is expressed in terms of two sets of valid and invalid traces (OMG, 2015). In this paper, we don't handle the rarely used constructs for defining invalid traces (such as the `neg` operator), so only the valid traces are relevant here. A *trace* is a sequence of event occurrences (OMG, 2015), corresponding, in this context, to the emission or reception of messages at lifelines. Figure 3 shows the structure of traces in VDM with timing information, by means of an optional timestamp associated with each event. Depending on the context, the timestamp may be a variable (as in $(e1,t1)$) or a concrete value (as in $(e1,1)$). In this paper we assume a discrete time scale (in an appropriate unit, such as seconds or microseconds), but a continuous scale would equally work. Hence, time instants (`Time`) and elapsed time (`Duration`) are represented by natural numbers.

Since we are dealing with distributed systems, we assume there is no global clock, so time instants are measured with the local clocks. Although it is impossible to ensure perfect clock synchronization between lifelines, in practice we can assume, like other authors did (see Related Work), that there is a maximum difference (or skew) between the readings of any two clocks (`MaxClockSkew`). For example, the Network Time Protocol (NTP) (Mills, 1991), designed to synchronize the clocks of computers over a network to a common timebase (usually UTC), achieves synchronization accuracies of 10 ms over Internet, and 1 ms on LAN.

For computing the set of valid traces defined by an Interaction with time constraints (see function `validTraces` in Figure 4), we first compute the valid traces ignoring time constraints (`validTracesUntimed`), following a procedure presented in our previous work for SDs without time constraints (Lima and Faria, 2017), and subsequently exclude the traces for which the defined time constraints

```
types
Interaction ::
  lifelines         : Lifeline-set
  messages          : Message-set
  combinedFragments: CombinedFragment-set
  timeConstraints   : TimeConstraint-set;

Message ::
  id           : MessageId         -- unique
  sendEvent    : LifelineLocation -- unique
  receiveEvent : LifelineLocation -- unique
  signature    : MessageSignature -- may not be unique
  sendTimestamp: [Variable]        -- unique
  recvTimestamp: [Variable];       -- unique

LifelineLocation = Lifeline × Location;

TimeConstraint ::
  firstEvent : Variable --identified by timestamp var.
  secondEvent: Variable-- identified by timestamp var.
  min        : [Duration]
  max        : [Duration];

Variable :: name : String;

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

Event ::
  type      : EventType
  signature: MessageSignature
  lifeline : Lifeline
  timestamp: [Variable|Time];

EventType = <Send> | <Receive>;
Time = ℕ;
Duration = ℕ;
TimeInterval = [Time] × [Time]; -- (min, max) pair

values
MaxClockSkew : Duration = 10; -- configurable
```

Figure 3: Interactions and traces.

```
functions
-- Gives the set of valid traces defined by an
-- Interaction.
validTraces: Interaction → Trace-set
validTraces(sd) ≜ {t | t ∈ validTracesUntimed(sd) •
                  isSatisfiable(t, sd.timeConstraints)};

-- Checks if a set of time constraints (C), imposing
-- min/max delays between pairs of events, is
-- satisfiable for a trace t.
isSatisfiable: Trace × TimeConstraint-set → 𝔹
isSatisfiable(t, C) ≜
 let cp = getConstrainedPairs(t, C),
     minGiven= {mk_(i,i+1)↦0 | i ∈ {1,...,|t|-1}} ++
               {mk_(i,j)↦c.min |
                    mk_(i,j,c) ∈ cp • c.min ≠ nil},
     maxGiven= {mk_(i,j)↦c.max |
                    mk_(i,j,c) ∈ cp • c.max ≠ nil},
     minDeriv= longestDistances(minGiven),
     maxDeriv= shortestDistances(maxGiven)
  in ∄ p ∈ dom minDeriv ∩ dom maxDeriv •
        minDeriv(p) > maxDeriv(p);

-- Given trace t and time constraints C, returns
-- tuples (i,j,c) of indices i and j of events in t
-- that are subject to a  constraint c in C.
getConstrainedPairs: Trace × TimeConstraint-set
→ (ℕ × ℕ × TimeConstraint)-set
getConstrainedPairs(t, C) ≜
 {mk_(i,j,c) | i ∈ inds t, j ∈ inds t, c ∈ C •
    i < j
    ∧ tᵢ.timestamp = c.firstEvent
    ∧ tⱼ.timestamp = c.secondEvent
    ∧ ∄ k ∈ {i+1, ..., j-1} •
         tₖ.timestamp ∈ {c.firstEvent,c.secondEvent}};
```

Figure 4: Valid traces defined by an Interaction with time constraints.

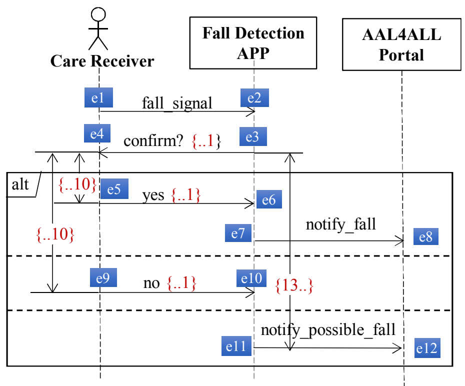
are not satisfiable (function `isSatisfiable`). A set C of time constraints is satisfiable for a trace t if there is an assignment of non-decreasing time instants to the event occurrences in t that satisfies all the time constraints in C. Before explaining the procedure for determining satisfiability, we start by presenting a running example.

Figure 5 presents a SD that describes a real scenario of a fall detection service. 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 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 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 time constraint of 10 seconds between a reception of the confirmation request and the user response, other time restrictions are also represented, namely one 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 this example, if time constraints are ignored, there are five valid traces (see Valid Traces Untimed in Figure 5). The last two traces correspond to permutations of the third trace in which event $e4$ occurs after events $e11$ and/or $e12$. This happens because, if time constraints are ignored, the presented SD doesn't impose any relative ordering between event $e4$ and events $e11$ and $e12$. However, if time constraints are considered, those traces are no longer valid. In fact, it is easy to check that the defined time constraints are not satisfiable for those traces: event $e4$ must occur up to 1 time unit after $e3$, whilst event $e11$ must occur at least 13 time units after $e3$, so $e4$ cannot occur after $e11$.

Let's return to the procedure for determining satisfiability of time constraints. For the type of time constraints we are considering in this paper, satisfiability can be determined in polynomial time, following the steps depicted in Figure 4 (function `isSatisfiable`) and explained next:

1. compute the pairs of event occurrences (identified by their indexes in t) that are subject to a time constraint (variable `cp`); as illustrated in Figure 7, in case of multiple occurrences of the same event in an execution trace (caused by loops), we assume



Valid Traces Untimed:
[e1, e2, e3, e4, e5, e6, e7, e8],
[e1, e2, e3, e4, e9, e10],
[e1, e2, e3, e4, e11, e12] ,
[e1, e2, e3, e11, e4, e12] ,
[e1, e2, e3, e11, e12, e4]

Valid Traces Timed:
[e1, e2, e3, e4, e5, e6, e7, e8],
[e1, e2, e3, e4, e9, e10],
[e1, e2, e3, e4, e11, e12]

Figure 5: Example of a fall detection scenario with time constraints.

that time constraints apply only to pairs of closest occurrences;

2. create two weighted directed acyclic graphs (dag) representing the minimum and maximum delays between pairs of event occurrences, based on the ordering of event occurrences in *t* and the result of the first step (variables `minGiven` and `maxGiven`);
3. derive weighted dags with the longest and shortest distances between all pairs of connected vertexes in graphs `minGiven` and `maxGiven`, respectively (variables `minDerived` and `maxDerived`, representing derived minimum and maximum delays);
4. check that there aren't pairs of event occurrences (identified by their indexes in *t*) constrained by incompatible derived minimum and maximum delays.

Figure 6 illustrates the application of this procedure for trace $[e1, e2, e3, e11, e4, e12]$ in Figure 5. We conclude that the trace is invalid because of the conflicting minimum and maximum delays required between events *e*3 and *e*4.

3.3 Conformance Checking based on Distributed Observations

In general an observed global trace *t* conforms to the specification if it belongs to the set of valid traces (computed as explained in the previous section) and satisfies the time constraints.

However, in distributed testing, global traces are not directly observed, but only the local traces observed at each lifeline, which raises the need to infer

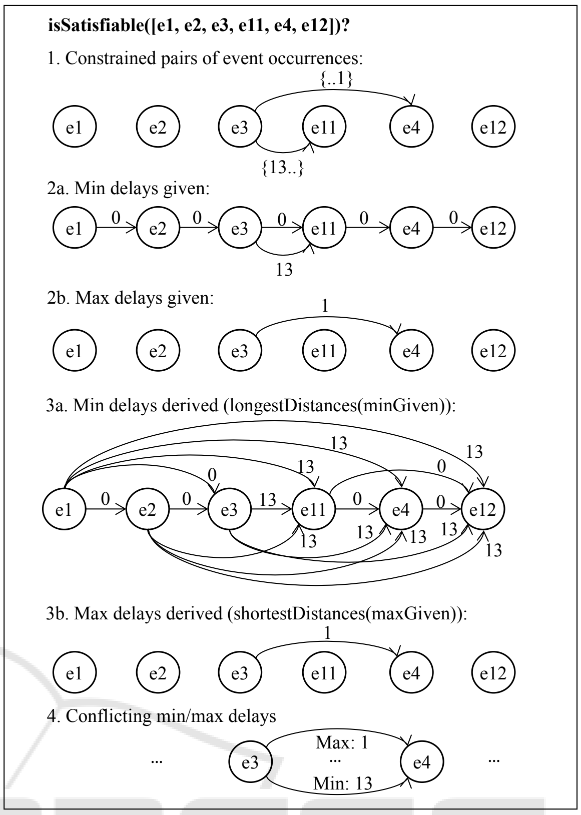

isSatisfiable([e1, e2, e3, e11, e4, e12])?
1. Constrained pairs of event occurrences:
{..1}
{13..}
2a. Min delays given:
0
13
2b. Max delays given:
1
3a. Min delays derived (longestDistances(minGiven)):
13
0
3b. Max delays derived (shortestDistances(maxGiven)):
1
4. Conflicting min/max delays
Max: 1
Min: 13
e1
e2
e3
e11
e4
e12
...

Figure 6: Example of satisfiability checking.

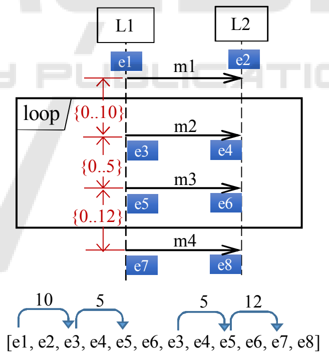


Figure 7: Example of computation of constrained pairs in the presence of multiple occurrences of the same event.

global traces from the observed local traces, possibly leading to inconclusive verdicts.

In distributed testing, conformance checking is best performed in two phases: first, local conformance checking is performed at each lifeline in and incremental way and, in case a local failure is detected, the test fails globally; secondly, in case all local checks pass, conformance checking is performed globally. For some scenarios (called *locally observable*), the second step is not needed, but the determination of conditions upon which such step is not needed is outside the scope of this paper.

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

We present in Figure 8 the primitives to perform a final conformance check globally by the central tester. Our procedure for determining if the observed local traces (including timestamp values) conform to the specification (SD), comprises the following steps (see function `finalConformanceCheck`):

Step 1. Obtain the set V of valid traces (with timestamp variables, but not timestamp values) defined by the given SD, as explained in the previous section. This step is performed only once per SD.

Step 2. Compute the set J of all feasible joins of the traces observed locally (with timestamp values). By a feasible join we mean a global trace obeying the following conditions: (i) the projections onto the lifelines yield the local traces (ensured incrementally by `joinActualTraces`); (ii) messages cannot be received before being sent, although they can be lost in the transmission channel (ensured incrementally by `isFeasibleAddition`); (iii) event occurrences from different lifelines must be ordered in a way consistent with their time instants, considering the maximum clock skew defined (ensured incrementally by `isFeasibleAddition`). Since we assume that only the message signature is guaranteed to be observable, in the presence of multiple message occurrences with the same signature, we can only make sure that, at any given point in the trace under analysis, the number of 'send' occurrences is greater or equal to the number of 'receive' occurrences (see `isFeasibleAddition`). In practice, there is only one or a few possible joins.

Step 3. If all feasible joins match at least one valid global trace, the test passes. A trace with timestamp values (actual trace) matches a trace with timestamp variables (formal trace) if they are identical when timing information is removed (`matchesUntimed`), and the former satisfies the time constraints associated with the latter. Because of the clock skew between lifelines, checking inter-lifeline constraints, and hence matching, may yield an inconclusive result (see `checkConstraint` in Figure 8 and Figure 9).

Step 4. If all the feasible joins fail to match all valid global traces, the test fails. Otherwise, we get an inconclusive result.

Examples of traces yielding different verdicts are shown in Figure 10.

For the sake of completeness, we present in Figure

```
types
Verdict = <Pass> | <Inconclusive> | <Fail>;
          -- dislayed from best to worse
functions
-- Global conformance checking, given the observed
-- local traces.
                                                 m
finalConformanceCheck: Interaction ×(Lifeline →
Trace) → Verdict
finalConformanceCheck(sd, localTraces) ≜
 let V = validTraces(sd), C = sd.timeConstraints,
     J = joinActualTraces([], localTraces),
 in if ∀j∈J • ∀v∈V • matches(j,v,C)=<Fail> then <Fail>
   else if ∀j∈J • ∃v∈V • matches(j,v,C)=<Pass> then <Pass>
   else <Inconclusive>;

-- Gives the feasible joins of traces from different
-- lifelines, respecting the order of events per trace
-- and message. The first argument is an accumulator
-- for already processed events (initially empty).
                                      m
joinActualTraces: Trace × (Lifeline→Trace) → Trace-set
joinActualTraces(t, m) ≜
 if ∀ l ∈ dom m • m(l) = [] then {t}
 else ⋃ {joinActualTraces(t⁀[hd m(l)], m++{l↦ tl m(l)})
         | l ∈ dom m • m(l) ≠ [] ∧
           isFeasibleAddition(t, hd m(l))};

-- Checks if an event occurrence is a feasible
-- addition to a trace, i.e., respects the fact that
-- messages can only be received after being sent, and
-- respects timestamp ordering.
isFeasibleAddition: Trace × Event → 𝔹
isFeasibleAddition(t, e) ≜
 (e.type = <Receive> ⇒
   #{i | i ∈ inds t • t_i.type = <Send> ∧
           t_i.signature = e.signature} >
   #{i | i ∈ inds t • t_i.type = <Receive> ∧
           t_i.signature = e.signature})
∧(∀ f ∈ t •  e.timestamp ≥ f.timestamp -
   (if f.lifeline=e.lifeline then 0 else MaxClockSkew));

-- Checks if an actual trace (a) matches a formal
-- trace (f), given a set of time constraints (C).
matches: Trace × Trace × TimeConstraint-set → Verdict
matches(a, f, C)  ≜
 if ¬ matchesUntimed(a, f) then <Fail>
 else worseVerdict({checkConstraint(a_i,a_j,c) |
              mk_(i,j,c) ∈ getConstrainedPairs(f,C)});

-- Checks if an actual trace (a) matches a formal
-- trace (f), ignoring time constraints.
matchesUntimed: Trace × Trace → 𝔹
matchesUntimed(a, f)  ≜
 |a| = |f| ∧ ∀ i ∈ inds a •
   mu(a_i, timestamp ↦ nil) = mu(f_i, timestamp ↦ nil);

-- Checks a time constraint (c) between two events (e1
-- before e2).
checkConstraint: Event × Event × TimeConstraint → Verdict
checkConstraint(e1, e2, c) ≜
 let d = e2.timestamp - e1.timestamp,
     s = (if e1.lifeline = e2.lifeline then 0
          else MaxClockSkew),
     ds = mk_(max(d-s, 0), d+s)
 in cases intersect({mk_(c.min, c.max), ds}):
     (ds) → <Pass>,
     nil → <Fail>,
     others → <Inconclusive>
 end;
```

Figure 8: Global conformance checking in the presence of time constraints.

11 the primitive needed to perform incremental conformance checking locally in each lifeline by each local tester. Because only intra-lifeline time constraints are checked locally, matching is never inconclusive.

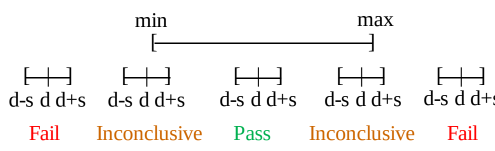


Figure 9: Checking imprecise duration observations (d±s) against a required interval (function `checkConstraint`).

#	**Observed local traces** (locally valid)			**Possible Joins *** (with first problematic event underlined)	**Verdict ***
	Care Receiver	**Fall Detection App**	**AAL4ALL Portal**		
1	[(e1, 0), (e4, 4.2), (e5, 14.2)]	[(e2, 2), (e3, 4), (e6, 14.5), (e7, 14.6)]	[(e8,16)]	[(e1, 0), (e2, 2), (e3, 4), (e4, 4.2), (e5,14.2), (e6,14.5), (e7,14.6), (e8,16)]	Pass
2	[(e1, 0), (e4, 4.2), (e5, 14.2)]	[(e2, 2), (e3, 4), (e6, 15.2), (e7, 15.6)]	[(e8,16)]	[(e1, 0), (e2, 2), (e3, 4), (e4, 4.2), (e5,14.2), (e6,15.2), (e7,15.6), (e8,16)]	Inconclu sive (a)
3	[(e1, 0), (e4, 4.2), (e5, 14.2)]	[(e2, 2), (e3, 4), (e6, 18), (e7, 18.6)]	[(e8,19)]	[(e1, 0), (e2, 2), (e3, 4), (e4, 4.2), (e5, 14.2), (e6, 18), (e7,18.6), (e8,19)]	Fail (b)
4	[(e1, 0), (e4, 16.8)]	[(e2, 2), (e3, 4), (e11, 17)]	[(e12,18)]	[(e1, 0), (e2, 2), (e3, 4), (e4,16.8), (e11,17), (e12,18)] [(e1, 0), (e2, 2), (e3, 4), (e11, 17), (e4,16.8), (e12,18)]	Fail (c)

* Assuming *MaxClockSkew* = 0.5s
(a) Time constraint between *e5* and *e6* (1s) possibly not respected.
(b) Time constraint between *e5* and *e6* (1s) not respected.
(c) Time constraint between *e3* and *e4* (1s) not respected.

Figure 10: Examples of traces with different conformance checking verdicts in the fall detection scenario.

```
functions
-- Checks if the next observed event occurrence (e) in
-- a lifeline is valid, given a valid sequence of
-- previously observed event occurrences in the
-- lifeline (p), the set of valid local traces (V) and
-- the set of local time constraints (C).
checkNextEvent: Trace × Event × Trace-set ×
TimeConstraint-set → 𝔹
checkNextEvent(p, e, V, C) ≙
 ∃ t ∈ V •
  |t| > |p| ∧ matches(p ⁀ [e], t1, …, |p|+1, C) = <Pass>;
```

Figure 11: Incremental conformance checking in the presence of time constraints.

4 RELATED WORK

Several works in the literature use temporal constraints in scenario-based specifications similar to the ones considered in our work, although with different objectives. One of them is the work from Zheng et al. (Zheng et al., 2002), in which the authors use an alternative representation of timed traces that incorporate time events between normal events. They derive the valid traces for Timed Message Sequence Charts (T-MSCs), which have many similarities with UML SDs, but do not address the problem of conformance checking based on distributed observations.

Another work is presented by Gaston et al. (Gaston et al., 2013). As in our work, they assume a distributed test architecture and no global clock. They provide a composition result, which allows checking conformance in two phases: in the first phase, each local tester checks local conformance according to the *tioco* conformance relation; in the second phase, the traces observed locally are brought together and it is analyzed if events are exchanged following some communication rules. In our case, conformance checking is also performed in two phases. However, different assumptions are considered in both works: they use timed input output transition systems whilst we use UML SDs; they assume that internal communication is multicast, whilst we assume unicast communication according to the UML standard (OMG, 2015); they assume that messages are never lost, contrarily to our assumption.

Akshay et al. (Akshay et al., 2007) presented 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. Compared to our work, we use similar time constraints to MSCs with timing constraints (TC-MSCs) where they associate lower and upper bounds on the time interval between certain pairs of events. However they do not address the problem of conformance checking.

In a more recent work (Akshay et al., 2015), the authors address model checking message-passing systems with real-time requirements. As behavioral specifications, they use TC-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, whilst in our work we want to check the conformance of traces exhibited by the actual system implementation and not the system model.

Regarding timed traces, in (Hierons et al., 2012), the authors take advantage of time information to refine the conformance relation, assuming that there is a local clock at each port (point of interaction with the environment) and that events at port p are timestamped with the current time of the local clock at port p. They assume that the local clocks differ by at most a know value α. In our work we have something similar, but further distinguish inconclusive traces and also check time constraints.

5 CONCLUSIONS

We presented decision procedures and criteria to check the conformance of observed execution traces against a test specification set by a UML SD, in the context of integration testing of time-constrained distributed system. As a consequence of the distributed nature of the system, the test verdict reached may be inconclusive in some cases.

As future work, we will investigate conditions for local observability and local controllability of distributed integration test scenarios with time constraints. We will also investigate procedures to overcome the lack of local observability and/or local controllability.

ACKNOWLEDGEMENTS

This work was performed in scope of project “NanoSTIMA: Macro-to-Nano Human Sensing: Towards Integrated Multimodal Health Monitoring and Analytics/NORTE-01-0145-FEDER-000016”, financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). This work was also financed by the Portuguese Foundation for Science and Technology (FCT), under research grant SFRH/BD/115358/2016.

REFERENCES

Akshay, S., Bollig, B., and Gastin, P. (2007). Automata and logics for timed message sequence charts. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 290–302. Springer.

Akshay, S., Gastin, P., Mukund, M., and Kumar, K. N. (2015). Checking conformance for time-constrained scenario-based specifications. *Theoretical Computer Science*, 594:24–43.

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

Gaston, C., Hierons, R. M., and Le Gall, P. (2013). An implementation relation and test framework for timed distributed systems. In *IFIP International Conference on Testing Software and Systems*, pages 82–97. Springer.

Hierons, R. M. (2014). Combining Centralised and Distributed Testing. *ACM Trans. Softw. Eng. Methodol.*, 24(1):5:1–5:29.

Hierons, R. M., Merayo, M. G., and Núnez, M. (2012). Using time to add order to distributed testing. In *International Symposium on Formal Methods*, pages 232–246. Springer.

Larsen, P. G., Lausdahl, K., Battle, N., Fitzgeral, J., Wolff, S., Sahra, S., Verhoef, M., Tran-Jørgensen, P., Oda, T., and Chisholm, P. (2016). VDM-10 Language Manual. Technical report.

Lima, B. and Faria, J. P. (2016). *Software Technologies: 10th International Joint Conference, ICSOFT 2015, Colmar, France, July 20-22, 2015, Revised Selected Papers*, chapter Automated Testing of Distributed and Heterogeneous Systems Based on UML Sequence Diagrams, pages 380–396. Springer International Publishing, Cham.

Lima, B. and Faria, J. P. (2017). Towards decentralized conformance checking in model-based testing of distributed systems. In *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE Ninth International Conference on*, pages 356–365. IEEE.

Mills, D. L. (1991). Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493.

OMG (2015). OMG Unified Modeling Language TM (OMG UML) Version 2.5. Technical report, Object Management Group.

Taylor, A. G. (2015). Keeping active with the activity app. In *Get Fit with Apple Watch*, pages 63–69. Springer.

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

Zheng, T., Khendek, F., and Hélouët, L. (2002). A semantics for timed msc. *Electronic Notes in Theoretical Computer Science*, 65(7):85–99.