Model-Based Testing of Timed Distributed Systems : A Constraint-Based Approach for Solving the Oracle Problem

Nassim Benharrat

université PARIS-SACLAY

NNT : 2018SACLC021

Thèse de doctorat

CentraleSupélec

Test à base de modèles de systèmes temporisés distribués : une approche basée sur les contraintes Pour résoudre le problème de l'oracle

Thèse de doctorat de l'Université Paris-Saclay
Préparée à CentraleSupélec

École doctorale n°573 : interfaces : Approches interdisciplinaires, Fondements, applications et innovation (Interfaces)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Gif Sur Yvette, le 14/02/2018, par

Nassim BENHARRAT

Composition du Jury :

Marc AIGUIER Professeur des universités, CentraleSupélec (Laboratoire MICS)	Président
Xavier URBAIN Professeur des universités, Université Lyon 1 (Équipe Drim, LIRIS)	Rapporteur
Ioannis PARISSIS Professeur des universités, Grenoble INP (Laboratoire LCIS)	Rapporteur
Delphine LONGUET Maitre de conférences, Université Paris-Sud (Laboratoire LRI)	Examinateur
Arnault LAPITRE Ingénieur de recherche, CEA Saclay (Laboratoire LIST)	Examinateur
Pascale LE GALL Professeur des universités, CentraleSupélec (Laboratoire MICS)	Directeur de thèse
Christophe GASTON Ingénieur de recherche, CEA Saclay (Laboratoire LIST)	Examinateur

To my loved ones
To my mentors

Acknowledgments

Prof. Pascale Le Gall introduced me to research when I first visited MOSAIC team for an internship at Ecole Centrale de Paris. Her dedication, suggestions, and guidance converted the lost student that arrived in 2014 into someone able to have his own ideas, express them and convert them into a Ph.D. thesis. Most importantly, she supported my work with valuable suggestions and stimulating discussions that strongly influenced the content of my thesis. Thanks Pascale for your patience, for teaching me how an article should be written and presented. I extend my gratitude to Christophe Gaston, for his support, encouragement, and guidance through every step of this work and for all he has taught me during this thesis. It was an honor and a pleasure to work with you both. Especial thanks to Arnault Lapitre for his help and the interesting discussions and suggestions about my work.

I sincerely thank the reviewers and the jury: Xavier Urbain and Ioannis Parissis for accepting reviewing this thesis and for all the comments they made; Marc Aiguier and Delphine Longuet for accepting to be in my Ph.D. thesis committee.

I would like also to thank my colleagues in CEA LIST-LISE research team for stimulating discussion and providing peer review of the work described in this thesis. In particular, I would like to thank Frédérique Descreaux for her patience, welcome, and assistance. I want to thank my colleagues with whom I have shared not only the office and lunch breaks, but also amusing and unforgettable moments: Amel Belaggoun, Anthony Legendre, Mohamed Benazouz, Gabriel Pedroza, Jean-Yves Pierron, Jean-Pierre Gallois, Alain Faivre and François Le Fevre, for their understanding, friendship and for their good humor.

I owe unconditional thanks for the support of my family who had to encourage me. Thank you for the trust you have given me, a big thank you for your support during my studies. I would also like to thank my parents for their unfailing support, and thanks to whom I could reach this stage. They always have been there for me, in all situations and I am infinitely grateful to them. My heartfelt thanks go to my dear wife for her patience and daily unwavering support. You have not ceased to comfort me and you have always been able to cheer me up.

Nassim Benharrat
Paris, France
December 2017

Abstract

Model-based testing of reactive systems is the process of checking if a System Under Test (SUT) conforms to its model. It consists of handling both test data generation and verdict computation by using models. In this thesis, we specify the behavior of reactive systems with so-called *Timed Input Output Symbolic Transition Systems* (TIOSTS), that are timed automata enriched with symbolic mechanisms to handle data.

When TIOSTSs are used to test systems with a centralized interface, that is, a system with a single user interface, the user interacts with the interface and may then completely order events occurring at this interface (i.e., inputs sent to the system and outputs produced from it). Interactions between the tester and the SUT are sequences of inputs and outputs named *traces*, separated by delays in the timed framework, to form so-called *timed traces*.

Distributed systems are collections of communicating local components which interact with their environment at physically distributed interfaces. The distributed nature of any observation of such systems is known to make distributed testing hard to solve. In addition, interacting with such a distributed system requires exchanging values with it by means of several interfaces in the same testing process. Different events occurring at different interfaces cannot be ordered any more since it is not possible to compare their respective moments at which they occurred. In this regard, this thesis focuses on conformance testing for distributed systems where a separate tester is placed at each localized interface and may only observe what happens at this interface. In our work, we assume that there is no global clock but only local clocks for each localized interface. The semantics of such systems can be seen as *tuples of timed traces* (one timed trace per localized interface representing a local vision of the system in question). We consider a framework for distributed testing from TIOSTS along with corresponding test hypotheses and a distributed conformance relation called *dtioco*. Global conformance can be tested in a distributed testing architecture using only local testers without any communication between them. We propose an algorithm to check valid communication policy for a tuple of timed traces by formulating the verification of message passing in terms of *Constraint Satisfaction Problem* (CSP). Therefore, we were able to implement the computation of test verdicts by orchestrating both localised off-line testing algorithms and the verification of constraints defined by message passing that can be supported by a constraint solver. Lastly, we validated our approach on a real case study of a telecommunications distributed system.

Keywords: Model-based testing, Distributed testing, Timed Input Output Symbolic Transition Systems, Off-line testing, Constraint Satisfaction Problem, Constraint-based testing.

Résumé

Le test à base de modèles des systèmes réactifs est le processus de vérifier si un système sous test (SUT) est conforme à sa spécification. Il consiste à gérer à la fois la génération des données de test et le calcul de verdicts en utilisant des modèles. Dans cette thèse, nous spécifions le comportement des systèmes réactifs à l'aide des systèmes de transitions symboliques temporisées à entrée sortie (TIOSTS).

Quand les TIOSTSs sont utilisés pour tester des systèmes avec une interface centralisée, l'utilisateur interagit avec toute l'interface et peut alors ordonner complètement les événements (i.e., les entrées envoyées au système et les sorties produites). Les interactions entre le testeur et le SUT consistent en des séquences d'entrées et de sortie nommées *traces*, pouvant être séparées par des durées dans le cadre du test temporisé, pour former ce que l'on appelle des *traces temporisées*.

Les systèmes distribués sont des collections de composants locaux communiquant entre eux et interagissant avec leur environnement via des interfaces physiquement distribuées. La nature distribuée des observations est connue pour rendre le test distribué difficile à résoudre. Différents événements survenant à ces différentes interfaces ne peuvent plus être ordonnés car il n'est pas possible de comparer leurs moments respectifs auxquels ils se sont produits. Cette thèse concerne le test de conformité pour les systèmes distribués où un testeur séparé est placé à chaque interface localisée et peut seulement observer ce qui se passe à cette interface. Dans notre travail, nous supposons qu'il n'y a pas d'horloge commune mais seulement des horloges locales pour chaque interface localisée. La sémantique de tels systèmes est définie comme des *tuples de traces temporisées* (une trace temporisée par interface localisée représentant une vision locale du système distribué en question). Nous considérons une approche du test des systèmes distribués dans le contexte de la relation de conformité distribuée appelée *dtioco*. La conformité globale peut être testée dans une architecture de test distribuée en utilisant uniquement des testeurs locaux sans aucune communication entre eux. Nous proposons un algorithme pour vérifier la politique de communication pout un tuple de traces temporisées en formulant le problème de message-passing en termes de problème de satisfaction de contraintes (CSP). Nous avons mis en œuvre le calcul des verdicts de test en orchestrant à la fois les algorithmes du test off-line de chacun des composants et la vérification des communications par le biais d'un solveur de contraintes. Enfin, nous avons validé notre approche sur un cas étude de taille significative.

Mots clés : Test à base de modèles, Test distribué, Systèmes de transition symboliques temporisés à entrée sortie, Test off-line, Problème de satisfaction de contraintes, Test à base de contraintes.

List of Figures

List of Tables

List of Abbreviations

SDL	Specification and Description Language
FIFO	First In First Out
LTS	Labelled Transition System
FSM	Finite State Machine
SMT	Satisfiability Modulo Theories
TLC	Train Local Controller
TCS	Train Control System
MBT	Model-Based Testing
SE	Symbolic Execution
PC	Path Condition
EC	Execution Context
ST	Symbolic Tree
TPC	Time Path Condition
DPC	Data Path Condition
CSP	Constraint Satisfaction Problem
TIOSTS	Timed Input/Output Symbolic Transition System
IOSTS	Input/Output Symbolic Transition Systems
TIOLTS	Timed Input/Output Labelled Transition Systems
SUT	System Under Test
DUT	Distributed System Under Test
ioco	Input/Output Conformance Relation
tioco	Timed/Input Output Conformance Relation
dtioco	Distributed Timed Input/Output Conformance Relation
DFS	Depth First Search
BFS	Breadth First Search
RFS	Random First Search
HoJ	Hit-Or-Jump
DS	Distributed System
LAN	Local Area Network
WAN	Wide Area Network
CDO	Correct Distributed Observation
FDO	Faulty Distributed Observation
RTC	Round-Trip Communication
UML	Unified Modeling Language
CPU	Central Processing Unit
VM	Virtual Machine
xLIA	eXecutable Language for Interaction and Assemblage
API	Application Programming Interface

DOM Document Object Model

Contents

Chapter 1

Introduction

Contents

1.1 Motivation

Distributed systems [65, 20, 83] consist of a number of independent subsystems running concurrently on different machines that interact with each other through communication networks to meet a common goal. In other words, the subsystems are autonomous, i.e., they possess full control over their parts at any time and have to take into account that they are being used by other subsystems and have to react properly to requests.

Distributed systems are challenging to implement correctly because they must handle concurrency and failure. Messages can be delayed, duplicated or reordered. Coordination and resource sharing can be difficult if proper protocols or policies are not in place. In addition, most of them are real time and hence they display a less deterministic global behavior than centralized systems. The complexity of distributed systems and their inherent concurrency leads to a complex design and implementation that must address both communication scheduling and computation. Generally, in these systems there is no global clock which can schedule distributed events i.e., each localized subsystem has its own local clock and the delays in the message communications or even the occurrence order of the events are unknown. These systems also exhibit concurrency, in which the timing of events in the system can affect the output results.

Because of the complexity of distributed systems and with the aim to prevent a faulty behavior, testing and verifying distributed systems are paramount in order for them to behave as expected, however, the issues described above make testing of these systems very hard to accomplish [80].

A system can be tested at different levels in its development process and abstraction. Based on the degree of visibility of the system's implementation, there is black-box testing and white-box testing [68]. Black-box testing assumes only access to the interface of the system's implementation and not its code. On the other hand, white-box testing assumes that tests are derived based on the internal details of the system. Between the two previous extreme situations, the degree of visibility between these two can vary, leading to grey-box testing [49]. In the context of black-box testing, we only have the specification of the System Under Test (SUT) from which all the expected behaviors can be derived and that provides the information to build the test scenarios. When the specification is described by a formal model, we are in the domain of Model-Based Testing (MBT).

MBT is a software testing technique in which the test cases are derived from a model that describes the functional aspects of the SUT [90]. This technique usually means functional testing for which the test specification is given as a test model. The test model is derived from the system requirements and it describes how user actions and system states relate to each other. The MBT process comprises four steps: (a) Modeling the expected behavior of the system; (b) Generation of a set of test cases; (c) Execution of the test cases on the implementation and (d) Checking of the test results to detect differences between the SUT and the specification using a mathematical conformance relation. Input/Output Conformance Relation (*ioco*) was the first conformance relation to be considered in MBT [86, 87] for reactive systems as one of the most established relation. Then Timed/Input Output Conformance Relation (*tioco*) was introduced by Krichen and Tripakis in [60] in the context of MBT of real-time reactive systems as *ioco* does not consider timed systems. In [37] Distributed Timed Input/Output Conformance Relation (*dtioco*) was introduced for testing timed distributed systems.

With the goal to analyse the consistency of distributed system's implementation against it specification model, we focus our attention on the fourth step of the MBT process that we call the *oracle problem*. Dealing with the other steps of MBT process is left as future work.

Moreover, the difficulty within distributed systems is to define a global coherent time which schedules all local events. Several works have defined clock synchronization mechanisms [56, 41] to solve the problem of ordering events in a distributed system. However using these approaches often is difficult to implement in practice and may require a significant amount of computational resources, in particular memory. To overcome this issue, logical clocks were first introduced by Lamport [65] as a concept to schedule events in a distributed system. Later, Fidge [33] then Mattern [66] enhanced this concept to produce a global ordering of distributed events which corresponds to a real scheduling. In the same context, Hierons *et al.* in [44] and Gaston *et al.* in [37] treated causality of events for solving the oracle problem in a distributed system in a similar way to the one used by Lamport, Fidge and Mattern. However, the authors of [44, 37] do not explicitly ground their approach on logical clocks. Yet, the problem addressed by their approaches concerns more the issue of finding an order; which would make a group of observations of such an execution on different remote interfaces; the witness of a correct global distributed system execution.

Our work revolves around the following thesis statement:
"The focus of my thesis is to design and develop a testing methodology and architecture for distributed systems focusing on the oracle problem in order to check the consistency of a

distributed system execution".

More precisely, we address the following issue:

"When considering a distributed testing architecture where a separate tester is localized on each subsystem of the system, how consistency check of the global view of the distributed system behavior can be realized?"

The observations made hitherto lead us to the following scope which drives the work described in this thesis.

1.2 Thesis Scope and Contributions

1.2.1 Scope of the Thesis

In this thesis, a Distributed System (DS) can be defined as a tuple of communicating subsystems with no global clock but only local clocks for each local subsystem. In this context, an observation made of a distributed system can be seen as a tuple of so-called *timed traces*: one timed trace to describe the behavior of each subsystem. Testers are able to measure durations between the communication actions of the timed traces and time units are identical for all clocks with no clock drift. We might not synchronise the instant at which testers start and end observing. However, each local tester start observing when its associated localized sub-system is reset.

We consider the SUT as black-box system which means that we do not have knowledge about its internals, thus, we can only rely on its observable inputs proposed by the environment and the outputs produced by it.

Let us suppose that we are testing from a model $\mathcal{M}$. Our aim is to formally reason about the correctness of a concrete SUT $\mathcal{S}$ w.r.t its specification model $\mathcal{M}$. We focus on the problem of producing an automated solution to the oracle problem. In our work, the oracle problem is the problem of checking that an observation made by a distributed system (here a tuple of timed traces) is the one which is allowed by a model $\mathcal{M}$. In the case of a distributed system, a model $\mathcal{M}$ carries out the local specification of local subsystems together with the communication pattern of the distributed system.

1.2.2 Research Approach and Contributions

To address the needs described above and to make testing of distributed systems easier we have been developing in this dissertation an orchestration framework that is able to solve the oracle problem in distributed testing and provides an automated solution to the oracle problem. The verdicts resulting from checking conformance of the distributed SUT against its distributed specification are produced according to the *dtioco* conformance relation [37, 12]. Furthermore, our framework carries out the following two activities: (1) the problem of checking each local observation against its corresponding local model; and (2) checking that the tuple of observations respects a valid communication pattern. In particular, this dissertation involves a combination of the following contributions:

- We adapt the offline testing algorithm for verdict computation given in [7]. Indeed, in [7], the authors proposed a centralized offline testing approach (from test case generation to verdict computation), which provides an algorithm for verdict computation based on *tioco* conformance relation [60, 61]. The work of [7] cannot be used directly for our goal since it cannot be used for our system semantics. Moreover, our centralized offline testing algorithm will not consider test purposes as the one presented in [7].

- We propose a constraint-based algorithm for solving the oracle problem for multicast communications in a timed setting. In other words, we propose an algorithm to check that a tuple of observations represents a valid communications pattern. This algorithm expresses the communication policy as a Constraint Satisfaction Problem (CSP): it constructs a set of constraints that can be satisfied if and only if the given tuple of local observations has a valid communication pattern. Therefore, a standard constraint solver can be used to solve this problem. In other words, we characterize the set of possible synchronizations in a symbolic manner by constructing constraints that carry on symbolic durations occurring in local observations.

- We implement an orchestration framework combining the two following activities: (1) we analyze tuple of observations from the communication perspective by executing our proposed algorithm to check communication in terms of CSP and also (2) we analyze each observation of the tuple with respect to its associated local model by executing our centralized off-line testing algorithm.

- In order to validate our tooling, we propose an approach for generating distributed observations. The approach consists of two random generation algorithms: *Correct Distributed Observations* (CDOs) and *Faulty Distributed Observations* (FDOs). A fault injection technique is used to generate an FDO. Both CDOs and FDOs will be submitted to our testing framework in order to observe corresponding testing verdicts. A CDO must never cause a fail verdict whereas an FDO may cause fail verdicts.

1.3 Thesis Outline

In accordance with the contributions we specified above, we structure our thesis as follows:

- Chapter 2: provides a preliminary definition of data structures regarding the formal definition of observation that can be made in distributed testing and describes system models. Indeed, in this thesis, we use Timed Input Output Symbolic Transition Systems to model the expected behavior of a distributed system.

- Chapter 3: presents centralized model-based testing and provides an algorithm for solving the oracle problem in the context of local conformance testing.

- Chapter 4: reviews the state of the art relevant to the context of distributed testing and describes our distributed testing architecture. This chapter provides an algorithm for checking the oracle problem, i.e., analysing a tuple of localized traces with regard to a communication policy, in terms of CSP. We also discuss implementation issues.

- Subsequently, Chapter 5, is dedicated to the validation of our implementation framework and the evaluation of the scalability of our approach with regard to the soundness of our algorithms. An experimentation of our testing approach on a real-sized case study of a telecommunication distributed system is given as an illustration.
- Chapter 6: emphasizes the contribution of our work and identifies challenges and research gaps that require further exploration.

1.4 Publications

A portion of our work has been published in the following conference paper:

- Nassim Benharrat, Christophe Gaston, Robert M Hierons, Arnault Lapitre, and Pascale Le Gall. Constraint-based oracles for timed distributed systems. In IFIP International Conference on Testing Software and Systems, pages 276-292. Springer, 2017.

Chapter 2

Formal Background

Contents

In this chapter, we introduce our formal preliminaries. We start in Section 2.1 by defining the classical typed equational logic whose syntactic part will be used later as a mean to define data in the TIOSTS formalism. In Section 2.2, we recall some information about CSP together with related tools, i.e, constraint solvers. We present the syntax of the TIOSTS formalism and give their semantics in Section 2.3. We conclude the chapter by providing in Section 2.4 the definitions related to Symbolic Execution techniques for TIOSTS.

2.1 Typed Equational Logic

We use classical typed equational logic to represent and reason about data. The typed equational logic is a restriction of the logic of the first-order predicates in meaning that the only predicate used is equality ($=$). The typed logic consists in partitioning the data according to a finite set of types S.

For two sets A and B, we use the notation B^A to denote the set of applications from A to B. For all sets A_i with $i \in \{1, \dots n\}$, the notation $\coprod_{i\in\{1,\dots n\}} A_i$ denotes the disjoint union of the sets $A_1 \dots A_n$. The notation $\mathbb{R}_{\geq}$ (resp. $\mathbb{R}_{>}$) denotes the set of (resp. strictly) positive real numbers. For a set A, A^* denotes the set of all finite sequences of elements of A, ε denotes the empty sequence and the symbol '.' is used for concatenation.

We start by presenting the syntax of typed equational logic by introducing the concepts of *signature*, first-order *terms* and $formulas$.

Definition 2.1 (Signature)**.** *A signature is a couple $\Omega = (S, Op)$ where S is a set of type names and Op is a set of function names provided with a profile $s_1 \dots s_{n-1} \to s_n$ in S^+.*

A function name of the form f which is associated with a profile $s_1 \dots s_{n-1} \to s_n$ is denoted $f : s_1 \dots s_{n-1} \to s_n$ and represents a function taking $n-1$ arguments of types $s_1 \dots s_{n-1}$ and computing a value of type s_n. A function name of the form $f :\to s$ denotes a constant value of type s.

In the sequel, we suppose that S contains primitive types such as *Integer* and *Real* and a particular type *Boolean* to denote two possible truth values: *true* and $false$.

Example 2.1 (Primitive real number data type)**.** *As an example of a signature, we introduce $\Omega_{Real} = (S_{Real}, Op_{Real})$ which is associated with the specification of real numbers arithmetic with:*

- $S_{Real} = \{Real, Boolean\}$
- $Op_{Real} = \{0 :\to Real, 1 :\to Real$
  $true :\to Boolean,$
  $false :\to Boolean,$
  $+ : Real.Real \to Real,\ (addition)$
  $- : Real.Real \to Real,\ (subtraction)$
  $* : Real.Real \to Real,\ (multiplication)$
  $\leq : Real.Real \to Boolean\ (inequality\ "equal\ to\ or\ less\ than").$
  $< : Real.Real \to Boolean\ (inequality\ "strictly\ less\ than").$
  $\dots\}$

A set V of so-called *variables* typed in S is a set of the form $\coprod_{s \in S} V_s$. The function $type : V \to S$ is the function that associates the type s to the variable x if and only if $x \in V_s$.

The set of terms $\mathcal{T}_\Omega(V)$ and formulas $\mathcal{F}_\Omega(V)$ over V are defined as follows:

Definition 2.2 (Term and Formula)**.** *Let $V = \coprod_{s \in S} V_s$ be a set of typed variables in S. The set of Ω-terms with variables in V is denoted $\mathcal{T}_\Omega(V) = \coprod_{s \in S} \mathcal{T}_\Omega(V)_s$ and is inductively defined as follows:*

- *if $x \in V_s$ then $x \in \mathcal{T}_\Omega(V)_s$*
- *if f has a profile $s_1 \dots s_{n-1} \to s_n$ (with $n > 0$) and $(t_1 \dots t_{n-1}) \in \mathcal{T}_\Omega(V)_{s_1} \times \dots \times \mathcal{T}_\Omega(V)_{s_{n-1}}$ then $f(t_1 \dots t_{n-1}) \in \mathcal{T}_\Omega(V)_{s_n}$ (with for the particular case $n = 0$, $f \in \mathcal{T}_\Omega(V)_{s_0}$)*

The set of typed equational Ω-formulas over V is denoted $\mathcal{F}_\Omega(V)$ and is inductively defined as follows:

- *$True$ and $False$ are in $\mathcal{F}_\Omega(V)$.*
- *for any $s \in S$, for any t and t' in $\mathcal{T}_\Omega(V)_s$, we have $t = t'$ is in $\mathcal{F}_\Omega(V)$*

- *for any* φ_1 *and* φ_2 *in* $\mathcal{F}_\Omega(V)$, *we have* $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\neg\varphi_1$ *are in* $\mathcal{F}_\Omega(V)$

The function *type* is extended canonically to $\mathcal{T}_\Omega(V)$ as usual.

Example 2.2. *Using the signature* $\Omega_{Real} = (S_{Real}, Op_{Real})$, *we consider the variable names typed in* S_{Real}, $V = V_{Boolean} \coprod V_{Real}$ *where* $V_{Boolean} = \emptyset$ *and* $V_{Real} = \{x, y\}$. *The following are terms in* $\mathcal{T}_\Omega(V)$:

0, x, y, $+(0,0)$ *and* $+(x,y)$, $-(x,y)$ *and* $*(x,y)$ *are in* $\mathcal{T}_\Omega(V)_{Real}$ *and term* $<(x,y)$ *is in* $\mathcal{T}_\Omega(V)_{Boolean}$. *For terms with two operands, we often use the infix notation, e.g.* $x + y$ *instead of* $+(x,y)$.

We may define the following formulas in $\mathcal{F}_\Omega(V)$: $x = y, \neg(0 = x), < (x,y) =\leq (x+0, y+0)$.

A substitution over V is an application $\rho : V \to \mathcal{T}_\Omega(V)$ preserving types. The identity substitution over V is denoted id_V. Any substitution ρ in $\mathcal{T}_\Omega(V)^V$ may be extended canonically to the set of terms and formulas as usual.

Example 2.3. *Consider the signature* Ω_{Real} *and the set of variables* $V_{Real} = \{x, y\}$. *We define the following substitutions* $\rho : V_{Real} \to \mathcal{T}_\Omega(V)$ *such that:*
$\rho(x) = x + 1$ *and* $\rho(y) = y + 1$. *We have for example* $\rho(x + y) = (x + 1) + (y + 1)$.

As we treat the question of testing timed reactive systems [61, 7], we consider signatures Ω that include a particular type *time* in S to denote values representing durations, provided with usual operations $<, +$ such that $<$: *time.time* $\to$ *Boolean* and $+$: *time.time* $\to$ *time*. Type *time* is the restriction of the type *Real* to positive real numbers. Variables of type *time* are named *clocks* and variables of any type $s \in S \backslash \{time\}$ are named *data variables*

A model associated with a signature is a mathematical structure used to interpret all symbols of the signature.

Definition 2.3 (Model)**.** *A* Ω*-model is a set* $M = \coprod_{s \in S} M_s$ *provided with a function* $f_M : M_{s_1} \times \cdots \times M_{s_{n-1}} \to M_{s_n}$ *for each* $f : s_1 \ldots s_{n-1} \to s_n$ *in Op.*

The set M_{time} is denoted D (for the set of durations) and is isomorphic to the set of positive real numbers $\mathbb{R}_{\geq}$. D^+ is the set of strictly positive durations is isomorphic to the set of strictly positive real numbers $\mathbb{R}_{>}$. D^+ is provided with usual operations: $+ : D^+ \times D^+ \to D^+$; and $<, \leq$: $D^+ \times D^+$ which have their usual meanings in $\mathbb{R}_{>}$.

To give semantical meaning to variables, we introduce the notion of interpretation. An interpretation is an application $\nu : V \to M$ preserving types.

Any interpretation ν in M^V may be extended canonically to the set of terms and formulas as usual. Now we define the notion of *formula satisfaction.*

Definition 2.4 (Formula satisfaction)**.** *For any interpretation* $\nu \in M^V$ *and a formula* $\varphi \in \mathcal{F}_\Omega(V)$, *we say that the interpretation* ν *satisfies the formula* φ *denoted* $M \models_\nu \varphi$ *if and only if:*

- ***Boolean values:*** *we have* $M \models_\nu True$ *and* $M \not\models_\nu False$,
- ***Equality:*** *if* φ *is of the form* $t = t'$ *with* t, t' *in* $\mathcal{T}_\Omega(V)_s$ *for any type* s *in* S, *we have* $\nu(t) = \nu(t')$,

- ***Conjunction*** *if φ is of the form $\varphi_1 \wedge \varphi_2$ with φ_1, φ_2 in $\mathcal{F}_\Omega(V)$, we have $M \models_\nu \varphi_1$ and $M \models_\nu \varphi_2$,*
- ***Disjunction*** *if φ is of the form $\varphi_1 \vee \varphi_2$ with φ_1, φ_2 in $\mathcal{F}_\Omega(V)$ we have $M \models_\nu \varphi_1$ or $M \models_\nu \varphi_2$,*
- ***Negation:*** *if φ is of the form $\neg\psi$ with ψ in $\mathcal{F}_\Omega(V)$, we have $M \not\models_\nu \psi$,*

Notation 2.1. *A formula φ in $\mathcal{F}_\Omega(V)$ is said to be satisfiable if there exists an interpretation ν in M^V such that $M \models_\nu \varphi$. We use the function $IsSat : \mathcal{F}_\Omega(V) \to \{True, False\}$ such that $IsSat(\varphi)$ returns $True$ if and only if φ is satisfiable.*

2.2 Solving Constraints

In the following, given a formula $\varphi \in \mathcal{F}_\Omega(V)$, $Var(\varphi)$ denotes the set of all variables occurring in φ. When a formula φ in $\mathcal{F}_\Omega(V)$ is satisfiable, we use the notation $Sat(\varphi)$ to represent a solution of the satisfaction problem for φ, that is an interpretation $\nu' \in M^{Var(\varphi)}$ satisfying:

There exists ν in M^V such that $M \models_\nu \varphi$ and $\forall x \in Var(\varphi), \nu'(x) = \nu(x)$ (by construction, we have $M \models_{\nu'} \varphi$).

At the tooling level, we use usual Satisfiability Modulo Theories (SMT)-solvers [13, 11] like CVC4 [8], Z3 [25] and Yices [29] where it is possible to setup the adequate typed equational logic in the aim to interpret typed variables and check satisfiability of formula built over those variables. In practice, most of SMT-solvers implement typed equational logic with primitive types such as *Real*, *Integer* and *Boolean*. For example we use the set of positive real numbers $\mathbb{R}_{\geq}$ to check satisfiability of constraints built over variables in D.

Example 2.4. *Let $\Omega = (S, Op)$ be a signature and V be a set of variables. We define $V_{time} = \{d_0, d_1\}$ and we give the formula $\varphi = (d_0 > d_1) \wedge (d_1 + 2 > d_0 + 1)$ in $\mathcal{F}_\Omega(V_{time})$ We have $Var(\varphi) = \{d_0, d_1\}$. We check satisfiability of φ using a standard SMT-solver like Yices [29]. We have $IsSat(\varphi)$ is $True$, indeed, there exists an interpretation ν in D^V such that $\nu(d_0) = 1$ and $\nu(d_1) = 1/2$ and we have $D \models_\nu \varphi$. We have then, $Sat(\varphi)$ may denote the interpretation $[d_0 \mapsto 1, d_1 \mapsto 1/2]$.*

In the sequel, we suppose the existence of a signature $\Omega = (S, Op)$ and a model M.

2.3 Timed Input Output Symbolic Transition Systems (TIOSTS)

In this section, we present the TIOSTS specification formalism. TIOSTS [32, 6, 7] are symbolic timed automata employed to specify the behavior of reactive timed systems. That is, open systems whose behavior depends on external stimuli (inputs from the environment) and where time to produce an output is as important as the output produced itself. TIOSTS are extensions of so-called Input/Output Symbolic Transition Systems (IOSTS) [31, 36] introducing constraints over execution delays of transitions.

2.3.1 Syntax

TIOSTSs are defined over *TIOSTS-signatures* which are used to introduce particular variables whose valuations define abstractly a state of the system. A TIOSTS-signature also introduces a set of *channels names* to communicate with the environment.

Definition 2.5 (TIOSTS-signature)**.** *A TIOSTS-signature is defined as a triple* $\Sigma = (A, T, C)$ *where:*

- $A = \coprod_{s \in S \setminus \{time\}} A_s$ *is a set of data variables where for all* $s \in S \setminus \{time\}$ *we have* $A_s \subseteq V_s$*;*
- $T \subseteq V_{time}$ *is a set of clocks;*
- $C = \coprod_{s \in S \setminus \{time\}} C_s$ *is a set of so-called channels provided with a type* s *in* $S \setminus \{time\}$ *and where any* C_s *can be partitioned as* $C_s^{in} \coprod C_s^{out}$ *where* C_s^{in} *is a set of input channels of type* s *and* C_s^{out} *is a set of output channels of type* s*.* $C^{in} = \coprod_{s \in S \setminus \{time\}} C_s^{in}$ *is the set of all input channels and* $C^{out} = \coprod_{s \in S \setminus \{time\}} C_s^{out}$ *is the set of all output channels.*

Elements of A are used to store input values, to denote system state evolutions and to define guards. Clocks are used to denote durations between occurrences of receptions and emissions of values through channels. Those durations may be constrained by defining guards over those clocks. As in the case of terms and variables, we use a function $type : C \to S \setminus \{time\}$ associating channels with their types.

We now define so-called *communication actions* over typed channels. Communication actions can be inputs or outputs sent or received through channels.

Definition 2.6 (Communication actions)**.** *Let* $\Sigma = (A, T, C)$ *be a TIOSTS-signature. The set of communication actions over* Σ *is defined as* $Act(\Sigma) = I(\Sigma) \cup O(\Sigma)$ *where:*

- $I(\Sigma) = \{c?x \mid c \in C^{in}, x \in A_{type(c)}\}$
- $O(\Sigma) = \{c!t \mid c \in C^{out}, t \in \mathcal{T}_\Omega(A)_{type(c)}\}$

Elements of $I(\Sigma)$ and $O(\Sigma)$ are called inputs and outputs respectively. In order to simplify the exposition, at the level of our modeling framework, we consider messages that contain only a single piece of data. However, at the tooling level, without adding any particular difficulties, messages may contain 0 (signals $c!$ or $c?$), 1 or n data ($c!(t_1, \ldots, t_n)$ or $c?(x_1, \ldots, x_n)$, the x_i being different variables of A).

TIOSTSs are structures composed of a set of *states*, an *initial state* and *labeled transitions* going from one state to another. Those latter transitions are composed of *data guards* and *time guards* which are conditions to be satisfied on data variables and time variables respectively in order to execute the transition; *communication actions* introduced in Definition 2.6 and *substitutions* representing modifications on both time and data variables when firing the transition.

Definition 2.7 (TIOSTS)**.** *Let* $\Sigma = (A, T, C)$ *be a TIOSTS-signature. A TIOSTS over* Σ *is a triple* (Q, q_0, Tr)*, where:*

- Q *is a set of states,*
- $q_0 \in Q$ *is the initial state,*
- Tr *is a set of labeled transitions of the form* $(q, \mathbb{T}, \phi_t, \phi_d, act, \rho, q')$ *where:*
  - $q, q' \in Q$,

- $\mathbb{T} \subseteq T$,
- $\phi_t \in \mathcal{F}_\Omega(T)$,
- $\phi_d \in \mathcal{F}_\Omega(A)$,
- $act \in Act(\Sigma)$,
- $\rho : A \to \mathcal{T}_\Omega(A)$ *is a substitution.*

For a transition defined by the tuple $(q, \mathbb{T}, \phi_t, \phi_d, act, \rho, q')$, q (resp. q') is the source (resp. target) state of the transition. ϕ_t and ϕ_d are firing conditions respectively on clocks and data variables. ϕ_t is called time guard ϕ_d is called data guard. $\mathbb{T} \subseteq T$ is a set of clocks (to be reset to 0 when the transition is executed). Values assigned to variables occurring in $\mathbb{T}$ are updated implicitly and refer to the instant of occurrence of act. act is a communication action and ρ assigns new values to data variables in A when the transition is executed in order to represent state evolutions.

Notation 2.2. *In the following, for any TIOSTS $\mathbb{G}$ of the form (Q, q_0, Tr) defined over a TIOSTS-signature $\Sigma = (A, T, C)$, we use the notations $states(\mathbb{G})$, $init(\mathbb{G})$ and $Trans(\mathbb{G})$ to refer to Q, q_0 and Tr. In the same way, for any transition tr in $Trans(\mathbb{G})$ of the form $(q, \mathbb{T}, \phi_t, \phi_d, act, \rho, q')$ we use the notations $source(tr)$, $target(tr)$, $clocks(tr)$, $\phi_t(tr)$, $\phi_d(tr)$, $act(tr)$ and $\rho(tr)$ to refer to $q, q', \mathbb{T}, \phi_t, \phi_d, act$ and ρ respectively.*

In the sequel, in particular, when considering TIOSTS examples, in order to depict a transition of the form $(q, \mathbb{T}, \phi_t, \phi_d, act, \rho, q')$ we use the graphical convention:

$$q \xrightarrow{\mathbb{T}\ [\phi_t]\ [\phi_d]\ act\ \rho} q'$$

When there are no new substitutions, it corresponds to the identity function id_A (that is, the variables are substituted by themselves) and we omit it in the depiction. When there are no necessary conditions for firing a transition (either a data condition or a time condition), this corresponds to the fact that the guards are $True$, in this case, we omit the guards.

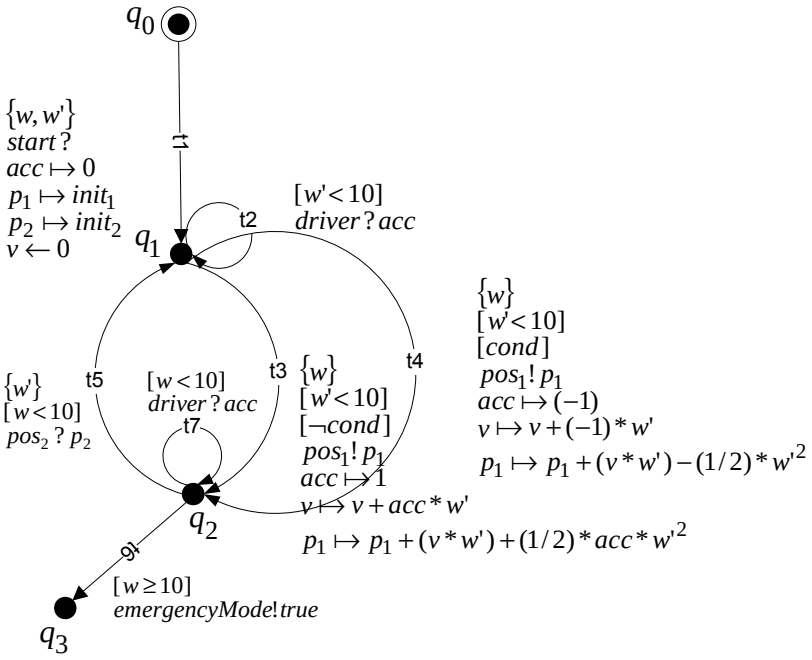
Example 2.5 (Train Local Controller)**.** *In the remaining of this chapter, we use a toy example for illustration. Train Local Controller (TLC) [37] is a system designed to manage safety by monitoring the location of a central train by ensuring that the train automatically decreases its speed when safety is threatened. The TLC system is specified by a TIOSTS $\mathbb{G}_{TLC}$ containing 4 states (q_0 the initial state, q_1 , q_2 and q_3) as depicted in Figure 2.1.*

$\mathbb{G}_{TLC}$ is defined over the TIOSTS-signature $\Sigma_{TLC} = (A, T, C)$ where:

- *$A = \{acc, v, p_1, p_2\}$ is a set containing 4 data variables: acc whose associated values are in the set $\{-1, 0, 1\}$, $acc = 0$ means that the central train does not accelerate. $acc = 1$ (respectively $acc = -1$) means that the central train increases (respectively decreases) its speed. v is used to store the speed of the central train and p_1 for storing the position of the train and p_2 for storing the position (received form the environment) of another train which may have a symmetric role as the one modeled by $\mathbb{G}_{TLC}$;*
- *$T = \{w, w'\}$ is a set containing 2 clocks: w, which is reset at each emission of the position p_1 and w', which is reset at each reception of the position p_2;*

- $C = \{start, driver, pos_1, pos_2, emergencyMode\}$ *is the set of channels through which data variables are communicating.*

$\mathbb{G}_{TLC}$ *specifies the following behavior: After an initialization phase (transition* t_1*), the central train sends its position* p_1 *to the environment (indeterministic behavior illustrated through transitions* t_3 *and* t_4*), and in return, it is supposed to receive* p_2 *(the environment) which is an estimation of another train's position that has a symmetric role as the one modeled by* $\mathbb{G}_{TLC}$ *(transition* t_5*). In this loop, two consecutive communication actions are supposed to be separated by a delay of less than 10 time units. If the estimated position* p_2 *is not received on time, the central train goes into an emergency mode (not detailed here) (transition* t_6*). At any moment in the loop, the driver may ask to modify the train acceleration (transitions* t_2 *and* t_7*). The new value is taken into account only if it does not affect the safety of the system (transition* t_3*). Safety is threatened if the condition named cond holds, that is, the distance between trains is less than the distance that can be covered by the train with the current acceleration. If safety is threatened, then the acceleration of the central train is set to -1 in order to reduce its speed (transition* t_4*).*



With: $init_1 = 42$, $init_2 = 300$ and $cond \equiv (p_1 < p_2) \wedge (p_2 \leq (v * 20) + 200))$

Figure 2.1: Train Local Controller TIOSTS

2.3.2 Semantics

TIOSTSs specify sequences of actions separated by numeric durations. Those sequences are called *timed traces*. In [62, 7] authors propose to accept any possible decomposition of durations in timed traces. For example, the duration 0.7 may be decomposed as the sum of delays 0.4 and 0.3 since $0.7 = 0.4 + 0.3$. It may also be represented as $0.1 + 0.1 + 0.1 + 0.4$, etc. In order to take into account all such durations, authors in [62, 7] define timed traces as the set of all sequences obtained by applying arbitrary decompositions

of durations. In [81] Schmaltz *et al.* raises the issue of timed trace normalization and mentioned that if there can be two successive delays in a timed trace, e.g., $\sigma = i?.d_1.d_2.o!$, it would be more natural to normalize to timed traces with no consecutive delays. Hence $\sigma = i?.d.o!$ with d a duration is a normalized timed trace such that $d = d_1 + d_2$. Hence, it is possible to associate to each timed trace a normalized one.

In this thesis, we deal with timed traces in a way that considers durations in a normalized way [81, 14]. This choice is a consequence of the way we represent durations and communication actions in Chapter 4, where we deal with distributed system semantics. Therefore the remaining of this section is a reformulation of the semantics of TIOSTS as presented in [7], in order to only consider normalized timed traces.

We introduce a definition to represent communication actions semantically.

Definition 2.8 (Concrete actions)**.** *Let $C = C^{in} \coprod C^{out}$ be a set of channels. The set of concrete actions over C is defined as $Act(C) = I(C) \cup O(C)$ where:*

- $I(C) = \{c?v \mid c \in C^{in}, v \in M_{type(c)}\}$
- $O(C) = \{c!v \mid c \in C^{out}, v \in M_{type(c)}\}$

The value v is the interpretation of the received or emitted terms.

Concrete actions are values exchanged through channels. Variable interpretations are canonically extended to symbolic actions ($\nu(c?x) = c?(\nu(x))$ and $\nu(c!t) = c!\nu(t)$).

Notation 2.3. *Given $act \in Act(C)$ of the form $c\Delta v$ with $\Delta \in \{!, ?\}$, $chan(act)$ refers to c, $\overline{act}$ refers to its so-called mirror action, $c\overline{\Delta}v$ with $\overline{!} = ?$ and $\overline{?} = !$.*

A concrete action is generally observed after a delay has occurred since the previous occurrence of a concrete action. This is captured by the notion of *concrete events*. When one cannot observe an action, following [87], we use the symbol 'δ' used to denote the absence of observation of a concrete action (*i.e.* quiescence).

Definition 2.9 (Concrete events)**.** *The set of* concrete events *over C is defined as $Evt(C) = (D^+ \cup \{\_\}) \times (Act(C) \cup \{\delta\})$. The set of* initialised concrete events *over C is defined as $IEvt(C) = D^+ \times (Act(C) \cup \{\delta\})$.*

Roughly speaking, events are structures representing the observation of an emission or a reception where the variables and terms present in the communication actions are interpreted in a model M after waiting for a (measured) non-null delay.

An initialised event of the form (d, a) is an observation of concrete action a after that a tester observes a positive delay d. In fact, in a centralized testing framework, a tester is supposed to measure duration from the initial instant or between two consecutive actions. Sometimes, it is not possible to observe such a common initial instant. Therefore, we define uninitialised events in which its duration cannot not be observed. Hence, symbol '_' denotes the absence of the observation of a delay (i.e $(\_, a)$) so that the observation needs not be stamped with a strictly positive duration. In addition, between two consecutive concrete actions, we require that the delay is greater than zero so that two events do not occur simultaneously.

Let us point out that usually, in a pure timed framework, the symbol δ is used to represent quiescence of a system may be useless (e.g. [58, 57, 37]). Here, the use of δ is a side effect of considering atomic actions as events. Indeed, expressing that a system is quiescent after a duration d has to be representable as an event, and thus, we need a symbol to represent these quiescent situations as a couple (d, δ).

Notation 2.4. *Given $ev \in Evt(C)$, we let $act(ev) = a$ and $delay(ev) = d$ if $ev = (d, a)$ with $d \in D^+$, else $delay(ev) = 0$ (i.e, $ev = (\_, a)$).*

Example 2.6. *Consider the signature Σ_{TLC} of TLC system illustrated in Example 2.5. The pair $ev = (6, pos_2?50)$ is a concrete initialised event where 6 is the time measured before the observation of the concrete action $pos_2?50$ and 50 is the interpretation value of the received position through channel pos_2 in the latter action. We have $delay(ev) = 6$ and $act(ev) = pos_2?50$.*

In the sequel $\delta Evt(C)$ (resp. $\delta IEvt(C)$) denotes the set of *unobservable events* (resp. unobservable initialized events) $\{ev \mid ev \in Evt(C), act(ev) = \delta\}$ (resp. $\{ev \mid ev \in IEvt(C), act(ev) = \delta\}$).

We now define *concrete timed traces* as sequences of events:

Definition 2.10 (Concrete timed traces)**.** *The* set $ITraces(C)$ of initialised timed traces over C *is* $(IEvt(C) \setminus \delta Evt(C))^*.(\varepsilon + \delta Evt(C)) + \delta IEvt(C)$.
The set $UTraces(C)$ of uninitialised timed traces over C *is* $\{u(\sigma) \mid \sigma \in ITraces(C)\}$ *where* $u(\sigma)$ *denotes* ε *if* $\sigma = \varepsilon$ *and* $(\_, a).\sigma'$ *if* σ *is of the form* $(d, a).\sigma'$.
The set $TTraces(C)$ of timed traces over C *is* $UTraces(C) \cup ITraces(C)$.

Any event of an initialised timed trace contains a duration and a concrete action. For the first event, this duration represents a delay between some distinguished moment (e.g. since the time at which a tester started to measure the duration) and the first observed action. Uninitialised traces are timed traces for which no initial instant is identified. Finally, note that quiescence is only observed at the end of traces, when no communication action has been observed.

Example 2.7. *This is an example of a concrete timed trace built over the signature Σ_{TLC} defined in Example 2.5.*

$\sigma = (1, start?).(3, driver?1).(3.pos_1!42).(12, emergencyMode!true)$ *denotes the following behavior: one waits for 1 time unit and enters initialization phase, the driver asks to modify the central train's acceleration and receives value 1 after waiting for 3 time units. The system waits for 3 time units and sends position 42 to the environment. The central train waits for the environment to send its position (here the environment may be represented by another train which has a symmetric role as the central one). However this action is not performed on time, hence, after waiting for more than 10 time units (here 12 time units), the train goes into an emergency mode.*

Timed traces of a TIOSTS are built from sequences of transitions. We start by introducing so-called *snapshots* representing numeric states before and after the transition execution.

Definition 2.11 (Snapshots)**.** *Let $\mathbb{G} = (Q, q_0, Tr)$ be a TIOSTS over Σ. The set of all snapshots of $\mathbb{G}$, denoted $Snp_M(\mathbb{G})$ is the set $Q \times M^{A \cup T}$.*

A snapshot characterizes a given numeric state of $\mathbb{G}$, that is supposed to be reached after some executions.

Definition 2.12 (Runs of transitions)**.** *Let $\mathbb{G} = (Q, q_0, Tr)$ be a TIOSTS over Σ. For a transition $tr = (q, \mathbb{T}, \phi_t, \phi_d, act, \rho, q') \in Tr$, the set of runs of tr, denoted as $Runs(tr) \subseteq Snp_M(\mathbb{G}) \times Evt(C) \times Snp_M(\mathbb{G})$ is defined as the set of triple $((q, \nu_i), ev, (q', \nu_f))$ such that $M \models_{\nu_i} \phi_d$ and there exists and intermediate interpretation $\xi \in M^{A \cup T}$ verifying:*

- *if act is of the form $c!t$, then for all $z \in A$ we have $\xi(x) = \nu_i(x)$.*
- *if act is of the form $c?x$, then all $z \in A \backslash \{x\}$ we have $\xi(z) = \nu_i(z)$,*
- *for all $\omega \in T$, $\xi(\omega) = \nu_i(\omega) + delay(ev)$ and $M \models_\xi \phi_t$ and such that for all $z \in A$ we have $\nu_f(z) = \xi \circ \rho(z)$, for all $\omega \in T \setminus \mathbb{T}$ we have $\nu_f(\omega) = \xi(\omega)$ and for all $\omega \in \mathbb{T}$ we have $\nu_f(\omega) = 0$. Finally $act(ev) = \xi(act(tr))$.*

A run of a transition is simply a triple that is defined by a snapshot denoting the numeric state before executing the transition; a concrete event associated with the firing of the transition; and finally a snapshot denoting the numeric state after the transition execution.

In Definition 2.12, ξ is an intermediate interpretation whose purpose is to let time pass from initial interpretation ν_i for all clocks ($\xi(w) = \nu_i(w) + delay(ev)$) and take into account a potential input value (denoted by $\xi(x)$ if $act = c?x$). Data guards of the transition should be satisfied by initial interpretation ν_i. Time guards of the transition should be satisfied by ξ and if it is the case then the transition can be fired resulting on a final interpretation ν_f updating data variables according to ρ and resetting clocks occurring in $\mathbb{T}$.

Example 2.8. *Let us consider the TIOSTS $\mathbb{G}_{TLC}$ depicted in Figure 2.1 and defined over the signature $\Sigma_{TLC} = (A, T, C)$ in Example 2.5. We recall that Σ_{TLC} is defined as follows:*

- $A = \{acc, v, p_1, p_2\}$
- $T = \{w, w'\}$
- $C = \{start, driver, emergencyMode, pos_1, pos_2\}$

Consider the transition t_2 as illustrated in $\mathbb{G}_{TLC}$:

$$q_1 \xrightarrow{[w'<10]\ driver?acc} q_1$$

We give a possible run r of the transition t_2 as follows:

$$(q_1, \nu_i) \xrightarrow{(8, driver?1)} (q_1, \nu_f)$$

where ν_i and ν_f are defined as follows:

Interpretation of variables
$\nu_i(acc) = 0, \nu_i(v) = 0, \nu_i(p) = 42, \nu_i(p') = 300, \nu_i(w) = 0, \nu_i(w') = 0$
$\xi(acc) = 1, \xi(v) = 0, \xi(p) = 42, \xi(p') = 300, \xi(w) = 8, \xi(w') = 8$
$\nu_f(acc) = 1, \nu_f(v) = 0, \nu_f(p) = 42, \nu_f(p') = 300, \nu_f(w) = 0, \nu_f(w') = 0$

Notation 2.5. *The set of runs of all transitions of $\mathbb{G}$, denoted $Runs(\mathbb{G})$, is in the set $Snp_M(\mathbb{G}) \times Evt(C) \times Snp_M(\mathbb{G})$. For any run r in $Runs(\mathbb{G})$ of the form $((q_i, \nu_i), ev, (q_f, \nu_f))$ we use the notations $source(r)$, $target(r)$ and $event(r)$ to refer to $(q_i, \nu_i), (q_f, \nu_f)$ and ev respectively.*

The paths of TIOSTS are finite sequences of consecutive transitions whose first transition starts at the initial state of the considered TIOSTS. We define the notion of *TIOSTS paths* as follows:

Definition 2.13 (Paths of a TIOSTS)**.** *Let $\mathbb{G} = (Q, q_0, Tr)$ be a TIOSTS defined over a TIOSTS-signature $\Sigma = (A, T, C)$. The set of paths of $\mathbb{G}$ denoted $Paths(\mathbb{G}) \subseteq Tr^*$ is the set which contains the empty sequence ε and all finite sequences $tr_1 \dots tr_n$ such that:*

- $source(tr_1) = q_0$
- $target(tr_i) = source(tr_{i+1})$ *for all* $i < n$.

Example 2.9. *Consider the TIOSTS $\mathbb{G}_{TLC}$ depicted in Figure 2.1 and defined over the signature $\Sigma = (A, T, C)$ in Example 2.5. A possible path p from $\mathbb{G}_{TLC}$ is:*

$$q_0 \xrightarrow[\substack{acc\mapsto 0\\ v\mapsto 0\\ p_1\mapsto 42\\ p_2\mapsto 300}]{start?} q_1.q_1 \xrightarrow[\substack{acc\mapsto 1\\ v\mapsto v+acc*w'\\ p_1\mapsto p_1+(v*w')+(1/2)*acc*w'^2}]{[w'<10]\ [\neg cond]\ pos_1!p_1} q_2.q_2 \xrightarrow{[w<10]\ pos_2?p_2} q_1$$

The semantics associated with a finite path is defined by the semantics that is given to the transitions that compose the path.

Definition 2.14 (Timed traces from a path of TIOSTS)**.** *Let $\mathbb{G} = (Q, q_0, Tr)$ be a TIOSTS defined over a TIOSTS-signature $\Sigma = (A, T, C)$. For a path p of $\mathbb{G}$, the* set of initialized timed traces of *p, denoted $ITraces(p)$ is defined as follows:*

- *$ITraces(p)$ is $\{\varepsilon\}$ if $p = \varepsilon$*
- *if p is of the form $tr_1.\cdots.tr_n$, $ITraces(p)$ contains all sequences of events $ev_1 \cdots ev_n$ such that there exists a sequence of runs $r_1 \cdots r_n$ satisfying: for all $i \leq n$, r_i is a run of tr_i of the form $(snp_i, ev_i, snp'_{i+1})$ and for all $j < n$ we have $snp'_j = snp_{j+1}$ and such that all events are initialised.*

The set of uninitialized timed traces of *p, denoted $UTraces(p)$ is the set $\{u(\sigma) \mid \sigma \in ITraces(p)\}$ where $u(\sigma)$ denotes ε if $\sigma = \varepsilon$ and $(\_, a).\sigma'$ if σ is of the form $(d, a).\sigma'$ with $d \in D^+$.*
The set of timed traces of *p, denoted $TTraces(p)$ is $ITraces(p) \cup UTraces(p)$.*

Example 2.10. *Consider the TIOSTS $\mathbb{G}_{TLC}$ depicted in Figure 2.1 and defined over the signature $\Sigma_{TLC} = (A, T, C)$ in Example 2.5. Consider the path p illustrated in Example 2.9:*

$$q_0 \xrightarrow[\substack{acc\mapsto 0\\ v\mapsto 0\\ p_1\mapsto 42\\ p_2\mapsto 300}]{start?} q_1.q_1 \xrightarrow[\substack{acc\mapsto 1\\ v\mapsto v+acc*w'\\ p_1\mapsto p+(v*w')+(1/2)*acc*w'^2}]{[w'<10]\ [\neg cond]\ pos_1!p_1} q_2.q_2 \xrightarrow{[w<10]\ pos_2?p_2} q_1$$

A possible initialized timed trace of p is: $\sigma = (1, start?).(3, pos_1!42).(5, pos_2?300)$. When one cannot observe the initial delay of σ, we have $\sigma = (\_, start?).(3, pos_1!42).(5, pos_2?300)$ is uninitialized timed trace of p. Both σ and σ' are defined in $TTraces(p)$.

Finally, the behaviors of a TIOSTS, also called its semantics, are defined as the set of all the timed traces that can be obtained from its paths.

Definition 2.15 (Timed traces of a TIOSTS)**.** *Let $\mathbb{G} = (Q, q_0, Tr)$ be a TIOSTS defined over a TIOSTS-signature $\Sigma = (A, T, C)$. The set of* initialized timed traces *of $\mathbb{G}$, denoted $TTraces(\mathbb{G})$, is defined as follows:*

- *For all $p \in Paths(\mathbb{G})$ we have $UTraces(p) \subseteq TTraces(\mathbb{G})$,*
- *For all $\sigma \in TTraces(\mathbb{G})$ such that there exists no path p and no event ev with $act(ev) \in O(C)$ satisfying $\sigma.ev \in UTraces(p)$, for all $d \in D^+$ we have $\sigma.(d, \delta) \in TTraces(\mathbb{G})$ if $\sigma \neq \varepsilon$ and $(\_, \delta) \in TTraces(\mathbb{G})$ if $\sigma = \varepsilon$.*
- *For all $\sigma.(d, a) \in TTraces(\mathbb{G})$ for all $d' \in D^+$ with $d' \leq d$ we have $\sigma.(d', \delta) \in TTraces(\mathbb{G})$.*

Timed traces of a TIOSTS are possible successions of events that are couples of observed delays before that a concrete action generated form a TIOSTS has occurred. Note that a succession of events of $Evt(C)$ of the form $ev_1 \cdots ev_n$ such that all events are initialised except for $i = 1$, (i.e, ev_1) is of form $(\_, a_1)$ and for all $i > 1$, ev_i is of form (d_i, a_i) define a unique way to represent *a timed trace* of $\mathbb{G}$.

Example 2.11. *Consider the TIOSTS $\mathbb{G}_{TLC}$ depicted in Figure 2.1 and defined over the signature Σ_{TLC} in Example 2.5. Some possible timed traces that can be generated from $\mathbb{G}_{TLC}$ are: $\sigma = (\_, start?).(3, pos_1!42).(5, pos_2?300)$, and $\sigma' = (\_, start?).(3, pos_1!42).(5, \delta)$. σ denotes the following behavior: one cannot observe the initial moment before entering the initialization phase, the system waits for 3 time units and sends position 42 to the environment. Finally, the central train waits for 5 time units before that the environment sends position 300 to it (here 300 is the position of another train which has a symmetric role as the central one). σ' denotes a prefix of the behavior of σ (without the last action) as the tester may not be able to observe the last action.*

2.4 Symbolic Execution

2.4.1 Symbolic Execution of Programs

Symbolic Execution (SE) was initially defined for programs [53, 22, 53, 54]. Its main usage is to analyze the feasibility of the executions paths of the program of interest. From a practical perspective, it consists in executing the program, not for actual input values, but for symbolic input parameters. The goal of this execution is to characterize constraints (called path conditions) on those input parameters for each execution path of the program. Any actual input satisfying a path condition will lead to a program execution following the path associated with the path condition. Of course, if the path condition is not satisfiable this means that its associated path is not executable.

A Path Condition (PC) is a condition on the input symbols of the program such that a path is feasible if and only if its PC is satisfiable.

Exploration of paths of a program leads to the construction of a Symbolic Tree (ST). Each path of the tree is obtained by following a sequence of instruction. Each time a new

instruction is added to the sequence, the SE process builds a structure whose purpose is to represent, as an abstract memory state, the possible values associated to the variables handled in the program. This structure, called symbolic state, is composed of:

- A control point, often denoted as a number, associated with the instruction to be processed;
- Symbolic values of the program variables in the current state (after having executed the instruction);
- The PC computed to reach the current state.

Example 2.12 (Symbolic execution of a program)**.** *Let us consider the program depicted in Figure 2.2 which computes the absolute value of an input variable x. We are interested in the instructions enumerated from* 1 *to* 6 *in this program.*

```
0  int absoluteVal(int x){
1      int y;
2      if(x≥ 0)
3          y:=x;
4      else
5          y:=-x;
6      return (y);
7  }
```

Figure 2.2: A program computing the absolute value of a variable

The construction of the symbolic tree of the program depicted in Figure 2.3 is performed as follows:

- *the initial symbolic state: is composed of the instruction number* (1)*, of the variables of the program x and y which are initialized with the symbolic variables x_0 and y_0 respectively, and the PC is set to $True$ since there is no condition on the variables;*
- *instruction* (2) *is executed symbolically. The associated symbolic state contains the number of instruction* 2*, state variables have not changed their values and the PC is still at $True$. As this is a conditional instruction, there are two possibilities and therefore two outgoing transitions.*
- *when $(x \geq 0)$ is considered as verified, the instruction $y := x$ is executed symbolically. The associated symbolic state is built with the instruction number* (3)*, the value of y is updated, and receives x_0 (the symbolic value of x), and the PC is updated to $x_0 \geq 0$ which is the condition for which this branch can be chosen;*
- *the instruction* (6) *is executed symbolically, the value of y which is x_0 is returned. The symbolic state contains the number of the instruction, the symbolic values of x and y which are both x_0. The PC is the same as that of the previous state;*
- *when $(y \geq 0)$ is not true, the instruction $y := -x$ is executed symbolically. The associated symbolic state is built with the instruction number* (4)*, the new value of y becomes $-x_0$ and the symbolic value of x does not change. The PC has been updated since the condition for which this path can be taken is $\neg(x_0 \geq 0)$;*

- *the instruction* (6) *is executed symbolically, the value of* y *which is* $-x_0$ *is returned. The symbolic state contains the number of the instruction, the symbolic values of* x *and* y *which are respectively* x_0 *and* $-x_0$*. The PC is still at* $\neg(x_0 \geq 0)$*.*

The symbolic tree obtained considers all the possible paths of the program. Each PC obtained at a leaf of the tree provides the necessary condition to take the path that ends by this leaf. In our case, if the input value represented by the symbolic value x_o *is positive (it satisfies the PC of the left leaf) then the left path is taken, if it is negative (it satisfies the PC of the right leaf) then it is the path on the right which is taken. The two PCs are complementary, in the sense that an input value of* x *cannot satisfy at the same time the two PCs associated with the two paths.*

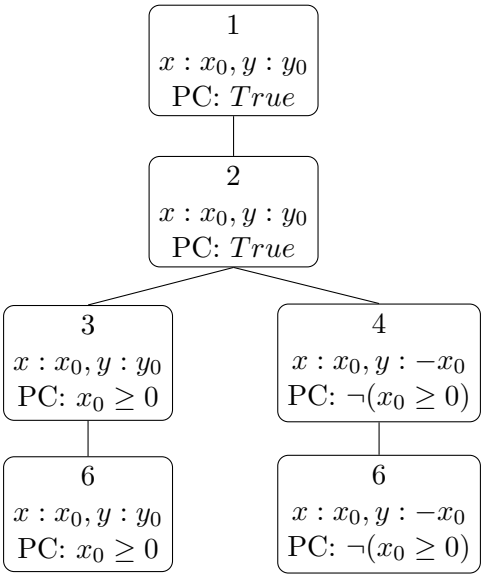


Figure 2.3: Symbolic tree of the program computing absolute value of a variable

Symbolic execution of programs was used for program verification [19, 16], program debugging [53, 54] and symbolic model-checking [50]. It is also used in the test cases generation from programs [18, 75].

2.4.2 Symbolic Execution of TIOSTS

Symbolic execution techniques have been extended to modeling formalisms, essentially to perform MBT. These formalisms are essentially used to the description of automata, as for example the IOSTS [38]. The main difference between SE of programs and SE of IOSTS is that the executions for IOSTSs are no longer reduced to couples (input of a program/output of the program) but must symbolically denote sequences of interactions (i.e inputs and outputs) between the SUT and the user because IOSTS automata denote reactive systems. Then SE techniques have been extended to TIOSTS [32, 7] where symbolic treatment of time variables is added.

In order to represent symbolic values of a TIOSTS, we define so-called *symbolic states*. For this purpose, we suppose that a set of so-called *fresh variables* $F = \bigcup_{s \in S} F_s$ is given. $F_t \subseteq F$ denotes the set of fresh variables of type *time*. $F_d = F \backslash F_t$ denotes the set of data

(or non-time) fresh variables.

In the sequel, a TIOSTS $\mathbb{G} = (Q, q_0, Tr)$ defined over a TIOSTS-signature $\Sigma = (A, T, C)$ is supposed given. *Symbolic states* are structures to store useful information characterizing constraints related to executions.

Definition 2.16 (Symbolic states)**.** *A symbolic state of* $\mathbb{G}$ *is a quadruple* $(q, \pi_t, \pi_d, \lambda)$ *where* $q \in Q$, $\pi_t \in \mathcal{F}_\Omega(F_t)$, $\pi_d \in \mathcal{F}_\Omega(F_d)$ *and* λ *is a substitution defined as* $\lambda : A \cup T \to \mathcal{T}_\Omega(F)$ *preserving types.*

q denotes the state reached after the execution leading to η, π_t is a constraint on durations between communication actions called Time Path Condition (TPC) and π_d is a constraint on symbolic data values called Data Path Condition (DPC). Both constraints (TPC and DPC) must be satisfied for the symbolic execution to reach symbolic state η. λ denotes the current terms (built over symbolic values) assigned to data variables of A and time variables of T.

Notation 2.6. *In the sequel,* Σ_F *stands for the TIOSTS-signature* (F_d, F_t, C)*. For any* $\lambda : A \cup T \to \mathcal{T}_\Omega(F)$ *we also note* $\lambda : \mathcal{T}_\Omega(A \cup T) \to \mathcal{T}_\Omega(F)$ *and* $\lambda : \mathcal{F}_\Omega(A \cup T) \to \mathcal{F}_\Omega(F)$ *its canonical extensions respectively to terms and formulas. We also note* $\lambda : Act(\Sigma) \to Act(\Sigma_F)$ *its extension to communication actions defined as* $\lambda(c?x) = c?\lambda(x)$, $\lambda(c!t) = c!\lambda(t)$ *and* $\lambda(\delta) = \delta$*.*

Notation 2.7. *We note* $\mathcal{S}(\mathbb{G})$ *the set of all symbolic states of* $\mathbb{G}$*. Let us introduce symbolic state* $Init = (q_0, True, True, \lambda_0) \in \mathcal{S}(\mathbb{G})$ *denoted by* $Init(\mathbb{G})$ *such that for all* $x, y \in A$ *with* $x \neq y$, $\lambda_0(x)$ *and* $\lambda_0(y)$ *are distinct fresh variables and forall* $z \in T$, $\lambda_0(z) = 0$ [1] *in* F*. For any symbolic state* $\eta = (q, \pi_t, \pi_d, \lambda) \in \mathcal{S}(\mathbb{G})$, $q(\eta)$, $\pi_t(\eta)$,$\pi_d(\eta)$ *and* $\lambda(\eta)$ *stand respectively for* q, π_t, π_d *and* λ*.*

In order to reason symbolically about events, we introduce the following definition.

Definition 2.17 (Symbolic Event)**.** *Let* $\Sigma_F = (F_d, F_t, C)$ *be a TIOSTS-signature.*
The set of symbolic events *over* Σ_F *is defined as* $Evt(\Sigma_F) = (F_t \cup \{\_\}) \times (Act(\Sigma_F) \cup \{\delta\})$*.*
The set of initialised symbolic events *over* Σ_F *is defined as* $IEvt(\Sigma_F) = F_t \times (Act(\Sigma_F) \cup \{\delta\})$*.*

Similarly to the way we defined TIOSTS semantics by starting to define runs of a transition, symbolic execution of a TIOSTS is based on the symbolic execution of a transition.

Definition 2.18 (Symbolic Execution of transitions)**.** *Let* $tr = (q, \mathbb{T}, \phi_t, \phi_d, act, \rho, q')$ *be a transition in* Tr *and* $\eta = (q, \pi_t, \pi_d, \lambda)$ *be a symbolic state in* $\mathcal{S}(\mathbb{G})$*. Let us define* λ^i *as:*

- *if act is of the form* $c?x$, $\lambda^i(x) = y$ *with* y *a new fresh variable in* F_d *and* $\forall z \in A, z \neq x, \lambda^i(z) = \lambda(z)$*, else (act is not of the form* $c?x$*)* $\forall x \in A, \lambda^i(x) = \lambda(x)$*;*
- $\forall \omega \in T, \lambda^i(\omega) = \lambda(\omega) + z$ *where* z *is a new fresh time variable in* F_t*.*

The symbolic execution $\mathcal{SE}_\eta(tr)$ *of* tr *from* η *is the symbolic transition* $st = (\eta, ev_s, \eta') \in \mathcal{S}(\mathbb{G}) \times Evt(\Sigma_F) \times \mathcal{S}(\mathbb{G})$ *with* ev_s *is a symbolic event in* $Evt(\Sigma_F)$ *where:*

- $act(ev_s) = \lambda^i(act)$
- $delay(ev_s) = z$

[1] *Init* is unique up to different fresh variable renaming in F.

- $\eta' = (q', \pi'_t, \pi'_d, \lambda')$ *where:*
  - $\pi'_t = \pi_t \wedge \lambda^i(\phi_t)$ *and* $\pi'_d = \pi_d \wedge \lambda(\phi_d)$
  - $\forall \omega \in \mathbb{T}, \lambda'(\omega) = 0$*;* $\forall \omega \in T \backslash \mathbb{T}, \lambda'(\omega) = \lambda^i(\omega)$*;* $\forall x \in A, \lambda'(x) = \lambda^i(\rho(x))$.

A symbolic execution of a transition tr is an event of the form (d_s, act_s) where d_s is a new fresh time variable (i.e. not used in the definition of $\mathbb{G}$) used to represent durations (they are typed as clocks) and each act_s is of the form $c?z$ or $c!t$ where z is a new fresh variable and t is a term built over the same equational logic signature as terms of $\mathbb{G}$ but on a set of new fresh variables.

Notation 2.8. *Given a symbolic transition* $st = (\eta, ev_s, \eta')$*, the variable* $delay(st)$ *is called the symbolic delay of* st *and is defined as* $delay(ev_s)$[2]*. The notations* $source(st)$*,* $event(st)$ *and* $target(st)$ *stand respectively for* η*,* ev_S *and* η'*. We note* $Fresh(st) = \{delay(st)\}$ *if* $act \in O(\Sigma)$ *and* $Fresh(st) = \{delay(st), \lambda^i(y)\}$ *if* act *is of form* $c?y$*.* $Fresh(st)$ *is called the set of fresh variables of symbolic transition* st*.*

The symbolic execution associated with a TIOSTS is then defined simply by executing exactly once all executable transitions from all symbolic states.

Definition 2.19 (Symbolic Execution of TIOSTS)**.** *A symbolic execution of a TIOSTS* $\mathbb{G} = (Q, q_0, Tr)$ *is a couple* $\mathcal{SE}(\mathbb{G}) = (Init, ST)$ *where:*

- $Init = (q_0, True, True, \lambda_0)$ *is such that* $\forall x \in T, \lambda_0(x) = 0$ *and* $\forall x, y \in A \cup T$*, with* $x \neq y$*, we have:* $\lambda_0(x)$*,* $\lambda_0(y)$ *in* F *with* $\lambda_0(x) \neq \lambda_0(y)$ *;*
- ST *is the set of all symbolic executions of all transitions* tr *occurring in* Tr *from any* $\eta \in \mathcal{S}(\mathbb{G})$ *such that* $q(\eta) = source(tr)$ *and* $Fresh(st) \cap \lambda_0(A \cup T) = \emptyset$*. Moreover for any two distinct symbolic transitions* st_1 *and* st_2 *in* ST*,* $Fresh(st_1) \cap Fresh(st_2) = \emptyset$*.*

$\mathcal{SE}(\mathbb{G})$ is a tree-like structure whose nodes are symbolic states used to capture information related to the possible executions of $\mathbb{G}$. Notice that all paths in $\mathcal{SE}(\mathbb{G})$ denote in an abstract way all possible executions of $\mathbb{G}$. At the beginning of the execution, there is no constraint on time and on data as it is signified by the two occurrences of $True$ respectively for TPC and DPC in the symbolic state $Init$. Always concerning the initial state, time variables are initialized to the null duration 0 and all other variables are initialized with fresh variables of F.

Example 2.13 (SE of TIOSTS $\mathbb{G}$ from Example 2.5)**.** *Figure 2.4 depicts the symbolic execution tree of TIOSTS* $\mathbb{G}_{TLC}$ *of the TLC system as depicted in Figure 2.1. Notice that the symbolic tree is cut after symbolic states* η_2*,* η_4*,* η_5 *and* η_6 *where we revisit states* q_1*,* q_2*,* q_1 *and* q_1 *respectively. Once, the TLC starts executing, the driver asks to receive the acceleration value (from the environment). Notice that in the branching state symbolic* η_1*, depending on the value of variables* p_1 *and* p_2 *there are two possibilities:*

- *if* $p_1 < p_2$ *then the central branch is taken*
- *if* $p_1 \geq p_2$ *then the right branch is taken*

From η_1*, the driver may always ask for the acceleration from the environment and then the system loops into* η_1*, in this case, the left branch is taken.*

[2] Given $st = (\eta, ev_s, \eta')$ a symbolic transition, we have $delay(st) = 0$ if η is the symbolic state $Init$

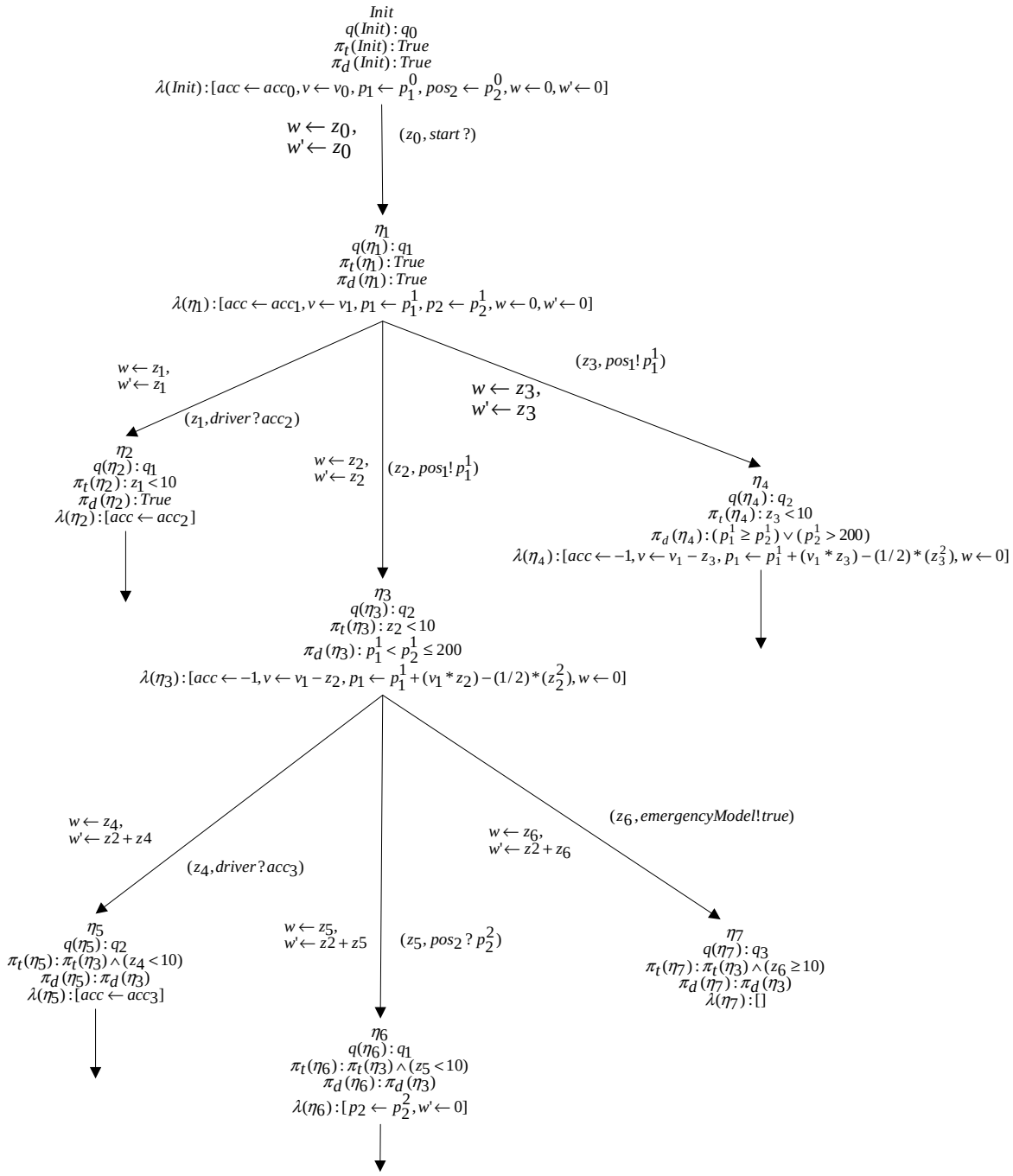


Figure 2.4: Symbolic tree produced by SE of TIOSTS $\mathbb{G}_{TLC}$

In order to deal with quiescence, we complete symbolic execution of TIOSTS by new transitions. Until now, transitions of TIOSTS carry events with actions that are necessary inputs or outputs. In order to manipulate quiescence inside symbolic execution trees, we consider the symbol "δ" as a special action modeling the absence of reaction.

The quiescence situation can be observed only in the case of a truncated trace: indeed, as long as a trace is not interrupted, the trace is only a sequence of events, each one consisting of a duration and an action. As soon as the trace is interrupted, it can be

interrupted just at the end of an event, that is just after an action, and thus the trace ends with an action, or else, the trace can be interrupted during an event itself. This last case corresponds to the observation of a duration, without the observation of an action, this duration being shorter than the duration associated with the interrupted event. This explanation emphasizes that the quiescence can be positioned, associated with a duration, only at the end of a trace. The symbolic quiescence transitions will thus appear only at the end of the paths of a symbolic execution tree, with as target states, new states, which will be ending states by nature, generically denoted as η^δ, with new control states of the form q^δ.

Notation 2.9. *In the sequel:*

- *for all $q \in Q$, let us note $Trans(q)$ the set of all transitions tr verifying $source(tr) = q$ and $React(q)$ the set all transitions tr verifying $source(tr) = q$ and $act(tr) \in O(\Sigma)$;*
- *for all $\eta \in \mathcal{S}(\mathbb{G})$, let us note $Trans(\eta)$ the set of all transitions st of ST verifying $source(st) = \eta$ and $React(\eta)$ the set of all transitions st verifying $source(st) = \eta$ and $act(st) \in O(\Sigma_F)$.*

Next definition shows how to complete a symbolic tree with transitions reflecting quiescence and time passing.

Definition 2.20 (Symbolic Execution of TIOSTS with quiescence enrichment)**.** *The quiescence and time passing enrichment of $\mathcal{SE}(\mathbb{G})$ denoted $\mathcal{SE}(\mathbb{G})_\delta$ is the couple $(Init, ST \cup ST_\delta)$ where ST_δ is defined as follows:*

- ***time based quiescence*** *(1): Let us note $\pi^\delta_{t_1}(\eta)$ the formula in $\mathcal{F}_\Omega(F_t)$ restricted to $True$ if $Trans(q(\eta)) = \emptyset$ and equal to $\bigvee_{str \in Trans(\eta)}(z < delay(str)) \wedge \pi_t(target(str)))$ otherwise.*

  Then, $(\eta, (z, \delta), \eta^\delta_t) \in ST_\delta$ with $\eta^\delta_t = (q^\delta_{t_1}, \pi_t(\eta) \wedge \pi^\delta_{t_1}(\eta), \pi_d(\eta), prog_z(\lambda(\eta)))$ where for $\lambda : A \cup T \to \mathcal{F}_\Omega(F)$, $prog_z(\lambda) : A \cup T \to \mathcal{F}_\Omega(F \cup \{z\})$ is defined by $\forall x \in A, prog_z(\lambda)(x) = \lambda(x)$ and $\forall cl \in T, prog_z(\lambda(cl)) = \lambda(cl) + z$. Here, z stands for an additionnal time variable that has to be considered for making $\pi^\delta_{t_1}(\eta)$ satisfiable;

- ***time based quiescence*** *(2): Let us note $\pi^\delta_{t_2}(\eta)$ the formula in $\mathcal{F}_\Omega(F_t)$ restricted to $True$ if $React(q(\eta)) = \emptyset$ and equal to $\bigwedge_{str \in React(\eta)} \forall delay(str), \neg\phi_t(tr)$ otherwise.*

  Then, $(\eta, (z, \delta), \eta^\delta_t) \in ST_\delta$ with with z a new fresh variable in F_t and $\eta^\delta_t = (q^\delta_{t_2}, \pi_t(\eta) \wedge \pi^\delta_{t_2}(\eta), \pi_d(\eta), prog_z(\lambda(\eta)))$.

- ***data based quiescence:*** *Let us note $\pi^\delta_d(\eta)$ the formula in $\mathcal{F}_\Omega(F_d)$ restricted to $True$ if $React(\eta) = \emptyset$ and equal to $\bigwedge_{str \in React(\eta)} \neg\pi_d(target(str))$ otherwise.*

  Then $(\eta, (z, \delta), \eta^\delta_d) \in ST_\delta$ with z a new fresh variable in F_t and $\eta^\delta_d = (q^\delta_d, \pi_t(\eta), \pi_d(\eta) \wedge \pi^\delta_d(\eta), prog_z(\lambda(\eta)))$.

Time based quiescence corresponds to two kinds of situations.

- The first kind of situations expresses that time-based quiescence transitions can be executed if whenever we might wait for a positive symbolic duration before an action (input or output) occurs then we wait for a shorter positive symbolic duration. By noting that $delay(str)$ is the symbolic delay in F_t associated with the transition str,

we have that the constraint $\pi^{\delta}_{t_1}(\eta)$ states that there exists a transition str of source η for which the condition $(z < delay(str))$ associated to the time path condition $(\pi_t(target(str)))$ is satisfiable with z a new symbolic delay in F_t.

- The second kind of situations expresses that time-based quiescence transitions can be executed if no transition labeled by an output can be executed anymore due to unsatisfiable time constraints. We have that the constraint $\pi^{\delta}_{t_2}(\eta)$ states that for all output transitions str of source η, whatever the delay is, the time path condition $(\pi_t(target(str)))$ cannot be satisfied.

Data based quiescence transitions can be executed only if no transition labelled by an output can be executed anymore due to unsatisfiable data constraints.

Example 2.14 (Quiescence enrichment of $\mathcal{SE}(\mathbb{G})$)**.** *Figure 2.5 depicts the application of the quiescence enrichment on the symbolic state Init of symbolic tree depicted in Figure 2.4. Notice that we add the following quiescence symbolic states:*

- $\eta^{\delta}_{t_1}$ *with condition* $z_1 < z_0$*, indeed,* $\exists str \in Trans(Init)$ *s.t* $str = (Init, (z_0, start?), \eta_1)$ *and* $delay(str) = z_0$
- $\eta^{\delta}_{t_2}$ *and* η^{δ}_{d} *with condition* $True$ *for data and time passing respectively. Indeed,* $React(Init) = \emptyset$*.*

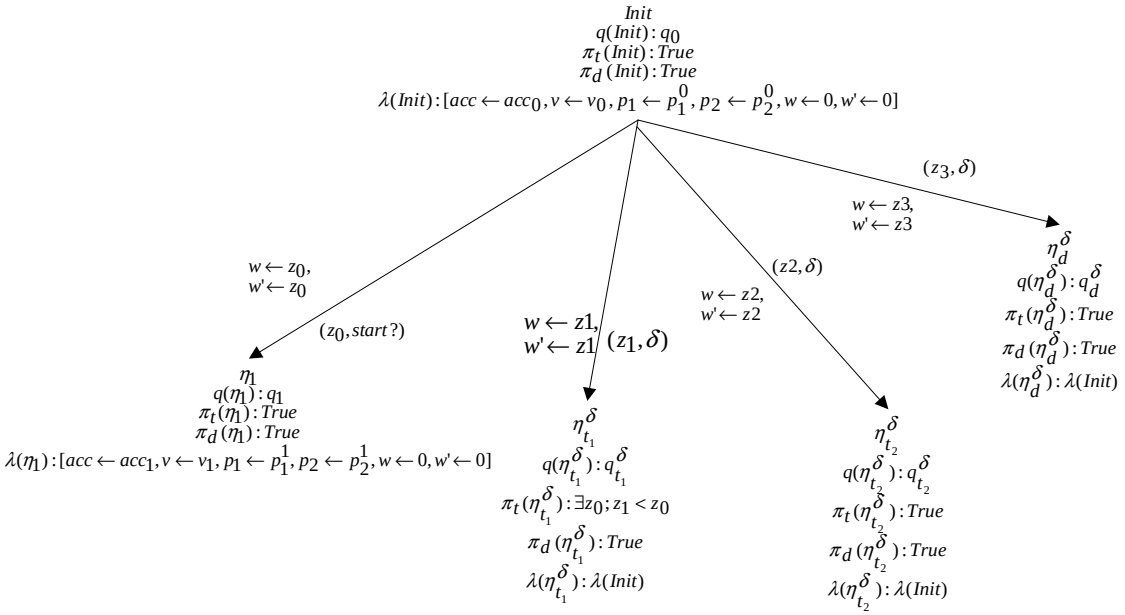


Figure 2.5: SE of symbolic state $Init$ with quiescence enrichment

Figure 2.6 depicts the application of the quiescence enrichment on the symbolic state η_1 *of symbolic tree depicted in Figure 2.4. In this case we have,* $React(Init) = \{pos_1!p_1\}$*, hence, we add the following quiescence symbolic states:*

- $\eta^{\delta}_{t_1}$ *where output* $pos_1!p_1$ *cannot be executed if we wait for a shorter positive duration* $z_3 < z_2$ *where* z_2 *is the symbolic delay before observing* $pos_1!p_1$*.*

- $\eta^{\delta}_{t_2}$ *where output* $pos_1!p_1$ *cannot be executed due to unsatisfiable time constraints* $\neg\pi_t(\eta_3)$
- η^{δ}_{d} *where output* $pos_1!p_1$ *cannot be executed due to unsatisfiable data constraints* $\neg\pi_d(\eta_3)$.

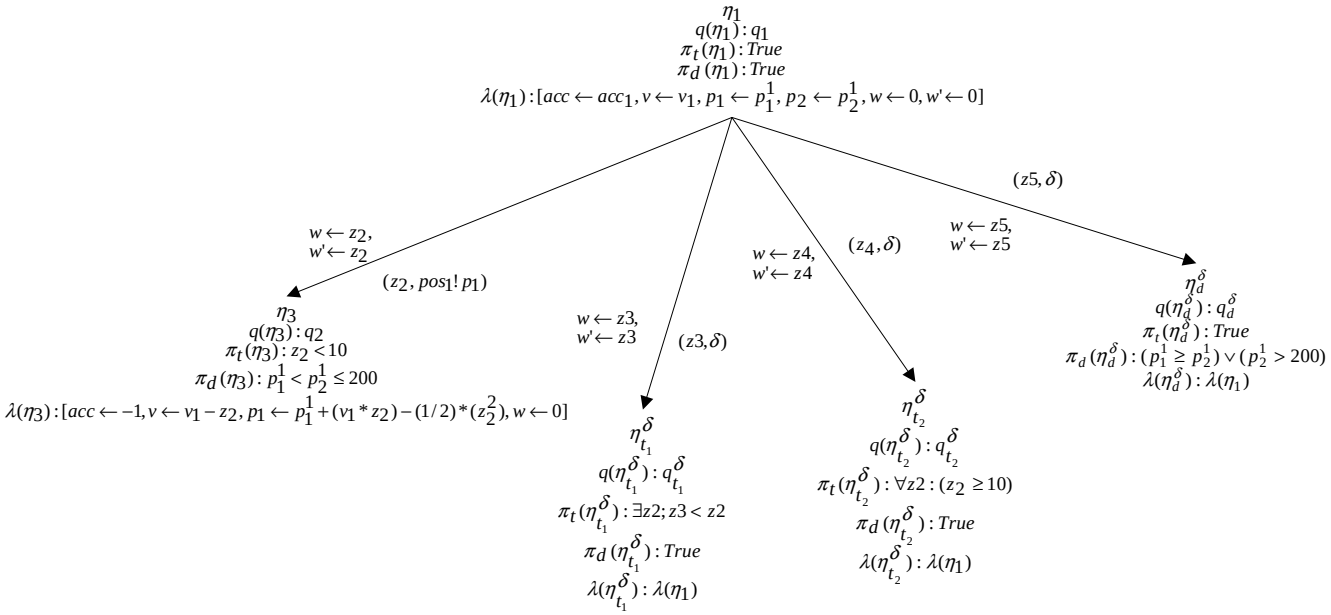


Figure 2.6: SE of symbolic state η_1 with quiescence enrichment

Symbolic execution tree $\mathcal{SE}(\mathbb{G})_\delta$ resulting from the symbolic execution of TIOSTS $\mathbb{G}$ characterizes in a natural way the set of all timed traces of $\mathbb{G}$. To represent timed traces of $\mathcal{SE}(\mathbb{G})_\delta$, we begin by characterizing so-called *symbolic paths* of $\mathcal{SE}(\mathbb{G})_\delta$.

Definition 2.21 (Symbolic Paths from SE of TIOSTS)**.** *Let* $\mathcal{SE}(\mathbb{G})_\delta = (Init, ST)$ *be the symbolic execution tree associated with* $\mathbb{G}$*. The set of symbolic paths of* $\mathcal{SE}(\mathbb{G})_\delta$ *denoted* $Paths(\mathcal{SE}(\mathbb{G})_\delta)$ *is the set which contains the empty sequence* ε *and all finite sequences of symbolic transitions* $st_1 \dots st_n$ *such that:*

- *for all* $i \leq n$, $st_i \in ST$
- $source(st_1) = Init$
- *for all* $j \leq n$, $q(target(st_j)) = q(source(st_{j+1}))$

A symbolic path p_s is a sequence of consecutive edges relating symbolic states and labelled by symbolic events.

Notation 2.10. *For a symbolic path* p_s *in* $Paths(\mathcal{SE}(\mathbb{G})_\delta)$ *of the form we denote* $final(p_s) = target(st_n)$*. By convention* $final(\varepsilon) = Init$*.*

The timed trace semantics for a symbolic execution tree are defined in a natural way. If we solve both data and time path condition of a given path (that is, the path condition of its last symbolic state), one can evaluate all the symbolic actions labeling this path and extract the corresponding timed trace. In general, it is not guaranteed that a given symbolic path p_s defines timed traces, as it depends on its associated data and time path condition.

Property 2.1 (Symbolic Paths Satisfiability). *Let $\mathcal{SE}(\mathbb{G})_\delta = (Init, ST)$ be the symbolic execution tree associated with $\mathbb{G}$. A symbolic path p_s in $Paths(\mathcal{SE}(\mathbb{G})_\delta)$ is satisfiable if there exists an interpretation $\nu \in M^F$ verifying $M \models_\nu \pi_t(final(p_s))$ and $M \models_\nu \pi_d(final(p_s))$. In this case, the notation $\nu(p_s)$ denote the set of all interpretations $\nu \in M^F$ such that $M \models_\nu \pi_t(final(p_s))$ and $M \models_\nu \pi_d(final(p_s))$. In the sequel, the set of all satisfiable paths of $\mathcal{SE}(\mathbb{G})_\delta$ is denoted $SPaths(\mathcal{SE}(\mathbb{G})_\delta)$*

Symbolic paths resulting from symbolic execution of a TIOSTS characterize event sequences called *symbolic initialized timed traces*.

Definition 2.22 (Symbolic initialized timed traces). *Let p_s be a symbolic path in $Paths(\mathcal{SE}(\mathbb{G})_\delta)$ of the form $st_1 \dots st_n$. The symbolic initialized timed trace tr_s associated with p_s is the sequence of symbolic events $ev_1 \dots ev_n \in Evt(\Sigma_F)^*$ accumulated along p_s.*

The set of executions (initialized timed traces) associated to p_s is characterised by the sequence $ev_1 \dots ev_n$ of symbolic events labelling the consecutive edges and by the final symbolic state $final(p_s)$. Each symbolic event of the sequence is of the form (d_i, act_i). Each d_i is a new fresh variable (i.e. not used in the definition of $\mathbb{G}$) used to represent durations (they are typed as clocks) and each act_i is of the form $c?z_i$ or $c!t_i$ where z_i is a new fresh variable and t_i is a term built over the same equational logic signature as terms of $\mathbb{G}$ but on a set of new fresh variables.

In the following, we state definition of so-called *concrete timed traces by interpretation* associated with a given symbolic timed trace of a satisfiable path. A concrete timed trace is the interpretation of a symbolic timed trace obtained by replacing all symbolic events by their interpretation.

Definition 2.23 (Concrete timed traces by interpretation). *Let $\mathcal{SE}(\mathbb{G})_\delta = (Init, ST)$ be a symbolic execution tree of $\mathbb{G}$. Let tr_s a symbolic timed trace of the form $ev_1^s \dots ev_n^s$ associated with its satisfiable symbolic path p_s in $SPaths(\mathcal{SE}(\mathbb{G})_\delta)$. A concrete timed trace get by interpretation from tr_s is the sequence $ev_1 \dots ev_n \in Evt(C)^*$ such that there exists an interpretation $\nu \in \nu(p_s)$ verifying:*

- *$delay(ev_1) = 0$ and $delay(ev_i) = \nu(delay(ev_i^s))$ for all $1 < i \leq n$*
- *$act(ev_i) = \nu(act(ev_i^s))$*

The set $TTraces(p_s)$ denotes the set of all concrete timed traces obtained by interpreting the associated symbolic timed trace of p_s with any interpretation in $\nu(p_s)$. The set of all concrete timed traces of $\mathcal{SE}(\mathbb{G})_\delta$ is

$$TTraces(\mathcal{SE}(\mathbb{G})_\delta) = \bigcup_{p_s \in SPaths(\mathcal{SE}(\mathbb{G})_\delta)} TTraces(p_s)$$

Since $\mathcal{SE}(\mathbb{G})_\delta$ is obtained from the symbolic execution tree of $\mathbb{G}$ we have that $TTraces(\mathcal{SE}(\mathbb{G})_\delta)$ characterizes in a natural way the set of all timed traces of $\mathbb{G}$ (i.e by removing only the unsatisfiable paths from $\mathcal{SE}(\mathbb{G})_\delta$). Finally, since an TIOSTS and its symbolic execution share the same semantics of timed trace, we can either consider a TIOSTS or its symbolic execution when tackling the issue of MBT over TIOSTS.

In this chapter, we have presented the formal concepts on which our work is founded. Chapter 3 tackles, in the context of MBT, the issue of detecting differences between a SUT

which has a single interface of communication and its TIOSTS model in order to decide conformance using a single tester connected to the SUT in question by implementing *tioco* conformance relation.

Chapter 3

Centralized Model-Based Conformance Testing from TIOSTS

Contents

In this chapter, we present main activities presented in the field of centralized model-based conformance testing. Section 3.1 provides a brief state of the art on MBT. We present the off-line approach for performing MBT with TIOSTSs in Section 3.2. In Section 3.2.1, we present the work of [7] that will be adapted in Section 3.2.2 for providing a centralized MBT approach well-suited to our formal background introduced in Chapter 2.

3.1 Model-Based Testing: State of the Art

When we are dealing with black box testing we only assume access to the interface (inputs/outputs) of the system implementation and not to its code. We also assume that we have the specification of the system describing all the expected behaviors of the SUT which can be stimulated via its interface in order to observe how it responses. We may then differentiate between the inputs or stimulus proposed by the environment and the outputs or reactions produced by the system. When the specification is described by a formal model, we are in the domain of MBT. MBT as presented in [90] is the process which evaluates conformance of a SUT w.r.t a reference model automatically. An advantage of using models in testing is that once the models have been built and assuming tools are available, the testing of the system can be performed more quickly and automatically. Another advantage is that, when the system evolves, it is often sufficient to update the model incrementally with the corresponding changes in the system.

3.1.1 Model-Based Testing Process

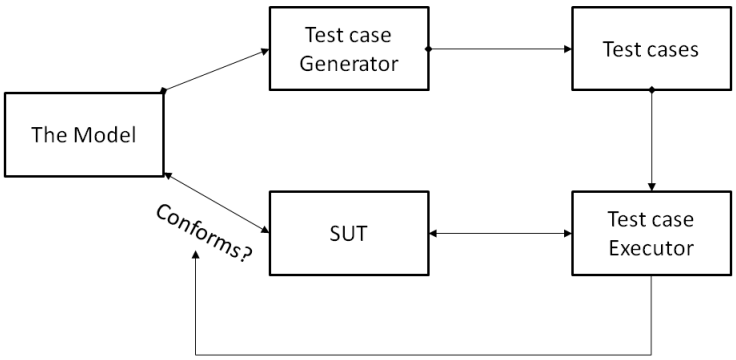


Figure 3.1: Model-Based Testing process

In software testing, there are several techniques to model the application behavior. In other words, there are several notations to describe the behavior of a system as a model. In [30], the authors present the most appropriate notations for testing. Some of these notations are: Labelled Transition System (LTS) [87], Finite State Machine (FSM) [71], Statecharts [42], Unified Modeling Language (UML) [78], Markov chains [48], and Petri nets [76].

Figure 3.1 illustrates a typical model-based testing process which comprises four steps:

- Building the model: The model represents a correct behavior of a system and should be as abstract as possible without missing important information.
- Generating test cases: The model that has been built in the previous activity is used to create the test cases. There are two types of test case generation: Off-line and online test generation.
- Executing the test cases: In order to execute test cases that have been previously generated from the model, we need to translate them into an executable form.
- Checking conformance (also called *solving oracle problem*): After executing the test cases and getting the actual outputs of the system, we have to evaluate and analyze the results, that is, to conclude if the System Under Test behaves as it is expected in the reference model.

Model building: Since the model is a description of the system's behavior, the model should be understandable by all testers, even if they do not have any experience or knowledge in the application domain, or if they do not know what the system performs. Moreover, the model should be precise, clear, and should be presented in a formal way. In [73] Prenninger *et al.* presented different model abstraction techniques that are applied in MBT. Different guidelines that can be used to improve understanding of the SUT to build a coherent model are suggested in [30, 82]. In **Test Case Generation,** the specification model that has been built in the previous activity is used to generate test cases. The tester typically has to specify or provide information about criteria or the purpose of the test cases, the inputs and sometimes the expected outputs of the system. When the SUT is complex, it often means that the number of test cases is very large or even infinite. So, automation makes possible to generate test cases for complex systems such as distributed systems. To

improve the quality of the system, we need to select good test cases that will help the tester to find as many failures as possible in the system. There are several model coverage criteria in the literature that help the tester to control the test generation process. Utting *et al.* [89] discuss the most commonly used criteria which are: structural coverage criterion, data coverage criterion, requirements-based coverage criterion, stochastic criterion[1] and Fault-based criterion (also called mutation coverage [2]). In **Test case execution**, the tester executes test cases generated from the model, after adapting them into an executable form. Then, the tester applies these executable test cases to the SUT to produce the actual outputs of the system. Finally, **Conformance checking** is the process consisting in a comparison[2] between the actual outputs of the SUT with the expected outputs provided by the test cases by implementing a mathematical conformance relation. Conformance relations *tioco* [60, 61, 7] and *dtioco* [37] will be discussed in the next chapters of this thesis. This activity produces results called *test verdicts*. A test verdict (verdict for short) may be *Pass*, *Fail*, or *Inconclusive* as follows:

- A test is *passed* when the real outputs produced from SUT conform to the expected outputs as given in the specification model.
- It *fails* if they do not conform to the specification.
- When the testing activity does not consider the whole specification as a target, but only a part of it, often called *test purpose*, then due to non-determinism issues that often occur when considering reactive systems, it is not possible to ensure that the test purpose has been reached, the test is then to be *inconclusive* (see the work of [7]).

3.1.2 Model-based Testing Classification

Utting *et al.* [89] presented a taxonomy of MBT approaches. Classification criteria used are related to the model, to the test generation algorithms, and to test execution techniques (On-line or Off-line). Some of the classification criteria are:

- The Model notation criterion: MBT can be classified according to the Model notation used in the process. Several notations such as state-based notations, trace-based notations, and transition-based notations are presented in [89].
- Test generation criterion: Test cases can be generated randomly, by a dedicated graph search (as Breadth First Search (BFS) [15], Depth First Search (DFS) [84], Random First Search (RFS) [4] or Hit-Or-Jump (HoJ) algorithms[4]) or through symbolic execution or model checking.
- On-line and off-line criterion (See Figure 3.2): this criterionof represents the relative timing between test generation and test execution activities. Off-line testing means that test cases are generated and stored for example in a file before running them, so that they can be executed many times without new computing efforts. In on-line testing, test cases are generated while the system is running: at any moment, test case generation depends on the previous actual outputs of the SUT. In Section 3.1.2, we will detail the differences between off-line and on-line case generation techniques.

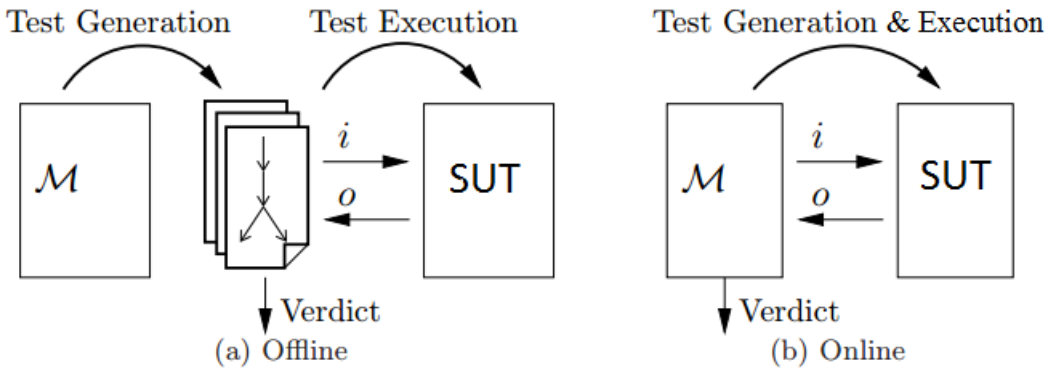


Figure 3.2: Online vs. Offline Test Generation[43]

In this thesis, in the context of MBT classification, we use TIOSTS (see Definition 2.7) which is a transition-based model notation. As we are only interested in solving the oracle problem (that is checking conformance), we suppose that the test cases are already present, thus, our testing approach that we will present in Chapter 4 is purely defined as off-line.

3.1.3 On-line versus Off-line MBT

There are two main techniques to deploy MBT; on-line and off-line MBT [92, 43]. As we mentioned previously, test cases can be generated off-line and later executed, or they can be generated and executed on-the-fly. In the sequel, we detail a little more those two techniques (See Figure 3.3).

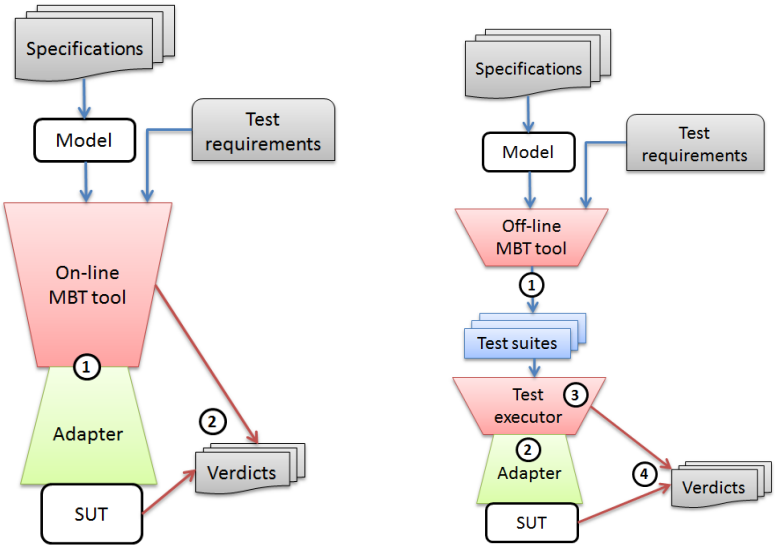


Figure 3.3: On-line vs Off-line testing activities [74]

On-line MBT Process: In on-line testing (see Figure 3.3(a)), we combine test generation and execution. Test generator and the SUT are connected. Hence, the test generation

[1] A *probabilistic model* is the environment model which represents the behavior of the environment of the SUT models. Test cases are generated based on the probabilities that are assigned to the transitions.
[2] Ideally, this comparison is performed automatically.

process can react to the actual outputs of the SUT and thus test data are generated and executed one after the other. A SUT is often hardly controllable at the test execution phase, typically because, for the sake of abstraction, its reference model may include non-deterministic situations. For this reason, when dealing with automatic test case generation, approaches in which test inputs to be sent to the SUT are computed on-the-fly are very popular: they permit to stimulate it in a flexible manner depending on observed SUT executions, and depending on the goal of the testing process in terms of behaviors to cover.

In reference to Figure 3.3(a), on-line testing consists in:

1. Submitting an input sequence computed from the model to the SUT after translation into an adaptable format.

2. Comparing (on-the-fly) between the actual outputs of the SUT with the expected outputs provided by the test cases and computing a test verdict.

There are several advantages of on-line testing. On-line testing may potentially continue for a long time. The state-space-explosion problem may be reduced because only a limited part of the state-space needs to be stored at any point in time. Moreover, on-line test generators often allow non-determinism in timed models: Since they are generated event-by-event they are automatically adaptive to the non-determinism of the specification and SUT.

A disadvantage of on-line testing is that the reference model must be analyzed on-line and in real-time which require very efficient test generation techniques. Although some guidance is possible, test generation is typically randomized which means that satisfaction of coverage criterion cannot be first guaranteed.

Off-line testing Process: The other alternative is the off-line testing (see Figure 3.3(b)) which means that test cases are generated strictly before they are run on the SUT. Test cases can be automatically re-generated to reflect the change, rather than manually updating every test case and test script. Off-line testing approaches are generally applied with timed systems. In fact, time allows expressing instants at which inputs have to be sent to the SUT. For example, in[43], the authors describe an off-line test generation approach for real-time systems specified as timed automata, in [7] the authors present an approach for off-line based testing systems.

In reference to Figure 3.3(b), off-line testing consists in:

1. Computing the full input sequence from test case generated from the model.

2. Submitting the sequence to the SUT after translation into an adaptable format.

3. Storing the real output sequence produced by the SUT during the execution phase.

4. Comparing between the actual outputs of the SUT with the expected outputs provided by the test cases and computing a test verdict.

Off-line testing approach presents several advantages going from test case generation to test phase implementation. In this alternative of testing, the tester can generate test suites once, and then execute them as many times as desired on SUT. Off-line testing

as depicted in Figure 3.3(b) allows to avoid the intertwining[3] of the test generation, test execution, and verdict computation processes. For example, this helps to separate both test generation and test execution and then to perform them on different machines and in different environments, as well as in different times.

There are two main disadvantages of off-line test generation. One advantage is that the reference model must be analyzed entirely, which often results in a state explosion which limits the size of the specification that can be managed. Another problem is non-deterministic implementations and specifications. In this case, the output and its timing cannot be predicted. Typically, the test case may take the form of a test-tree that branches for all possible outcomes. This may lead to very large test cases. In particular for real-time systems, the test case may need to branch for all time instances where an output could be executed.

In the following, in Section 3.2.1 we present an overview of the work of [7] as our baseline off-line approach of performing MBT over TIOSTS based on conformance relation *tioco* [58, 62]. In Section 3.2.2 we present an adaptation version of algorithm for solving the oracle problem of [7] which is well-suited for our new representation of timed traces as presented in Definition 2.15.

3.2 Off-line Centralized Conformance Testing from TIOSTS

In this section, we present *tioco* conformance relation [60, 61, 7] relating the correctness of a localized SUT against its localized specification TIOSTS model. We present briefly the approach of [7] as an existing baseline approach for centralized off-line testing where the authors exposed the oracle problem from a centralized testing perspective.

3.2.1 Overview

In [7], the authors presented an approach for applying a complete off-line testing approach over TIOSTS models (see Figure 3.4). Then they, introduced an off-line testing algorithm based on the timed conformance relation *tioco*[58, 62].

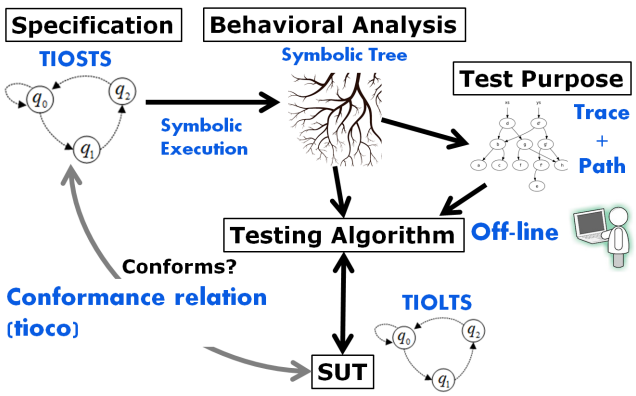


Figure 3.4: Off-line MBT approach process of [7]

[3]The intertwining consists in interlacing between test generation, test execution, and verdict computation processes in such a way that one can lose the test controllability.

The authors of [7] started by defining a testing framework for SUT described as Timed Input/Output Labelled Transition Systems (TIOLTS) which are automata[4] whose transitions are labeled either by concrete actions (inputs or outputs) or by delays. Given a TIOLTS $\mathbb{A}$, as for TIOSTS a path of $\mathbb{A}$ is defined as a sequence of consecutive concrete transitions. The set of all paths of $\mathbb{A}$ is denoted $Paths(\mathbb{A})$. A concrete timed trace of $\mathbb{A}$ is hence built by concatenating consecutive concrete actions and delays of its corresponding path and where we have the ability to split up and add up time values in the set of strictly positive delays D^+. The set of all concrete timed traces of $\mathbb{A}$ is denoted $TTraces(\mathbb{A})$.

System Under Test (SUT). The authors of [7] defined a SUT as a TIOLTS satisfying two properties *Input enableness* and *Time elapsing*. Input enableness condition expresses that an SUT is always able to receive an input. Indeed, most of the times, a tester can submit an input to SUT at any time within the testing process duration. Time elapsing condition expresses that the absence of a reaction amounts to observe no reaction during a strictly positive delay: i.e, for any state q in SUT, if there is no transition from q labeled by an output, we have that there exists a transition from q labeled with a duration. That is to say, if a SUT does not react to an input submitted by a tester, then SUT will wait throughout a given positive duration. The set of all concrete timed traces of an SUT $\mathbb{S}$ is denoted $TTraces(\mathbb{S})$.

Unlike the approach of [7], our approach for centralized testing which will be described in Section 3.2.2, defines an SUT as a set of timed traces (as defined in Definition 2.15) that respects some properties. We choose this new representation because a SUT may be only observable by means of timed traces that a tester builds while interacting with it.

Timed conformance relation. The authors of [7] which were interested in the problem of checking conformance between a SUT and its specification used the timed conformance relation *tioco* as a mathematical relation between concrete timed traces of the SUT and timed traces that can be generated from the reference model. *tioco* conformance relation states that after a specified sequence of interactions between the local tester and local SUT represented in terms of a concrete timed trace σ, any reaction r (i.e, either an output or an observation of a delay) of the SUT must be specified in the reference model.

As presented in Section 3.1.3, off-line testing approach is described as a process of four steps. After defining the specification model and generating test suites in step 1, the authors of [7] were interested in steps 2 and 3 which correspond respectively to input sequence submission to SUT and output sequence storing by a local tester when it interacts with a local SUT. They defined a *test data* as a couple (σ_i, σ_o) gathering submitted input sequence σ_i to SUT and output sequence σ_o produced during test execution step. The connection between σ_i and σ_o forms a timed trace of the SUT. For describing the connection mechanism, the authors introduced two functions to handle test execution step: projection function and merging function. *Projection function:* is the function to extract a sub-timed trace from the timed trace generated from the reference model containing only consecutive input actions and delays. The *Merging function:* allows combining an input sequence and an output sequence according to delays occurring in them. In the following, given a timed trace σ, the input projection of σ is denoted $\sigma_{\downarrow_I}$, and given two timed traces σ_1 and σ_2

[4] A TIOLTS is a numerical representation of a TIOSTS

the merging operation of σ_1 and σ_2 is denoted $Merge(\sigma_1, \sigma_2)$.

The authors of [7] introduced an off-line algorithm for verdict computation following the next steps: input sequence selection, test execution, and verdict computation:

- **Input sequence selection:** For a given TIOSTS $\mathbb{G}$, the authors of [7] used the notation $\mathcal{SE}(\mathbb{G})_\delta$ for symbolic execution of all symbolic transitions in Tr with quiescence enrichment. From the tree-like structure produced by symbolic execution of $\mathbb{G}$, they extract a path which corresponds to a so-called *test purpose* which is defined as a feasible path tp in $Paths(\mathcal{SE}(\mathbb{G})_\delta)$, then, they build input sequences corresponding to timed traces of the selected path tp. Given a test purpose tp, they selected an input sequence by the application of the projection function (previously presented) on a trace tr chosen in $TTraces(tp)$. Having presented how to select an input sequence from a test purpose tp in $Path(\mathcal{SE}(\mathbb{G})_\delta)$, the authors introduced $IS(p)$ as the set of all input sequences extracted from all timed traces in $TTraces(tp)$. Formally:

  $$IS(p) = \{\sigma_{\downarrow_I} | \sigma \in TTraces(tp)\}$$

  In the sequel, we suppose that an SUT $\mathbb{S}$, a test purpose tp and an input sequence $\sigma^{tp}_{\downarrow_I}$ are given ($\sigma^{tp}_{\downarrow_I}$ is the notation which stands for a input sequence extracted from tp). We describe the process of *test execution* as follows:

- **Test execution:** Having defined how to extract $\sigma^{tp}_{\downarrow_I}$ from a given test purpose tp, the execution phase denotes the stage where a tester submits an input sequence to SUT which in turn reacts by producing output sequence. The *test execution* is the function which submits $\sigma^{tp}_{\downarrow_I}$ to $\mathbb{S}$ and denoted $sigma^{tp}_{\downarrow_I} \rightsquigarrow_{\mathbb{S}} \sigma_o$ where σ_o is the produced timed trace as a reaction of the SUT.

In real time, one must wait for a given duration (which corresponds to the execution phase) until that SUT delivers output sequences. In this case SUT sends an output sequence σ_o and we note: $\sigma^{tp}_{\downarrow_I} \rightsquigarrow_{\mathbb{S}} \sigma_o$. We merge the two previous sequences (input and output sequences) in the aim of producing the corresponding execution timed trace $\sigma_{\mathbb{S}}$ such that $\sigma_{\mathbb{S}} = Merge(\sigma^{tp}_{\downarrow_I}, \sigma_o)$.

Verdict computation. The algorithm takes as inputs three arguments: $\mathcal{SE}(\mathbb{G})_\delta$ which denotes symbolic execution on $\mathbb{G}$; tp which designates the test purpose to cover and $\sigma_{\mathbb{S}}$: the timed trace to be analyzed in order to deliver a verdict concerning the correctness of $\mathbb{S}$ assessment together with the coverage of tp by $\sigma_{\mathbb{S}}$ w.r.t *tioco*. Off-line testing algorithm (as depicted in Figure 3.5) of [7] analyses elements of $\sigma_{\mathbb{S}}$ one element at a time and produces a test verdict in $\{PASS, FAIL, WEAK\_PASS, INCONC_i, INCONC_r\}$ as follows:

- Verdict $FAIL$ is emitted when an unspecified output o or delay d is read from $\sigma_{\mathbb{S}}$.
- Verdict $PASS$ is emitted when tp is the only path covered in $\mathcal{SE}(\mathbb{G})_\delta$ by $\sigma_{\mathbb{S}}$.
- Verdict $WEAK\_PASS$ is emitted when tp is covered in $\mathcal{SE}(\mathbb{G})_\delta$ by $\sigma_{\mathbb{S}}$ together with the covering of other paths in $\mathcal{SE}(\mathbb{G})_\delta$.
- Verdict $INCONC_r$ when some paths are covered in $\mathcal{SE}(\mathbb{G})_\delta$ by $\sigma_{\mathbb{S}}$ but not tp.
- Verdict $INCONC_i$ when an unspecified input i is read from timed trace $\sigma_{\mathbb{S}}$.

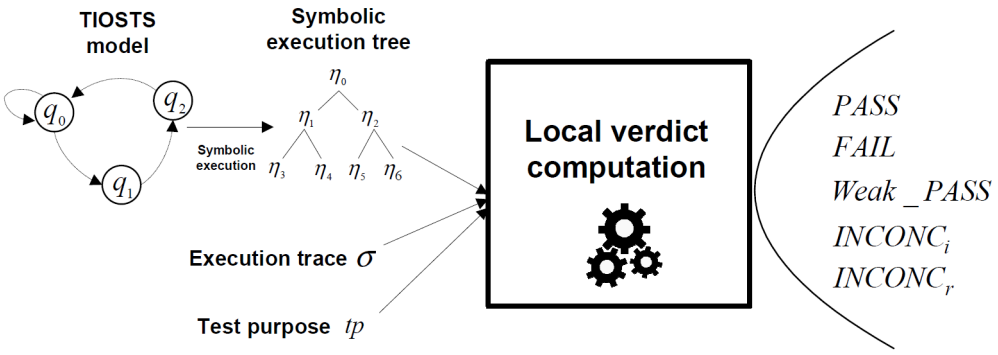


Figure 3.5: Verdict computation process and local verdicts [7]

3.2.2 An Adaptation of the Centralized off-line Testing Algorithm

Herein, we introduce our adaptation of centralized testing algorithm for solving the oracle problem of [7] which is based on our new presentation of timed traces defined as normalized sequences of events. Section 3.2.2.1 deals with modeling the SUT and introduces our adapted definition of *tioco* while Section 3.2.2.2 details the verdict computation algorithm.

3.2.2.1 System Under Test and Timed Conformance Relation

In the sequel, given a set of channels C, we introduce the following notations:

Notation 3.1. *Given a timed trace σ in $TTraces(C)$, we let $Pref(\sigma)$ denotes the set of* prefixes of *σ defined as $\{\varepsilon\}$ if σ is ε and $Pref(\sigma') \cup \{\sigma\}$ if σ is of the form $\sigma'.ev$.*

Definition 3.1 (System Under Test (SUT))**.** *Let C be a set of channels. A System Under Test (SUT) is defined over C as a non-empty subset $\mathbb{S}$ of $UTraces(C)$ such that:*

- ***Input completeness**: for any σ in $\mathbb{S}$ of the form $\sigma'.ev'$, for any $ev \in Evt(C)$ such that $act(ev) \in I(C)$ and $delay(ev) \leq delay(ev')$, we have $\sigma'.ev \in \mathbb{S}$. Moreover for any $i \in I(C)$ we have $(\_, i) \in \mathbb{S}$.*
- ***Quiescence** (1): for all $\sigma \in \mathbb{S}$ we have:*

$$\forall ev \in Evt(C).(act(ev) \in O(C) \Rightarrow \sigma.ev \notin \mathbb{S})$$

$$\Rightarrow$$

$$(\sigma \neq \varepsilon \Rightarrow (\forall d \in D^+, \sigma.(d, \delta) \in \mathbb{S})) \wedge ((\_, \delta) \in \mathbb{S})$$

- ***Quiescence** (2): $(\_, \delta) \in \mathbb{S}$ and for all $\sigma \in \mathbb{S}$ of the form $\sigma'.ev$ with $delay(ev) \neq 0$ (and thus $\sigma' \neq \varepsilon$) for all $d < delay(ev)$ we have $\sigma'.(d, \delta) \in \mathbb{S}$.*
- ***Reaction prefix**: for any σ in $\mathbb{S}$, we have $Pref(\sigma) \subseteq \mathbb{S}$.*

The SUT definition specifies that:

- **Input completeness**: any timed trace of a SUT can be completed by any input. This condition is required so that a SUT cannot refuse an input from the environment or any stimulation sent by the tester.

- **Quiescence** corresponds to two kinds of situations. The first kind of situations (**Quiescence** (1)) are the ones where the SUT will not react anymore until it receives a new stimulation. The second kind (**Quiescence** (2)) expresses that whenever we might wait for a positive duration before an action occurs, we can observe quiescence if we stop the observation of the trace in a shorter positive duration.

- **Reaction prefix**: the set of timed traces of a SUT is stable by prefix, i.e a prefix of an observation is an observation. in particular $\varepsilon \in \mathbb{S}$.

The conformance of a SUT $\mathbb{S}$ with respect to a TIOSTS $\mathbb{G}$ is defined as a mathematical relation between $\mathbb{S}$ and the set of timed traces of $\mathbb{G}$. Intuitively an SUT $\mathbb{S}$ conforms to $\mathbb{G}$ according to *tioco* if and only if for any timed trace σ common to $\mathbb{S}$ and $\mathbb{G}$, any reaction (waiting for a delay d and observing an emission of an output o or waiting for a quiescent reaction δ) of the SUT $\mathbb{S}$ after σ must be allowed by $\mathbb{G}$.

Definition 3.2 (tioco)**.** *Let C be a set of channels. Let $\mathbb{S}$ and $\mathbb{G}$ be respectively a SUT and a TIOSTS, both defined over the same set of channels C. $\mathbb{S}$ conforms to $\mathbb{G}$ denoted $\mathbb{S}$ tioco $\mathbb{G}$ iff for any $\sigma \in TTraces(\mathbb{G}) \cap \mathbb{S}$ and for any $ev \in Evt(C)$ with $act(ev) \in O(C) \cup \{\delta\}$ we have:*

$$\sigma.ev \in \mathbb{S} \Rightarrow \sigma.ev \in TTraces(\mathbb{G})$$

3.2.2.2 Our Off-line Centralised Testing Algorithm

We now present our off-line testing algorithm inspired and adapted from the one defined in [7]. As we presented in Section 3.2.1, the centralised testing algorithm of [7] which is designed for timed models analyses a timed trace which is an ordered sequence of actions and delays and computes a verdict in the set of keywords $\{PASS, FAIL, INCONC_i, WEAK\_PASS, INCONC_r\}$.

Our adaptation of [7] is first motivated by the need of dealing with normalized timed traces defined as sequences of events. Moreover, we are not concerned with test case generation, and thus, we will not consider test purposes. In fact, we are only interested in the process of verdict computation. Hence, we suppose the existence of a finite timed trace as a local observation on a SUT and which will be analyzed in order to check conformance of this observation in question against a TIOSTS model w.r.t *tioco* conformance relation. Thus, our testing algorithm is a simplified version of one defined in [7] as it does not specialize the verdict computation up to a test purpose that serves as a guide for test case generation and as the reference for computation verdict: then, our algorithm will not deliver verdicts $WEAK\_PASS$ and $INCONC_r$. Indeed, instead of checking whether or not a timed trace is appropriate up to a test purpose previously selected, we will only check that it is allowed or not by the reference model w.r.t the conformance relation. We adopt this weak position since in this document, we are essentially interested in the question of the analyse of observed traces up to a specification. The question of analysing a timed trace up to a selected test purpose is rather related to the challenge of generating test cases in charge of covering a given test purpose. As this point is not discussed in this document, this explains that we only consider a weak version of verdict computation.

In the sequel, we suppose the existence of a SUT $\mathbb{S}$, a TIOSTS model $\mathbb{G}$. We assume that a finite uninitialized timed trace σ has been computed as an execution of SUT $\mathbb{S}$ and we proceed to the verdict computation of σ.

3.2.2.3 Local Verdict Computation

Our local verdict computation algorithm takes as input the symbolic structure $\mathcal{SE}(\mathbb{G})_\delta$ (see Definition 2.20 from Chapter 2) computed from the reference model $\mathbb{G}$ obtained by symbolic execution techniques. We recall that $\mathcal{SE}(\mathbb{G})_\delta$ is a tree-like structure whose nodes are symbolic states that are used to capture all information related to the possible executions of $\mathbb{G}$.

As introduced in Definition 2.21 from Chapter 2, a symbolic path p in $\mathcal{SE}(\mathbb{G})_\delta$ is a sequence of consecutive edges relating symbolic states and labelled by symbolic events. The set of executions (e.g. timed traces) associated to p can be characterised by the sequence $ev_1 \dots ev_n$ of symbolic events labelling the consecutive edges.

In the sequel, a set F of fresh variables is supposed given. $F_t \subseteq F$ is the set of fresh time variables and $F_d = F \backslash F_t$ is the set of fresh data variables.

We recall that each symbolic event of the sequence is of the form (d_i, act_i) where each d_i is a new fresh variable in F_t used to symbolically represent durations and each act_i is of the form $c?z_i$ or $c!t_i$ where z_i is a new fresh data variable in F_d and t_i is a term in $\mathcal{T}_\Omega(F_d)$ built over the same equational logic signature Ω as terms in $\mathbb{G}$ and over the set F_d of new fresh variables.

The computation verdict concerns the conformance of the SUT $\mathbb{S}$ against TIOSTS $\mathbb{G}$. In other words, we seek to know if σ belongs to the set of uninitialised timed trace defined by a possible symbolic path belonging to $\mathcal{SE}(\mathbb{G})_\delta$. For this, we begin by introducing some intermediate definitions that are needed in the algorithm in order to define verdicts.

A *context* is a mathematical structure denoting paths of $\mathcal{SE}(\mathbb{G})_\delta = (Init, ST)$ potentially covered by a timed trace, together with additional identification constraints induced by the timed trace (the sequence of previously encountered inputs/outputs).

Definition 3.3 (Context)**.** *A context is a triple (η, ψ_t, ψ_d) where η denotes the end state of a symbolic path of $\mathcal{SE}(\mathbb{G})_\delta$, ψ_t is a formula of $\mathcal{F}_\Omega(F_t)$ expressing identification constraints on fresh time variables and ψ_d is a formula of $\mathcal{F}_\Omega(F_d)$ expressing identification constraints between fresh data variables and values emitted and received in the timed trace.*

Identification constraints are the conjunction of constraints of the form $z = v$, where z is a fresh variable and v, is a value, that is inherited from the concrete timed trace σ under analyse. These constraints allow to particularize symbolic paths that already partially match with the beginning of the trace σ.

Notation 3.2. *For a context $ct = (\eta, \psi_t, \psi_d)$ the notation $state(ct)$ stands for η, the target state of the potentially covered path. Since there may be more than one path that is covered by a timed trace, we manipulate sets of contexts generically denoted SC.*

We introduce some technical functions useful to reason about sets of contexts, in particular in order to compute the sequence of sets of contexts resulting from the successive observation of elementary actions. The function $Next(ev, SC)$ computes the set of all contexts that can be reached from a given set of contexts SC, when a new event $ev \in Evt(C)$ occurs.

Definition 3.4 ($Next(ev, SC)$: Execution of set of contexts driven by an event)**.** *Let SC be a finite set of contexts and ev be in $Evt(C)$ with $act(ev)$ of the form $c\triangle u$ with $\triangle \in \{?, !\}$ and u a value*[5]*. $Next(ev, SC)$ is the set of all contexts that can be reached by triggering a symbolic transition of $\mathcal{SE}(\mathbb{G})_\delta$ consistently with ev. We have $(\eta', \psi'_t, \psi'_d) \in Next(ev, SC)$ if and only if there exists a context (η, ψ_t, ψ_d) in SC and symbolic transition $st = (\eta, ev', \eta')$ in ST with $act(ev')$ of the form $c\triangle t$ such that:*

- *ψ'_t is the formula in $\mathcal{F}_\Omega(F_t)$ restricted to $True$ if $delay(ev) = 0$ and equal to $\psi_t \wedge (delay(st) = delay(ev))$ when $\pi_t(\eta') \wedge \psi_t \wedge (delay(st) = delay(ev))$ is satisfiable otherwise.*
- *ψ'_d is the formula in $\mathcal{F}_\Omega(F_d)$ equal to $\psi_d \wedge (t = u)$ when $\pi_d(\eta') \wedge \psi_d \wedge (t = u)$ is satisfiable.*

The general idea of the algorithm is to read one by one the elements of the timed trace σ, and either compute the next set of contexts or emit a verdict.

3.2.2.4 Rule-based Algorithm

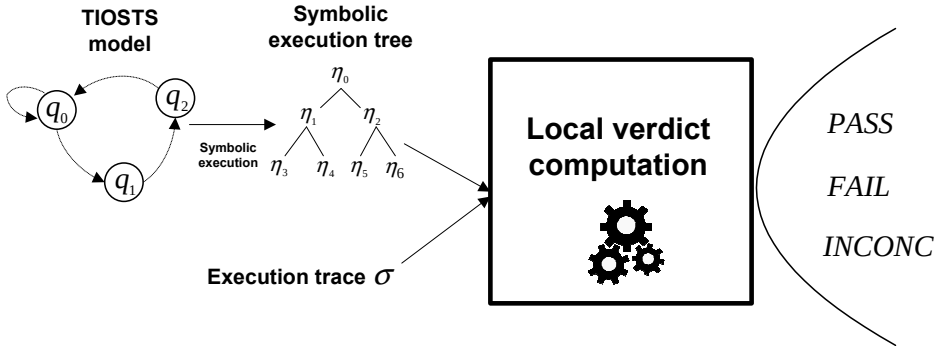


Figure 3.6: Our verdict computation process and local verdicts

As depicted in Figure 3.6, our goal is to compute a $Verdict$ belonging to the set of keywords: $\{FAIL, PASS, INCONC\}$ where:

- $Verdict$ is $FAIL$ if the situation in which we observe an output is not allowed by the specification.
- $Verdict$ is $PASS$ if the observed timed belongs to the specification
- $Verdict$ is $INCONC$ if the timed trace is allowed by the conformance relation *tioco*, but does not belong to the specification, i.e. the last event of the timed trace is an input that is not specified in the specification.

Notation 3.3. *For a non-empty timed trace σ of the form $ev.\sigma'$ we use the notation $head(\sigma)$ to denote ev and $tail(\sigma)$ to denote σ'.*

For that, we will take into account the knowledge of the associated contexts. The algorithm is then given as a set of rules of the form:

[5] Values are assimilated to constant terms

$$\frac{SC(ev) \quad \sigma_{suf}}{Result} \ cond$$

where:

- σ_{suf} is the remaining timed trace to be analyzed with respect to the first analyzes stored in $SC(ev)$ and to $\mathcal{SE}(\mathbb{G})_\delta$. If we are at the end of the timed trace we have $\sigma_{suf} = \varepsilon$. Otherwise, $\sigma_{suf} = ev'.\sigma'_{suf}$
- *cond* are the conditions under which the rule can be applied
- *Result* is either a verdict or of the form $SC'(ev') \ \ \sigma'_{suf}$. Moreover, if $\sigma_{suf} = \varepsilon$ then *Result* is necessarily a verdict since the initial timed trace σ is fully analyzed. If *Result* is $SC'(ev') \ \ \sigma'_{suf}$, then it means that σ_{suf} has been written as $ev'.\sigma'_{suf}$. We will access respectively to ev' and σ'_{suf} from σ_{suf} by using the notations $head(\sigma_{suf})$ and $tail(\sigma_{suf})$.

Notation 3.4. *Let us suppose that σ can be written as $\sigma_{pref}.ev.\sigma_{suf}$ where ev is an event. The notation $SC(ev)$ represents the set of contexts SC reached after reading the beginning $\sigma_{pref}.ev$ of the timed trace with the last analyzed element ev. At the initialization step, when no element of σ has been analyzed, then we use the symbol τ, that is $SC(\tau)$.*

Rules of our testing algorithm are described as follows:

Initialization Rule:

$$\frac{}{\{(Init, True, True)\}(\tau) \ \ \sigma}$$

Next Rule: An event with an action and a delay is read from the trace, SC is not empty.

$$\frac{SC(ev) \ \ \sigma}{Next(head(\sigma), SC)(head(\sigma)) \ \ tail(\sigma)} \ SC \neq \emptyset, \ \sigma \neq \varepsilon$$

Fail Rule: An event with an unspecified output and a delay is read from the trace.

$$\frac{SC(ev) \ \ \sigma}{FAIL} \ SC = \emptyset; \ act(ev) \in O(C) \cup \{\delta\}$$

Inconclusive Rule: An event with an unspecified input is read from the trace.

$$\frac{SC(ev) \ \ \sigma}{INCONC} \ SC = \emptyset; \ act(ev) \in I(C)$$

Pass Rule: The read event permits to cover a path in $\mathcal{SE}(\mathbb{G})_\delta$.

$$\frac{SC(ev) \ \ \sigma}{PASS} \ \sigma = \varepsilon; \ SC \neq \emptyset$$

- **Initialization**: corresponds to the initialization phase where the set of contexts contains only one context stating that we begin at the symbolic state $Init$, there are no constraints identified yet.
- **Next Rule:** is applied to compute a new set of contexts. This is done as long as SC is not empty and there are still elements of the timed trace to read

- **Fail Rule:** concerns the $FAIL$ verdict emitted when the timed trace denotes an incorrect behavior.
- **Inconclusive Rule:** introduces the verdict $INCONC$. According to this rule, $INCONC$ is emitted when σ is not included in $\mathcal{SE}(\mathbb{G})_\delta$ due to input under-specification. More precisely, σ is then of the form $\sigma_{pref}.ev.\sigma_{suf}$, with σ_{pref} in $\mathcal{SE}(\mathbb{G})_\delta$ but $\sigma_{pref}.ev$ is not with $act(ev) \in I(C)$.
- **Pass Rule:** introduces the $PASS$ verdict emitted when the trace (which is fully analyzed without generating any of the previous verdicts) denotes a correct behavior, i.e there exists a covered path p in $\mathcal{SE}(\mathbb{G})_\delta$.

We notice that unless σ can be decomposed as $\sigma_{pref}.ev.\sigma_{suf}$, where σ_{pref} is a specified timed trace and $\sigma_{pref}.ev$ is not (in which case we have $FAIL$ or $INCONC$ depending on the nature of $act(ev)$), all events of σ will be analyzed even though the emission of $PASS$ is not possible anymore. This choice allows us to always emit $FAIL$ if a timed race reveals a non conformance.

Example 3.1. *Let us apply our rule-based algorithm to the TLC system from Example 2.5. Consider symbolic tree $\mathcal{SE}(\mathbb{G}_{TLC})_\delta$ produced from application of symbolic execution on TIOSTS $\mathbb{G}$ with quiescence enrichment. Symbolic tree $\mathcal{SE}(\mathbb{G}_{TLC})_\delta$ specifies correct behavior of TLC system of Example 2.5. Let us assume that we have a timed trace σ which is an execution of the TLC system as illustrated in Example 2.11: $\sigma = (\_, start?).(3, pos_1!42).(5, pos_2?300)$. We proceed to verdict computation of σ as follows:*

(a)

$(\_,start?)$	$(3, pos_1!42)$	$(5, pos_2?300)$

ψ_t=$True$; ψ_d=$True$; SC={$(Init, \psi_t, \psi_d)$}(τ); ***(Initialization)***

(b)

$(\_,start?)$	$(3, pos_1!42)$	$(5, pos_2?300)$

$ev = (\_, start?)$; $delay(ev) = 0$
$\psi_t \leftarrow True$
$\pi_d(\eta_1) = True$
$\psi_d \leftarrow \psi_d \wedge \pi_d(\eta_1)$ *is satisfiable*
$Next(ev, SC) \longrightarrow SC$={$(\eta_1, \psi_t, \psi_d)$}$(ev)$;***(Next Rule)***

(c)

$(\_,start?)$	$(3, pos_1!42)$	$(5, pos_2?300)$

$ev = (3, pos_1!42)$
$\pi_t(\eta_3) = z_2 < 10$
$\psi_t \leftarrow \psi_t \wedge \pi_t(\eta_3) \wedge (z_2{=}3)$ *is satisfiable*
$\pi_d(\eta_3) = p_1 < p_2 \leq 200$
$\psi_d \leftarrow \psi_d \wedge \pi_d(\eta_3) \wedge (p_1{=}42)$ *is satisfiable*
$Next(ev, SC) \longrightarrow SC$={$(\eta_3, \psi_t, \psi_d)$}$(ev)$;***(Next Rule)***

(d)

$(\_,start?)$	$(3, pos_1!42)$	$(5, pos_2?300)$

$ev = (5, pos_2?300)$
$\pi_t(\eta_6) = \pi_t(\eta_3) \wedge z_5 < 10$
$\psi_t \leftarrow \pi_t(\eta_6) \wedge z_5{=}5$ *is satisfiable*
$\pi_d(\eta_6) = \pi_d(\eta_3)$
$\psi_d \leftarrow \psi_d \wedge \pi_d(\eta_6) \wedge (p'_2{=}50)$ *is satisfiable*
$Next(ev, SC) \longrightarrow SC$={$(\eta_6, \pi_t, \pi_d)$}$(ev)$;***(Next Rule)***

(e)

$(\_,start?)$	$(3, pos_1!42)$	$(5, pos_2?300)$

$\sigma = \varepsilon$; $SC \neq \emptyset$
(Pass Rule)

In this chapter, we have tackled the issue of performing centralized MBT over TIOSTSs and adapting the process of checking conformance to fit our formal background of Chapter 2.

In Chapter 4 we present and discuss our contributions in order to propose a verdict computation process and thus solving the oracle problem in distributed MBT.

Chapter 4

A Distributed Testing Framework for Solving the Oracle Problem

Contents

4.1 An Overview of Works Related to Distributed Testing

Several definitions of Distributed Systems have been given in the literature [65, 20, 83]. Lamport [65] characterizes a distributed system as a set of asynchronous communicating processes. A commonly used definition in software engineering community is the one of Tanenbaum *et al.* [83], in which authors define a DS as:

"A collection of independent computers that appear to the users of the system as a single computer".

This definition introduces two main features of distributed systems: The first one is that DS is a collection of nodes, each being able to behave independently of each other. Nodes can be either a hardware device or a software process. The second one is that users (i.e., people or applications) believe they are dealing with a single system. This means that one way or another the nodes need to collaborate. This collaboration lies at the heart of developing distributed systems.

For the purpose of this thesis, we propose to use the following definition of distributed systems:

"A DS consists of collection of localized autonomous entities, connected through a communication network, which enables those entities to coordinate their activities and to exchange the system resources, so that users perceive the whole distributed system as a one single executing entity".

A distributed system by Tannenbaum's definition would surely also be one by our definition; however, our definition is more in line with the current state of the art as perceived by today's users of distributed systems and it characterizes the kind of systems that we will study throughout this thesis.

Internet is considered as a distributed system with multiple clients and servers for acceding and sharing linked data. Within the Internet, Servers maintain collections of data while clients provide user-interfaces for presenting and accessing this data. A Web browser is the user-interface to Internet, it includes Web pages that link to other ones. On the other hand, Web servers can refer to either the hardware (the machine) or the software (the application) which runs on the server side of the system and delivers Web content that can be accessed through communication networks.

Figure 4.1 depicts a working arrangement of the Internet as a distributed system.

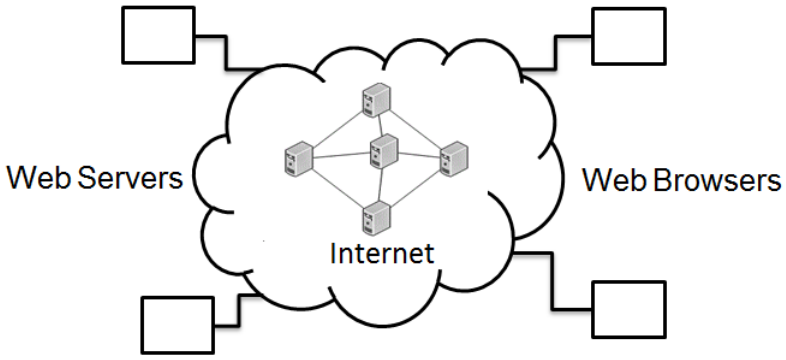
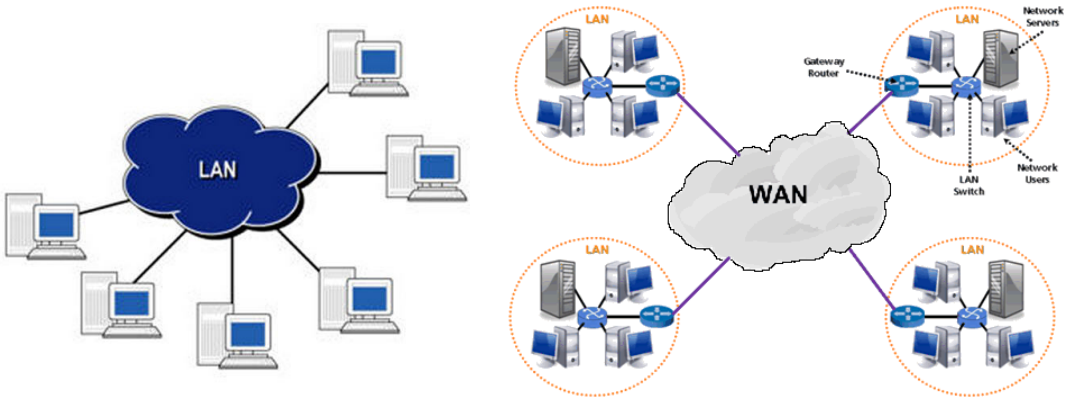


Figure 4.1: Internet considered as a distributed system

Internet communication networks can be classified into two categories: LAN and WAN. According to [79, 63] a LAN spans a small geographical area, typically a single building or a cluster of buildings, while a WAN spans a large geographical area (e.g. a nation) which needs a switched large network. A WAN can be defined as a network linking several LANs. Figure 4.2 depicts an architecture of a LAN and a WAN in Internet.



(a) (b)

Figure 4.2: Illustration of a LAN and a WAN distributed architectures [63].

Distributed systems can be especially difficult to program, for a variety of reasons. They can be difficult to design, difficult to manage, and, above all, difficult to test. Testing a normal system can be trying even under the best of circumstances, and no matter how diligent the tester is, bugs can still get through. Now take all of the standard issues and multiply them by multiple processes written in multiple languages running on multiple boxes that could potentially all be on different operating systems, and there is potential for a real disaster.

Individual component testing, usually done via automated test suites, certainly helps by verifying that each component is working correctly. Component testing, however, usually does not fully test all the bits of a distributed system. Testers need to be able to verify that data at one end of a distributed system makes its way to all other parts of the system and, perhaps more importantly, that it is visible to the various components of the distributed system in a manner that meets the consistency requirements of the system as a whole. In the next, we discuss the open issues in testing distributed systems.

As a DS is a collection of communicating localized subsystems, it might expose failure due to some of them. In addition, the network communication which is based on message passing mechanism might expose transmission errors. Those two kinds of errors have different natures from the oracle computation problem point of view. To identify errors related to localized systems, on has to analyze sequences of events which can be fully ordered since all of them can be associated with dates by a common clock. On the contrary identifying communication errors involve analyses that aim at logically ordering events that occur on different remote interfaces. Indeed as the DS does not have any global clocks, logging its execution does not come to build a unique sequence of events, but rather a collection of such sequences, one for each of the localized systems.

Lamport [65] characterizes a distributed system as a set of asynchronous communicating processes. Exchanged messages between local processes of a distributed system are essentially intended for the coordination of local tasks executed by those processes (i.e to synchronize their communication). The difficulty within distributed computing is to define a global coherent time which schedules all local events. To overcome this issue, logical time was first introduced by Lamport [65] as a concept to schedule events in a distributed system.

Causal dependency and Logical clocks: Let E be a set of so-called events. There exists a *causal dependency* between two events e and e' if an event must occur before the other one and we note $e \rightarrow e'$. Given a set of *events* E and a set of so-called *timestamps* T, Lamport [65] introduced the notion of *logical clock* as the function c which associates a date to a given event e in E defined as follows:

$$c : \begin{cases} E \rightarrow T \\ e \mapsto c(e) \end{cases}$$

For two given events e and e' we have: $e \rightarrow e' \Rightarrow c(e) < c(e')$.

Scalar Logical clocks: Lamport [65] has been a pioneer in proposing techniques to analyze logs of distributed systems execution, with a particular emphasis on identifying causality between messages exchanged by localized subsystems. This was done based on a formal setting called *"logical scalar clocks"*. For a distributed system composed of n localized subsystems, Lamport [65] characterizes a localized subsystem as a process identified with its number $p \in [1, \ldots, n]$. A process is then seen as an ordered sequence of events. Given a process composed of m events, a scalar logical clock [65] is a couple (p, e) which stands for a timestamp associated to an event where $p \in [1, \ldots, n]$ is the process number in and $e \in [1, \ldots, m]$ is the event number.

In [65] Lamport presented an algorithm for annotating distributed events of an execution of DS with scalar clock. For a distributed system $(P_1, \ldots, P_i, \ldots, P_n)$ composed of n processes, Lamport states that each process P_i ($i \in [1, \ldots, n]$) has a logical scalar clock c_i initiated at 0. Locally, for each local event of P_i, we increment c_i by 1 ($c_i \leftarrow c_i + 1$) and the event in question is timestamped locally by c_i. In the case of exchanging messages we distinguish two rules:

- Emission of a message m by P_i: we increment c_i by 1, then we send the message m with (i, c_i) as a timestamp.
- Reception of a message m with a clock c_j ($j \in [1, \ldots n]$ and $j \neq i$): $c_i \leftarrow max(c_i, c_j)+1$. In this case, we mark the reception of message m with c_i.

We illustrate Lamport clock advancing algorithm by means of an example. We give the traces tr_1, tr_2 and tr_3 as possible executions of three processes P_1, P_2 and P_3 as follows:

- $tr_1 = e_{11}.e_{12}.e_{13}.e_{14}.e_{15}$
- $tr_2 = e_{21}.e_{22}.e_{23}.e_{24}$
- $tr_3 = e_{31}.e_{32}.e_{33}.e_{34}.e_{35}$

Following Lamport algorithm [65] presented previously and by applying scalar clocks advancing mechanism, traces tr_1, tr_2 and tr_3 are annotated with timestamps as presented graphically in Figure 4.3. First, logical clocks c_i are initiated at 0. Then, we follow rules presented previously to advance timestamps logically. For example, we have:

- Timestamp of the event e_{23} is 6, indeed, message m_5 received had logical clock value of 5 and local clock is only at 3.

- Timestamp of the event e_{34} is 4, indeed, we increase local clock by one since its value ($c_i = 3$) is greater than clock value of message m_3.
- For events e_{11}, e_{12}, e_{13}, we increment local clock by one.

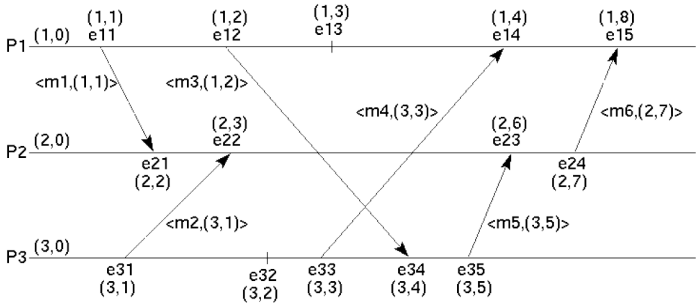


Figure 4.3: Scalar clocks annotating mechanism using Lamport algorithm [65]

Logical scalar clocks define an order in the set of events observed from the execution of a given DS. Indeed, for two events e and e', we have e is less than e' if and only if timestamp of e is strictly less than timestamp of e'. This order is only partial because several events can have the same timestamps. Lamport [65] extended this partial order to a total one denoted $\prec$ by assuming the following decision: if two events with same timestamps occur, the event on the process with the smallest identifier predates the other one, that is, if e and e' are two events that run on processes P_i and P_j then we have:

$$e \prec e' \Leftrightarrow (c_i(e) < c_j(e')) \vee (c_i(e) = c_j(e') \text{ with } i < j)$$

From the previous example, we can achieve the following total order: $e_{11} \prec e_{31} \prec e_{12} \prec e_{21} \prec e_{32} \prec e_{13} \prec e_{22} \prec e_{33} \prec e_{14} \prec e_{34} \prec e_{35} \prec e_{23} \prec e_{24} \prec e_{15}$. As we can notice, events e_{11} and e_{31} have the same timestamp value which is equal to 1 but as e_{11} occurred on process 1 and e_{31} occurred on process 3 and $1 < 3$ then $e_{11} \prec e_{31}$.

For two events e and e', the relation $c(e) < c(e')$ is not sufficient to decide about causal dependency between e and e'. Yet, it is useful to determine whether or not there is a causal dependency between two events. The notion of *vector logical clock* was introduced later in [33] to ensure that the reciprocal of the causal dependency holds. In other words to ensure that:

$$c(e) < c(e') \Rightarrow e \rightarrow e'.$$

Vector Logical clocks: Later, Fidge [33] presented the limitations of using logical scalar clocks to schedule distributed events. Indeed, global scheduling obtained by using scalar clock is indeed arbitrary and does not necessarily correspond to a real scheduling. For example, we have $c(e_{32}) = 2$ and $c(e_{22}) = 3$, yet, in practice, e_{22} can occur before e_{32}. Indeed, for two events e and e' such that $c(e) < c(e')$, one cannot decide about causal dependency between e and e'. Specifically, the relation $c(e) < c(e') \Rightarrow e \rightarrow e'$ does not hold. For this reason, Mattern [66] then Fidge [34] introduced vector logical clocks that ensure that previous relation holds.

Mattern [66] and Fidge [33, 34] define a vector logical clock of a distributed system of n processes as a data vector v of n logical clocks, one clock per process. In [66] Mattern

presented an algorithm for annotating a distributed observation of DS with vector clocks as follows:

Let us consider a distributed system $(P_1, \ldots, P_i, \ldots, P_n)$ of n communicating processes. Locally, each process P_i has a vector clock v_i of n elements and where each element $v_i[j]$ contains clock values of process P_j $(i, j \in \{1, \ldots, n\}$. We update vector clocks v_i w.r.t the following rules:

1. Initially, we have $v_i[j] = 0$.
2. Before process P_i timestamps an event, it executes $v_i[i] = v_i[i] + 1$
3. Whenever a message m is sent from P_i to P_j:
   - Process P_i executes $v_i[i] = v_i[i] + 1$ and sends v_i with m.
   - Process P_j receives v_i with m and merges vector clocks v_i and v_j as follows:

$$v_j[k] = \begin{cases} max(v_j[k], v_i[k]) + 1, & \text{if } j = k(\text{ as in Lamport clocks}) \\ max(v_j[k], v_i[k]), & \text{otherwise} \end{cases}$$

Initially, all clocks are initialized at 0. Each time a process P_i experiences an internal event, it increments its own logical clock in the vector by one. Each time a process prepares to send a message, it sends its entire vector together with the message being sent. Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element). This last part ensures that everything that subsequently happens at P_j is now causally related to everything that previously happened at P_i.

Vector clock formalism defines a partial order relation on the set of dates w.r.t the following equivalence relations given that v and v' are two vectors of n logical clocks:

- $v \leq v' \Leftrightarrow \forall i \in \{1, \ldots, n\}$, $v[i] \leq v'[i]$.
- $v < v' \Leftrightarrow v \leq v'$ and $\exists i \in \{1, \ldots, n\}$ s.t $v[i] < v'[i]$.
- $v \parallel v' \Leftrightarrow \neg(v < v')$ and $\neg(v' < v)$.

We illustrate vector clock advancing algorithm by means of an example. We give the traces tr_1, tr_2 and tr_3 as possible executions of three processes P_1, P_2 and P_3 as follows:

- $tr_1 = e_{11}.e_{12}.e_{13}.e_{14}.e_{15}$
- $tr_2 = e_{21}.e_{22}.e_{23}.e_{24}$
- $tr_3 = e_{31}.e_{32}.e_{33}.e_{34}.e_{35}$

Following vector clock advancing algorithm presented previously, traces tr_1, tr_2 and tr_3 are annotated with timestamps as presented graphically in Figure 4.4. For example, we have:

- $v(e_{13}) = (3, 0, 0)$ and $v(e_{14}) = (4, 0, 3)$. We have $v(e_{13}) < v(e_{14})$, hence, $e_{13} \rightarrow e_{14}$.

- $v(e_{32}) = (0, 0, 2)$ and $v(e_{13}) = (3, 0, 0)$. We have $v(e_{32}) \| v(e_{13})$, then, neither $(e_{32} \rightarrow e_{13})$ holds nor $(e_{13} \rightarrow e_{32})$ holds.

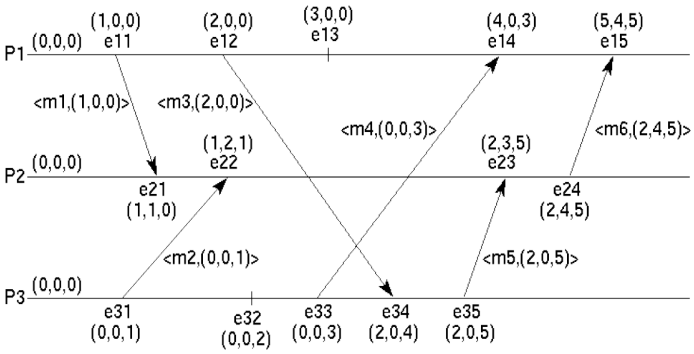


Figure 4.4: Vector clocks annotating mechanism.

Related work: Some testing approaches used logical time, as a technique for testing distributed systems. The following is a brief list of some works on testing distributed (concurrent) systems using logical clocks:

- In [51], Kim *et al.* studied the problem of testing concurrent distributed systems as black-boxes modeled as *asynchronous communicating finite state machines (ACFSM)*. The authors defined and presented with illustrative examples an approach to derive test cases in a formal way for concurrent distributed systems. The approach defined a technique to avoid the state explosion problem by introducing a *causality relation model* based on logical clocks advancing mechanism. By adopting a causality relation model, the authors of this paper expressed a true concurrency model and hence avoided classical approaches in distrusted testing that use interleaving methods for the events in a concurrent system. Kim *et al.* [51] introduced the *Minimal Causality Path (MCP)* notion using logical clocks as global event sequence path with minimal length. Those paths were later used in test case generation in order to avoid the state space explosion problem. This work also introduced for new definitions such as *Observationally Rate (OR), Stable State (SS)* and *Controllability Rate (CR)*. Yet, this work assumed that atomic actions in the model consume exactly one unit of logical time, hence, the model cannot be considered as applicable to the real world situations.

- In [52], Kim *et al.* extended work presented in [51] by relaxing the unit-time assumption to any natural or real numbers in describing timing constraints and by presenting a computationally efficient algorithm for deriving test cases from the model with respect to the relaxed event duration assumed previously.

- In [17], Choi *et al.* proposed a test sequence generation algorithm in a formal way using logical clocks. Their work aims to solve both controllability and observationally problems occurred in distributed testing with concurrent events. The proposed algorithm is generic and can be used for any possible communication paradigm. Authors of the paper took benefit from the use of logical clocks and hence they can make difference between concurrent events and causal ones by labeling and comparing the logical clock values of the events of a test sequence. In this new approach, the local testers generate additional signals to control concurrent events when these last can be

identified. Using reachability tree generation techniques, the authors demonstrated that the proposed algorithm can solve the so-called *contro-observation problem* in a formal way. This work proposed a new test architecture for solving the latter problem. A Specification and Description Language (SDL) tool is used to verify the correctness of the proposed algorithm. Yet, authors of [17] applied their algorithm to the message exchange for the establishment of $Q.2971$ point-to-multipoint call/connection[1] as a case of study.

- Recently, in [24], Ponce *et al.* extended the **ioco** conformance relation [85] to test concurrent distributed systems specified with true concurrency and hence they defined the **co-ioco** conformance relation. In [72], Ponce *et al.* assumed that global observation in distributed testing cannot be reconstructed from local observations made in local interfaces of a distributed system. Hence, they proposed to use vector logical clocks in order to regain global conformance from local testing. In this work, authors presented a framework which only considers synchronous communication for concurrent systems specified; for a first time; as a network of *Labeled Transition systems (LTSs)*; and then as one distributed *Petri net*. An adaptation of the previous test generation algorithm for **co-ioco** for handling vector timestamps was presented in this work.

Even though the works of Hierons *et al.* [44] and Gaston *et al.* [37] do not explicitly ground their approach on logical scalar or vector clocks, the way they treat causality of events for solving the oracle problem in distributed systems is similar to the one used by Lamport, Fidge and Mattern. However, the main goal of Lamport was to build a causal order between events observed when a distributed systems execute. The problem addressed in [44] and [37] concerns more the question:

Does it exist such an order which would make a group of observations of such an executions on different remote interfaces the witness of a correct global distributed system execution?

A part of the answer was tackled and a dressed in [37]. The work in [37] will be presented in more details in Section 4.3.

The complexity of the problem just discussed illustrates the consequences of the lack of observability when trying to solve the oracle problem in Distributed systems. Indeed, even though all internal communications happens in a total order, this order cannot be easily observed due to the lack of the global clock.

Observability problem is a situation where a tester cannot distinguish between the global sequence produced by the SUT and the one which is expected according to the distributed specification model despite those two traces being different. As depicted in Figure 4.5, let us consider a distributed architecture where there are two observers called, for instance, Obs_1 at local interface L_1 and Obs_2 at local interface L_2 respectively. In Figure 4.5(a), we consider that SUT produces the global sequence $\sigma_{SUT} = input(i_1).output(o_1).input(i_1).output(o_2).output(o_1)$ where the response to a

[1] In telecommunications, point-to-multipoint communication is communication which is accomplished via a distinct type of one-to-many connection, providing multiple paths from a single location to multiple locations[63].

first input i_1 at L_1 leads to output o_1 at L_1 and a second input i_1 leads to o_1 at L_1 and o_2 at L_2. After projection, local observer Obs_1 expects to observe the sequence $input(i_1).output(o_1).input(i_1).output(o_1)$ and local observer Obs_2 expects to observe the sequence $output(o_2)$. On the other hand, in Figure 4.5(b), the specification $SPEC$ contains global sequence $\sigma_{SPEC} = input(i_1).output(o_2).output(o_1).input(i_1).output(o_1)$ where o_2 at L_2 is an output in response to the first input i_1 rather than the second one. Although global observations σ_{SUT} and σ_{Spec} are different, they have the same projections at local interfaces L_1 and L_2, and hence, it is not possible to distinguish between them in distributed testing with multiple observers.

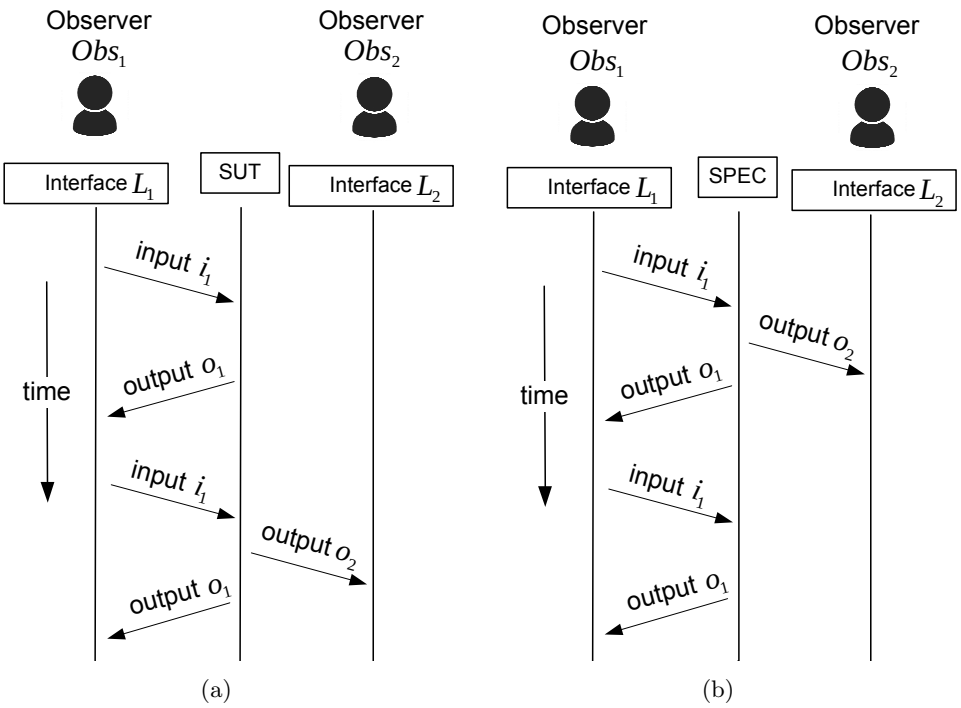


Figure 4.5: Observability problem in distributed testing with multiple observers

The lack of observability in distributed system testing is a cause of difficulties to identify deadlocks. A **deadlock** is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does. In a DS there is a difficulty to detect deadlocks. Hence, it is desirable to detect such problem while testing, indeed deadlock problem may hamper a DS that need to function extremely efficiently.

Figure 4.6 illustrates a deadlock situation in a distributed system composed of two local systems. Let us consider two systems labeled Sys_1 and Sys_2 that are exchanging internal messages i_1 and i_2. Sys_1 cannot send i_2 towards Sys_2, in fact, it is waiting to receive input i_1 which supposed to be sent from Sys_2, however, Sys_2 cannot send i_1 towards Sys_1, in fact, it is waiting to receive input i_2 which supposed to be sent from Sys_1 after the reception of i_1 from Sys_2. Both Sys_1 and Sys_2 are each waiting for the other to send its message, and thus neither ever does.

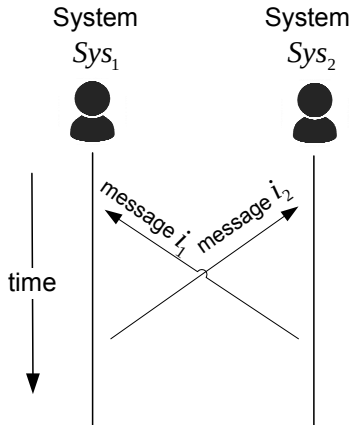


Figure 4.6: A deadlock situation in a distributed system

From the point of view of the tester having to stimulate the system under test, the counterpart of the observability problem is the so-called controllability problem.

Controlability problem is a situation where a local tester cannot determine when to apply a particular input to a SUT. This problem introduces non-determinism into testing.

For example, as depicted in Figure 4.7, let us consider a situation where a local tester T_1 at local interface L_1 applies an input i_1 to an SUT. This should lead to output o_1 at interface L_1, and a tester T_2 at local interface L_2 should then send input i_2. Here local tester T_2 is not able to know when to send i_2 since it does not observe the previous input and output at interface L_1.

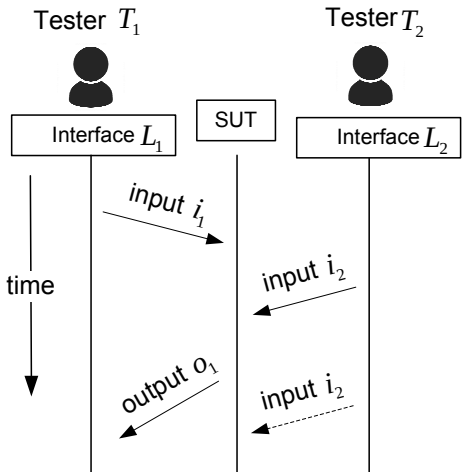


Figure 4.7: Controlability problem in distributed testing with multiple local testers

A consequence of the lack of controllability is the problem of *reproducibility of event* as Ghosh *et al.* underline [39] reproducing a specific behavior of a system is often required for testing. Yet, in distributed testing, reproducing specific execution behavior is often hard to achieve because of concurrent processing along with the presence of asynchronous communication and the lack of full control over the environment.

4.2 Distributed Testing Architectures

The activity of a tester consists in interacting with the SUT in order to execute the available test cases and then to observe the response of the SUT due to this stimulation[2].

According to [94] and [59], a given tester may be global or local. Hence, we may either associate only one global tester with the whole SUT or associate one local tester with each localized subsystem. A more general configuration exists and allows both previous situations [59]. These are the three possible testing architecture for testing a DS. They are respectively referred to *global-tester-based testing architecture*, *local-tester-based testing architecture* and *hybrid testing architecture* [94, 59]. Section 4.2.1, Section 4.2.2 and Section 4.2.3 are devoted, respectively, to discuss those three kind of testing architectures.

4.2.1 Global-tester-based testing architecture

We first start with the simplest architecture provided with a global tester, which entirely simulates the environment of the distributed SUT during a test run by means of a dedicated communication network . In Figure 4.8, we present a global-tester-based testing architecture[3] by interaction with it through the black connectors denoting the dedicated network. A global tester T_G may have total control over the distributed (SUT). The global tester is implemented as sequential machine T_G and runs in parallel with the distributed SUT observing and controlling if necessary all external and internal actions of the SUT (grey-box testing approach). In some global-tester-based testing approaches, the global tester centrally collects local observations of the distributed SUT made at local interfaces (without controlling) and derives a global test verdict.

An advantage of using this type of testing architecture is its simplicity. In fact, when a global tester is employed, a global view on the distributed SUT can be provided as a unique sequence which preserves the correct causal dependencies between the actions of the distributed SUT. However, one of the drawbacks of this architecture is that it requires strict control over the execution of distributed SUT when it presents concurrency. In particular, since the tester uses a dedicated network to communicate with localized sub-systems, one has to deal with the problem of introducing communication latency between the tester and the different sub-systems. The techniques dealing with this problem fall in the class of so-called *remote testing* techniques. For example, to illustrate this kind of approaches, in [46], Jard *et al.* used logical clocks to prove that remote asynchronous testing can gain the same power as local testing. Authors of [46] said that the tester needs to reorder events of the SUT using logical stamps in order to reach the same testing power as in synchronous local testing. Authors of [46] presented an operational technique to derive the correct test cases for remote asynchronous testing. In [23], David *et al.* presented a testing framework on black box remote testing of real-time systems using Uppaal-TIGA[4] testing tool. In [23] authors addresses the challenge of communication latency between the tester and the SUT in remote testing that may lead to interleaving of inputs and outputs.

[2] In case of monitoring, the role of the tester is limited to observe the behavior of the SUT and to decide whether the generated behavior is accepted or not

[3] Also referred to as "centralized" testing architecture

[4] http://people.cs.aau.dk/ adavid/tiga/

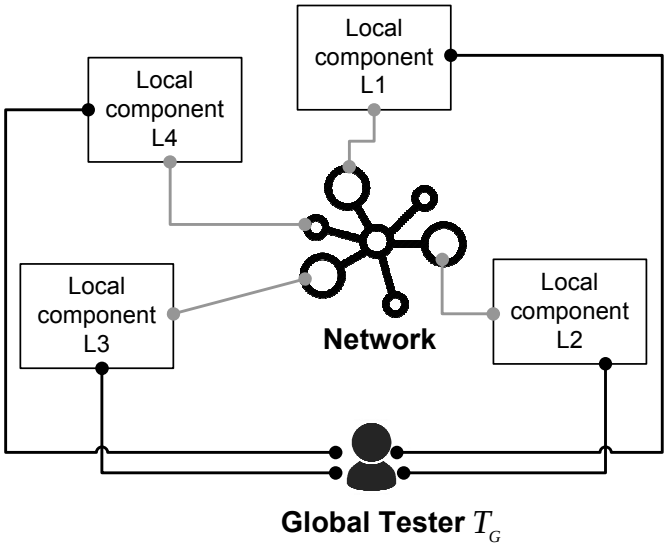


Figure 4.8: Global-tester-based testing architecture

4.2.2 Local-tester-based testing architecture

In Figure 4.9 we present a local-tester-based testing architecture[5] where a local tester T_i is associated with each local component and derives a local test verdict. It consists of several concurrently operating local testers which process together, but independently. In this testing architecture, the local tester T_i may control inputs from the environment and observe outputs occurring on channels connected to its local interface. In this architecture, the tuple consisting of all local testers (T_1, T_2, T_3) is referred as a *distributed tester*

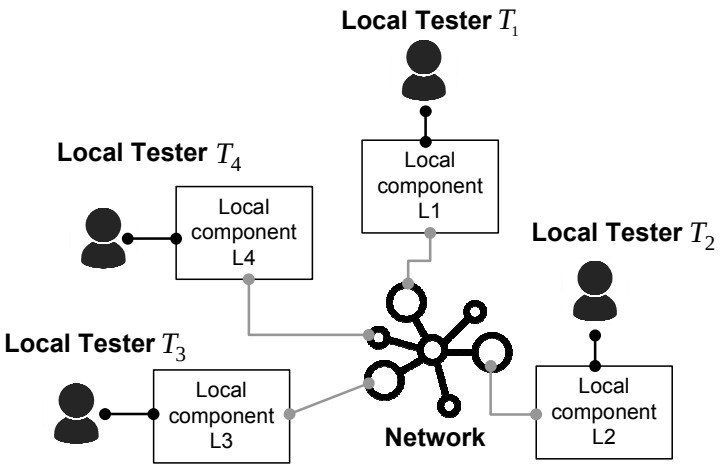


Figure 4.9: Local-tester-based testing architecture

[5] Also referred to as "decentralized" testing architecture

4.2.3 Hybrid testing architecture

In Figure 4.10 we present a hybrid-tester-based testing architecture. Hybrid testing architecture allows both global and local-based testing situations. That is, for a given component of the SUT, the latter may be connected to more than one tester simultaneously. On the other hand, a given tester may be either associated with one or several components of the SUT as well.

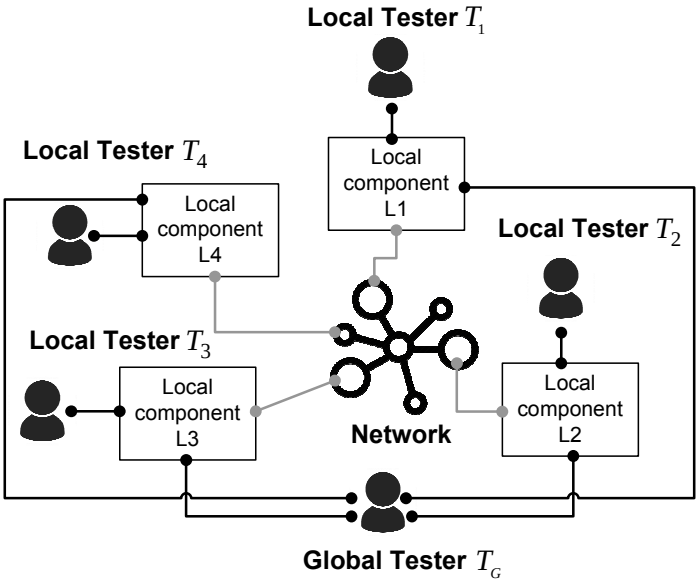


Figure 4.10: Hybrid testing architecture

In case of the presence of more than one tester (we refer to local-based or hybrid-based architectures), the correct global view on the behavior of the distributed SUT must be maintained by all existent testers. This can be achieved if coordination procedures are established between all testers. Coordination between testers can be implemented by exchanging messages between testers by means of communication shared channels in order to synchronize their activities. In this case, the different testers communicate with each other by exchanging messages about their respective observations about the SUT. Since we may deal with real-time systems, these exchanged messages should contain the instants at which these observations are made. In the absence of coordination between local testers, one of the drawback, is that a distributed tester may assign a successful verdict to a test run although the SUT contains faults. This is possible because there is no global view on the SUT if a distributed tester is used.

In case of local testing architecture, local testers testers T_i may coordinate their activities and communicate with each other via a shared channel (depicted in orange) like it is illustrated in Figure 4.11. In case of hybrid testing architecture, all testers (global one T_G and local ones T_i) may either coordinate their activities and communicate with each other via a shared channel (depicted in orange) like it is illustrated in Figure 4.12. In all presented testing architectures testers may also observe values (inputs and outputs) sent and received through internal channels. This technique is referred as *grey-box testing approach* [37].

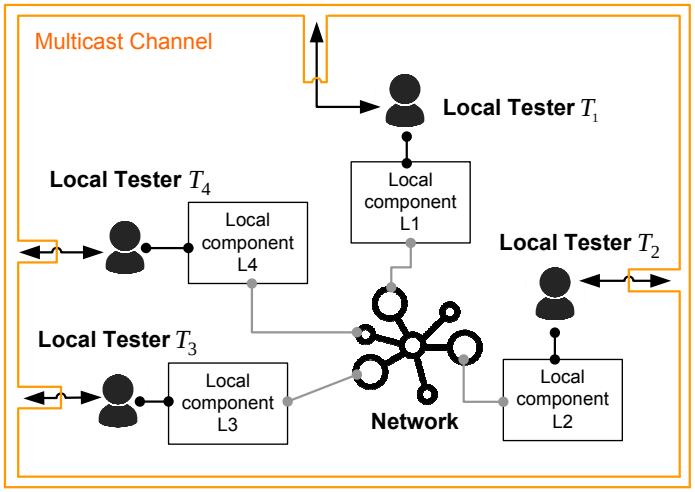


Figure 4.11: Local testing architecture with communication between testers

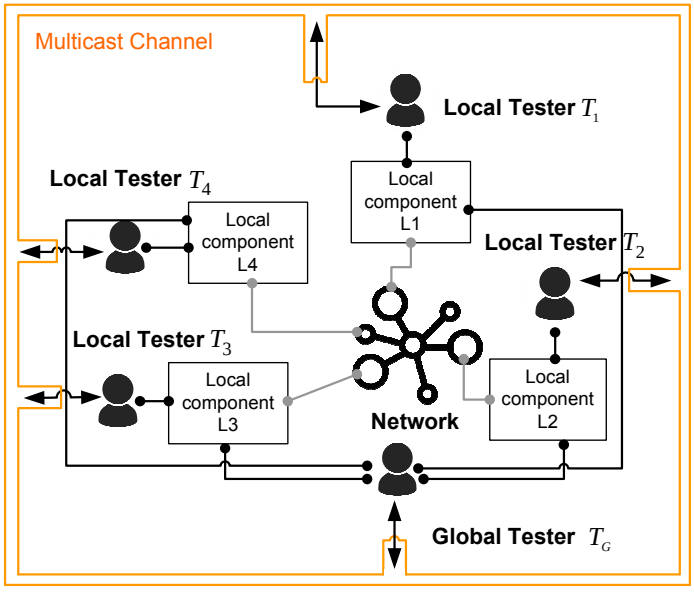


Figure 4.12: Hybrid testing architecture with communication between testers

4.3 A Baseline Approach to solve the Oracle Problem for Timed Distributed Systems

In this section, we present the general context on testing timed distributed systems. We use the works of [7] and [37] as a baseline approaches for our distributed testing approach.

In [7], the authors have proposed an off-line centralized symbolic testing framework (from test case generation to verdict computation), which provides algorithms for both test case generation and verdict computation based on *tioco* conformance relation. The work of [7] needed to be adapted to be used for our goal. This adaptation has been presented in Section 3.2.2. The core of the adaptation consisted in taking into account our new definition of timed traces based on events [12]. Moreover, our new centralized off-line

testing algorithm does not consider test purposes as the one presented in [7]. It only takes an execution time trace together with an execution symbolic tree of the TIOSTS reference model as an input and delivers a verdict as an output.

In the same context, the authors of [37] have proposed an extension of *tioco* [7] to deal with testing timed distributed systems and introduced the *dtioco* conformance relation. Moreover, they proposed an algorithm for solving the oracle problem in this context. Our work aims to provide a new timed distributed testing solution focusing on the oracle problem in the context of the *dtioco*. The main difference with the one in [37] is that we use constraints to symbolically deal with durations occurring in timed traces. Details about our approach are given in Section 4.4.

In Section 4.3.1 we will introduce the distributed testing architecture together with the testing assumptions used in [37]. This architecture, as well as those assumptions, are also assumed in our work. Details about how Gaston *et al.* [37] verified valid communication in a tuple of timed traces is given in Section 4.3.2.

4.3.1 The Distributed Testing Architecture and Hypotheses

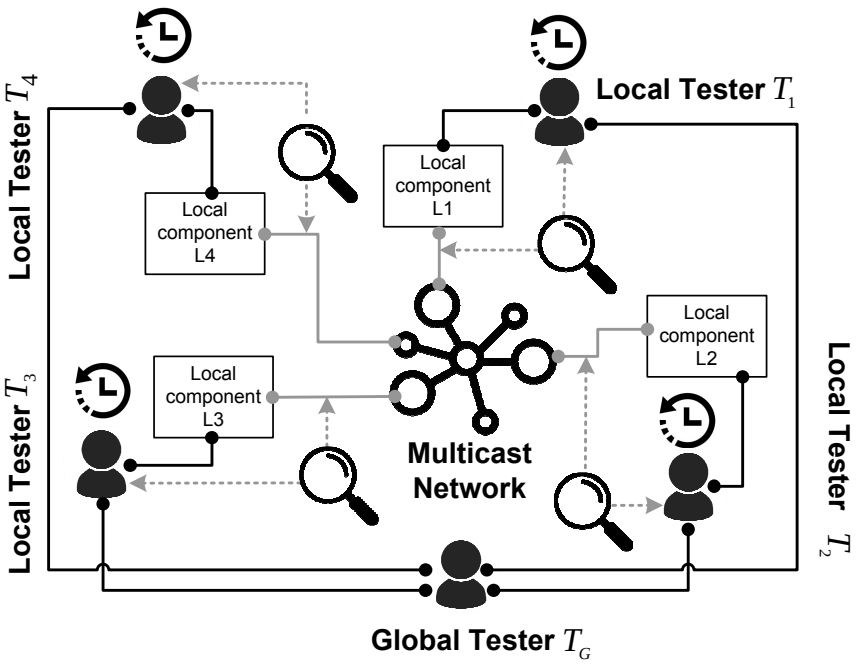


Figure 4.13: Our distributed testing architecture

Figure 4.13 depicts the testing architecture. It is a particular kind of the *hybrid testing architecture*. In Figure 4.13 we have limited the number of localized subsystems composing the distributed SUT to three for the sake of clarity. Hence, the SUT is composed of components L_1, L_2 and L_3. Each L_i has external channels connected to the environment and internal channels shared with other components in order to exchange values. A local tester T_i is associated with each localized subsystem L_i and a global tester T_G communicates with local testers T_i and collects observations made at local interfaces.

Moreover, as DSs present a network part, in testing distributed, we may allow a tester (local or global) to either observe or/and control internal communications between local components. Thus, so-called *grey-testing approach* is introduced as a distributed testing architecture [94].

In our testing context, each T_i may control inputs it submits to local component L_i and it may observe outputs occurring on the external channels connected to the environment. The local tester may also observe values passing through internal channels (grey-box testing). Each L_i executes in a centralized way so that behaviors observed by each local tester T_i can be viewed as timed traces (i.e, the local tester can observe the order of the actions occurring on its channels and can measure duration between consecutive actions).

In some real-time testing approaches [59], since each tester has its own local clock, a phase of clock synchronization [56, 41] is needed between the clocks of the testers. Yet, in our work, we assume that there is no global clock in the distributed system but only local clocks for each localized subsystem with no possibility of clock synchronization. By assumption, we assume that all local testers use clocks progressing at the same rate. In other words, there is no clock drift and therefore time units in the different timed traces of the different testers are the same.

Moreover, each local tester starts observing when its associated localized sub-system is reset. This is called the *local reset assumption*.

The semantics of such distributed systems using testing architecture described previously with these testing assumptions can be seen as tuples of timed traces (one component per localized system representing a local vision of the subsystem in question). In addition, we do not accept any communication between local testers.

4.3.2 Communication Checking

The aim of [37] was to define and implement an algorithm to test conditions which define consistency communication rules within a distributed system. Communications in the internal network are considered as *one-to-many multicast* (multicast for short). Multicast [21, 70, 3, 79] is the term used in network communication protocols standards [63] to describe communication where a piece of information is sent from one point called *source* to a subset of other points of the network called *destinations*.

Figure 4.14 depicts a multicast routing scheme in a communication network. In the case of multicast communication, as we have multiple receivers, we are immediately faced with two problems: (1) how to identify the receivers of a multicast message and (2) how to address the message sent to these receivers. A possible solution by using our architecture and deploying multicast communication is that a message sent by a unique component can be received by several recipients that listen on the shared channel of interest.

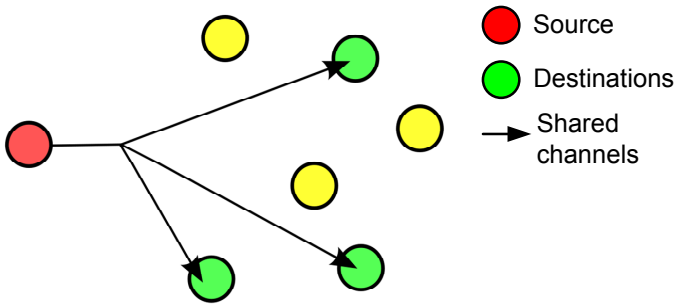


Figure 4.14: A routing scheme for multicast communication

In our context, messages are represented (referring to Chapter 3) as ordered sequences of events. In our case, multicast is modeled by the fact that localized subsystems share internal communication channels. Indeed, when a localized subsystem emits a message m on an internal channel c, we have that all other localized subsystems sharing the same channel c may receive the exchanged message m. The global tester retrieves all timed traces of each local tester and is in charge of analyzing them with respect to so-called *communication rules* [37].

An execution that may be observed by a global tester is denoted by a tuple of timed traces. An empty tuple whose all elements denote empty sequences corresponds to no interaction having occurred with the system is a correct tuple. We can extend a correct tuple by adding to any component either an input from the environment or an output. Outputs are considered as non-blocking when sent to the environment and when sent to other entities of the distributed system. There are two fundamental types of communications: internal ones are on shared channels and external ones that concern the local interfaces.

The time that takes a message to reach a recipient is not quantifiable, in fact, messages travel between interfaces within a multicast network and there is no global clock (we cannot measure the global time). Causality of communications is maintained in a correct tuple if a message cannot be received more often than the total number of emissions in the system. In a correct tuple, time elapses in the same manner for all local interfaces whose corresponding trace is not empty. Moreover, in distributed testing, we cannot make any suppositions on the different moments at which the different testers stop observing their corresponding interfaces. To take into account this issue, we accept as admissible observations, tuples of observations made of traces prefixes. In the sequel, we call all admissible tuples of observations of DS, *observable multitraces*. A tuple of timed traces which respects the previous rules is said to respect the so-called *observable multitrace property*.

Communication errors are observed only when an internal reception is identified for which no associated internal emission is found. Yet, no other type of communication action can reveal a communication error. In practice, such an error may represent a wrong message sent by the network itself. In [37] it is assumed that the local testers do not miss internal emissions because otherwise, one could not always know if an internal input has its corresponding internal output.

Communication Checking Process [37]

In [37], authors defined an algorithm to check whether a tuple of observations respects or not the observable multitrace property defined by system communication rules introduced previously. The algorithm introduced in [37] aims of studying all the possible temporal ordering of the communication actions occuring in a tuple under study. The execution of this algorithm can be represented as a tree whose initial root node is the tuple to be analyzed and each branch leads to a new node where either a communication action has been identified as the latest communication action to be added in the order or time has passed. In order to consider all the possible temporal synchronization, the durations occurring in traces of the tuple are decomposed in a basic common unit of time (for example the duration (3) is decomposed into the sequence (1).(1).(1)). Indeed, this is exactly what is done in practice since the clock itself imposes the basic delay defining the unity.

Process to check the observable multitrace property in a tuple of timed traces as presented in [37] consists in three main steps:

- Read the initial tuple of observations $(\sigma_1^c, \ldots, \sigma_n^c)$ from the beginning to the end.
- Store elements already read of $(\sigma_1^c, \ldots, \sigma_n^c)$ in a tuple observations $ot = (\mu_1, \ldots, \mu_n)$ which is considered as correct from a communication perspective (i.e, forms an observable multi-trace).
- Keep elements still to be read of $(\sigma_1^c, \ldots, \sigma_n^c)$ in a tuple of observations $mt = (\sigma_1, \ldots, \sigma_n)$.

The algorithm of [37] analyzes all configurations for interleaving emissions and receptions of different local components. Hence, it produces all combinations of couples (mt, ot) where mt stores elements still to be read of the original tuple of observations to check its observable property and ot stores elements already read by the algorithm. We note that any element μ_i (resp. σ_i) of mt (resp. of ot) is a prefix (resp. is a suffix) of σ_i^c (*i.e* for each i in $\{1, \ldots, n\}$ $\sigma_i^c = \mu_i.\sigma_i$).

Figure 4.15 depicts overall process to check the observable multitrace property. The algorithm produces a tree-like structure formed by all possible couples (mt, ot) for interleaving emissions and receptions of different local components. In each branch of this tree, the algorithm returns a verdict (True or False) about the correctness of the tuple of observations with respect to system communication properties. Process for checking observable multitrace property returns $True$ (ends with success) when it returns a configuration (mt, ot) where the tuple of observations to be analyzed mt is the empty tuple $(\varepsilon, \ldots, \varepsilon)$ and the read tuple of observations ot is identical to the initial tuple $(\sigma_1^c, \ldots, \sigma_n^c)$. It returns $False$ when all generated configurations (mt, ot) in the produced tree-structure are those where either $mt \neq (\varepsilon, \ldots, \varepsilon)$ or $or \neq (\sigma_1^c, \ldots, \sigma_n^c)$; in other words, if the reading of mt cannot be continued until reaching the empty tuple of observations $(\varepsilon, \ldots, \varepsilon)$, then the initial tuple to be checked does not valid observable multitrace property.

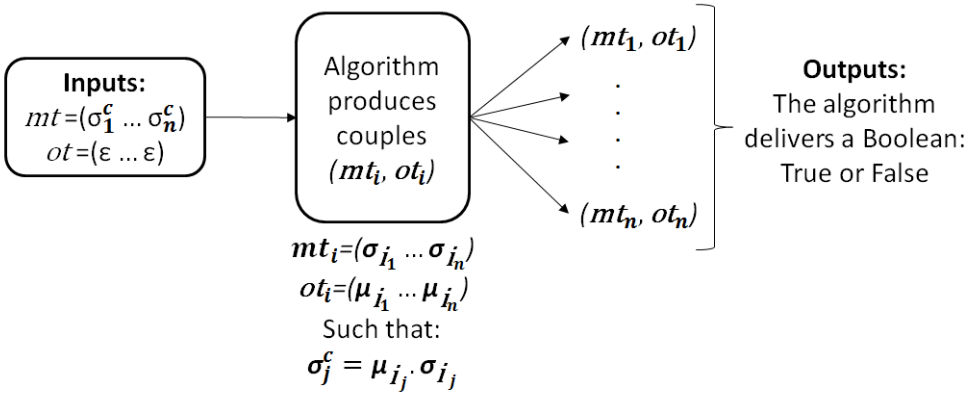


Figure 4.15: Communication checking process as introduced in [37]

Example 4.1. *We give a simple example of deploying the communication checking algorithm of [37]. We consider two communicating subsystems Sys_1 and Sys_2 (as depicted in Figure 4.16) exchanging internal messages m_1 and m_2 through channels c_1 and c_2 respectively. In addition Sys_1 may communicate through channels start for receiving messages from the environment and end for sending messages towards the environment. We consider the tuple of timed traces $\mu = (\sigma_1 = start?.(1).c_1!m_1.(2).c_2?m2.(1).c_2?m2.(1).end!, \sigma_2 = c_1?m_1.(1).c_2!m_2)$. Recall that in [37] the authors uses the classical formalism of timed traces defined as ordered sequences of actions (i.e, emissions annotated with ! and reception annotated by ?) separated by positive integers to denote durations elapsed between those actions.*

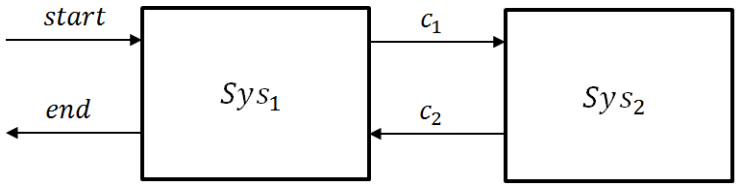


Figure 4.16: An example of a distributed system

We have that the tuple of timed traces $\mu = (\sigma_1, \sigma_2)$ constitutes a correct tuple of timed traces. Indeed, all receptions in μ have been preceded by an emission. Figure 4.17 depicts the tree-like structure produced by the execution of communication checking algorithm of [37]. Actions already read by the algorithm (i.e, emissions and receptions) are printed in orange, while, elapsed durations are printed in blue.

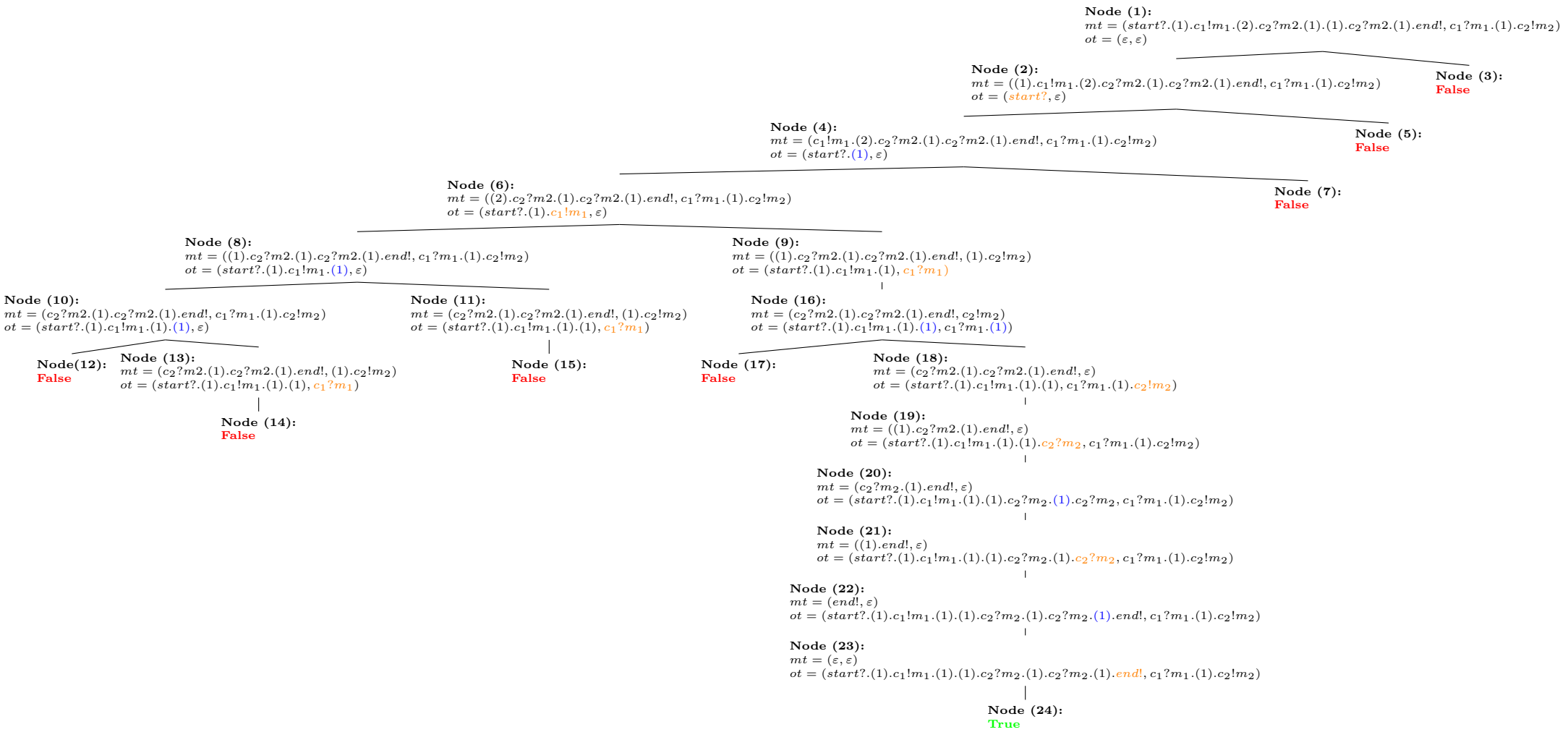


Figure 4.17: An execution of communication checking algorithm of [37] on a correct tuple of timed traces

Communication checking algorithm of [37] returns the value $True$ (that means ends with success) when the tuple to be analyzed mt is the empty tuple $(\varepsilon, \ldots, \varepsilon)$ and the read multitrace ot is identical to the complete initial multitrace $(\sigma_1^c, \ldots, \sigma_n^c)$. For example, in the node (24) we have succeeded to empty the initial tuple mt, hence, we end with $True$ at node (24). On the other hand, if the reading cannot be continued until reaching the empty tuple, then the initial multitrace does not represent an observable multitrace and the algorithm returns the value $False$.

The algorithm analyzes mt following two cases: either there exists a timed trace σ_i of mt beginning with an action a_i (i in $\{1, \ldots, n\}$) (Case (1)), or a duration d can be read on all admissible timed traces (Case (2)):

Case 1. *An action a_i can be read by the algorithm from σ_i. Hence, a_i will be removed from timed trace still to be read and added to the timed trace already read only if one of following conditions if fulfilled:*

- *Action a_i is a non-blocking reception from the environment. For example, in the node* (2)*, we have consumed start? as non-blocking reception from environment.*
- *Action a_i is an emission toward the environment or other subsystems. For example, in the node* (6)*, we have consumed $c_1!m_1$ as non-blocking emission from Sys_1 towards Sys_2. In the node* (23)*, we have consumed end! as non-blocking emission towards environment.*
- *Action a_i is a reception of a message m on the channel c coming from one of the other subsystems and the number of occurrences of a_i in μ_i (elements already read by the algorithm for the subsystem i) is strictly less than the number of emissions already read by the algorithm (i.e the number of $c!m$ occurring in ot) provided that none of subsystems j ($j \neq i$) that can emit on the channel c has a trace fully read i.e, $\forall j|$ subsystem j can emit on c, $\sigma_j \neq \varepsilon$. For example, at node* (6)*, we have a valid internal reception $c_1?m_1$ observed at subsystem Sys_2 because there is a sufficient internal emissions $c_1!m_1$ observed at subsystem Sys_1 and the trace σ_1 is not fully read.*
- *Action a_i is a reception of a message m on the channel c coming from one of the other subsystems and there exists a subsystem j that can emit on the channel c whose timed trace is already fully read i.e, $\exists j|$ subsystem j can emit on c, $\sigma_j = \varepsilon$. Indeed, when a local tester does not wait enough to observe the whole local timed trace, algorithm to check the observable multitrace property of [37] assume that the system can consume the internal input in question if there exists a subsystem which can emit on the same channel as the internal input's channel and whose its observed timed trace is fully read by the local tester in question. For example, at node* (21)*, we have a valid internal reception $c_2?m_2$ observed at subsystem Sys_1 because the trace σ_2 observed at Sys_2 is already fully read and subsystem Sys_2 may send internal message m_2 on channel c_2.*

In [37], the authors used the predicate $FullyRead(chan(a), mt)$ which is True when there exists $j \neq i$ s.t subsystem j can emit on $chan(a_i)$ where $chan(a_i)$ is the channel of action a) and the timed trace σ_j (occurring in mt) is the empty timed trace ε. This predicate is used for both sub-cases (3) *and* (4)*.*

Case 2. *A duration $d = 1$ can be read by the algorithm if one of the non-empty timed traces σ_i starts with a duration $d_i > 0$ and if for all timed traces σ_i starting with an action, the reading of the timed trace has not been started yet, i.e. $\mu_i = \varepsilon$. In this case, the duration d is subtracted from all durations d_i occurring at the beginning of traces σ_i (d_i is simply removed if $d_i = 1$) and added to the corresponding μ_i. For example, at node (4), we may elapse duration (1) because the reading of timed trace σ_2 has not been started yet. At node (16), as we have already started the reading of σ_1 and σ_2, we must elapse time identically for both subsystems Sys_1 and Sys_2.*

Example Analysis

Communication checking algorithm of [37] applied on the latter example generates a tree-like structure denoting all function calls that stand for couples created (mt, ot) where ot is the tuple already ready of timed traces which constitutes an observable multitrace and mt is the tuple to be read. Since there is no global clock but only local ones with no drift, the algorithm considers all configuration of interleaving emissions and receptions of different subsystems sharing the communication network. In this example, we note that the execution generates 24 recursive calls. The algorithm of [37] aims naively to empty elements of tuple mt and insert those elements into ot following rules to constitute an observable multitrace. As the execution time of the algorithm depends on the number of recursive calls, we note that this time will be clearly high if we choose to increase the number of timed traces forming the tuple in question.

The latter analysis shows that the latter algorithm's execution time depends naively on the number of couples (mt, ot) generated after every call. This can have a direct consequence to get this execution time increase clearly high. Hence, the question to ask is how to improve the algorithm's performance and then how to reduce its execution time.

Limitations of Algorithm presented in [37]

The algorithm of [37] for checking observable multitrace property presents an important limitation in terms of combinatorial explosion problem. We discuss this issue and we give a hint to a solution that avoids this problem in Section 4.4.2 when we introduce our new approach for checking communication.

The algorithm presented in [37] to check valid communication pattern in a tuple of observations of a DS deals with time in a fully numerical manner and required the analysis of all possible temporal synchronizations of local traces, one unit of time per unit of time; clearly, this lead to a combinatorial explosion. A solution for resolve this problem is to characterize the set of possible synchronizations in a symbolic manner, by constructing constraints carrying on durations occurring in local traces, and so, combinatorial explosion of the previous algorithm of [37] is circumvented.

Our contribution consists in proposing an algorithm for analysing tuples of observations according to or communication rules. This algorithm expresses the communications policy as a CSP and so standard constraint solvers can be used to solve it.

4.4 Constraint-based Oracle Algorithm

Herein, we introduce our distributed testing framework. As glimpsed in Section 4.1, a distributed system is described as a collection of localised components exchanging data through a communication network. To test distributed systems, we adapt and extend centralized testing framework of [7] devoted to timed unitary testing and based on an adaptation of conformance relation *tioco* [60, 61, 7]. Our associated testing architecture is the same as the one presented in [37] described in Section 4.3.1. In short, local testers have the power to observe what happens when localized subsystem communicate with the environment through external channels and what happens when localized subsystems communicate between them via the network through internal channels. Local testers are in charge of analyzing local traces with respect to a TIOSTS model modeling the intended behaviors of the localized subsystem to which it is connected. The global tester collects local observation within a tuple and emits a global verdict relating to the correctness of the distributed system w.r.t an adaptation of conformance relation *dtioco* [37]. As compared to [37] the main difference in our approach [12] is that the global tester relies on constraint solving techniques while in [37] it is based on enumeration techniques aiming at studying all the possible ordering of communication actions.

The remaining of this chapter is structured as follows: In Section 4.4.1 we give our formal preliminaries about the concepts of a valid communication in a distributed system. In Section 4.4.3 we model a distributed system as a collection of communicating TIOSTSs and we define an observation of a distributed SUT as a tuple of local timed traces as presented in Definition 2.15. In Section 4.4.2 we introduce a new algorithm to decide if a distributed observation respects a valid communication pattern based on constraint solving. Finally, in Section 4.4.3.3 we give our definition of the distributed conformance relation *dtioco*.

4.4.1 Distributed Systems and Communication

4.4.1.1 Observation of a Distributed System

We now define a *distributed interface* as a collection of localised interfaces.

Definition 4.1 (Distributed interface)**.** *A distributed interface is a tuple* $\Lambda = (C_1, \cdots, C_n)$*, with* $n \geq 1$*, where for all* $i \leq n$*,* C_i *is a set of channels such that for any* $i \neq j$ *we have* $C_i^{out} \cap C_j^{out} = \emptyset$*.* $C(\Lambda)$*, which is equal to* $\bigcup_{i \leq n} C_i$*, is the set of channels of* Λ *with* $C(\Lambda)^{in} = \bigcup_{i \leq n} C_i^{in}$ *and* $C(\Lambda)^{out} = \bigcup_{i \leq n} C_i^{out}$*.*

The condition $C_i^{out} \cap C_j^{out} = \emptyset$ ensures that for a channel c, messages emitted through c can only be emitted from a unique sender to model one-to-many multicast communication mechanism. This is a simplification hypothesis that makes the later formalisation lighter.

Definition 4.2 (Internal and External channels)**.** *Let* $\Lambda = (C_1, \cdots, C_n)$ *be a distributed interface. For a given localised interface* C_i *of* Λ*,* C_i^{int} *defined as* $\bigcup\{C_i \cap C_j | j \leq n \wedge j \neq i\}$ *is the set of internal channels of* C_i*.* C_i^{ext} *defined as* $C_i \backslash C_i^{int}$ *is the set of external channels of* C_i*.*

In our distributed architecture, the set of internal channels is used to exchange messages with other localised subsystems. On the other hand, the set of external channels is used to exchange messages with the system environment.

Notation 4.1. *Given a distributed interface* $\Lambda = (C_1, \cdots, C_n)$. *We let* $C^{int}(\Lambda)$ *denote* $\bigcup_{i \leq n} C_i^{int}$, $C^{ext}(\Lambda)$ *denote* $\bigcup_{i \leq n} C_i^{ext}$, *and* $Act(\Lambda)$ *denote* $I(\Lambda) \cup O(\Lambda)$ *with* $I(\Lambda) = \bigcup_{i \leq n} I(C_i)$ *and* $O(\Lambda) = \bigcup_{i \leq n} O(C_i)$. $I^{int}(\Lambda)$ *(resp.* $O^{int}(\Lambda)$*) is the subset of* $I(\Lambda)$ *(resp.* $O(\Lambda)$*) whose elements are inputs (resp. outputs) through internal channels. We let* $Act^{int}(\Lambda) = I^{int}(\Lambda) \cup O^{int}(\Lambda)$, $Evt(\Lambda) = Evt(C(\Lambda))$, *and* $Evt_{in}^{int}(\Lambda)$ *be the set of events whose action is an internal input. For any* $c!v \in O(\Lambda)$, $Sender(\Lambda, c!v)$ *stands for the index* j *such that* $c \in C_j^{out}$.

A *distributed observation* will be a tuple of timed traces where each timed trace represents a local observation.

Definition 4.3 (Distributed observation)**.** *Let* $\Lambda = (C_1, \cdots, C_n)$ *be a distributed interface. A distributed observation is a tuple* $(\sigma_1, \ldots, \sigma_n)$ *where each* σ_i *is in* $TTraces(C_i)$ *with* $i \leq n$. $Tup(\Lambda)$ *denotes* $TTraces(C_1) \times \ldots \times TTraces(C_n)$ *is the set of all distributed observations made over* Λ.

Example 4.2 (Distributed interface and distributed observation)**.** *In Example 2.5 an illustration of a TLC system is given. This TLC communicates with the environment to perform some control and safety process. Herein, we model the environment as another TLC which has a symmetric role as the central TLC. The collocation between the two TLCs defines a so-called Train Control System (see Figure 4.18) that may be represented by a distributed interface* Λ_{TCS}.

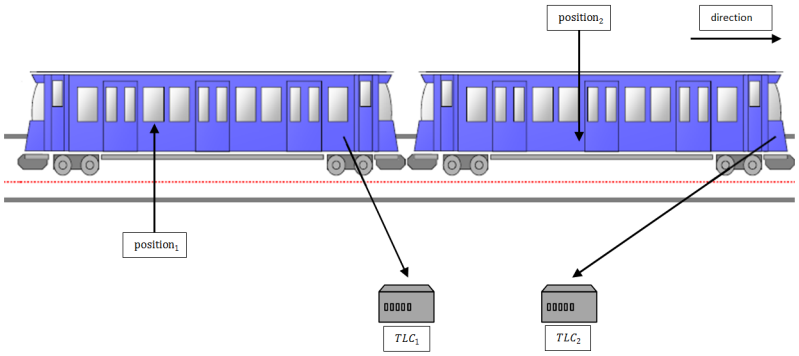


Figure 4.18: Train Control System Example

Distributed interface. *A Train Control System (TCS) which is depicted Figure 4.19 as two black-box communicating systems; is a system designed to ensure safety by monitoring locations of trains and locomotives, providing analysis and reporting, and automation of track warrants and similar orders. In this example we analyze a TCS involving two TLC components, one per train (say train 1 and train 2), going in the same direction on a rolling stock. The couple* $\Lambda_{TCS} = (C_{TLC_1}, C_{TLC_2})$ *defines a distributed interface through which the two TLCs may communicate to ensure that the train on the rear side automatically decreases speed as soon as the one in front of it is too close. The* C_{TLC_i} *is a localized interface that is used for exchanging messages through external channels:* $start_i, driver_i$ *and* $emergencyMode_i$ *for communicating with the environment and through internal channels:* pos_i *for sending internal messages and* pos_{3-i} *for receiving internal messages.*
Namely, we have $C_{TLC_i}^{ext} = \{start_i, driver_i, emergencyMode_i\}$, $C_{TLC_i}^{int} = \{pos_i, pos_{3-i}\}$ *and* $C_{TLC_i} = C_{TLC_i}^{ext} \coprod C_{TLC_i}^{int}$ *with* i *in* $\{1, 2\}$.

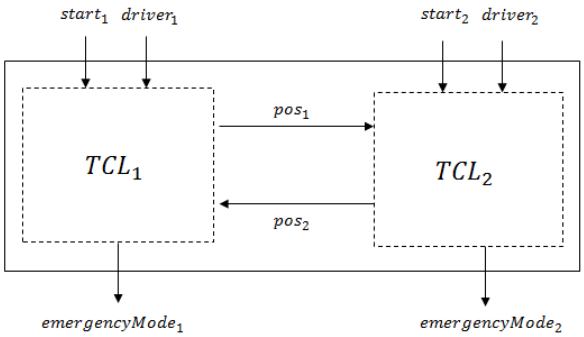


Figure 4.19: Distributed interface of the Train Control System as two communicating black boxes: TLC_i, for $i = 1, 2$

Distributed observation. *An example of a distributed observation* $\mu = (\sigma_1, \sigma_2)$ *is depicted in Figure 4.20.*
$\mu = (\sigma_1 = (1, start_1?).(1, pos_1!42).(3, pos_2?300), \sigma_2 = (2, start_2?).(1, pos_2!300).(2, pos_1?42))$
μ *defined in* $Tup(\Lambda_{TCS})$ *with* σ_i *a timed trace defined in* $TTraces(C_{TLC_i})$ *for* i *in* $\{1, 2\}$.

Time trace σ_1 *corresponds to the following behavior: the train* 1 *starts its execution after waiting* 1 *time unit from initialization and sends its relative position after that* 1 *time unit elapses; then it is notified with the relative position of train* 2 *after* 3 *time units. Time trace* σ_2 *corresponds to the following behavior: the train* 2 *starts its execution after waiting* 2 *time units from initialization and sends its relative position after that* 1 *time unit elapses; then it is notified with the relative position of train* 1 *after* 2 *time units.*

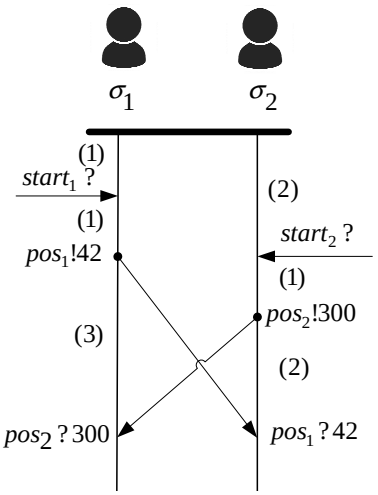


Figure 4.20: An example of a distributed observation of TCS

In the sequel, a distributed interface $\Lambda = (C_1, \ldots, C_n)$ is supposed given.

4.4.1.2 Valid Communication of a Distributed System

A valid observation of a distributed system is a tuple of local observations which respect a set of communication rules transposed from [37] to fit with our definition of timed traces based on events. We use the notion of a *multi-trace*, which is a tuple of timed traces

characterizing compatible communications between a collection of localised components as follows:

- A tuple of timed traces with empty interactions is a multi-trace
- External events with non-blocking outputs are always sent to the environment.
- Extremal events with user inputs are never refused from the environment.
- Internal exchanged events between local components must respect a so-called *causality of communications*. i.e.
  - A n^{th} occurrence of an internal input can be received if earlier at least n occurrences of the corresponding output have already been sent.

In the following, we introduce some intermediate functions useful to the definition of *multi-traces*.

Notation 4.2. *For $\sigma \in TTraces(C)$, $dur(\sigma)$ denotes the* duration of σ*, which is* 0 *if σ is ε, and otherwise is the sum of all delays of events in σ. Moreover, for an action a in $Act(C)$, $|\sigma|_a$ denotes the number of occurrences of a in σ. $pref(\sigma, a, n)$ stands for the smallest prefix of σ that contains n occurrences of a when this prefix exists. Finally, using the $pref$ operation, we introduce an operation that measures the elapsed time at the nth occurrence of an event a from the beginning of the trace. By convention, if a trace contains strictly fewer than n occurrences of a, then the associated duration is that of the entire trace.*

$$dur\_occ(\sigma, a, n) = \begin{cases} dur(pref(\sigma, a, n)) & \text{if } pref(\sigma, a, n) \text{ exists} \\ dur(\sigma) & \text{othewise} \end{cases}$$

We define the notion of a *multi-trace*, which is a tuple of timed traces characterizing compatible communications between a collection of localised components.

Definition 4.4 (Multi-traces)**.** *The set of* initialized multi-traces of Λ*, denoted $IMTraces(\Lambda)$, is the subset of $ITraces(C_1) \times \cdots \times ITraces(C_n)$ defined as follows:*

- ***Empty multi-trace:*** *$(\varepsilon, \cdots, \varepsilon) \in IMTraces(\Lambda)$,*
- ***multi-trace Extension:*** *for any $\mu = (\sigma_1, \ldots, \sigma_n) \in IMTraces(\Lambda)$, for $ev \in IEvt(C_i)$ for $i \leq n$, $(\sigma_1, \ldots, \sigma_i.ev, \ldots, \sigma_n) \in IMTraces(\Lambda)$ provided that: if $act(ev) \in I(C_i) \cap I^{int}(\Lambda)$, we have $|\sigma_j|_{\overline{act(ev)}} \geq |\sigma_i|_{act(ev)} + 1$ and $dur\_occ(\sigma_j, \overline{act(ev)}, |\sigma_i|_{act(ev)} + 1) < dur(\sigma_i.ev)$ with $j = Sender(\Lambda, \overline{act(ev)})$.*

The set $UMTraces(\Lambda)$ of uninitialised multi-traces *of Λ is $\{(u(\sigma_1), \cdots, u(\sigma_n)) | (\sigma_1, \cdots, \sigma_n) \in IMTraces(\Lambda)\}$.*
Finally, the set $MTraces(\Lambda)$ of multi-traces *of Λ is the set $UMTraces(\Lambda) \cup IMTraces(\Lambda)$.*

An empty multitrace whose all elements denote empty sequences corresponds to no interaction having occurred with the environment. We can extend a multitrace by adding to any trace either an input from the environment or an output. Outputs are considered as non-blocking when sent to the environment and when sent to other localized subsystems of the distributed system. There are two fundamental types of communications: internal ones are on shared channels and external ones that concern the local interfaces. We consider an

internal communication as one- to-many multicast: This means that a message sent (from a unique sender) can be caught by the multiple receivers who might listen on the channel in question. Finally, a message cannot be received more often than the total number of emissions in the system provided that time elapses in the same manner for all local interfaces.

Initialized multi-traces denote tuples of traces observed at local interfaces given a common distributed execution. Each timed trace occurring in an initialized multi-trace starts with an event introducing a duration. All those durations are supposed to start at a common initial instant. Yet, in the context of communicating distributed systems, it is generally not possible to observe a common initial instant which synchronises distributed executions. Indeed, there is no global clock for ordering distributed events but only local clocks to order events on localized interfaces. Therefore, we define *uninitialized multi-traces* as the tuple of distributed observations in which the initial durations are not observable.

Example 4.3 (Multi-traces)**.** *From Example 4.2 we have that tuple of timed traces* $\mu = (\sigma_1, \sigma_2)$ *defines a distributed observation defined in* $Tup(\Lambda_{TCS})$.

Initialized multi-traces. *The tuple* μ *is an initiated multi-trace in* $IMTraces(\Lambda_{TCS})$ *since there is an initial observable instant at the beginning of timed traces* σ_1 *and* σ_2*and since it respects valid communication pattern described in Section 4.4.1.2. Indeed,* μ *corresponds to a situation in which all receptions have been preceded by an emission; in particular, there are exactly as many emissions as receptions, and measured duration to observe a reception is indeed strictly greater than observed duration before sending input corresponding to the reception in question. For instance, we need* 4 *time units before that we receive position of train* 2 *(observation of action* $pos_2?300$ *in* σ_1*), this reception is indeed possible when we have emission of position of train* 2 *after elapsing* 3 *time units in* σ_2 *(observation of action* $pos_2!300$ *in* σ_2*) and we have* $4 > 3$.

Uninitialized multi-traces. *As it is generally not possible to observe a common initial instant, we may consider all configurations for different initial moments of each subsystems and then multiple configurations of the uninitialized multi-traces. we define* $\mu' = (\sigma'_1, \sigma'_2)$ *as an uninitialized multi-trace (see Figure 4.21 for the two possible configurations).* μ' *in* $UMTraces(\Lambda_{TCS})$ *where:*

- $\sigma'_1 = u(\sigma_1) = (\_, start_1?).(1, pos_1!42).(3, pos_2?300)$
- $\sigma'_2 = u(\sigma_2) = (\_, start_2?).(1, pos_2!300).(2, pos_1?42)$

We note that both μ *and* μ' *are in* $MTraces(\Lambda_{TCS})$*, the set of all multi-traces.*

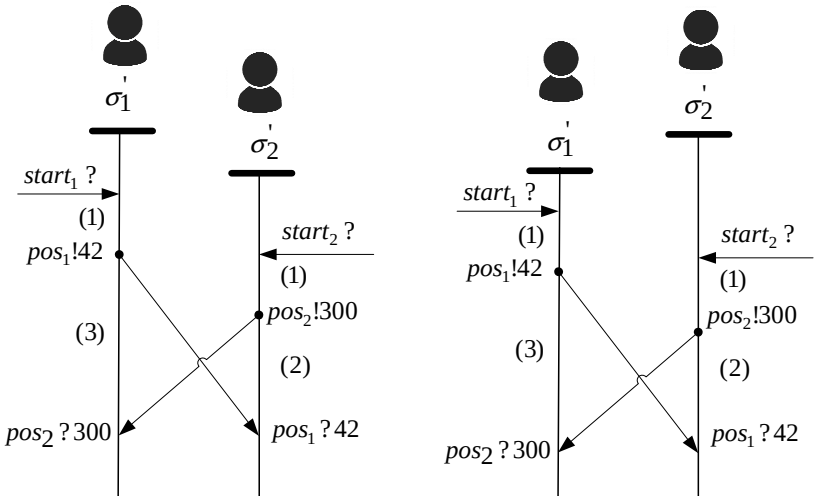


Figure 4.21: Multiple configurations for uninitialized multi-trace μ'

In distributed testing [37], we assume that there is a separate tester at each localized interface and there is no global clock for globally ordering distributed events. Hence, we cannot make any assumption on the different moments at which the different local testers stop observing their associated interfaces. To capture this, we accept as valid observations, tuples made of multi-trace prefixes.

Definition 4.5 (Observable multi-traces)**.** *The* set of initialised observable multi-traces of Λ, denoted $IOTraces(\Lambda)$, is the smallest set containing $IMTraces(\Lambda)$ and such that for any $(\sigma_1, \cdots, \sigma_i.ev, \cdots, \sigma_n) \in IOTraces(\Lambda)$ we have $(\sigma_1, \cdots, \sigma_n) \in IOTraces(\Lambda)$.
The set of *uninitialised observable multi-traces of* Λ, denoted $UOTraces(\Lambda)$, is the set $\{(u(\sigma_1), \cdots, u(\sigma_n)) | (\sigma_1, \cdots, \sigma_n) \in IOTraces(\Lambda)\}$.
Finally, the set $OTraces(\Lambda)$ of *observable multi-traces* of Λ is the set $UOTraces(\Lambda) \cup IOTraces(\Lambda)$.

Initialised observable multi-traces characterize observations starting at a common initial instant but ending at different instants depending on the considered component of the interface. Of course, since there is a common initial instant it is possible to order the moments at which the observations of the different traces of the tuple occur (σ_i ends before σ_j if $dur(\sigma_i) < dur(\sigma_j)$). However, as for multi-traces, in general, such an initial instant cannot be identified in testing. Therefore, real observations of system executions should be defined by tuples containing only uninitialised timed traces, which is captured by uninitialised observable multi-traces.

Example 4.4 (Observable multi-traces)**.** *We have that tuple of timed traces* $\mu = (\sigma_1, \sigma_2)$ *defines an initialized multi-trace in* $IMtraces(\Lambda_{TCS})$*. As we cannot make any assumption on the different moments at which the different local testers stop observing their associated interfaces, we define* $\sigma_1^{pref} = (1, start_1?).(1, pos_1!42)$ *as a prefix of* σ_1*; we have then* $\mu^{pref} = (\sigma_1^{pref}, \sigma_2)$ *is an initialized observable multi-trace in* $IOTraces(\Lambda_{TCS})$*. As in the case of uninitialized multi-traces, we define* $\mu'' = (\sigma_1'', \sigma_2'')$ *(see Figure 4.22) as an uninitialized observable multi-trace in* $UOTraces(\Lambda_{TCS})$ *where:*

- $\sigma_1'' = u(\sigma_1^{pref}) = (\_, start_1?).(1, pos_1!42)$

- $\sigma_2'' = u(\sigma_2) = (\_, start_2?).(1, pos_2!300).(2, pos_1?42)$

We note that both μ_{pref} and μ'_{pref} are in $OTraces(\Lambda_{TCS})$, the set of all observable multi-traces.

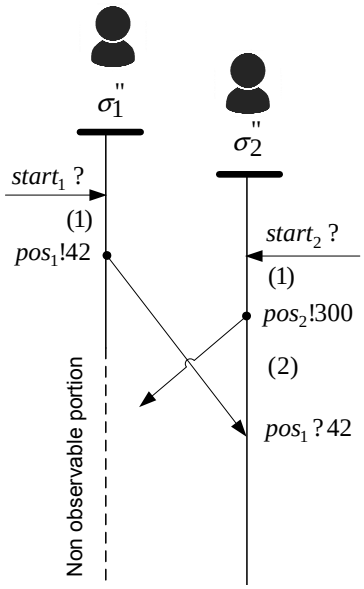


Figure 4.22: An example of an observable multi-trace

4.4.2 Constraint-based analysis for Communication Checking

Approach overview

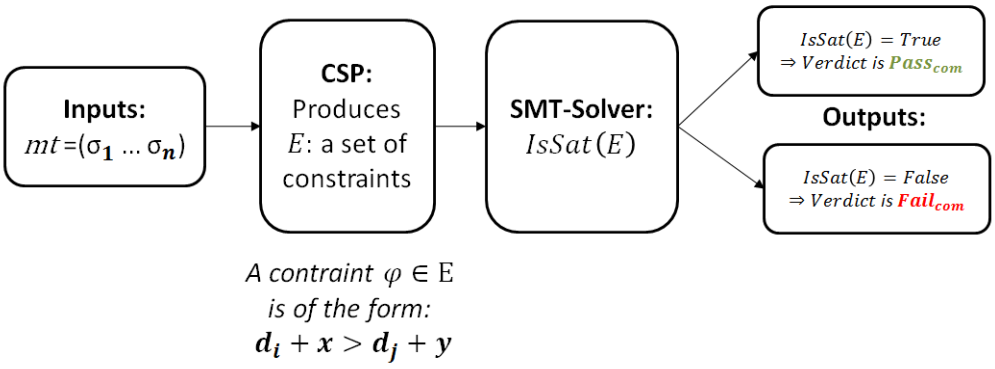


Figure 4.23: Constraint-based process for Communication Checking

In our approach for communication checking (as depicted in Figure 4.23), we use a constraint-based analysis for analyzing tuples of timed traces and checking whether or not those tuples satisfy the property of being an observable multitrace. This problem is formalized as a CSP [88] and so a standard constraint solver can be used (for example CVC4 [9], Yices2 [28] or Z3[26]).

Intuitively, our technique is based on the following intuition: any uninitialised observable multi-trace $\mu = (\sigma_1, \cdots, \sigma_n)$ is such that each σ_i is either empty or of the form $(\_, a_i).\sigma_i'$. Furthermore, μ has been obtained from an initialised observable multi-trace of the form $\mu' = (\sigma_1'', \cdots, \sigma_n'')$ where σ_i'' is ε if σ_i is ε and of the form $(d_i, a_i).\sigma_i'$ if σ_i is of the form $(\_, a_i).\sigma_i'$. Therefore, for some $ev \in Evt_{in}^{int}(\Lambda)$ and for some $(\sigma_1, \cdots, \sigma_n) \in UOTraces(\Lambda)$, one can decide whether $(\sigma_1, \cdots, \sigma_i.ev, \cdots \sigma_n) \in UOTraces(\Lambda)$ by determining whether there exists such durations $d_1, \cdots, d_n$ satisfying $\mu'' = (\sigma_1'', \cdots, \sigma_i''.ev, \cdots \sigma_n'') \in IOTraces(\Lambda)$.

Based on this principle, we will consider that those symbolic durations exist if by considering them as n variables $d_1, \cdots, d_n$ typed in D, we are able to construct a constraint on those variables characterizing these properties. More precisely, we construct a constraint such that each interpretation of $\{d_1, \cdots, d_n\}$ that satisfies the constraint can be used to define such an initialised observational multi-trace μ''. In the sequel, we present formally the algortihm to check valid communication in a distributed observation.

Constraint-based Algorithm

We introduce our constraint-based approach for checking communication of a distributed system. We propose a (new) algorithm which expresses the communications policy as a constraint satisfaction problem and so a standard constraint solver can be used to check communication of a distributed system.

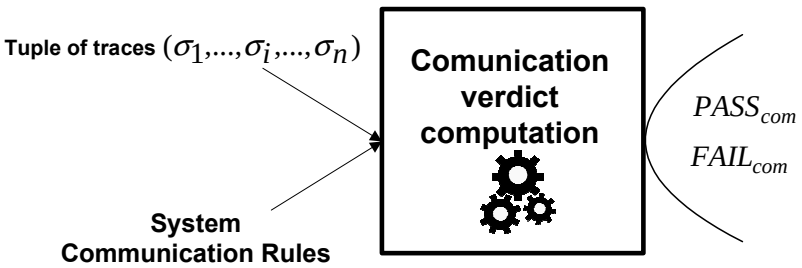


Figure 4.24: Communication testing algorithm and produced verdicts

For a tuple of timed traces $\mu = (\sigma_1, \ldots, \sigma_n)$ we may check its so-called *observable multitrace property*, by implementing the function:

$$ComChek_\Lambda : Tup(\Lambda) \times \{d_1, \ldots, d_n\} \to \{PASS_{com}, FAIL_{com}\}$$

As presented in Figure 4.24, the process to check valid communication pattern in a distributed observation μ returns a verdict $Verdict_{com}$ in the set of keywords $\{PASS_{com}, FAIL_{com}\}$ as follows:

- $Verdict_{com}$ is $PASS_{com}$: if and only if $\mu \in UOTRaces(\Lambda)$.
- $Verdict_{com}$ is $FAIL_{com}$: if and only if $\mu \notin UOTRaces(\Lambda)$.

The goal of Algorithm 1 is to check whether those observations reveal communication errors by checking whether they are in $UOTraces(\Lambda)$. Algorithm 1 is based on the property that an uninitialised observable multi-trace $\mu = (\sigma_1, \cdots, \sigma_n)$ is such that each σ_i is either empty or of the form $(\_, a_i).\sigma_i'$, but in the latter case μ has been obtained from an initialised

Algorithm 1: $ComChek_{\Lambda}(\mu, d)$: An algorithm to check valid communication pattern

Data: $\mu = (\sigma_1, \cdots, \sigma_n)$ tuple of timed traces, Λ distributed interface
Result: a verdict stating whether or not μ is an observable multi-trace

```
1  begin
2      E ← ∅;
3      for i ∈ [1 ··· n] do
4          ρ ← ε ;
5          foreach ev ∈ σ_i do
6              ρ ← Extend_C(ρ, ev) ;
7              if act(ev) ∈ I(C^int(Λ)) then
8                  a ← act(ev);
9                  j ← Sender(Λ, act(ev)‾);
10                 E ← E ∪ {(d_i + dur(ρ) > d_j + dur_occ(σ_j, a‾, |ρ|_a)}/* E ∈ F_Ω({d_1, ··· , d_n})
                   */;
11                 if ¬IsSat(E) then
12                     return FAIL_com   /* It's not an observable multi-trace  */;
13 return PASS_com              /* It is an observable multi-trace  */;
```

observable multi-trace of the form $\mu' = (\sigma''_1, \cdots, \sigma''_n)$ where σ''_i is ε for $\sigma_i = \varepsilon$ and of the form $(d_i, a_i).\sigma'_i$ for σ_i of the form $(\_, a_i).\sigma'_i$. Thus, $(\sigma_1, \cdots, \sigma_i.ev, \cdots \sigma_n) \in UOTraces(\Lambda)$ if and only if there exist durations $d_1, \cdots, d_n$ where $\mu'' = (\sigma''_1, \cdots, \sigma''_i, \cdots \sigma''_n) \in IOTraces(\Lambda)$.

We check whether such durations exist by considering them as n variables $d_1, \cdots, d_n$ (of type D); we construct constraints on these variables characterizing the properties of observable traces (line 10). By definition, only the occurrence of an internal input might break the property. There are two reasons for allowing an initialised observable multi-trace to be extended by an internal input. The first is that a sufficient number of corresponding internal outputs have previously been emitted. The second is that at the time when the extension is performed, the trace emitting the corresponding internal output is no longer observed.

If σ_i is the trace extended by internal input a, $\rho = \sigma_i.a$ (line 6) and σ_j is the trace at the interface that sends $\overline{a}$ (line 9), the first case correspond to situation in which $pref(\sigma_j, \overline{a}, |\rho|_a)$ exists and C: $d_i + dur(\rho) > d_j + dur\_occ(\sigma_j, \overline{a}, |\rho|_a)$ holds.

The latter case corresponds to situations in which $pref(\sigma_j, \overline{a}, |\rho|_a)$ does not exist and C': $d_i + dur(\rho) > d_j + dur(\sigma_j)$ holds. However, by definition of dur_occ, when $pref(\sigma_j, \overline{a}, |\rho|_a)$ does not exist we have that $dur\_occ(\sigma_j, \overline{a}, |\rho|_a) = dur(\sigma_j)$, which means that the constraints C and C' are equivalent. Therefore both cases can be treated in the same way by requiring that C holds, as is done in Algorithm 1. Every new constraint to be considered is added to the set E (line 10).

Example 4.5 (Checking communication in a distributed observation)**.** *In the following, we apply Algorithm 1 for checking communication on tuple of timed traces $\mu'' = (\sigma''_1, \sigma''_2)$ defined in Example 4.4 as an uninitialized observable multi-trace (see Figure 4.25(a)).*

To check communication in μ'' by using Algorithm 1, we first consider symbolic durations

d_1, d_2 in V_{time} and build the initialized tuple of timed traces (as depicted in Figure 4.25(b)): $\mu''_{init} = (\sigma''_{1_{init}}, \sigma''_{2_{init}})$ *where:*

- $\sigma''_{1_{init}} = (d_1, start_1?).(1, pos_1!42)$
- $\sigma''_{2_{init}} = (d_2, start_2?).(1, pos_2!300).(2, pos_1?42)$

We check communication in μ'' by determining whether there exists such durations d_1, d_2 in D satisfying $\mu''_{init} \in IOTraces(\Lambda_{TCS})$. Based on this principle, our algorithm analyses tuple of traces μ'', detects the only internal reception $(2, pos_1?42)$ and construct the constraint $d_2 + 1 + 2 > d_1 + 1$ relating the principle stating that in a multi-trace every internal reception must be observed before that its corresponding internal emission has been observed.

We have then, tuple μ'' is correct form communication perspective if and only if there exists d_1, d_2 in D such that $d_2 + 1 + 2 > d_1 + 1$ is satisfiable. In this case, a standard constraint solver (as Yices2) can be used to solve this problem and find for example interpretations, $d_1 = 1$, $d_2 = 1$ that may be a solution.

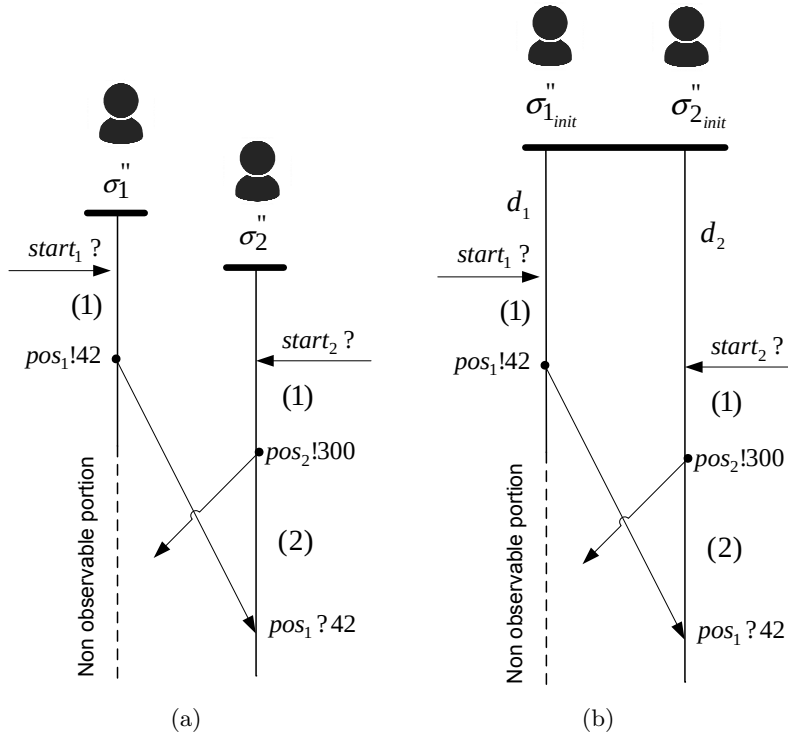


Figure 4.25: Communication checking of tuple of traces μ''_{init} using Algorithm 1

4.4.3 Modeling Timed Distributed Systems and Conformance relation

In this section we present our theoretical testing framework, that is how we denote distributed system specifications (distributed specifications for short), distributed systems under test and give the definition of our conformance relation.

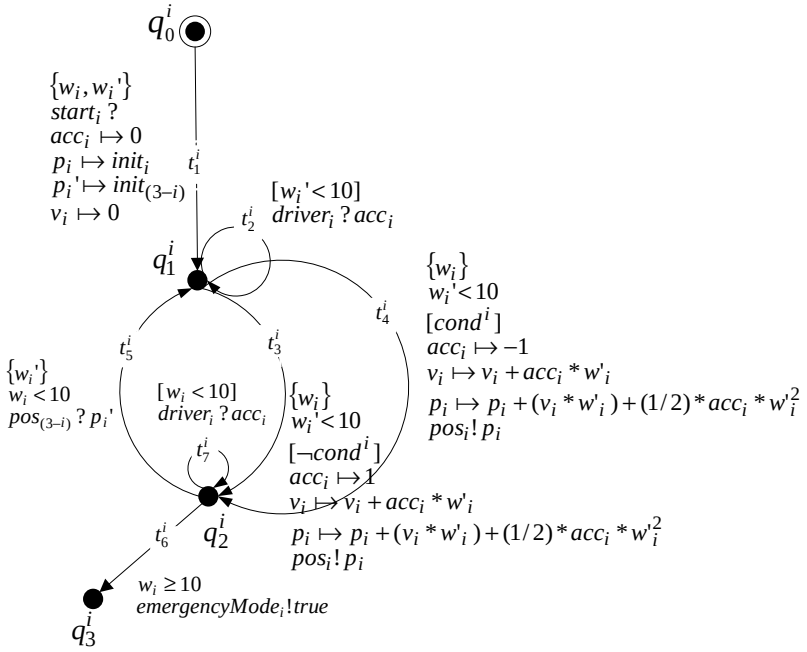
4.4.3.1 Distributed Specification

The next definition captures the notion of *distributed specification* and its associated *semantics*. We define distributed specifications over distributed interfaces as tuples of communicating TIOSTSs.

Definition 4.6 (Distributed Specification)**.** *Let $\Lambda = (C_1, \cdots, C_n)$ be a distributed interface. A* distributed specification over Λ *is a tuple $Spec = (\mathbb{G}_1, \cdots \mathbb{G}_n)$ such that for all $i \leq n$ we have $\mathbb{G}_i$ is a TIOSTS over a signature of the form $\Sigma_i = (A_i, T_i, C_i)$. The* semantics of *Spec, denoted $OTraces(Spec)$ is defined as $(TTraces(\mathbb{G}_1) \times \cdots \times TTraces(\mathbb{G}_n)) \cap UOTraces(\Lambda)$.*

Example 4.6 (Distributed specification)**.** *In Example 4.2 we defined the distributed interface $\Lambda_{TCS} = (C_{TLC_1}, C_{TLC_2})$. Herein, we model a TCS distributed specification as two communicating TLCs: TLC_1 and TLC_2 . A TLC_i is a TIOSTS communicating through localized interface C_{TLC_i} as described in Example 2.5.*

The symbol i should be replaced by two possible values, 1 and 2. The distributed specification $TCS = (TLC_1, TLC_2)$ as depicted in Figure 4.26 ensures that the train on the rear side automatically decreases speed as soon as the one in front of it is too close. The relative position of trains is given by their positions, which can be accessed by consulting the value of variable p_i: if $p_1 < p_2$, then train 1 is behind train 2. The TLC_i is an automata containing 4 states (q_0^i the initial state, q_1^i , q_2^i and q_3^i), communicating through C_{TLC_i} and having 4 data variables (acc_i in $\{-1, 0, 1\}$ for the acceleration of the train, v_i for the speed of the train, p_i for the position value of the train, p_i' for the estimation of the position of the other train) and 2 clocks (w_i, which is reset at each emission of the position and w_i', which is reset at each reception of the position of the second train).



TLC_i, for $i = 1, 2$

With: $init_1 = 42$ and $init_2 = 300$ and $cond^i \equiv (p_i < p_i') \wedge p^i \leq (v^i * 20) + 200$

Figure 4.26: Distributed Specification of the Train Control System

We recall that a TLC_i specifies the following behavior: After an initialization phase, the train of interest sends its position to the other train, and in return, the other train is supposed to send its position. In this loop, two consecutive communication actions are supposed to be separated by a delay of less than 10 units of time. If the remote train does not send its position on time, the local train goes into an emergency mode (not detailed here). At any moment in the loop, the driver may ask to modify the train acceleration. The new value is taken into account only if it does not affect the safety of the system (safety is threatened if condition ρ holds, that is, the distance between trains is less than the distance that can be covered by the rear train with the current acceleration). If safety is threatened, then the acceleration of the rear train is set to -1 in order to reduce its speed.

4.4.3.2 Distributed System Under Test

Similarly to distributed specifications, a Distributed System Under Test (DUT) is defined as a tuple of SUTs (as introduced in Definition 3.1).

Definition 4.7 (Distributed System Under Test)**.** *Let $\Lambda = (C_1, \cdots, C_n)$ be a distributed interface. A DUT over Λ is a tuple $\mathbb{DS} = (\mathbb{LS}_1, \ldots, \mathbb{LS}_n)$ where for all $i \leq n$ we have $\mathbb{LS}_i$ is an System Under Test defined over C_i. Moreover, $\mathbb{DS}$ is associated with a semantics denoted as a set $Obs(\mathbb{DS}) \subseteq \mathbb{LS}_1 \times \cdots \times \mathbb{LS}_n$. Elements of $Obs(\mathbb{DS})$ are called distributed executions of $\mathbb{DS}$.*

$Obs(\mathbb{DS})$ is a mathematical object representing all the possible executions of $\mathbb{DS}$. As such it is naturally defined as a subset of $\mathbb{LS}_1 \times \cdots \times \mathbb{LS}_n$. However, note that some tuples in $Obs(\mathbb{DS})$ may reveal communication faults due to a faulty communication network. Therefore we may not assume that we have $Obs(\mathbb{DS}) \subseteq UOTraces(\Lambda)$.

4.4.3.3 The *dtioco* Conformance Relation

In this section, we introduce a slightly adapted version of the conformance relation *dtioco* [37] that grounds our distributed testing framework. In our testing architecture, we made the assumption that for each localised interface, there is a local tester that controls external inputs at this interface and observes external outputs as well as internal inputs and outputs. Therefore, at each localised interface C_i, the corresponding tester does not interact with the full SUT $\mathbb{LS}_i$, but it rather interacts with $\mathbb{LS}_i$ in the context of the system $\mathbb{DS}$. This restriction of $\mathbb{LS}_i$ consists in all the traces of $\mathbb{LS}_i$ that occur in at least one tuple of $Obs(\mathbb{DS})$. This restriction is in our context defined as follows.

Definition 4.8 (SUT projection)**.** *Let $\mathbb{DS} = (\mathbb{LS}_1, \ldots, \mathbb{LS}_n)$ be a DUT defined over Λ. The projection of $\mathbb{LS}_i$ on $\mathbb{DS}$ is the SUT $\mathbb{LS}_i|_{\mathbb{DS}}$ defined over C_i that contains all timed traces σ s.t. there exists $(\sigma_1, \ldots, \sigma_i, \ldots, \sigma_n)$ in $Obs(\mathbb{DS})$ with $\sigma = \sigma_i$.*

We now introduce the conformance relation *dtioco*.

Definition 4.9 (*dtioco*)**.** *Let $Spec = (\mathbb{G}_1, \ldots, \mathbb{G}_n)$ and $\mathbb{DS} = (\mathbb{LS}_1, \ldots, \mathbb{LS}_n)$ be resp. a distributed specification and a DUT over the same interface Λ.*
*$\mathbb{DS}$ **dtioco** Spec if and only if:*

*Local conformance: $\forall i \leq n, \mathbb{LS}_i|_{\mathbb{DS}}$ **tioco** $\mathbb{G}_i$ and,*

Communication correctness: $Obs(\mathbb{DS}) \subseteq UOTraces(\Lambda)$.

Note that in [37], the *dtioco* definition was given in a different manner which consisted mainly in transposing the *tioco* definition to observable multitraces. Then a formulation very similar the one in Definition 4.9 was given by means of a theorem claiming $\mathbb{DS}$ **dtioco** *Spec* was equivalent to local conformance and communication correctness. For sake of simplicity here, we directly defined *dtioco* based on those two properties reflecting precisely our two steps testing process:

- Centralized testing of each projection of $\mathbb{LS}_i$ on $\mathbb{DS}$ using conformance relation *tioco* as introduced in Definition 3.2,
- Checking the correctness of internal communications to satisfy the observable multitrace property.

Section 4.5 presents the implementation of this testing process.

4.5 Implementation: Distributed Testing by Orchestration

Herein, we describe our implementation framework for distributed testing where an observation made in testing is a tuple of timed traces: one timed trace for each location. We focus on the problem of producing an automated solution to the oracle problem. The verdicts are produced according to the *dtioco* conformance relation as introduced in Definition 4.9. Producing those verdicts requires carrying out two activities: the (standard) problem of checking each local trace against its corresponding model (according to *tioco*); and checking that the tuple of timed traces conforms to a valid communication pattern. For that reason, the results produced by our algorithm will consist of a local verdict associated to each timed trace and a communication verdict associated to the tuple of timed traces.

To solve the standard problem of checking each local observation against the corresponding model, we use our laboratory tool: DIVERSITY [27] which offers MBT facilities based on SE technique. An algorithm was implemented in DIVERSITY [7] for off-line centralized testing. The implementation of this latter algorithm follows rules of the algorithm presented in [7] to compute a verdict. Furthermore, the implementation was developed in a way that a user has the ability to whether declare a test purpose or not, so to have a complete set of verdicts as presented in [7] when considering a test purpose in testing or a restricted set of verdicts as we presented in Chapter 3 when we do not consider a test purpose. To match our off-line testing framework presented in Chapter 3 we used the implementation without test purpose.

To solve the problem of checking that a tuple of timed traces conforms to a valid communication pattern, we give an implementation of Algorithm 1 to check communication in a distributed observation which translates the latter problem in terms of a constraint satisfaction problem so that an SMT-solver can be used to solve the problem.

Figure 4.27 depicts our implementation which orchestrates the different trace analysis processes previously discussed. Its constituents will be detailed in the remaining of that section.

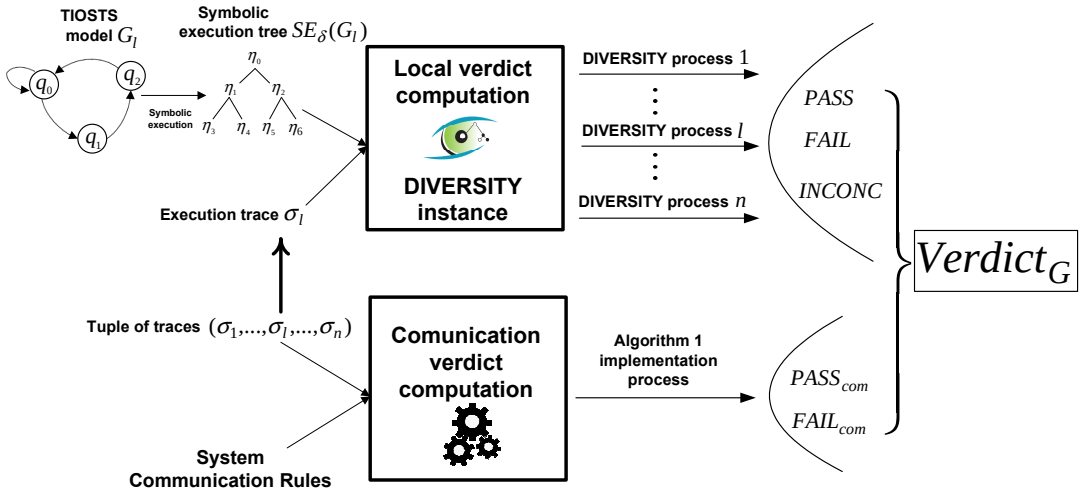


Figure 4.27: Implementation framework work-flow for distributed testing by orchestration

4.5.1 Off-line Centralized Testing

DIVERSITY [27, 5] is a multi-purpose and customizable platform for formal analysis based on symbolic execution. It has been designed for the purpose of managing the diversity of different semantics, but also the diversity of possible analysis based on symbolic execution (test generation, proof, deadlock search, etc.). DIVERSITY provides a pivot language called eXecutable Language for Interaction and Assemblage (xLIA) which is a generic language with a wide variety of primitives. xLIA has previously been used to encode specifications in SDL [93], UML statemachines [91], Simulink stateflow [1], etc. In particular, xLIA supports classical automata syntax involving symbolic data and communication actions. In our case, we encode the TIOSTS formalism with xLIA. We illustrate in Figure 4.28 an example of coding a transition with timing constraint depicted in Figure in DIVERSITY platform. We recall that a transition is defined by a tuple $(q, \mathbb{T}, \phi_t, \phi_d, act, q')$ (see Definition 2.7). Consider a transition $trans = (SourceState, \{clk\}, clk \leq TIMEOUT, True, c?x, TargetState)$. The action of tr is an reception of a symbolic variable x through channel c, $\mathbb{T}$ contains time variable clk, tr is executed provided that $clk \leq TIMEOUT$ is True and there is no data guard for tr (ϕ_d is $true$).

```
0  state SourceState {
1      transition trans --> TargetState {
2          input c( x );
3          tguard( clk <= TIMEOUT );
4      }
5  }
```

Figure 4.28: xLIA code for a symbolic transition

DIVERSITY employs symbolic execution techniques to generate a symbolic tree which represents all the possibles executions of the system. The symbolic tree is obtained by simulating the system specification with input symbols rather than concrete values for data. Each path of the tree has a constraint on input symbols, for the execution to follow that particular path. Sequences of concrete test inputs are computed by solving these path conditions using a constraint solver. For that purpose, DIVERSITY integrates solvers such

as CVC4 [6][9], Yices2 [7][28] and Z3 [8][26].

Based on those symbolic execution facilities, an off-line centralized testing algorithm for *tioco* is already implemented in DIVERSITY. It produces different verdicts $Verdict \in \{PASS, FAIL, WEAK\_PASS, INCONC_i, INCONC_r\}$ (see Section 3.2.1). It is possible to edit a configuration file in DIVERSITY in order to relax some input parameters of the algorithm, for example, to avoid considering the test purposes. We used such a configuration of DIVERSITY where we do not consider a test purpose. The only inputs of our restricted testing algorithm are the reference TIOSTS model and a timed trace to be analyzed. It produces a verdict in the set of keywords $\{PASS, FAIL, INCONC\}$ as discussed in Section 3.2.2.4 (see Figure 3.6):

- Verdict is $FAIL$ when an event with an unspecified output or delay is read from σ.
- Verdict is $PASS$ when there is a path in $\mathcal{SE}(\mathbb{G})_\delta$ covered by σ.
- Verdict $INCONC$ when an unspecified input is read from σ.

4.5.2 Communication Checking

In this section, we present our implementation of Algorithm 1 for checking communication of distributed systems. We have chosen Java as our main programming language and Eclipse as its developing tool for their flexibility. Figure 4.29 describes our implementation process for Algorithm 1. In the sequel, we present main functions that are used to implement Algorithm 1 to check the observable multitrace property.

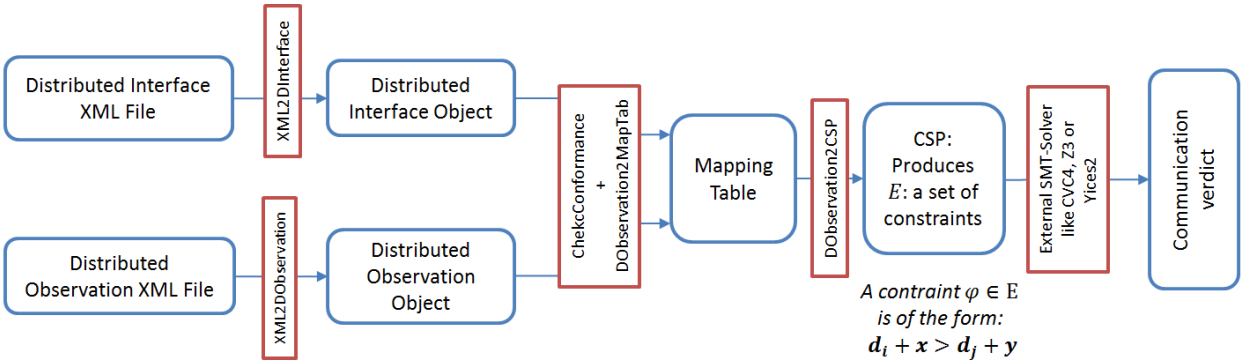


Figure 4.29: Implementation process of Algorithm 1

Main functions in our implementation framework are implemented within Java classes. Each class incorporates so-called *attributes* which denote variables and *methods* that are used when a class is instantiated as an object.

Table 4.1 describes those main functions and their associated packages.

[6] http://cvc4.cs.stanford.edu/web/
[7] http://yices.csl.sri.com/
[8] https://z3.codeplex.com/

Function	Package	Description
XML2DInterface	DInterface	Read an XML file containing information of a distributed interface and build its distributed interface object
XML2DObservation	DObservation	Read an XML file containing information of a distributed observation and build its distributed observation object
CheckConformance	CheckCom	Checks conformance between a distributed observation and a distributed interface
DObservation2MapTab	CheckCom	Read elements of a distributed observation and build a mapping table which associates each internal event of a trace with its duration measured form initialization
BuildConstraint	CheckCom	Constructs a constraint on the detection of an event whose action is an internal input at some designated place in a distributed observation
DObservation2CSP	CheckCom	Read elements of the mapping table associated with a distributed observation and build a file (in SMT-Lib) format containing the set of constraints related to the observation of the reception of each internal event

Table 4.1: Main functions used in our implementation framework to check communication

Distributed Interface Implementation

Package *DInterface* has a central class which includes one single attribute used to store data of a distributed interface after the reading of an XML file. The constructor of this class takes as argument XML file's name which stores the distributed interface data. Thanks to the use of an open source library from Java Application Programming Interface (API) for parsing XML files[9], The parser traverses the XML file and creates the corresponding Document Object Model (DOM)[10] nodes. These DOM objects are linked together in a tree structure. Once the parser is done, the user gets this DOM object structure back from it. Then we can traverse the DOM structure back and create the distributed interface object.

Here is an example of an XML file including a distributed interface data, and a DOM tree that illustrates the principle of turning XML into DOM:
Listing 4.1 depicts an example of XML file of TCS distributed interface illustrated in Example 4.2.

Listing 4.1: XML file containing a distributed interface description

```
<?xml version="1.0"?>
<DInterface name="TCS">
    <LInterface name="TLC1">
        <extern>
            <chan type="signal">start1:?</chan>
            <chan type="real">driver1:?</chan>
            <chan type="boolean">emergencyModel1:!</chan>
        </extern>
        <intern>
            <chan type="real">pos1:!</chan>
            <chan type="real">pos2:?</chan>
        </intern>
    </LInterface>
        <LInterface name="TLC2">
```

[9] see: http://docs.oracle.com/Javase/tutorial/jaxp/dom/readingXML.html
[10] The Document Object Model provides API that let the programmer create, modify, delete, and rearrange nodes

```
        <extern>
            <chan type="signal">start2:?</chan>
            <chan type="real">driver2:?</chan>
            <chan type="boolean">emergencyModel2:!</chan>
        </extern>
        <intern>
            <chan type="real">pos1:?</chan>
            <chan type="real">pos2:!</chan>
        </intern>
    </LInterface>
</DInterface>
```

The corresponding DOM structure of Listing 4.1 is depicted in Figure 4.30:

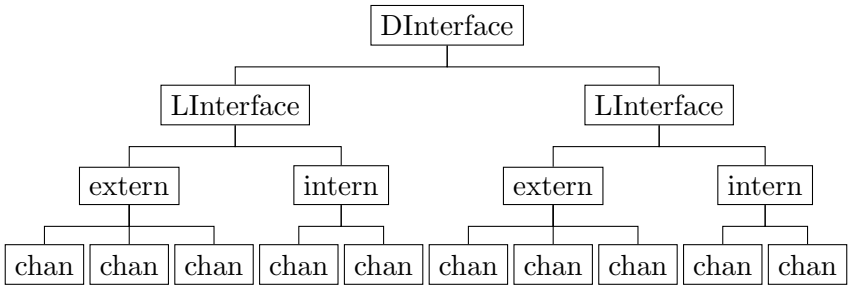


Figure 4.30: DOM generated after parsing XML file containing a distributed interface data

We give the corresponding output captured from Eclipse console as follows:

```
Distributed Interface "TCS":
0) Localized interface "TLC1":
    External channels: [start1:?:signal, driver1:?:real, emergencyMode1:!:boolean]
    Internal channels: [pos1:!:real, pos2:?:real]
1) Localized interface "TLC2":
    External channels: [start2:?:signal, driver2:?:real, emergencyMode2:!:boolean]
    Internal channels: [pos1:?:real, pos2:!:real]
```

Distributed Observation Implementation

Package *DObservation* has a central class package which includes one main class with one single attribute used to store data of a distributed observation from the reading of an XML file. The parser traverses the XML file and creates the corresponding DOM object which in its turn is used to generate the distributed observation object.

Listing 4.2 depicts the XML file of distributed observation illustrated in Example 4.4.

Listing 4.2: XML file containing a distributed observation data

```
<?xml version="1.0"?>
<DObservation dinterface="TCS">
    <trace linterface="TLC1">
        <event><duration>0</duration><action type="signal">start1?</action></event>
        <event><duration>1</duration><action type="real">pos1!42</action></event>
    </trace>
    <trace linterface="TLC2">
        <event><duration>0</duration><action type="signal">start2?</action></event>
        <event><duration>1</duration><action type="real">pos2!300</action></event>
        <event><duration>2</duration><action type="real">pos1?42</action></event>
    </trace>
</DObservation>
```

The corresponding DOM structure of Listing 4.2 is depicted in Figure 4.31:

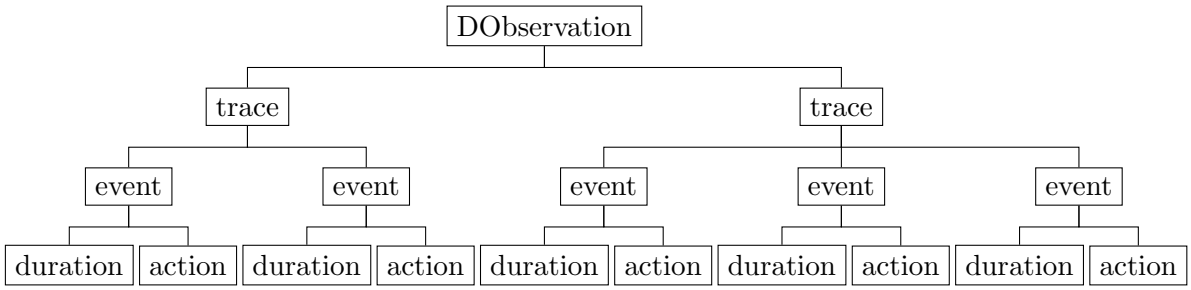


Figure 4.31: DOM generated after parsing XML file containing multitrace data

We give the corresponding output captured from Eclipse console as follows:

```
Tuple Of Traces TCS:[
    TLC1:(_,start1?).(1,pos1!42),
    TLC2:(_,start2?).(1,pos2!300).(2,pos1?42)
]
```

Communication Checking Implementation

In implementation of our communication checking algorithm, given a distributed interface $\Lambda = (C_1, \ldots, C_n)$, we provide a tuple of timed traces $\mu = (\sigma_1, \ldots, \sigma_n)$ defined in $Tup(\Lambda)$ with an a hash map used to associate each timed trace σ_i defined in $TTraces(C_i)$ with a array containing the duration measured from initialization of each internal input that might be received by localized interface C_i. This associative array[11] is updated in an iterative manner by analysing all elements of μ as follows: for a timed trace σ_i in μ, when encountering an event with input internal action act to be multicasted to localized interfaces C_j that might receive it (with $j \neq i$), we measure the duration from initialization of act and add it to the array associated with σ_j (with $j \neq i$). In this way, Our implementation of Algorithm 1 to check the observable multitrace property of μ may then easily analyse elements of μ and build the constraint associated to each detection of an internal communication between elements of μ. It builds a file (written in an SMT-Lib format[10]) that contains all constraints built after completely reading the tuple of traces μ.

Example 4.7. *To check distributed observation μ illustrated in Section 4.5.2, our implementation associates μ with the mapping table depicted in Table 4.2.*

Trace	Mapping Table
TLC1	[pos2!300:{1}]
TLC2	[pos1!42:{1}]

Table 4.2: Mapping associated with tuple of traces of Example 4.4

Listing 4.3 depicts a the corresponding SMT-Lib file created after execution of our implementation of Algorithm 1.

Listing 4.3: Output from Eclipse console after checking distributed observation of Example 4.4

[11] An associative array is an abstract data type composed of a collection of $(key, value)$ pairs, such that each possible key appears at most once in the collection.

```
(set-option :produce-models true)
(set-logic QF_LRA)
(declare-fun d1 () Real)
(declare-fun d2 () Real)
(assert (> d1 0))
(assert (> d2 0))
(assert (> (+ d1 1) (+ d2 3)))
(check-sat)
(get-model)
(exit)
```

Finally, an external call of an SMT-solver like CVC4, Z3 or Yices2 may solve the problem by finding an interpretation of symbolic duration that satisfies the formula built in Listing 4.3. Otherwise, the solver may return an error code.

Implementation of Algorithm 1 in Java is presented in Appendix C. In Appendix C.1, we present Java source code of function BuildConstraint to build a constraint on the detection of an internal input in a distributed observation. Appendix C.2 present Java source code of function DObervation2CSP which translate the problem of checking communication in a distributed observation into a CSP by building a file written in SMT-Lib format containing all constraints built on the detection of each internal input.

Evaluation

First, note that a class for carrying out unitary validation of all previous packages is also implemented. Our new algorithm to check communication in a distributed observation is more efficient than the one presented in [37]. Indeed, we have avoided the combinatorial problem of the algorithm of [37] by translating the problem of checking an observable multitrace property in a tuple if traces into a constraint satisfaction problem and so an SMT-Solver may return a solution. To validate the efficiency of our implementation in Java, we encountered a difficulty in measuring the execution time of the main function. The problem is that execution time was, in fact, different at each function run in Java and in a huge range. Professional engineer-testers report that benchmarking is indeed a difficult science especially if the programming language interacts with Central Processing Unit (CPU) through a Virtual Machine (VM) [12].

To figure out how to obtain correct measurements we choose to run the program multiple times and discarding the first run. Hence, a useful solution, for example, is to compute the mean value of the last 10 measurements.

4.5.3 Global Verdicts

We focus on the problem of producing an automated solution to the oracle problem. We require then, the implementation relation *dtioco* in order to determine the verdict of a test run (whether it is pass or fail) and produce a global verdict and we showed that to solve the oracle problem it is sufficient to carry out two activities: (1) we analyze the tuple of timed traces from the communication perspective by executing Algorithm 1 to check communication and (2) also each local timed trace of the tuple with respect to its associated local model by using the off-line testing algorithm implementation in DIVERSITY (without

[12] A virtual machine is an emulation of a particular computer system

considering a test purpose).

For this, we implemented an approach of testing a distributed system (focusing on the oracle problem) by separating the verification of local traces using DIVERSITY tool and the verification of the tuple of traces with respect to the definition of observable multi-traces. If there are n subsystems, the global verdict $Verdict_G$ has $n+1$ verdicts written in the form $(Verdict_1, \ldots, Verdict_n, Verdict_{com})$ where for l in $[1, \ldots, n]$, $Verdict_l$ is the local verdict in the set of keywords $\{PASS_l, FAIL_l, INCONC_l\}$ associated to the l^{th} component and where $Verdict_{com}$ which is the communication verdict in the set of keywords $\{PASS_{com}, FAIL_{com}\}$ is the verdict relating to the verification of the communication policy.

For this, we implement an orchestrator under which it is possible to analyse the tuple of traces from the communication perspective and also each trace of the tuple with respect to its associated local TIOSTS model. Our implementation runs (in parallel) $n+1$ processes $(Process_1, \ldots, Process_n, Process_{com})$ where:

- $Process_i$ (with $i \leq n$): is an instance of DIVERSITY tool for centralized testing a localized subsystem in a system of n components
- $Process_{com}$: is an instance of our implementation of Algorithm 1 for communication checking.

Orchestration Process

We build a script which orchestrates the work of running in parallel $n+1$ processes as we described previously. our script does not take into account what order the different processes completed in. However, it does not exit until all the spawned processes had exited. In other words, we do not exit the script until having all the testing verdicts.

The simplest way to achieve this is to use the *wait* command. We have simply forked all processes with & Linux command to run them in parallel, and then follow them with a wait command as follows (Listing 4.4):

Listing 4.4: An example of running in parallel n+1 processes to orchestrate testing

```
#!/bin/sh

/local/test/my-process-1 --args1 &
/local/test/my-process-2 --args2 &
/local/test/my-process-3 --args3 &
⋮
/com/test/my-process-n+1 --args-n+1 &

wait
echo all processes complete
print verdicts
exit
```

It is the simplest way to implement our orchestrator and print all testing verdict once we have analyzed fully the distributed observation. When one runs the script, all (n+1) processes will be forked in parallel, and the script will wait until all the processes have completed before exiting. Anything after the *wait* command will execute only after the

forked processes have exited.

Another way to implement our orchestrator is to determine the exit codes of the processes we forked. We may program our processes to associate each test verdict (either local or communication verdict) with an exit code. Since we need to know if any of the tests failed and return an error code from the parent shell script if they did. We, hence, may stop the orchestrator for example, once we have a process has returned a fail test verdict.

In Chapter 5 we present an approach to validate our distributed testing framework.

Chapter 5

Validating our Testing Approach

Contents

In order to validate our distributed testing approach described in Section 4.4, we adopt a step-by-step validation approach as presented in Figure 5.1. First, we generate so-called *correct distributed observations*; then, we follow a *mutation-based approach* in order to inject faults in them and produce so-called *faulty distributed observations*. Both correct distributed observations tuples and mutated ones will be submitted to our testing framework in order to observe corresponding testing verdicts and analyze results.

A Correct Distributed Observation (CDO) is a tuple of timed traces that respect valid communication patterns and where each local timed trace describes a correct behavior of its local component. On the other hand, a Faulty Distributed Observation (FDO) is a tuple of timed traces that either do not respect valid communication patterns or contain at least a timed trace that does not represent a correct behavior of the corresponding local component.

To generate and check a CDO, we adopt the following process: first, we introduce an algorithm to randomly generate observable multitraces (see Definition 4.5), i.e, prefixes of tuples of timed traces that respect valid communication patterns (see Definition 4.4). Those tuples will be used as inputs for Algorithm 1 in order to validate our communication

verification approach. Then, we use DIVERSITY tool[1] [5] to build local timed traces by projection, focusing on the behaviors of local components. Those local traces will be used as inputs for the rule-based algorithm for verdict computation described in Section 3.2.2.3. Moreover, DIVERSITY tool allows us to generate tuples of timed traces that are correct by construction with respect to both local analyses and communication rules by coupling composition and projection mechanisms. In fact, by using DIVERSITY, we are able to generate a global trace that represents the global behavior of a distributed system built by composing its local components. Then, resulting multitraces are directly constructed by considering a tuple made of all projections for each component. Those generated correct tuples of traces are submitted to our orchestrated testing framework.

Mutation Testing [69, 45, 55, 47] is a fault–based software testing approach which was first used in programs to inject faults that represent the mistakes that programmers often make. Such faults are deliberately seeded into the original program, by a simple syntactic change, to create a set of faulty programs called *mutants*, each containing a different syntactic change. Mutation-based testing promises to be effective in identifying adequate test data which can be used to find real faults. We adapt mutation-based techniques used in mutation testing with the aim to generate FDO from CDO (by injecting faults in them). For that, we apply random mutations on CDO and hence modify their data with the aim to inject communication errors or modify the correct behavior of localized components. Generated tuples of traces (after faults injection) are submitted to our orchestrator testing framework.

In order to assess the scalability of our testing framework, we apply our distributed testing approach on a case study of a significant size, called PhoneX, which is a telecommunication system for multiple call management provided by Ericsson company [77]. This case study will serve us to illustrate our approach of testing of distributed systems. We will be led first to design the distributed specification model, then to generate (from the model) *CDOs* with long local traces that are as representative as possible of normal use by using DIVERSITY. From those correct tuples of traces, we will simulate an execution of a faulty PhoneX distributed system by applying a mutation-based approach. Finally, we apply our testing approach by orchestration on those tuples and analyze corresponding results.

Overview. Section 5.1 describes our framework to generate observable multitraces randomly. In Section 5.2 we present our technique to generate correct distributed observations using DIVERSITY tool. In Section 5.3, we present a mutation-based technique to generate *faulty distributed observations*. That is, distributed observations with either potential local faults or communication ones. In Section 5.4, we apply our testing approach on the PhoneX study case and then, we give and comment some experimental results.

[1] http://projects.eclipse.org/proposals/eclipse-formal-modeling-project

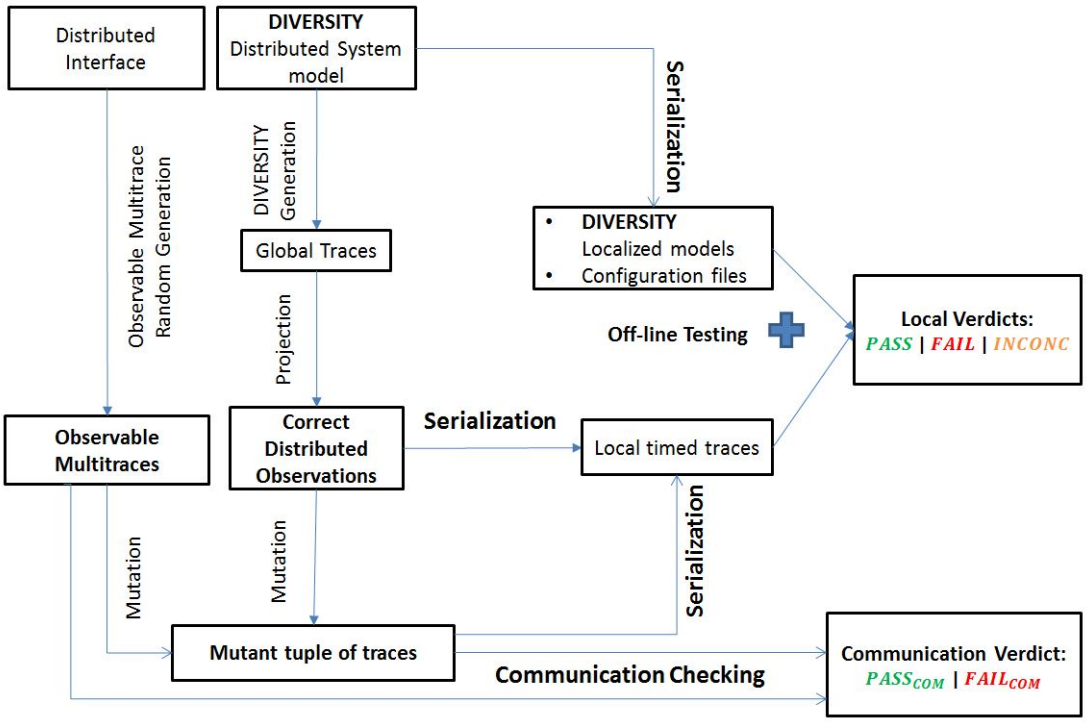


Figure 5.1: Validation process of our testing approach

5.1 Randomly Generating Observable Multitraces

Herein, we present our framework to generate randomly an observable multitrace. First, we define a function to generate randomly an uninitialized multitrace by following rules for building a correct tuple of timed traces accordingly to Definition 4.4. Randomness is used in choosing a localized interface in a distributed interface or generating non-null positive durations and data values to build actions and events. An observable multitrace may be generated then, by simply considering a prefix of a generated multitrace.

5.1.1 Generating multitraces

We build a correct tuple of timed traces by following communication policy rules by choosing randomly a channel of an internal output in a distributed interface and a data value (of the same data type) to be multicasted through this channel. The notion of multicast will be taken into account by storing those multicasted messages in queues of actions for components that might receive this message (i.e, which listen on the same channel) and dequeuing them when we choose the same shared channel for an internal input.

Like many other works, we use the notion of *queuing* to capture message-passing multicasting mechanism in communication between local components of a distributed system. Exchanged messages will be stored in the queue by preserving the same order of receipt.

FIFO Queues. A First In First Out (FIFO) queue is a type of data collection structure where any value inserted first, will be the first to be consumed. We introduce the signature $\Omega_{Elem} = (S_{Elem}, Op_{Elem})$ associated to the specification of so-called *elements* where S_{Elem} contains type $Elem$ to represent elements provided with successor and predecessor

operations as follows:

- $S_{Elem} = \{Elem\}$
- $Op_{Elem} = \{\emptyset :\rightarrow Elem,$
  $succ : Elem \rightarrow Elem\ (successor),$
  $pred : Elem \rightarrow Elem\ (predecessor)\}$

We give now the signature $\Omega_{Queue} = (S_{Queue}, Op_{Queue})$ for a queue of elements and we define the following classical operations:

- $S_{Queue} = S_{Elem} \cup \{Queue\}$
- $Op_{Queue} = Op_{Elem} \cup \{[\,] :\rightarrow Queue,$
  $isEmpty : Queue \rightarrow Boolean,$
  $dequeue : Queue \rightarrow Elem,$
  $enqueue : Queue.Elem \rightarrow Queue\}$

Table 5.1 describes the role of each operation related to the queuing mechanism:

Operation	Description
$[\,]$	Returns the empty queue
$isEmpty$	Returns $True$ if empty. It returns $False$ otherwise.
$dequeue$	For a non empty queue, returns the queue after deletion deprived of its first element
$enqueue$	Inserts a new value in the back of the queue.

Table 5.1: Main operations used in the definition of Ω_{Queue}

In the sequel, we denote by $Queue(E)$ the set of all queues defined over a generic set of elements E.

When multicasting internal messages between local components, we introduce the notion of *memory* as a way to store internal actions and the delay elapsed as perceived by local components. A *memory* is a couple $(d, \mathcal{Q})$ where d is the duration in D^+ accumulated when elapsing time synchronously for all timed traces and $\mathcal{Q}$ is a FIFO queue in $Queue(Act(C))$ in which we store received actions each time we multicast an internal output action from other timed traces. Those multicasted actions will be later consumed. The set of all memories defined over a distributed interface Λ is the set $Mems(\Lambda)$.

We assume that each tuple of timed traces $\mu = (\sigma_1, \cdots, \sigma_n)$ defined in $Tup(\Lambda)$ is given with a *tuple of memories* $(m_1, \ldots, m_n)$. We now introduce some useful technical functions:

Auxiliary Functions

Notation 5.1. *The notation $\mathbb{Z}_>$ denotes the set of strictly positive integers.*

In the sequel, let $\Lambda = (C_1, \ldots, C_n)$ be a distributed interface and C be a set of channels. Table 5.2 describes auxiliary technical functions that we will use in the implementation of generating multitraces mechanism.

Function - Might Receive an internal output

$MightReceive_\Lambda : Act(\cup_{i\leq n} C_i) \times \mathbb{Z}_> \rightarrow \{True, False\}$

Let act be and action in $Act(\cup_{i\leq n} C_i)$ of an internal output and i a position in $\{1, \ldots, n\}$.

Function	Description
$MightReceive_\Lambda(act, i)$	Returns $True$ if local interface C_i might receive internal output act. Returns $False$ otherwise.
$MultitcastMessage_\Lambda(act, i)$	Multicasts an internal output action to all memories (except the sender memory itself) that might receive it. It stores that action in the corresponding queue of actions of the receiver component
$Extend_C(\sigma, ev)$	Extends a timed trace with a new event
$mirror(act)$	Returns the mirror of an action.
$length(\sigma)$	Length of trace is defined as the total number of events in that trace
$size(\mu)$	Returns the total number of events of μ.
$uninitialized(\mu)$	Returns the uninitialized tuple of timed traces from μ.

Table 5.2: Technical functions used in generating a multitrace

$MightReceive_\Lambda(act, i)$ is $True$ if $chan(act) \in C_i$ and $False$ otherwise.

Function - Multicast an internal output action
$MultitcastMessage_\Lambda : Mems(\Lambda) \times Act(\cup_{i\leq n} C_i) \to Mems(\Lambda)$
Let $mem = (m_1, \ldots, m_n)$ be a tuple of memories in $Mems(\Lambda)$ where $m_i = (d_i, \mathcal{Q}_i)$ for all $i \leq n$ and act is the internal output to be multicasted in mem. $MultitcastMessage_\Lambda(mem, act)$ is the tuple of memories $(m'_1, \ldots, m'_n)$ where $m'_i = (d_i, enqueue(\mathcal{Q}_i, act))$ if $MightReceive_\Lambda(act, i)$ is $True$ and $m'_i = m_i$ otherwise.

Function - Timed trace extension with an event
$Extend_C : TTraces(C) \times Evt(C) \to TTraces(C)$
$Extend_C(\sigma, ev)$ is defined if and only if either σ is ε, or σ is of the form $\sigma'.ev'$ with $act(ev') \neq d$ and $delay(ev) \neq 0$. In both cases we have: $Extend_C(\sigma.ev) = \sigma.ev$.

Function - Mirror action
$mirror : Act(C) \to Act(\cup_{i\leq n} C_i)$
Let act be an action in $Act(C)$, $mirror(act)$ is defined as $\overline{act}$.

Function - Length of a timed trace
$length : TTraces(C) \to \mathbb{Z}_>$
Let σ be a timed trace in $TTraces(C)$ of the form $ev_1 \ldots ev_n$, $length(\sigma)$ is n (with the particular case $n = 0$ if $\sigma = \varepsilon$).

Function - Size of tuple of timed traces
$size : Tup(\Lambda) \to \mathbb{Z}_>$
Let $\mu = (\sigma_1, \ldots, \sigma_n)$ in $Tup(\Lambda)$ be a tuple of timed traces, $size(\mu)$ is defined as $\sum_{i\leq n} length(\sigma_i)$.

Function - Uninitialized tuple of timed traces
$uninitialized : Tup(\Lambda) \to Tup(\Lambda)$
Let $\mu = (\sigma_1, \ldots, \sigma_n)$ in $Tup(\Lambda)$ be a tuple of timed traces , $uninitialized(\mu)$ is the tuple of traces $(\sigma'_1, \ldots, \sigma'_n)$ where σ'_i is ε if σ_i is ε and is of the form $(\_, a).\sigma''_i$ if σ_i is of the form $(d, a).\sigma''_i$.

In the sequel, a signature $\Omega = (S, Op)$ and a model $M = \coprod_{s \in S} M_s$ are given. Randomness in generating a multitrace is illustrated in (randomly) generating: (1) a positive duration in D^+ and (2) a data value in M. Thus, we require the use of following technical functions:

- $randomDuration : D^+ \times D^+ \rightarrow D^+$ such that for two positive durations x and y in D^+ such that $x < y$ we have $randomDuration(x, y)$ returns a random positive duration in $[x, \ldots, y]$.
- $randomData_M : S \rightarrow M_s$ such that for a type $s \in S$ we have $randomData_M(s)$ returns a random data value in M_s
- We also need to choose a (random) element in a set of elements. Given a generic set of elements E, $randomElement(E)$ returns a random element $e \in E$.

Main Function

Function - Generating randomly an uninitialized multitrace
$GenerateMultitrace_\Lambda : \mathbb{Z}_> \times M \rightarrow MTraces(\Lambda)$
Let *size* be an integer which designates the size of the multitrace to be generated. $GenerateMultitrace_\Lambda(size, M)$ returns a multitrace μ in $MTraces(\Lambda)$ built by following rules of valid communication patterns.

Algorithm 2: Random generation of a multitrace

This algorithm generates randomly an uninitialized multitrace by following rules defining valid communications. We suppose that $LIMIT$ is maximum positive integer value used in computing.

Input: $\Lambda = (C_1, \ldots, C_n)$: a non-empty and valid distributed interface,
M: a set of data,
$size$: the size of the multitrace to be generated

Output: $\mu = (\sigma_1, \cdots, \sigma_n)$: a randomly generated uninitialized multitrace.

1 $GenerateMultitrace_{\Lambda}(size, M)$:
2 /*Initialization*/
3 $\mu \leftarrow (\sigma_1, \ldots, \sigma_n)$/*Where $\sigma_i = \varepsilon$*/
4 $mem \leftarrow ((d_1, \mathcal{Q}_1), \ldots, (d_n, \mathcal{Q}_n))$/*Where $d_i = 0$ and $\mathcal{Q}_i = [\,]$*/
5 /*Random generation*/
6 **while** $size(\mu) < size$ **do**
7 $i \leftarrow randomElement(\{1, \ldots, n\})$
8 $c \leftarrow randomElement(C_i)$
9 $d \leftarrow randomDuration(1, LIMIT)$
10 $mem \leftarrow ((d_1 + d, \mathcal{Q}_1), \ldots, (d_n + d, \mathcal{Q}_n))$/*Synchronous time elapsing*/
11 /*We have $C_i = C_{in}^{int} \coprod C_{out}^{int} \coprod C_{in}^{ext} \coprod C_{out}^{ext}$*/
12 **switch** c **do**
13 **case** *External_Output* /*$c \in C_{out}^{ext}$*/
14 $v \leftarrow randomData_M(type(c))$
15 $ev \leftarrow (d_i, c!v)$
16 $\mu \leftarrow (\sigma_1, \ldots, Extend_{C_i}(\sigma_i, ev), \ldots, \sigma_n)$
17 **case** *External_Input* /*$c \in C_{in}^{ext}$*/
18 $v \leftarrow randomData_M(type(c))$
19 $ev \leftarrow (d_i, c?v)$
20 $\mu \leftarrow (\sigma_1, \ldots, Extend_{C_i}(\sigma_i, ev), \ldots, \sigma_n)$
21 **case** *Internal_Output* /*$c \in C_{out}^{int}$*/
22 $v \leftarrow randomData_M(type(c))$
23 $mem \leftarrow MultitcastMessage_{\Lambda}(mem, c!v)$/*Multicast action */
24 $ev \leftarrow (d_i, c!v)$
25 $\mu \leftarrow (\sigma_1, \ldots, Extend_{C_i}(\sigma_i, ev), \ldots, \sigma_n)$
26 **case** *Internal_Input* /*$c \in C_{in}^{int}$*/
27 **if** $\neg isEmpty(\mathcal{Q}_i)$ **then**
28 $act \leftarrow dequeue(\mathcal{Q}_i))$
29 $ev \leftarrow (d_i, mirror(act))$
30 $\mu \leftarrow (\sigma_1, \ldots, Extend_{C_i}(\sigma_i, ev), \ldots, \sigma_n)$
31 /*Reset the accumulated duration for current component*/
32 $mem \leftarrow (m_1, \ldots, (0, \mathcal{Q}_i), \ldots, m_n)$/
33 **return** $uninitialized(\mu)$
34

Algorithm 2 describes the process of (randomly) generating a multitrace. It generates randomly an uninitialized multitrace by following rules of building a correct tuple of timed traces which respects valid communication rules. Randomness is illustrated in both choosing (randomly) a localized interface with its index i and choosing (randomly) a channel c in this latter selected localized interface (lines 7 and 8). We produce (randomly) a non-null positive duration in a range $[1, \ldots, LIMIT]$ (line 13) and a data value in the typed set $M_{type(c)}$.

Process: First, we initialize our function ingredients: we have an empty tuple of timed traces $(\varepsilon, \ldots, \varepsilon)$ and an empty tuple of memories $((0, [\,]), \ldots, (0, [\,]))$ (lines 3 and 4). As long as the size of the tuple of timed traces to be generated is less than our predetermined $size$ we select (randomly) a channel c. together with a non-null duration d (lines 7 to 9). We elapse time synchronously for all local memories (we update delays d_i in local contexts with the random produced delay d) (line 10). Following the nature of the selected channel c: we choose randomly a data value v in $M_{type(c)}$. If the latter channel is an external output (resp. input) we extend the trace of position i with event $(d_i, c!v)$ (resp. $(d_i, c?v)$) (lines 16 and 20). If the latter channel is an internal output we multicast the action $c!v$ (line 23) and we extend the trace i with event $(d_i, c!v)$. If the latter channel is an internal input we poll an action a form the back of the queue in context at position i (if the corresponding queue is not empty) and we extend the trace at position i with event $(d_i, \overline{a})$ (lines 27 to 30). In line 32, the elapsing duration associated to the selected component (i) is reset, while elapsing durations associated to the other components have been increased (line 10) and will be reset only when the corresponding component will be later selected. Finally, we return the uninitialized (without observed duration in initial events of each timed trace) correct tuple of timed traces (line 33).

5.1.2 Generating observable multitraces

An observable multitrace is defined as a prefix of a multitrace. A strategy to generate randomly an observable multitrace $\mu = (\sigma_1, \ldots, \sigma_n)$ which size is $size$ is first to generate randomly an uninitialized multitrace $\mu' = (\sigma'_1, \ldots, \sigma'_n)$ of size $ratio \times size$ where $ratio$ is a positive integer (that one may enter randomly), and then to select the prefixes of timed traces σ'_i (chosen randomly) until $size(\mu) = size$. Therefore, we get an observable multitrace μ which is the prefix of a random generated multitrace μ' and whose size corresponds to the targeted $size$. $size(\mu) = size(\mu')/ratio$.
Before introducing algorithm to generate observable multitraces, we introduce the following functions:

Function - Prefix (without the last event) of a timed trace
$prefix_C : TTraces(C) \to TTraces(C)$
Let σ be a timed trace in $TTraces(C)$ of the form $ev_1, \ldots, ev_n$, $prefix_C(\sigma)$ is ε if σ is ε and σ' if σ if of the form $\sigma'.ev$ (and $\sigma' \neq \varepsilon$).

Function - Generating randomly an observable multitrace
$GenerateObsMultitrace_\Lambda : \mathbb{Z}_> \times \mathbb{Z}_> \times M \to OTraces(\Lambda)$
Let $\Lambda = (C_1, \cdots, C_n)$ be a distributed interface, $size$ be a positive integer which designates the size of the multitrace to be generated and $ratio$ is a positive integer. Function $GenerateObsMultitrace_\Lambda(size, ratio, M)$ returns a tuple of traces μ in $OTraces(\Lambda)$ of size $size$.

Algorithm 3 describes the process of (randomly) generating an observable multitrace.

Algorithm 3: Random generation of an observable multitrace

This algorithm generates an observable multitrace randomly. It returns an observable multitrace by following the two next steps: Generate a multitrace, then get the prefix of the generated multitrace using the previously defined strategy

Input: $\Lambda = (C_1, \ldots, C_n)$: a distributed interface
M: a set of data
$size$: the size of the generated observable multitrace $ratio$: a user-defined prefix quotient.

Output: $\mu = (\sigma_1, \cdots, \sigma_n)$: a randomly generated observable multitrace.

```
1 GenerateObsMultitrace_Λ(size, ratio, M) :
2 μ ← GenerateMultitrace_Λ(size × ratio, M)
3 while size(μ) ≠ size do
4 |   i ← randomElement({1, ..., n})
5 |   μ ← (σ_1, ..., prefix_{C_i}(σ_i), ..., σ_n)
6 return μ
```

5.2 Generating CDOs with DIVERSITY

5.2.1 Global Trace Generation

In the context of DIVERSITY platform, a system is defined by a set of communicating xLIA models (equivalent to TIOSTS formalism of Definition 2.7) where communication is modeled y asynchronous data passing. In fact, for any output of a value on a given channel (i.e, written on the associated buffer), that value may be consumed later by another xLIA model considering this value as an input action on the same channel (i.e, read from the associated buffer).

DIVERSITY [4] implements symbolic execution processing (depicted in Figure 5.2) which can be customized by some on-fly using *filtering mechanisms*: steps $(i), \ldots, (v)$. The scheduling of these steps is cyclic. Each cycle consists in updating a queue of Execution Context (EC). At the initialization of the first iteration of the cycle, the queue contains EC_0 (equivalent to symbolic state $Init$ when using TIOSTS formalin-see Notation 2.7) which characterizes the initial symbolic values associated with the variables where the PC is restricted to $True$ because no constraint has yet been encountered. Each iteration step consists in: selecting one or more ECs (removed from the queue); computing their children ECs by symbolically executing all outgoing transitions from the control states reached in the parent ECs; deciding whether or not the parent ECs are added to the tree; in which case, their children ECs are added to the queue. The whole symbolic processing is based on the notation of $filtering$. The purpose of a filter is to dynamically accept or reject ECs according to a specific user coverage purpose. It can be seen as a selection strategy to complement the traversal strategy in order to increase the chances of reaching the targeted coverage while avoiding combinatorial explosion.

Steps of the symbolic processing. In the following, we present steps of symbolic processing depicted in Figure 5.2.

- Step (i) Selection of EC candidates for Step (ii): One or more EC are selected from the queue according to a customizable strategy such as RFS, BFS and DFS and HoJ.

Some heuristics may, however, associate a weight with each of the EC, and thus induce an order thereof in the queue which becomes a priority queue.

- Step (ii) Pre-filtering: It consists in applying one or more filters to reason on ECs before computing their children. If the EC successfully passes the control of each of the chained filters, it continues its way in the symbolic processing flow, through Step $(ii-a)$. Otherwise, Step $(ii-b)$, the EC will be ignored or possibly tagged and inserted into the symbolic tree under construction. In the favorable case where all user coverage objectives are met, the symbolic processing stops.
- Step (iii) Symbolic execution: Each EC issued from Step $(ii-a)$ is evaluated symbolically. During the evaluation its children $EC_1, \ldots, EC_n$ are computed by symbolically executing outgoing transitions.
- Step (iv) Post-filtering: It is similar to Step (ii), except that the filter involved in post-filtering reasons on the EC and its children to decide of the future of the symbolic processing. After passing the post-filters, there are two possibilities:
  1. Step $(iv-a)$ If successful, the symbolic processing continue with Step (v) in which case the EC is added to the symbolic tree.
  2. Step $(iv-b)$ If failed, the EC and, its children $EC_1, \ldots, EC_n$ are ignored or inserted in the symbolic tree.

  As in Step (ii), in the favorable case where all user coverage objectives are met, the symbolic processing stops.
- Step (v): All the children $EC_1, \ldots, EC_n$ resulting from Step $(iv-a)$ are enqueued and the symbolic processing iterates with Step (i).

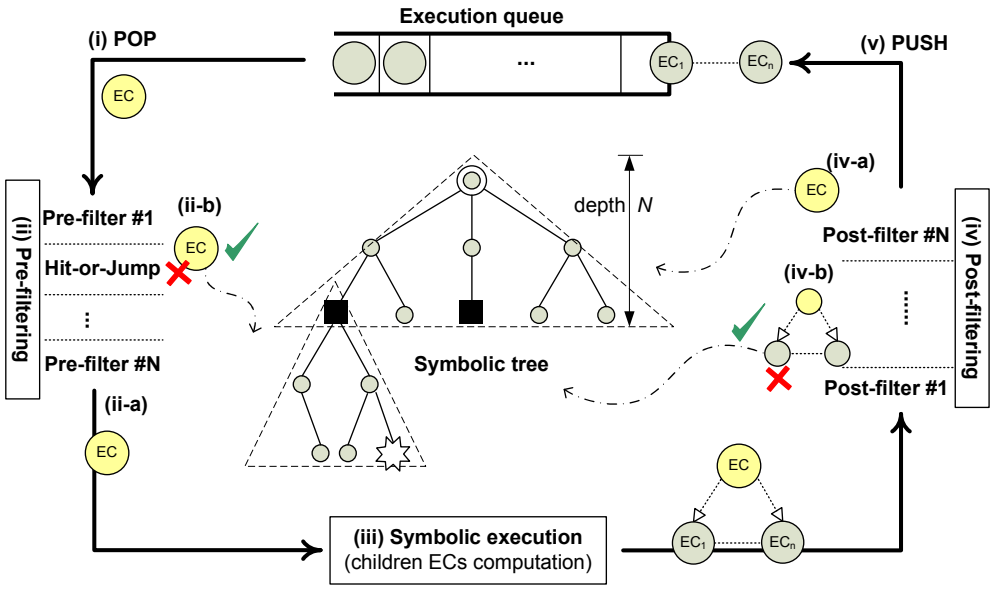


Figure 5.2: The running process of symbolic execution in DIVERSITY [27, 4].

Following the steps of the symbolic processing of a xLIA model describing a distributed specification in DIVERSITY, we may choose a path in the symbolic tree, and then, a timed trace that represents the global correct behavior of the distributed system.

5.2.2 From Global Timed Traces to CDOs by Projection

To generate a correct tuple of timed traces by construction, we first generate a global trace using SE techniques in DIVERSITY platform as presented in Section 5.2.1. For this, we use buffers to store internal messages to be exchanged between local components. Those buffers simulate internal communications using one timed queue per component. Since the reception of a message can be delayed, the composition of xLIA models specifies asynchronous communications. Indeed, as DIVERSITY may model timed systems, it manages time elapsing by generating a symbolic duration at each transition evaluation. DIVERSITY builds a global trace following a given coverage criterion, focusing on some targeted behaviors in order to construct global traces that are as representative as possible of normal use. Those global traces describe a correct behavior of the distributed system built using asynchronous communications via buffering and time elapsing techniques. We may build a CDO of a distributed system modeled by xLIA models by considering a tuple made of all projections of the global trace for each local component. For this, we use a projection technique to get local timed traces which describe correct behaviors of local components. The tuple of traces made of those local traces is by definition correct from a communication point of view. Indeed, any internal output of a value on a given channel can be consumed later by other components (that listen on the same shared channel, and thus represent potential receivers).

Let $Spec = (\mathbb{G}_1, \ldots, \mathbb{G}_n)$ be a distributed specification defined over an interface $\Lambda = (C_1, \ldots, C_n)$. Let us suppose that $Spec$ is written in the xLIA formalism and we suppose the existence of the equivalent xLIA model $Spec^X = (\mathbb{G}_1^X, \ldots, \mathbb{G}_n^X)$ where each equivalent xLIA local model $\mathbb{G}_i^X$ is defined over the same signature $\Sigma_i = (A_i, T_i, C_i)$. The xLIA model $Spec^X$ simulates a composition of local TIOSTSs $\mathbb{G}_i$ asynchronously via buffering internal messages that might be sent and received.

A global trace σ which describes a correct behavior of the distributed specification $Spec$ may be generated using SE techniques from the xLIA model $Spec^X$. We have timed trace σ is defined as an element of the set of timed traces $TTraces(\bigcup_{i=1}^{n} C_i)$.

Let C be a local interface. The projection of σ on C is denoted $\pi_C(\sigma)$. Algorithm 4 describes the process of projecting a global timed trace on a local interface.

Algorithm 4: Projection of a global timed trace on a local interface

This algorithm returns the projection of a global timed trace σ on a local interface C
Input: $\sigma = [ev_1, \ldots, ev_n]$: a global generated timed trace,
C: a local interface (set of channels),
Output: $\pi_C(\sigma)$: the projection of σ on C

```
1  π_C(σ) :
2  if σ = ε then
3      return ε
4  else
5      d ← 0
6      σ' ← ε
7      for ev ∈ [ev_1, ..., ev_n] do
8          d ← d + delay(ev)
9          a ← act(ev)
10         if chan(a) ∈ C then
11             ev' ← (d, a)
12             σ' ← Extend_C(σ', ev')
13             d ← 0
14 return σ'
```

Process: Algorithm 4 outputs the projection of a global timed trace σ (that represents a behavior of a distributed specification built over xLIA models) on a local interface C. The projection of an empty timed trace is an empty timed trace (line 2-3). For all events ev in σ, if the channel of $act(ev)$ belongs to the set of channels modeled by the local interface C (line 10) then, the algorithm builds an event $ev' = (d, a)$ where a is the action $act(ev)$ and d is the accumulated delay that when reading elements of σ (line 11). Then, the algorithm stores ev' in the timed trace to be produced as the projection of σ on C (line 12) which is denoted $\pi_C(\sigma)$. The accumulated delay d is reset to 0 at each step, an event from σ is stored in $\pi_C(\sigma)$ (line 13).

Now, given a distributed interface $\Lambda = (C_1, \ldots, C_n)$, a global timed trace σ in $TTraces(\bigcup_{i=1}^{n} C_i)$, we have that the tuple of traces $(\pi_{C_1}(\sigma), \ldots, \pi_{C_n}(\sigma))$ is in $OTraces(\Lambda)$.

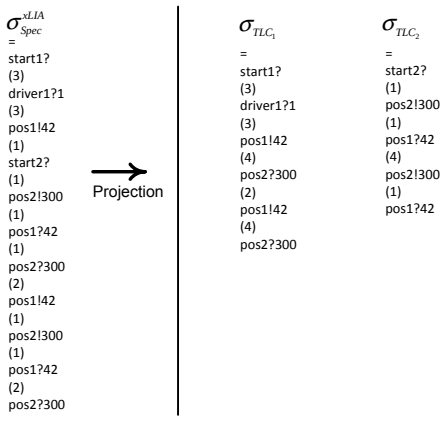
Example 5.1 (Generating a CDO and Checking valid communication)**.** *Consider distributed interface $\Lambda_{TCS} = (C_{TLC_1}, C_{TLC_2})$ from Example 4.2 and distributed specification $TCS = (\mathbb{G}_{TLC_1}, \mathbb{G}_{TLC_2})$ illustrated in Example 4.6. Let us suppose that DIVESITY produces global trace σ_{TCS}^{xLIA} depicted in Figure 5.3(a) using composing mechanism by queuing described in Section 5.2.1. σ_{TCS}^{xLIA} corresponds to a situation in which all receptions have been preceded by an emission. Now by applying projection mechanism described in Algorithm 4, we generate the tuple of timed traces $(\sigma_{TLC_1}, \sigma_{TLC_2})$ as illustrated in Figure 5.3.*

The tuple of traces $(\sigma_{TLC_1}, \sigma_{TLC_1})$ which is depicted in Figure 5.4(a) by the means of interaction diagram, defines by construction an observable multitrace (i.e correct tuple of traces that respects valid communication pattern). In fact, each local trace of this tuple σ_{TLC_1} and σ_{TLC_1} is the projection on its local interface C_{TLC_1} and C_{TLC_2} respectively of the global trace σ_{TCS}^{xLIA} where all receptions have been preceded by an emission. To check the correctness of this tuple, we suppose the existence of a virtual common instant at which

both local components TLC_1 and TLC_2 start their execution and hence the existence of symbolic durations d_1 and d_2 in V_{time} as depicted in Figure 5.4(b). In this tuple:

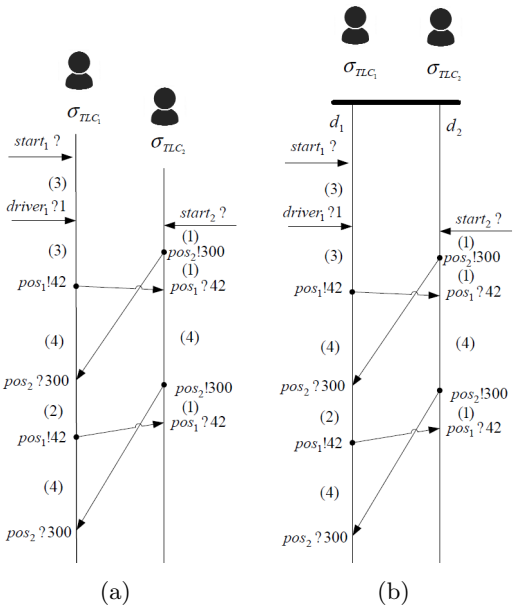
- 1^{st} *reception* $pos_1?42$ *is observed correctly in* σ_2 *provided that:* $d_2 + 2 > d_1 + 6 \ldots \varphi_1$
- 1^{st} *reception* $pos_2?300$ *is observed correctly in* σ_1 *provided that:* $d_1 + 10 > d_2 + 1 \ldots \varphi_2$
- 2^{nd} *reception* $pos_1?42$ *is observed correctly in* σ_2 *provided that:* $d_2 + 7 > d_1 + 12 \ldots \varphi_3$
- 2^{nd} *reception* $pos_2?300$ *is observed correctly in* σ_1 *provided that:* $d_{16} + 13 > d_2 + 6 \ldots \varphi_4$

We let $V_{time} = \{d_1, d_2\}$*, and we give the formula* $\varphi = \bigwedge_{i \in \{1,\ldots,4\}} \varphi_i$ *in* $\mathcal{F}_\Omega(V_{time})$*. We have* $IsSat(\varphi)$ *is* $True$*, indeed, there exists an interpretation* $\nu \in (D^+)^V$ *such that* $\nu(d_1) = 1/2$ *and* $\nu(d_2) = 6$ *and we have* $D^+ \models_\nu \varphi$*.* $Sat(\varphi)$ *may denote the set of solutions* $[d_1 \mapsto 1/2, d_2 \mapsto 6]$*. Hence,* $(\sigma_{TLC_1}, \sigma_{TLC_2}) \in OTraces(\Lambda)$*.*



(a) (b)

Figure 5.3: From a global trace to a correct distributed observation



(a) (b)

Figure 5.4: Communication checking of a CDO get by projection

5.3 A Mutation-based Approach to generate FDOs

Generated correct distributed observations may be modified by applying some simple mutation schemes. We apply mutations on either a correct tuple of traces generated by DIVERSITY tool or an observable multitrace generated randomly in order to modify either local traces (in the aim to get local traces which do not respect correct behavior of local components) or breaking the so-called RTC in the aim to inject a communication fault.

We describe the mutation schemas in Table 5.3. While Mutation schemas #2 and #4 do not require any conditions on system signature, mutation schemas #1 and #3 require that added or modified events respect syntactic requirements from the system signature and concerning channels and data types.

Mutation schema #5 is designed to break the key property of multitraces, that is that time is necessarily elapsing when messages are transmitted. Applying mutation schema #5 consists of breaking a detected RTC.

While the first four mutation schemas do not necessarily create faulty multitraces, mutation schema #5 creates by construction at least a communication fault. In the following, our framework to generate distributed observation with potential non-conformance faults will be presented as a couple of two frameworks: A framework to apply so-called *classical mutations* and a framework to apply so-called *RTC mutations*.

Mut. schema	Description
#1	Choose (randomly) a position in μ and insert an event ev
#2	Choose randomly an event ev in μ and delete it
#3	Choose randomly an event ev in μ and modify its data
#4	Choose randomly an event ev in μ and modify its duration
#5	Choose randomly a Round-Trip-Communication in μ and break it.

Table 5.3: Mutation schemas on multitraces

5.3.1 Classical mutations

Classical mutations are those listed in Table 5.3 from #1 to #4. Table 5.4 describes some functions useful to perform classical mutations:
The following functions perform respectively mutations on events, timed traces and a tuple of timed traces.

Function - Random mutation on an event
$MutateEvent_C : Evt(C) \times M \rightarrow Evt(C)$
Let ev be an event. $MutateEvent_C(ev)$ returns an event in $Evt(C)$ where either data or delay of ev is mutated.

Function - Random mutation on a timed trace
$MutateTrace_C : TTraces(C) \times M \rightarrow TTraces(C)$
Let σ be an a timed trace in $TTraces(C)$. $MutateTrace_C(\sigma, M)$ may either choose (randomly) an event ev in σ and performs a random mutation on ev by calling function

Function	Description
$changeDuration_C(ev, D^+)$	Returns a new event $(d, act(ev))$ where d is a random duration in D^+
$changeData_C(ev, M)$	Returns a new event $(delay(ev), a)$ where $a = c\triangle v'$ if $act(ev)$ is of the form $c\triangle v$ and v is a random data value in $M_{type(c)}$
$newEvent_C(M)$	Returns a new event $(d, c\triangle v)$ where d is a random duration in D^+, $c \in C$, $\triangle \in \{?, !\}$ and v is a random data value in $M_{type(c)}$
$removeEvent_C(\sigma)$	If σ is of the form $ev_1, \ldots, ev_n$ it returns a new timed trace in $TTraces(C)$ where event ev_i at position i chosen randomly in $[1, \ldots, n]$ is removed.
$insertEvent_C(\sigma)$	Extends σ with a new event $ev = newEvent_C(M)$ by calling $Extends_C(\sigma, ev)$.

Table 5.4: Technical functions used in implementing classical mutation on a tuple of traces

$MutateEvent_C(ev, M)$, or delete a random event by calling function $removeEvent_C(\sigma)$ or add a new event at the tail of σ by calling function $insertEvent_C(\sigma)$. It then returns the new mutant timed trace in $TTraces(C)$.

Function - Random mutation on a tuple of timed traces
$MutateTuple_\Lambda : Tup(\Lambda) \times M \to Tup(\Lambda)$
Let $\mu = (\sigma_1, \cdots, \sigma_n)$ be a tuple of timed traces in $Tup(\Lambda)$. $MutateMultitarce_\Lambda(\mu, M)$ chooses randomly a time trace σ_i in μ and performs a mutation on σ_i by calling function $MutateTrace_C(\sigma_i)$. It then returns the new mutant tuple of timed trace defined in $Tup(\Lambda)$.

In the sequel, we give only Algorithm 5 to describe the process of event mutation. Algorithms describing the process of timed trace mutation and tuple of traces mutation are trivial.

Algorithm 5: Random generation of an event mutant

This algorithm performs a random mutation on an event. It returns a mutant event after applying a random designated mutation.
Input: C: a non-empty set of typed channels,
ev: an event,
M: a model,
Output: A mutant event ev'.

```
1 MutateEvent_C(ev, M) :
2 Select_Mut ← chooseMutationIn(MUTATE_EVENT)
3 switch Select_Mut do
4     case Change_Duration
5         ev' ← changeDuration(ev, D+)
6     case Change_Data
7         ev' ← changeData(ev, M)
8 return ev'
```

Algorithm 5 performs (randomly) a classical mutation on a given event ev. Following the type of the mutation, the function will select randomly: we may either change the duration of event ev (line 5), or its data (line 7).

Example 5.2 (Classical mutation of a tuple of timed traces)**.** *Consider distributed interface* $\Lambda_{TCS} = (C_{TLC_1}, C_{TLC_2})$ *from Example 4.2 and distributed specification* $TCS =$

$(\mathbb{G}_{TLC_1}, \mathbb{G}_{TLC_2})$ *illustrated in Example 4.6. Consider tuple of timed traces* $\mu_{TCS} = (\sigma_{TLC_1}, \sigma_{TLC_1})$ *depicted in Figure 5.5(a). The tuple of traces* μ_{TCS} *denotes an observable multitrace as we demonstrated previously in Example 5.1.*

Function $MutateTuple_{\Lambda_{TCS}}(\mu_{TCS})$ *may return the new mutated tuple of traces* $\mu^{mut}_{TCS} = (\sigma_{TLC_1}, \sigma^{mut}_{TLC_1})$ *depicted in Figure 5.5(b). In this tuple, we notice that event* $(1, pos_2?42)$ *is deleted from timed trace* σ_{TLC_1}*, probably, by calling function* $removeEvent_{C_{TLC_2}}(\sigma_{TLC_2})$ *as described in Table 5.4.*

The new tuple of traces (after mutation) μ^{mut}_{TCS} *does not denote a correct timed trace of TIOSTS* $\mathbb{G}_{TLC_2}$ *(which is the local specification model of component* TLC_2 *of the TCS distributed system). To demonstrate this, we apply our rule-based algorithm for verdict computation to* TLC_2 *component from Example 2.5. Consider symbolic tree* $\mathcal{SE}(\mathbb{G}_{TLC_2})_\delta$ *depicted in Figure 2.5 produced from application of symbolic execution on TIOSTS* $\mathbb{G}_{TLC_2}$ *which specifies correct behavior of TLC system of Example 2.5. Let us assume timed trace* $\sigma^{mut}_{TLC_2}$ *denotes an execution of* TLC_2 *system:* $\sigma^{mut}_{TLC_2} = (\_, start?).(1, pos_2!300).(4, pos_1!300)(1, pos_1?42)$.

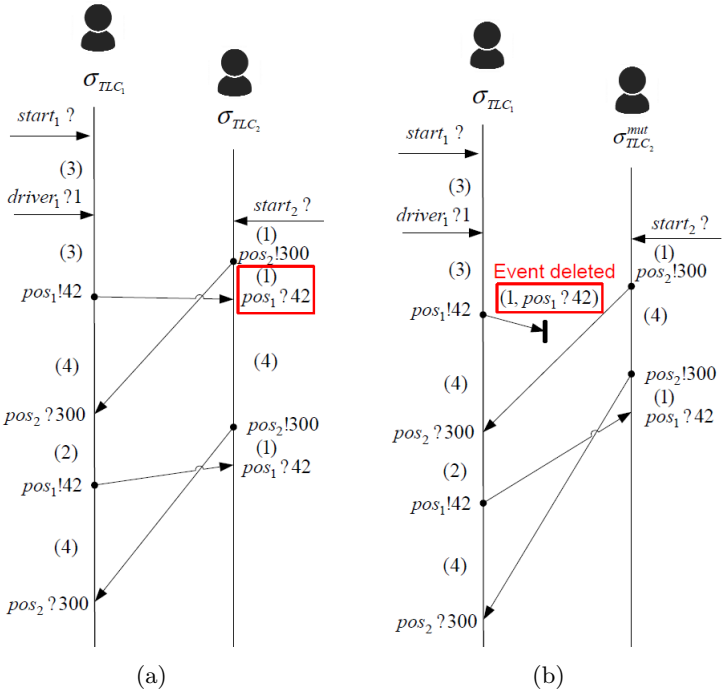


Figure 5.5: Application of a classical mutation on an observable multitrace

We proceed to verdict computation of $\sigma^{mut}_{TLC_2}$ *as follows:*

(a)

$(\_, start_2?)$	$(1, pos_2!300)$	$(4, pos_2!300)$	$(1, pos_1?42)$

ψ_t=$True$; ψ_d=$True$; SC=$\{(Init, \psi_t, \psi_d)\}(\tau)$; ***(Initialization)***

(b)

$(\_, start_2?)$	$(1, pos_2!300)$	$(4, pos_2!300)$	$(1, pos_1?42)$

$ev = (\_, start?)$; $delay(ev) = 0$
$\psi_t \leftarrow True$
$\pi_d(\eta_1) = True$
$\psi_d \leftarrow \psi_d \wedge \pi_d(\eta_1)$ *is satisfiable*
$Next(ev, SC) \longrightarrow SC$=$\{(\eta_1, \psi_t, \psi_d)\}(ev)$;***(Next Rule)***

(c)

$(\_, start_2?)$	$(1, pos_2!300)$	$(4, pos_2!300)$	$(1, pos_1?42)$

$ev = (1, pos_2!300)$
$\pi_t(\eta_4) = z_3 < 10$
$\psi_t \leftarrow \psi_t \wedge \pi_t(\eta_4) \wedge (z_3{=}1)$ *is satisfiable*
$\pi_d(\eta_4) = (p_2 \geq p'2) \vee (p'2 > 200)$
$\psi_d \leftarrow \psi_d \wedge \pi_d(\eta_4) \wedge (p_2{=}300)$ *is satisfiable*
$Next(ev, SC) \longrightarrow SC{=}\{(\eta_4, \psi_t, \psi_d)\}(ev)$; ***(Next Rule)***

(d)

$(\_, start_2?)$	$(1, pos_2!300)$	$(4, pos_2!300)$	$(1, pos_1?42)$

$ev = (4, pos_2!300)$
$SC{=}\emptyset$; $act(ev) \in O(C_{TLC_2})$; $delay(ev) \in D^+$
(Fail Rule)

5.3.2 Breaking a round-trip communication (RTC mutation)

RTC mutation is the mutation number #5 described in Table 5.3. Herein, we present an approach to mutate an observable multitrace (i.e, a CDO) in order to generate another tuple of timed traces with a communication fault (i.e, a FDO). We recall that there is no global clock which schedules distributed events observed on local interfaces, but only local clocks (with the hypothesis which states that time must elapse in the same way for all local interfaces).

To mutate a CDO in order to generate a FDO by implementing an RTC mutation, we first have to detect a RTC in it, then, we apply several classical mutations on delays of events in the RTC of interest in the aim to inject a communication fault in it. We first define the notion of *communication* between two local subsystems in a tuple of timed traces.

Communication. Let C_1 and C_2 be two local interfaces such that $\exists c \in C_1^{int} \cap C_2^{int}$. Let σ_1, σ_2 be two non empty timed traces in $TTraces(C_1)$ and $TTraces(C_2)$ respectively. A communication com is a couple (ev_s, ev_d) where $ev_s \in Evt(C_1)$ is an event of an internal output action and ev_d is an event in $Evt(C_2)$ where its action $act(ev_d) = \overline{act(ev_s)}$. The set of all communications defined over C_1 and C_2 is denoted $Coms(C_1, C_2)$.

Notation 5.2. *Let C_1 and C_2 be two local interfaces such that $\exists c \in C_1^{int} \cap C_2^{int}$. Given a communication $com = (ev_s, ev_d)$ in $Coms(C_1, C_2)$, $src(com)$ returns so-called source event ev_s of com, $dest(com)$ returns so-called destination event ev_d of com.*

In the sequel, we need to compute the time elapsed between two sequential events ev_i and ev_j that exist in the same timed trace. For this we introduce the function $time(ev_i, ev_j)$ defined as follows:

Time elapsed between two events. Let C be a local interface. Let σ be a non empty timed trace in $TTraces(C)$ of the form $ev_1 \dots ev_n$. Let i and j two positions in $[1, \dots, n]$ with $i \leq j$. Time elapsed between ev_i and ev_j is denoted as $time(ev_i, ev_j)$ in D^+ and defined as $\sum_{k=i+1}^{k=j} delay(ev_k)$.

Notation 5.3. *Let C be a local interface. Let σ be a non-empty timed trace in $TTraces(C)$ of the form $ev_1, \dots, ev_n$. We note $ev_i \prec ev_j$ if $i < j \leq n$.*

We define the notion of RTC as a couple of two communications that are connected by some conditions.

Round-trip Communication (RTC). Let $\Lambda = (C_1, \ldots, C_n)$ be a distributed interface where there exists an ordered subset of indexes $\{j_1, \ldots, j_m\} \subseteq \{1, \ldots, n\}$ with $m \leq n$ and where $\forall i < m \; \exists com_i \in Coms(C_{j_i}, C_{j_{i+1}})$ and for $i = m$, $\exists \; com_m \in Coms(C_{j_m}, C_{j_1})$. Let $(\sigma_1, \ldots, \sigma_n)$ be a tuple of timed traces defined in $Tup(\Lambda)$. An RTC rtc is a tuple $(com_1, \ldots, com_m)$ of communications where:

- $\forall i < m$, $com_i = (ev_s^i, ev_d^i)$ is a communication in $Coms(C_{j_i}, C_{j_{i+1}})$ where ev_s^i and ev_d^i are two events of σ_{j_i} and $\sigma_{j_{i+1}}$ respectively.
- $\forall i < m$, $ev_d^i \prec ev_s^{i+1}$.
- for $i = m$, $com_m = (ev_s^m, ev_d^m)$ is a communication in $Coms(C_{j_m}, C_{j_1})$ where ev_s^m and ev_d^m are two event of σ_{j_m} and σ_{j_1} respectively.
- for $i = m$, $ev_s^1 \prec ev_d^m$.
- ev_s^1, ev_d^m are two events of σ_{j_1} and ev_d^{m-1}, ev_s^m are two events of σ_{j_m}
- if $time(ev_s^1, ev_d^m) = d$ and $time(ev_d^{m-1}, ev_s^m) = d'$ then $d > d'$.

The set of all Round-trip communications defined over Λ is denoted $RTComs(\Lambda)$. The process of breaking a RTC is described as follows:

Process of breaking an RTC. When one detects an RTC $rtc = (com_1, \ldots, com_m)$ in a tuple of timed traces $(\sigma_1, \ldots, \sigma_n)$ with $m \leq n$ such that we have:

$time(src(com_1), dest(com_m)) = d$ and,

$time(dest(com_{m-1}), src(com_m)) = d'$ and,

$d > d'$.

We may mutate randomly delays of events $src(com_1)$,$dest(com_m)$,$dest(com_{m-1})$ and $src(com_m)$ in order to have:

$$time(src(com_1), dest(com_m)) < time(dest(com_{m-1}), src(com_m))$$

In this case, we guarantee that we have broken the RTC in question.

Example 5.3 (Breaking a RTC). *Consider distributed interface $\Lambda_{TCS} = (C_{TLC_1}, C_{TLC_2})$ from Example 4.2. The tuple of timed traces (σ_1, σ_1) depicted in Figure 5.6(a). (σ_1, σ_2) is an observable multitrace as we demonstrated in Example 5.1.*
In (σ_1, σ_2), we detect the RTC $rtc = (com_1, com_2)$ where:

- *$com_1 = (ev_1, ev_2) \in Coms(C_{TLC_1}, C_{TLC_2})$ where $ev_1 = (3, pos_1!42)$ is an event of σ_1 and $\underline{act}(ev_1) \in O(C_{TLC_1})$ and $ev_2 = (4, pos_1?42)$ is an event of σ_2 and $act(ev_2) = \overline{act(ev_1)}$.*
- *$com_2 = (ev_2', ev_1') \in Coms(C_{TLC_2}, C_{TLC_1})$ where $ev_2' = (3, pos_2!300)$ is an event of σ_2 and $\underline{act(ev_2')} \in O(C_{TLC_2})$ and $ev_1' = (2, pos_2?300)$ is an event of σ_1 and $act(ev_1') = \overline{act(ev_2')}$*
- *ev_1, ev_1' are events of σ_1 and ev_2, ev_2' are events of σ_2 such that:*

$$- \ time(ev_1, ev_1') = 4 + 2 + 4 = 10 \ and \ time(ev_2, ev_2') = 4 \ and \ 10 > 4$$

We apply a delay mutation on event ev_2' *such that* $ev_2' = (8, pos_2!300)$. *Then, we build (with the mutated event) a new tuple of timed traces* $(\sigma_1, \sigma_2^{mut})$ *which is defined in* $Tup(\Lambda)$ *where we have:*

$$time(ev_1, ev_1') = 10 \ and \ time(ev_2, ev_2') = 11 \ and \ 10 < 11$$

The new mutated tuple of traces $(\sigma_1, \sigma_2^{mut})$ *(depicted in Figure 5.6(b)) denotes an incorrect tuple of timed traces. In fact, to check the correctness of this tuple, we suppose the existence of a virtual common instant at which both local components* TLC_1 *and* TLC_2 *start their execution and hence the existence of symbolic durations* d_1 *and* d_2 *in* V_{time}. *In this tuple:*

- *First internal reception* $pos_1?42$ *in* σ_2 *may be observed correctly provided that:* $d_2+2 > d_1+6 \ldots \varphi_1$
- *First internal reception* $pos_2?300$ *in* σ_1 *may be observed correctly provided that:* $d_1 + 10 > d_2 + 1 \ldots \varphi_2$
- *Second internal reception* $pos_1?42$ *in* σ_2 *may be observed correctly provided that:* $d_2 + 14 > d_1 + 12 \ldots \varphi_3$
- *Second internal reception* $pos_2?300$ *in* σ_1 *may be observed correctly provided that:* $d_1 + 16 > d_2 + 13 \ldots \varphi_4$

We let $V_{time} = \{d_1, d_2\}$, *and we give the formula* $\varphi = \bigwedge_{i \in \{1,\ldots,4\}} \varphi_i$ *in* $\mathcal{F}_\Omega(V_{time})$. *We have* $IsSat(\varphi)$ *is* $False$ *indeed, there does not exist an interpretation* $\nu \in (D^+)^V$ *such that* $D^+ \models_\nu \varphi$. *We have* $Sat(\varphi) = \emptyset$. *Hence,* $(\sigma_1, \sigma_2^{mut}) \notin OTraces(\Lambda)$

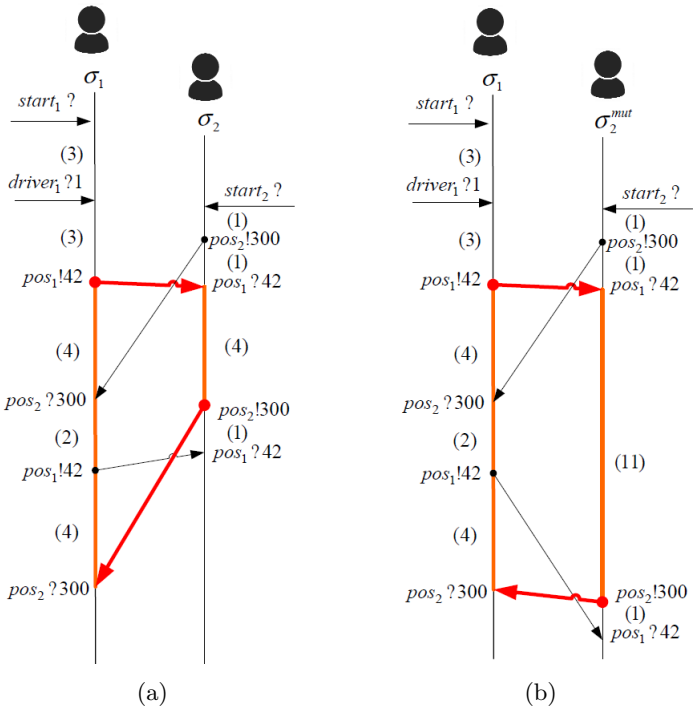


Figure 5.6: Breaking an RTC and generating a Faulty Distributed Observation (FDO)

5.4 The PhoneX Case Study

We illustrate our distributed testing approach for global verdict computation by applying our testing tool on correct distributed observations generated directly from PhoneX model or faulty ones generated by applying adapted mutation-based techniques. The main goals of this case study are listed as follows:

- Create a PhoneX distributed specification model in DIVERSITY modeling platform.
- Generate correct distributed observations[2] using the PhoneX model with DIVERSITY tool using composition and projection techniques.
- Simulate a faulty execution of the PhoneX system. This may be done by applying mutations on generated correct distributed observations with the aim to inject communication errors or to modify correct behaviors of local components of the system.
- Validate generated either correct distributed observations or faulty ones by applying our distributed testing toolchain and compute global test verdicts.

We use a distributed system of 10 components and generated correct distributed observations benchmarks containing from 100 to 10000 events. On those benchmark data, we apply adapted mutation-based techniques to generated 1000 mutated distributed observations from one correct distributed observation. We report on these experimentations, for this purpose, we compare the measured time of checking communication with the measured timed to perform local conformance checking and discuss results.

5.4.1 PhoneX System Overview

In this section, we present PhoneX as a case study developed first to promote Model-driven software development principles. The distributed nature of PhoneX makes it suitable for illustrating our testing approach.

PhoneX [77] is a central telecommunication system with communicating entities over the network. PhoneX (Figure 5.7) is a toy protocol created for demonstration purposes to promote Model-driven software development principles. It is signaling protocol to establish sessions that resembles well-known telecommunication protocols. PhoneX clients are devices that can register themselves to the PhoneX server which acts like a telecommunication switch. After checking registration, the caller client can call other clients (named called clients) via the server by providing the number of the called device.
The main goal with PhoneX is to create an end to end (i.e. from models to test execution) toolchain to apply our distributed testing approach.
PhoneX server provides very basic services. Examples of these services:

- Message routing: It keeps track of the registered clients and routes the messages based on the called numbers.
- Called client not available: In case a number is called which is not registered, then it informs the client which initiated the session, that the called number is not available.

[2] Observable multitraces where each local timed trace describes a correct behavior of its local component

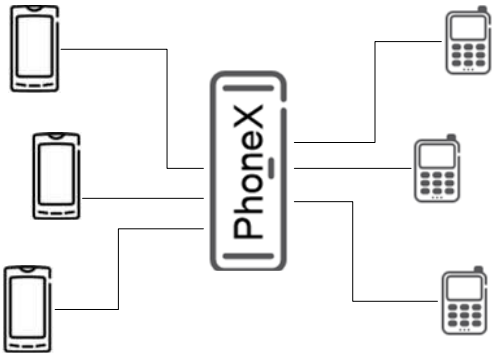


Figure 5.7: PhoneX clients and server

- Called client busy: In case the device with the called number is already online, then the client who initiated the session is informed, that the called device is busy.

In the following, we give basic scenarios of PhoneX system execution.

Successful call scenario. An example signaling flow for the successful session setup and call establishment between two clients can be seen in Figure 5.8. In this scenario, a caller with $Phone_{112}$ initiates a call ($doCall(113)$) to the user of $Phone_{113}$. The PhoneX server, after receiving $Calling(112, 113)$, checks if $Phone_{113}$ is registered, available, and then starts $StartSession(112, 113)$ for communication management and remains available. $Session_{112}^{113}$ informs $Phone_{113}$ that $Phone_{112}$ tried to get in contact ($CalledBy(112)$). The user of $Phone_{113}$ can accept the call ($doAcceptCall$) and informs $Session_{112}^{113}$ using $AcceptingCall$ which can establish communication (multicasting $InitCall$). Each user can end the call (the user of $Phone_{112}$ hangs up, $doEndCall$) and report it ($EndingCall$) to $Session_{112}^{113}$ that closes the connection by multicasting $TermCall$ and becomes available ($EndSession(112, 113)$) again.

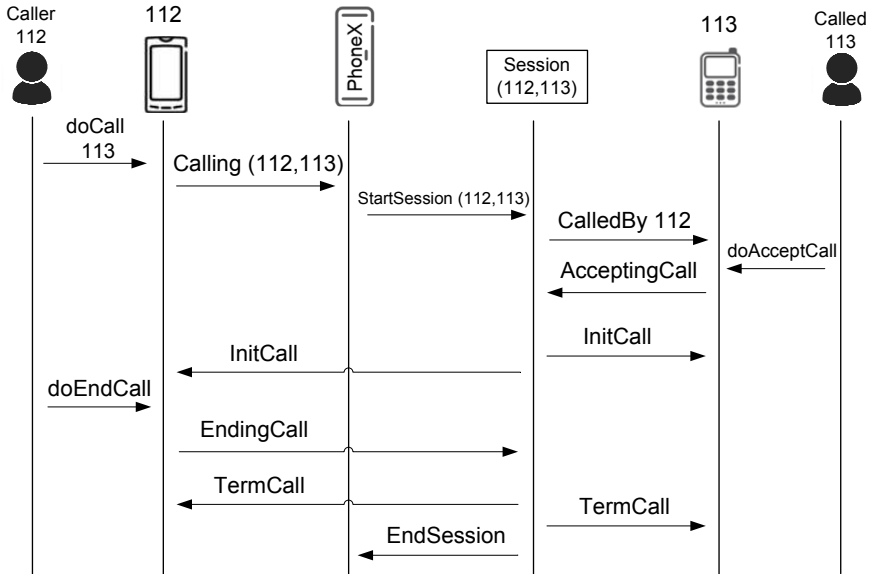


Figure 5.8: Interaction scenario of a successful call operation

Other scenarios of PhoneX system execution are given in Appendix A. Appendix A.1 describes a Line busy scenario. Appendix A.2 depicts a No Answer scenario.

5.4.2 PhoneX System Interface

Herein, we present PhoneX system interface as a set of channels (internal and external) through which local components of PhoneX system exchange messages. PhoneX system interface (as depicted in Figure 5.9) defines set of channels (internal and external) through which local components of PhoneX system exchange messages. Components Caller client and Called client define two roles that registered phones can have. PhoneX is the component that plays the role of the telecommunication central. An active session is a generic representation of sessions created by PhoneX central to manage communications caller and called clients. Communication channels model the media used by components.

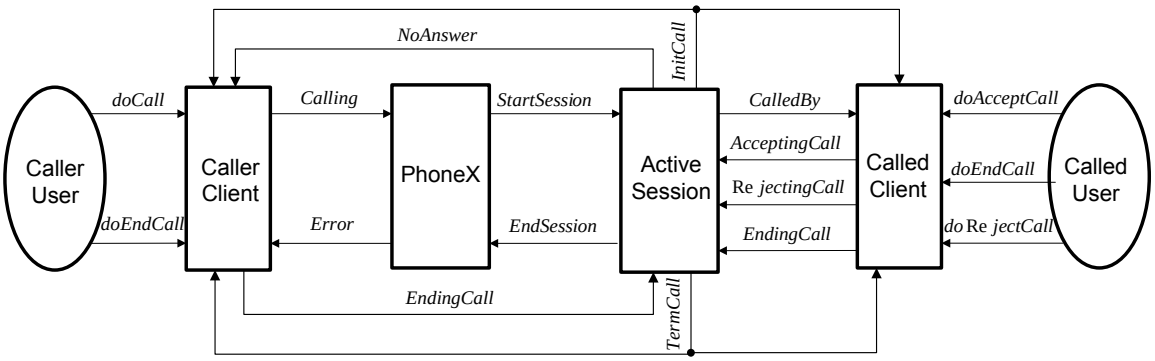


Figure 5.9: The PhoneX architecture

The description of all channels used to construct system interface of PhoneX case study is given in Appendix D.

5.4.3 PhoneX Modeling Effort

In our context, the modeling effort relates to the identification of the different modeling features used in TIOSTS from the requirement of PhoneX case study. We model PhoneX distributed system as a collection of TIOSTS communicating via a communication network. We model PhoneX distributed system as a collection of TIOSTS communicating via a communication network. Our system is comprised of four communicating components, a caller client, a Called client, the PhoneX central and an Active session when a call can be established between two different clients.

- Caller client specified with TIOSTS $\mathbb{G}_{src}$ (see Figure B.1(a) in Appendix B.1): describes the behavior of a client, which initiates a call. It sends call signals to be treated by PhoneX central. At any moment it may end the call it has emitted.
- Called client specified with TIOSTS $\mathbb{G}_{dest}$ (see Figure B.1(b) in Appendix B.2): describes the behavior of a client, which may receives a call. It receives call signals from the active session in order to either accept the call or reject it. At any moment it may end the call he/she has received.
- PhoneX specified with TIOSTS $\mathbb{G}_X$ (see Figure B.2 in Appendix B.3) is in charge of establishing a call between the two clients. If a no session between a caller and a called clients is already active then it starts and registers a new session between the two clients. PhoneX central keeps track of both registered clients in the Client database and active sessions in the Session database. If no session is active between

the caller and called clients, then PhoneX central starts a session provided that called number is registered in the database and is a valid one. It rejects calls with an unknown number, or when the called number is busy or when calling number is a not valid one.

- An active session specified with TIOSTS $\mathbb{G}_S$ (see Figure 5.10): is in charge of initiating (if it is already established) a terminating a call between two different[3] and registered clients in the Client database. Once a session is started between a caller and a called client, it initiates a call. PhoneX central gets notified that a session is finished by emitting a no answer signal when the called client does not answer within a giving timeout delay. It may also notify PhoneX central that the session is finished when the call is terminated between the caller and the caller clients.

There can be several instances of a TIOSTS model $\mathbb{G}_{src}$ (resp. $\mathbb{G}_{dest}$) during a test: each instance is identified with a *id* which corresponds to the caller (resp. called) number. For example, for a caller client 112 (resp. called client 113), we associate a TIOSTS model $\mathbb{G}_{src}^{112}$ (resp. $\mathbb{G}_{dest}^{113}$). There must be as many instances of a Session TIOSTS model as the number of instances of the called model, for a session between caller 112 and called client 113, we associate model $\mathbb{G}_S^{(112,113)}$. In the following, we describe only the active session model behavior. The rest of models behaviors are given in Appendix B.

Session behavior (depicted in Figure 5.10). When a new session is started, a Session TIOSTS is instantiated. At the *Idle* state, Session receives *src* and *dest* numbers, it then reaches *Starting*. It notifies called client *dest* with a call operation emitted by caller client *src* and reaches *Initiating*. At *Initiating*, it may reach either *Accepted* when called client accepts the call during a waiting delay or *Terminating* if a no-answer is observed during a waiting delay or the call get rejected. At *Accepted*, active session initiates and establishes a call between caller *src* and called *dest* and reaches state *Established*. At the latter state, either caller or called client may end the call (session reaches *Terminating* state). At *Terminating* state session sends a terminating signal to both caller and called clients and reaches *Ending*. Finally, it return to *Idle* by notifying PhoneX central of ending the active session $(src, dest)$.

[3] A client cannot call itself

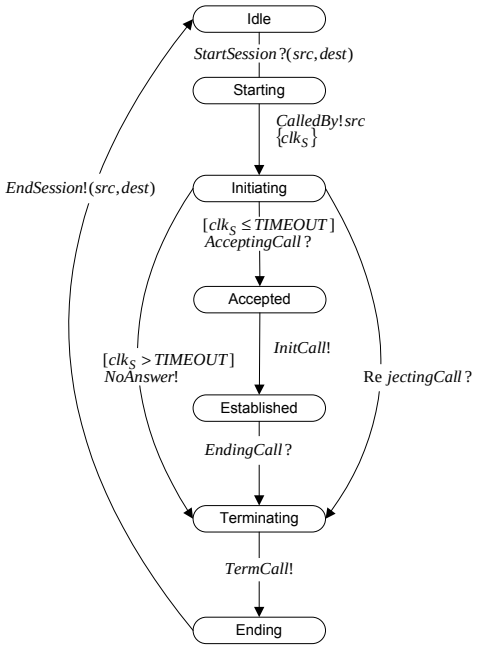


Figure 5.10: TIOSTS model $\mathbb{G}_S$ of the Active Session

TIOSTS $\mathbb{G}_{src}$, $\mathbb{G}_{dest}$ are described in Appendix B.1 and B.2 respectively, $\mathbb{G}_X$ is described in Appendix B.3.

5.4.4 Testing PhoneX

In this section, we present our PhoneX testing architecture. We apply our distributed testing approach by orchestration on PhoneX. Then, we draw experimental results of testing PhoneX system.

PhoneX Testing Architecture

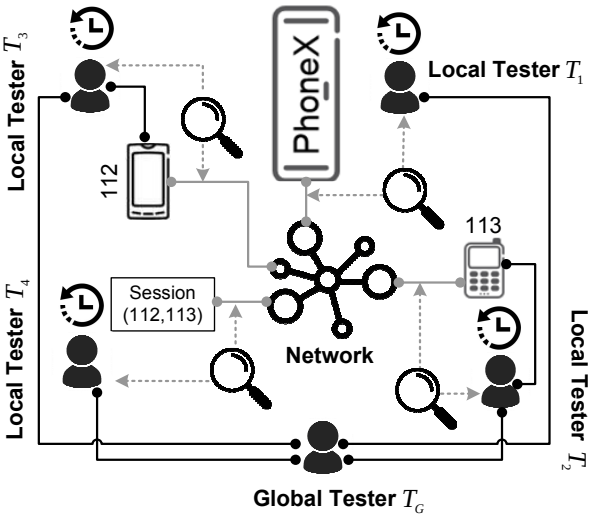


Figure 5.11: PhoneX distributed testing Architecture

Figure 5.11 illustrates the architecture used to carry our testing approach. We give an example of a client $Phone_{112}$ trying to call another client $Phone_{113}$ through PhoneX central which creates a session[4] $Session_{112}^{113}$. Only local components which are clients (here caller $Phone_{112}$ and called $Phone_{113}$) can communicate with environment. Hence each client has channels connected to the environment (in black connections) to receive signal from users and internal channels (in gray connections) to exchange values with other components (here $PhoneX$ central, active session $Session_{112}^{113}$, caller client $Phone_{112}$ and called client $Phone_{113}$). A tester T_i with $1 \leq i \leq 4$ is associated with each local component and T_i may control inputs (here from client users) and observe outputs occurring on channels connected to the environment. The tester may also observe values sent through internal channels (represented by the magnifying glasses). Each localized subsystem executes in a centralised way so that the local tester can observe the order of actions occurring on its channels and can measure durations between consecutive actions. Therefore, behaviors observed by each tester T_i can be viewed as timed traces and may be analyzed with respect to the set of timed traces of the model specifying the local component in question. We cannot directly combine the timed traces observed at different local interfaces since there is no global clock but only local clocks ordering local events. Internal communications, represented by a network are multicast: a message sent can be received by several recipients (all those who listen on the channel of interest). Messages are never lost but the time to reach a recipient is not quantifiable since it travels between interfaces and there is no global clock (we cannot measure it). Recall, however, that we assume that all testers use clocks progressing at the same rate. A global tester (also called a global checker) T_G links to local interfaces and collects local observations to build a tuple of timed traces, then, it performs a communication analysis to check whether or not the collected observed tuple of traces respects valid communication pattern.

Setting Up the Experiments

The size of the PhoneX system depends on the number of clients. In the following, we consider a system of 10 localised subsystems (depicted in Figure 5.12): 3 caller clients, 3 called clients, 3 active sessions and a PhoneX central.

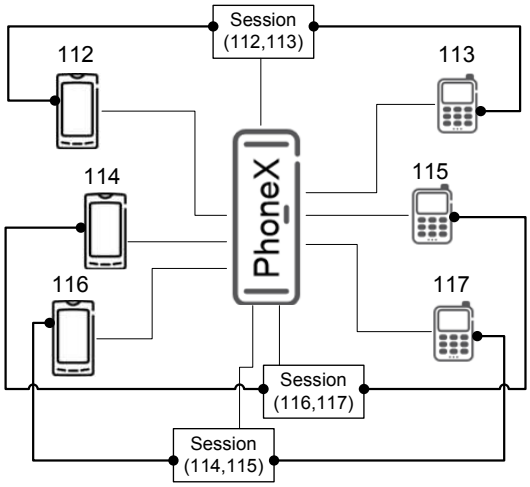


Figure 5.12: PhoneX distributed system composed of 10 components

[4]We may create as much active session as the number of called clients in the distributed architecture

Experimental Results

Correct multitraces generated by Diversity				1000 Mutated tuples of traces	
#events	#internal. com	com. checking	local. testing (for all traces)	#com. errors	average of com. checking
759	340	17ms	6s519ms	713	17.371ms
1587	700	28ms	21s761ms	729	27.648ms
3633	1589	49ms	1m34s178ms	800	40.934ms
6486	2830	59ms	7m5s797ms	737	60.140ms
7797	3400	69ms	10m52s378ms	722	66.315ms
9999	4357	88ms	24m14s860ms	738	80.825ms

Table 5.5: Experimental data for correct multitraces and their mutants

In Table 5.5, the third column (com. checking) gives the time[5] needed to solve the constraint associated to the verification of communications described in a multitrace whose number of events is given in the first column and number of internal communications is given in the second column. The fourth column provides the time[6] needed to analyse all local traces. For each multitrace, we generate 1000 mutated tuples of traces and we count the ratio of multitraces that are faulty with regards to communication policy (before the last column). Finally, in the last column, we give the average time to check the communication constraint of the mutated tuples. Experiments have been performed on a 3.10Ghz Intel Xeon E5-2687W working station with 64 GB of RAM on Linux Ubuntu 14.04.
We can easily observe in the table the exponential explosion in the time of local testing. Indeed, DIVERSITY uses exploration strategies that would have to reach a number of evaluation steps (used as a stop criterion), and that is costly. For information purposes, using BFS strategy to explore a trace of 6384 elements, we computed more than 4000 execution steps.

Communication checking time which is the measured time to check that tuple of timed traces respects or not valid communication pattern is the addition of two times: $time2build$ which is the time necessary to build the conjunction of constraints relating to the detection of all internal receptions in the tuple of traces. and $time2solve$ which is the measured time to solve the conjunction of constraints relating to the detection of all internal receptions in the tuple of traces using a constraint solver (in our case we have used the Yices2 constraint solver).

Time to build constraints (that we denote as $time2build$) is $O(N)$ with N the number of events in a tuple of traces. In fact, the translation of a multitrace into a CSP performs a classical loop over events occurring in the tuple of traces.

Time to solve constraints (that we denote as $time2solve$) is an experimental time whose complexity depends on the type of constraint solver to be used and the structure of constraint to be built. In our case, we have experimented $CVC4$[9], $Z3$[26] and $Yices2$[28] constraint solvers and we have observed that $Yices2$ seems to produce the best results in solving constraints of the form $d_i + x > d_j + y$ where $d_i + x$ is the time measured from

[5] using the Yices2 constraint solver[28]
[6] using the CVC4 constraint solver[9] embedded in DIVERSITY platform.

initialization to the observation of and internal input in component i and $d_j + y$ is time measured from initialization to the observation of the corresponding internal output in component j.

Chapter 6

Conclusions and Perspectives

Contents

6.1 Summary

Distributed Systems (DSs) consist of a number of independent components running concurrently on different machines that interact with each other through communication networks to meet a common goal. In this thesis, we showed that testing DSs is more difficult than testing centralized systems. Difficulties of testing DSs were highlighted in this thesis.

In the context of model-based testing of distributed systems, the oracle problem is the problem of checking of test results to detect differences between a DUT and its specification in order to decide conformance. This thesis presented a model-based distributed testing approach which focused on solving the oracle problem where the DUT consists of multiple localized subsystems under test communicating through a communication network. In this context, we specified, in Chapter 2, local entities composing the DUT using TIOSTS formalism.

In Chapter 3, we presented a model-based testing approach where we proposed an algorithm for solving the oracle problem in the context of local conformance testing based on *tioco* conformance relation.

Chapter 4 presents our contributions in the context of distributed MBT. Section 4.2 describes our distributed testing architecture and shows that non conformance can be detected locally (this is the case when the error belongs locally to a subsystem), yet, communication errors coming from the interaction between localized subsystems cannot be detected locally. In this regard, we assumed there is a separate tester at each localized interface which only obverses the interactions made at its interface. Since there is no global clock, we assumed that we cannot directly combine the timed traces observed at each local interface. We show that under the assumption that each local interface has a local clock, we can reconstruct a global view of the distributed system from local timed traces, using rules of valid communication pattern, and therefore global conformance can be reached

by using only local testers. Section 4.4.1 presented semantics of distributed systems that can be seen as *tuples of local timed traces*. To ensure the consistency of communications between localized SUTs, we introduced a set of properties that reflect correct interactions between components providing local timed traces. To capture this, we have introduced the notion of *multitrace* to denote a tuple of timed traces where elements that can be observed by each local tester respect valid communication pattern. Yet, as we cannot make any assumption on the different moments at which the different local testers stop observing their associated interfaces, we may accept as valid observations, tuples made of multi-trace prefixes that we denote as *observable multitraces*.

Conformance is decided by implementing *dtioco* conformance relation which verifies, first, the correctness of each stimulated localized SUT against its TIOSTS model, then, it checks the consistency of internal communications by testing whether properties that reflect correct interactions between local subsystems hold or not.

Our implementation framework for testing timed distributed systems consists in timed unitary testing for each stimulated localized SUT along with testing for internal communications. We developed in Section 4.5 of Chapter 4 our testing framework for solving the oracle problem. We have proposed an algorithm to check valid communication pattern by formulating the latter in terms of constraint solving problem. This latter algorithm was implemented in Java. Then we have presented an orchestration-based technique to implement the computation of global test verdicts.

In Chapter 5 we presented a validation approach to our implementation framework. We evaluated the scalability of our approach with regard to the soundness and performance of our algorithms. An experimentation of our testing approach on a real case study of a telecommunication distributed system was illustrated in this chapter.

6.2 Future Research

Although this thesis tackles several issues of model-based conformance testing for distributed systems, there are many challenges in this research domain that need further investigation.

Extensions to other communication protocols (e.g., smart grids protocols [67, 64, 40], TCP/IP [35]) may be one of the challenges to be considered as a motivation to enhance our communication checking approach in a distributed system.

In our work, we have assumed that each local tester starts observing when its associated localized sub-system is reset. Relaxing this assumption is left for future work.

This work does not give a complete framework to test timed distributed systems. First, there is a need to define and implement suitable test case generation algorithms. In particular, it is necessary to define distributed test purposes and to find test generation strategies to drive system execution so that they follow those test purposes. Second, in this work, we assume that local clocks progress at the same rate; it should be possible to generalize the results to the case where clocks can drift. Finally, we also intend to consider

the case where the sending and receiving of internal messages is hidden.

Another open issue that needs to be tackled is the problem of detecting deadlock in DSs. To the best of our knowledge, we could not find a work talking about using the MBT activities to detect deadlock in distributed systems. So having a technique that uses all the activities of MBT to detect deadlock in DSs is left as future work.

From the practical point of view, although the algorithm to check valid communication pattern presented in this thesis have been implemented and the prototype allows us to do some experiments (see Appendix C), we plan, as a future work, to incorporate this implementation in the DIVERSITY core in order to have a centralized tool used to implement a model-based testing approach (from test case generation to verdict computation) for distributed systems.

Appendix A

PhoneX other call scenarios

Contents

A.1 PhoneX Line busy scenario

An example that represents a call pattern with a line busy notification is depicted in terms of multiple life-lines in Figure A.1. It denotes the following behavior: Once a call is initiated between the caller client $Phone_{112}$ and the called client $Phone_{113}$, an active session $Session_{113}^{112}$ is registered in the Session database. Now, a calling signal is emitted by another caller $Phone_{111}$ to join called client $Phone_{1}13$ (Calling (111, 113)). In this case, there is already an active session where the called number is 113, hence, PhoneX notifies, caller $Phone_{111}$ with a message *LineBusy*. Indeed, the call cannot be settled between clients $Phone_{111}$ and $Phone_{113}$ as long as client $Phone_{113}$ is already taken by an active session.

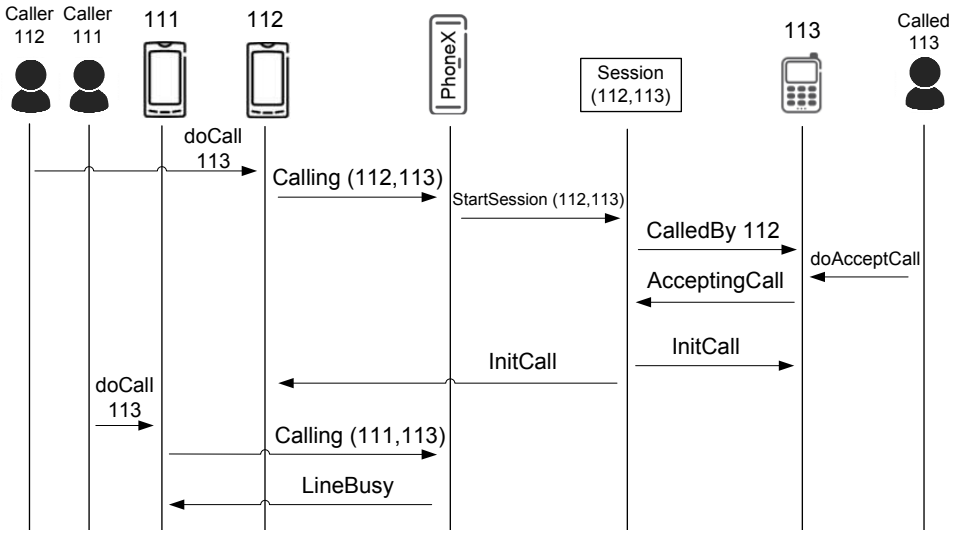


Figure A.1: Interaction scenario of a call scenario with Line Busy notification

A.2 No Answer scenario

An example that represents a call pattern with a no-answer notification is depicted in terms of multiple life-lines in Figure A.2. It denotes the following behavior: Caller $Phone_{112}$ initiates a call operation to join client $Phone_{113}$ (Calling(111, 113) is emitted). PhoneX central receives the Calling signal and after checking registration ($Phone_{113}$ is an allowed-to-call in the Client database and no active session is already opened with destination client $Phone_{113}$), it starts a new session $Session_{113}^{112}$. The latter session sends a $CalledBy$ signal to called client $Phone_{113}$ and waits for a response within a TIMEOUT duration.

In our scenario, active session $Session_{113}^{112}$ notifies caller client $Phone_{112}$ with a $NoAnswer$ signal, indeed, it has wait for a delay which is strictly greater than TIMEOUT which corresponds to a situation in which called client $Phone_{113}$ does not accept the call operation. The active session terminates the call process by multicasting signal $TermCall$ towards both caller and called clients $Phone_{112}$ and $Phone_{113}$ respectively. The current active session is finished when client identifiers 112 and 113 are emitted from active session $Session_{113}^{112}$ towards PhoneX central through channel $EndSession$.

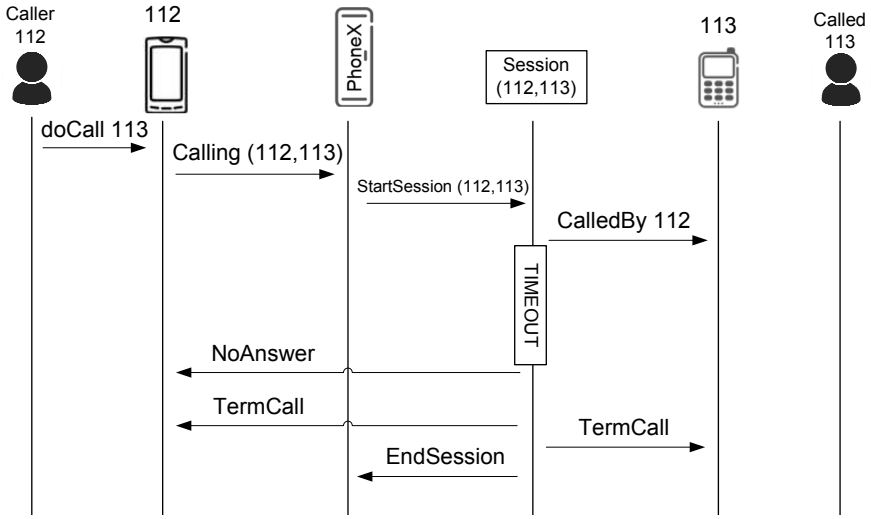


Figure A.2: Interaction scenario of a call scenario with No Answer notification

Appendix B

PhoneX TIOSTS models

Contents

B.1 Caller client TIOSTS model

Caller client behavior (depicted in Figure B.1(a)). At the *Idle* state, caller *src* receives a call from the environment (a caller) to make a call operation with called *dest* (caller reaches *Starting* state), then, it joins PhoneX central by sending to it *src* and *dest* (caller reaches *Initiating* state). Caller returns to *Idle* state when it receives an error code from PhoneX (PhoneX cannot establish a call due to violated condition of call establishment) or a signal to terminate the call from the active session (due to a call rejection by called client).

At *Initiating*, *src* may reach *Established* if a call is established by active session or state *Terminating* if a no-answer (from called client) is observed during a waiting delay. When a call is established (at *Established*), *src* may return to *Idle* by receiving a terminating signal from the active session (due to an ending call by called client) or receive a signal from the environment (a caller) to end the call in progress (caller reaches *UserEndingCall*). At *UserEndingCall*, the caller may also return to *Idle* by due to an ending call by the called client, it then, receives a terminating signal from the active session. From *UserEndingCall*, the caller notifies the active session for terminating the call and reaches *Terminating* state. At *Terminating*, caller *src* returns to *Idle* by receiving a terminating signal from the active session. Now, the caller is ready to make another call.

B.2 Called client TIOSTS model

Called client behavior (depicted in Figure B.1(b)). This role is symmetric (on the called client side) to the one described in Appendix B.1). At the *Idle* state, called client *dest* receives a called-by signal from the active session (caller reaches *Initiating* state). At this state, it may return to *Idle* when it receives a terminating signal from the active session (if a no-answer is observed during a waiting delay) or reach *Rejecting* if it receives

a rejection signal. At *Rejecting*, *dest* notifies the active session (of rejecting the call) and reaches *Terminating*.

From *Initiating*, *dest* reaches *Accepting* when called user accepts the call, it then, notifies the active session and reaches *Accepted*. *dest* may reach *Established* if a call is established. At *Established*, *dest* may return to *Idle* by receiving a terminating signal from the active session (due to an ending call by caller client). As described for caller client, *dest* may end the call and reach *UserEndingCall* and then *Terminating*. At *Terminating*, *dest* returns to *Idle* by receiving a terminating signal. Now, called client *dest* is free to accept another call.

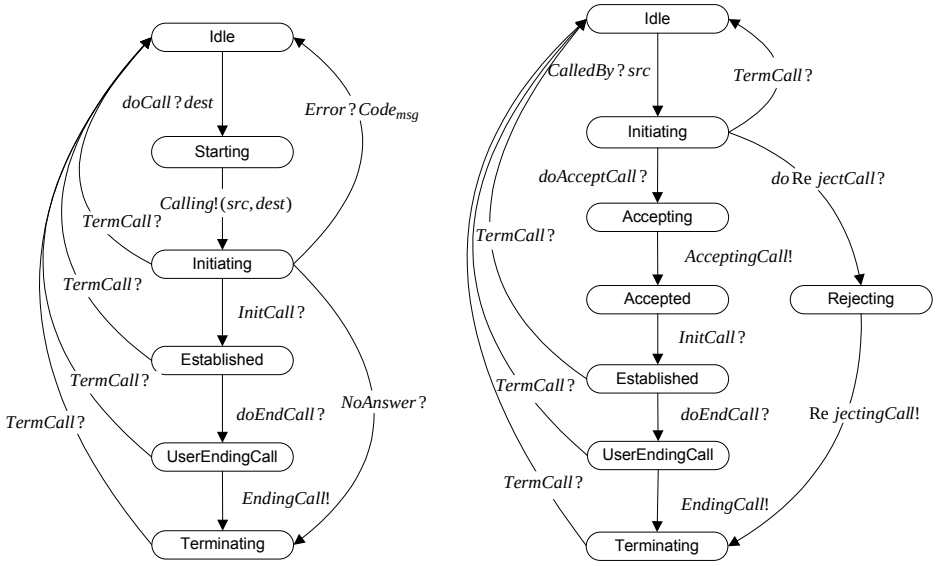


Figure B.1: TIOSTSs $\mathbb{G}_{src}$ and $\mathbb{G}_{dest}$ of Caller and Called clients

B.3 PhoneX central TIOSTS model

PhoneX central behavior (depicted in Figure B.2). At the *Idle* state, PhoneX may receive a new call with *src* and *dest* numbers and reach *Calling* or get notified of the ending of an already active session and return to *Idle*. At *Calling*, PhoneX may start a new session $(src, dest)$ and return to *Idle* provided that *dest* is a registered and allowed-to-call number in the Client database and there is no active session with called client *dest*. Otherwise, PhoneX may also return to *Idle* when *dest* is not registered in Client database (it notifies caller with code error *UnknownNumber*) or calling *dest* is not allowed (it notifies caller with code error *NotAllowed*) or called client *dest* is busy (it notifies caller with code error *LineBusy*).

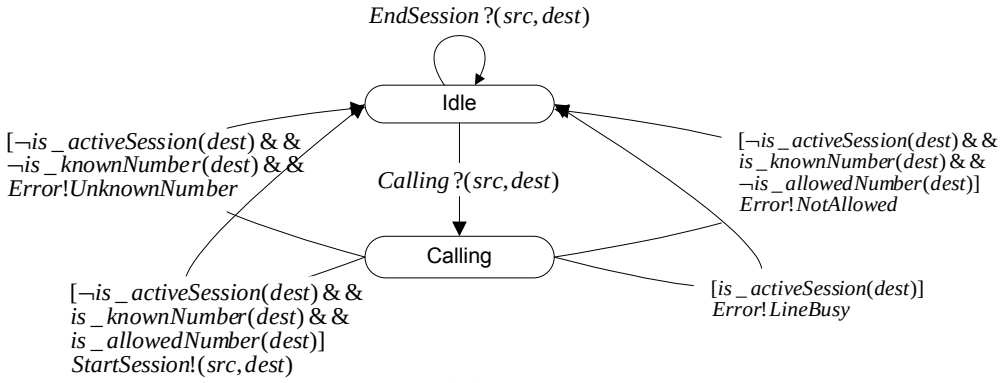


Figure B.2: TIOSTSs $\mathbb{G}_X$ of PhoneX Central

Appendix C

Algorithms Java Implementation

Contents

C.1 Java Implementation of function BuildConstraint

This appendix depicts Java source-code of function BuildConstraint which constructs a constraint on the detection of an event whose action is an internal input at some designated place in a distributed observation. This function takes as input a distributed observation, a distributed interface, an event and its date (from initialization) to validate its observation at a designated place and an array containing durations of all traces in the distributed observation. This function is supposed to return a constraint if there exists a subsystem that might send its corresponding internal output in a non-empty distributed observation. Otherwise, it returns an empty constraint.

Let us point out that there are two reasons for allowing an event whose action is an internal input to be accepted as a valid reception in a distributed observation. The first reason is that a sufficient number of corresponding internal outputs have been emitted prior to the consumption of this internal input. The second reason is that, at the time when the observation is performed, the trace emitting the corresponding internal output is no longer observed.

Listing C.1: Function to build a constraint on the detection of an internal input in a distributed observation

```
0 /*
1  *The function takes as input: a distributed observation: DO which denotes the tuple of traces which
       is supposed to contain an event to be correctly observed in a designated position in DO named
       trace_index, a distributed interface DI to provide DO with both internal and external channel
       sets, an event EV and its date (measured from initialization), the place and finally an array
       containing the durations of all traces in DO.
2  *The function builds a constraint on the observation of EV if its action is an internal input in DO.
       It returns an empty string of characters otherwise.
3  */
4 public static String BuildConstraint(DObservation DO, Event EV, int evDate, int trace_index,
      DInterface DI, Vector<Integer> traces_durations){
5     StringBuilder constraint=new StringBuilder();/*Build an empty string which is supposed to
          contain a constraint of the form Di+x>Dj+y */
```

```
6
7       if(DO.isEmpty())/*If DO is empty return the empty string as a constraint*/
8           return constraint.toString();
9
10      Trace TR= DO.get(trace_index);/*Get TR which is the trace at position trace_index*/
11      HashMap<String, Parameter> output_buffer= TR.getOutputBuffer();/*Retrieve the mapping table of
            TR*/
12
13      /*EV is an event whose action is an internal input at TR*/
14      if(EV.getAction().isInputInt(DI.getSystemSet().get(trace_index))){
15          int reception_index= -1;
16          if (output_buffer.get(EV.getAction().getOpposite().toString())!=null){
17              Queue<Integer> reception_buffer= output_buffer.get(EV.getAction().getOpposite().toString
                    ()).getReceive_buffer();
18              reception_index= output_buffer.get(EV.getAction().getOpposite().toString()).getId();
19
20              /*EV is an event whose action is an internal input at TR whose its corresponding output
                    is observed at trace whose position is reception_index*/
21              if (!reception_buffer.isEmpty()){
22                  if( ENABLED_DEBUG )
23                      System.out.println(EV+": is input internal && its corresponding output observed in
                            : "+reception_index);
24
25                  constraint
26                  .append("(> ").append("(+ D").append(trace_index).append(" ").append(evDate).append(")
                        ")
27                  .append("(+ D").append(reception_index).append(" ").append(reception_buffer.poll()).
                        append("))");
28
29                  if( ENABLED_DEBUG )
30                      System.out.println("==> Constraint to be checked for "+EV+" is: "+constraint);
31              }
32              /*EV is an event whose action is an internal input at TR whose its corresponding output
                    is not observed at trace whose position is reception_index*/
33              else{
34                  if( ENABLED_DEBUG )
35                      System.out.println(EV+": is input internal && corresponding output not observed in
                            : "+reception_index);
36
37                  /*Build the corresponding constraint*/
38                  constraint
39                  .append("(> ").append("(+ D").append(trace_index).append(" ").append(evDate).append(")
                        ")
40                  .append("(+ D").append(reception_index).append(" ").append(durations.get(
                        reception_index)).append("))");
41
42                  if( ENABLED_DEBUG )
43                      System.out.println("==> Constraint to be checked for "+EV+" is: "+constraint);
44              }
45          }
46
47          /*EV is an event whose action is an internal input at TR where its corresponding output is
                not observed at trace whose position is reception_index*/
48          else{
49              if( ENABLED_DEBUG )
50                  System.out.println(EV+": is input internal && corresponding output not observed in: "+
                        reception_index);
51
52              reception_index= Check.indexSender(EV.getAction(), DI);
53
54              /*Build the corresponding constraint*/
55              constraint
56              .append("(> ").append("(+ D").append(trace_index).append(" ").append(evDate).append(") ")
57              .append("(+ D").append(reception_index).append(" ").append(durations.get(reception_index)
                    ).append("))");
58
59              if( ENABLED_DEBUG )
60                  System.out.println("==> Constraint to be checked for "+EV+" is: "+constraint);
```

```
61          }
62      }
63      /*EV is an event whose action is not an internal input at TR*/
64      else{
65          if( ENABLED_DEBUG )
66              System.out.println(EV+" is not an internal input in sub-system: "+trace_index);
67      }
68      return constraint.toString();
69  }
```

C.2 Java Implementation of Main function DObervation2CSP

This appendix depicts Java source-code of function DObervation2CSP to check observable multitrace property as described in Section 4.5.2. The function takes as input a distributed observation, a distributed interface and a File object. It analyses elements of the distributed observation iteratively and produces a file written in SMT-Lib Format containing constraints related to the detection of each internal communication. In addition, the function is supposed to return the number of built constraints used as a metric in our evaluation.

Listing C.2: Function to translate the communication checking problem into a CSP

```
0   /*
1    *The function takes as input: a distributed observation: DO which denotes the tuple of traces to be
         analyzed, a distributed interface DI to provide DO with both internal and external channel
        sets and finally a File object file to be filled with constraints written in SMT-Lib.
2    *The function builds a file written in SMT-Lib format containing constraints related to the
        detection of each internal communication in DO.
3    *It also deliver the number of built constraints that we use as a metric.
4    */
5   public static int DObervation2CSP(DObservation DO, DInterface DI, File file) throws IOException{
6       int nbConstraints=0;/*We initialize the number of built constraints*/
7       int[] evDates = new int[DO.getComponents().size()];/*We use an array to store the measured
            duration of each analysed event*/
8       Arrays.fill(evDates, 0);
9       Vector<Integer> vect_durations= DO.getTracesDuration();/*This array contains the duration of
            each trace in DO*/
10
11      /*Initial filling of the SMT-file with positive symbolic durations*/
12      fw.write("(set-option :produce-models true)\n");
13      fw.write("(set-logic QF_LRA)\n\n");
14
15      for(int i=0;i<DO.nbTraces();i++){
16          fw.write("(declare-fun D");
17          fw.write(String.valueOf(i));
18          fw.write(" () Real)\n");
19          fw.write("(assert (> D");
20          fw.write(String.valueOf(i));
21          fw.write(" 0))\n");
22          nbConstraints++;
23      }
24      fw.write("\n");
25
26      /*Analysing elements of DO one by one*/
27      for(int trace_idx=0; trace_idx<DO.nbTraces(); trace_idx++){
28          Trace TR= DO.get(trace_idx);
29          for(int j=0; j<TR.size(); j++){
30              Event EV= (Event) TR.getElementList().get(j);/*Retrieve the head of each trace in DO*/
31              evDates[trace_idx]=evDates[trace_idx]+ EV.getDuration();/*Retrieve the delay measured
                    form initialization of the actual event*/
32              String constraint= BuildConstraint(DO, EV, evDates[trace_idx],trace_idx, DI,
                    vect_durations);/*Calling the BuildConstraint function to build a constraint when an
                     internal event is detected*/
33              /*Write the built expression in file if it is not empty. An empty expression is synonym
                    to the detection to no internal event*/
```

```
34              if (!constraint.isEmpty()){
35                  fw.write("(assert ");
36                  fw.write(constraint);
37                  fw.write(")\n");
38                  nbConstraints++;
39              }
40          }
41      }
42      /*Prepare the SMT-file by writing in it the SMT-Lib commands for the use of an SMT-Solver*/
43      fw.write("\n");
44      fw.write("(check-sat)\n");
45      fw.write("(get-model)\n");
46      fw.write("(exit)\n");
47
48      return nbConstraints; /*Return the total number of constraints built*/
49  }
```

Appendix D

PhoneX Distributed Interface

PhoneX system interface (as depicted in Figure 5.9) defines the set of channels (internal and external) through which local subsystems (of PhoneX system) exchange messages. The description of all channels used to construct system interface of PhoneX case study is given in the following table:

Channel	Type	Description
doCall	External	Used to send *dest* as a phone number to be called, from a caller user to a caller client in order to start a call conversion
doAcceptCall	External	Used to send a *doAcceptCall* signal from a called user to a called client to accept a call initiated by a caller client
doRejectCall	External	Used to send a *doRejectCall* signal from a called user to a called client to reject a call initiated by a caller client
doEndCall	External	Used to send a *doEndCall* signal from a caller (resp. called) a user to a caller (resp. called) client in order to end-up an already existing a call conversion
Calling	Internal	Used to send both caller and caller ids (*src* and *dest*) from a caller client to the PhoneX central
Error	Internal	If a call conversion cannot be established between clients with numbers *src* and *dest*, the PhoneX central sends an error code ($UnknownNumber$,$NotAllowed$,$LineBusy$) in destination of the caller client *src* with the aim to end-up the process of initiating a call
StartSession	Internal	After creating a session between clients with numbers *src* and *dest*, the latter numbers are sent by the PhoneX central to the created session through this channel
CalledBy	Internal	*src* id of the caller client is sent from the active session to the called client *dest* through this channel
NoAnswer	Internal	Due to a non responding timeout (if caller client *dest* is already in conversion within another active session), the created session sends a *NoAnswer* signal in destination of the caller client *src* through this channel
AcceptingCall	Internal	After receiving an *AcceptCall* from a caller user, the called client *dest* sends an *AcceptingCall* signal to the the active session through this channel
InitCall	Internal	After accepting a call by a called client *dest*, the active session initiates a call conversion between the caller client *src* and the called client *dest* by multicasting *InitCall* signal in destination of *src* and *dest* clients
RejectingCall	Internal	After receiving an *RejectCall* from the caller user, the called client *dest* sends an *RejectingCall* signal to the the active session
EndingCall	Internal	After receiving an *EndCall* from caller (resp. called) user, caller (resp. called) client may send an *EndingCall* signal to the the active session
TermCall	Internal	After ending-up the call by either the caller client *src* or the called client *dest*, the active session labeled with $(src, dest)$ terminates the call conversion between the caller client *src* and the called client *dest* by multicasting *TermCall* signal in destination of *src* and *dest* clients
EndSession	Internal	Once a call conversion is terminated between the caller *src* and the called client *dest*, the active session sends those numbers through *EndSession* channel in destination of the PhoneX central in order to terminate this active session

Table D.1: A summary of the channel names of the PhoneX system interface.

Bibliography

[1] Rajeev Alur, Aditya Kanade, S Ramesh, and KC Shashidhar. Symbolic analysis for improving simulation coverage of simulink/stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 89–98. ACM, 2008.

[2] Paul E Ammann, Paul E Black, and William Majurski. Using model checking to generate tests from specifications. In *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*, pages 46–54. IEEE, 1998.

[3] Susan M. Armstrong, Alan O. Freier, and Keith A. Marzullo. Multicast transport protocol. *RFC*, 1301:1–38, 1992.

[4] Mathilde Arnaud, Boutheina Bannour, and Arnault Lapitre. An illustrative use case of the diversity platform based on uml interaction scenarios. *Electronic Notes in Theoretical Computer Science*, 320:21–34, 2016.

[5] Mathilde Arnaud, Boutheina Bannour, and Arnault Lapitre. An illustrative use case of the DIVERSITY platform based on UML interaction scenarios. *Electr. Notes Theor. Comput. Sci.*, 320:21–34, 2016.

[6] B. Bannour, C. Gaston, and D. Servat. Eliciting unitary constraints from timed sequence diagram with symbolic techniques: application to testing. In *18th APSEC*. IEEE, 2011.

[7] Boutheina Bannour, Jose Pablo Escobedo, Christophe Gaston, and Pascale Le Gall. Off-line test case generation for timed symbolic model-based conformance testing. In *ICTSS*, pages 119–135. Springer, 2012.

[8] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[9] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, page 14, 2010.

[11] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[12] Nassim Benharrat, Christophe Gaston, Robert M Hierons, Arnault Lapitre, and Pascale Le Gall. Constraint-based oracles for timed distributed systems. In *IFIP International Conference on Testing Software and Systems*, pages 276–292. Springer, 2017.

[13] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[14] Laura Brandan Briones. *Theories for model-based testing: Real-time and coverage*. University of Twente, 2007.

[15] Alan Bundy and Lincoln Wallen. Breadth-first search. In *Catalogue of Artificial Intelligence Tools*, pages 13–13. Springer, 1984.

[16] Tsong Yueh Chen, TH Tse, and Zhiquan Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. *ACM SIGSOFT Software Engineering Notes*, 27(4):191–195, 2002.

[17] Young Choi, Hee Youn, Soonuk Seol, and Sang Yoo. Distributed test using logical clock. In *Formal Techniques for Networked and Distributed Systems*, pages 69–84. Springer, 2002.

[18] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, May 1976.

[19] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. *SIGSOFT Softw. Eng. Notes*, 26(5):142–151, September 2001.

[20] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. pearson education, 2005.

[21] Jon Crowcroft and Karen Paliwoda. A multicast transport protocol. In *SIGCOMM '88, Proceedings of the ACM Symposium on Communications Architectures and Protocols, Stanford, CA, USA, August 16-18, 1988*, pages 247–256, 1988.

[22] John A Darringer and James C King. Applications of symbolic execution to program testing. *Computer*, 11(4):51–60, 1978.

[23] Alexandre David, Kim G Larsen, Marius Mikučionis, Omer L Nguena Timo, and Antoine Rollet. Remote testing of timed specifications. In *IFIP International Conference on Testing Software and Systems*, pages 65–81. Springer, 2013.

[24] Hernán Ponce De León, Stefan Haar, and Delphine Longuet. Conformance relations for labeled event structures. In *TAP*, pages 83–98. Springer, 2012.

[25] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[26] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] Eclipse Formal Modeling Project (DIVERSITY) web site. https://projects.eclipse.org/proposals/eclipse-formal-modeling-project. Accessed: 2017-09-29.

[28] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[29] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[30] IK El-Far and JA Whittaker. Model-based software testing. encyclopedia of software engineering (edited by jj marciniak), 2001.

[31] Jose Pablo Escobedo, Christophe Gaston, Pascale Le Gall, and Ana Cavalli. Testing web service orchestrators in context: A symbolic approach. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, pages 257–267. IEEE, 2010.

[32] J.P. Escobedo, C. Gaston, and P. Le Gall. Timed Conformance Testing for Orchestrated Service Discovery. In *Proc. of Int. Conf. Formal Aspects of. Component Software (FACS)*. Springer, 2011.

[33] Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):56–66, 1988.

[34] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[35] Behrouz A Forouzan. *TCP/IP protocol suite*. McGraw-Hill, Inc., 2002.

[36] Lars Frantzen, Jan Tretmans, and Tim AC Willemse. Test generation based on symbolic specifications. In *International Workshop on Formal Approaches to Software Testing*, pages 1–15. Springer, 2004.

[37] Christophe Gaston, Robert M Hierons, and Pascale Le Gall. An implementation relation and test framework for timed distributed systems. In *IFIP International Conference on Testing Software and Systems*, pages 82–97. Springer, 2013.

[38] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *TestCom*, volume 3964, pages 1–18. Springer, 2006.

[39] Sudipto Ghosh and Aditya P Mathur. Issues in testing distributed component-based systems. In *First ICSE workshop on testing distributed component-based systems*, pages 211–220, 1999.

[40] Vehbi C Gungor, Dilan Sahin, Taskin Kocak, Salih Ergut, Concettina Buccella, Carlo Cecati, and Gerhard P Hancke. Smart grid technologies: Communication technologies and standards. *IEEE transactions on Industrial informatics*, 7(4):529–539, 2011.

[41] Riccardo Gusella and Stefano Zatti. The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3 bsd. *IEEE transactions on Software Engineering*, 15(7):847–853, 1989.

[42] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[43] Anders Hessel, Kim G Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In *Formal methods and testing*, pages 77–117. Springer, 2008.

[44] Robert M Hierons, Mercedes G Merayo, and Manuel Núnez. Using time to add order to distributed testing. In *FM*, pages 232–246. Springer, 2012.

[45] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8(4):371–379, July 1982.

[46] Claude Jard, Thierry Jéron, Lénaick Tanguy, and César Viho. Remote testing can be as powerful as local testing. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 25–40. Springer, 1999.

[47] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[48] John G Kemeny and James Laurie Snell. *Finite markov chains*, volume 356. van Nostrand Princeton, NJ, 1960.

[49] Mohd Ehmer Khan and Farmeena Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Sciences and Applications*, 3(6):12–1, 2012.

[50] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[51] Myungchul Kim, Jaehwi Shin, Samuel T Chanson, and Sungwon Kang. An approach for testing asynchronous communicating systems. *IEICE transactions on communications*, 82(1):81–95, 1999.

[52] Myungchul Kim, Jaehwi Shin, Samuel T Chanson, and Sungwon Kang. An enhanced model for testing asynchronous communicating systems. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 337–356. Springer, 1999.

[53] James C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, April 1975.

[54] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[55] Kim N King and A Jefferson Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[56] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Trans. Comput.*, 36(8):933–940, August 1987.

[57] M. Krichen. A formal framework for black-box conformance testing of distributed real-time systems. *Int. Journal of Critical Computer-Based Systems*, 3(1/2):26–43, 2012.

[58] M. Krichen and S. Tripakis. Black-box time systems. In *Proc. of Int. SPIN Workshop Model Checking of Software*. Springer, 2004.

[59] Moez Krichen. A formal framework for conformance testing of distributed real-time systems. In *International Conference On Principles Of Distributed Systems*, pages 139–142. Springer, 2010.

[60] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In *SPIN*, volume 2989, pages 109–126. Springer, 2004.

[61] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[62] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 34(3):238–304, June 2009.

[63] James F Kurose and Keith W Ross. *Computer networking: a top-down approach*, volume 4. Addison Wesley Boston, USA, 2009.

[64] Murat Kuzlu, Manisa Pipattanasomporn, and Saifur Rahman. Communication network requirements for major smart grid applications in han, nan and wan. *Computer Networks*, 67:74–88, 2014.

[65] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[66] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[67] Patrick McDaniel and Stephen McLaughlin. Security and privacy challenges in the smart grid. *IEEE Security & Privacy*, 7(3), 2009.

[68] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[69] Robert Nilsson, Jeff Offutt, and Jonas Mellin. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer Science*, 164(4):97–114, 2006.

[70] Katia Obraczka. Multicast transport protocols: a survey and taxonomy. *IEEE Communications magazine*, 36(1):94–102, 1998.

[71] Alexandre Petrenko and Nina Yevtushenko. Testing from partial deterministic fsm specifications. *IEEE Transactions on Computers*, 54(9):1154–1165, 2005.

[72] Hernán Ponce-de León, Stefan Haar, and Delphine Longuet. Distributed testing of concurrent systems: vector clocks to the rescue. In *International Colloquium on Theoretical Aspects of Computing*, pages 369–387. Springer, 2014.

[73] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116:59–71, 2005.

[74] Olli-Pekka Puolitaival. Model-based testing tools. *Presentation at Software Testing Day at TUT*, 2008.

[75] C. V. Ramamoorthy, S. B. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, Dec 1976.

[76] Wolfgang Reisig. Petri nets: an introduction, volume 4 of eatcs monographs on theoretical computer science, 1985.

[77] Ericsson Int. report. Investigation on how to integrate Diversity (MBT tool) and Titan (TTCN-3 executor) to provide an open source MBT tool chain, 2016-08-26.

[78] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[79] Laxman H Sahasrabuddhe and Biswanath Mukherjee. Multicast routing algorithms and protocols: A tutorial. *IEEE network*, 14(1):90–102, 2000.

[80] Ahmad Saifan and Juergen Dingel. Model-based testing of distributed systems. *School of Computing Queen's University Canada*, 2008.

[81] Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. *Formal Modeling and Analysis of Timed Systems*, pages 250–264, 2008.

[82] Ian Sommerville and Pete Sawyer. *Requirements engineering: a good practice guide*. John Wiley & Sons, Inc., 1997.

[83] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[84] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[85] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer networks and ISDN systems*, 29(1):49–79, 1996.

[86] Jan Tretmans. *Test Generation with Inputs, Outputs and Repetitive Quiescence*. Number TR-CTIT-96-26 in CTIT technical report series. Centre for Telematics and Information Technology (CTIT), Netherlands, 1996. CTIT Tecnnical Report Series 96-26.

[87] Jan Tretmans. *Model Based Testing with Labelled Transition Systems*, pages 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[88] Edward Tsang. *Foundations of constraint satisfaction: the classic text*. BoD–Books on Demand, 2014.

[89] M. Utting, B. Legeard, A. Pretschner, and University of Waikato. Department of Computer Science. *A Taxonomy of Model-based Testing*. Working paper series (University of Waikato. Department of Computer Science). Department of Computer Science, University of Waikato, 2006.

[90] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[91] Marlon E Vieira, Marcio S Dias, and Debra J Richardson. Object-oriented specification-based testing using uml statechart diagrams. In *Proceedings of the Workshop on Automated Program Analysis, Testing, and Verification (at ICSE'00)*, 2000.

[92] Stephan Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines*. PhD thesis, Humboldt University of Berlin, 2010.

[93] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, volume 3440, pages 365–381. Springer, 2005.

[94] Peter Zimmerer. Test architectures for testing distributed systems. *12th International software quality week (QW 99)*, 1999.

université PARIS-SACLAY

ÉCOLE DOCTORALE
INTERFACES
Approches Interdisciplinaires : Fondements, Applications et Innovation

Titre : Test à base de modèles de systèmes temporisés distribués : Une approche basée sur les contraintes pour résoudre le problème de l'oracle

Mots clés : Test Distribué Temporisé, Test à base de Modèles, Test à base de Contraintes, Problème de Satisfaction de Contraintes, Systèmes de Transition Symboliques Temporisés à Entrée Sortie, Test Off-line.

Résumé : Le test à base de modèles des systèmes réactifs est le processus de vérifier si un système sous test (SUT) est conforme à sa spécification. Il consiste à gérer à la fois la génération des données de test et le calcul de verdicts en utilisant des modèles. Nous spécifions le comportement des systèmes réactifs à l'aide des systèmes de transitions symboliques temporisées à entrée-sortie (TIOSTS). Quand les TIOSTSs sont utilisés pour tester des systèmes avec une interface centralisée, l'utilisateur peut ordonner complètement les événements (i.e., les entrées envoyées au système et les sorties produites). Les interactions entre le testeur et le SUT consistent en des séquences d'entrées et de sortie nommées traces, pouvant être séparées par des durées dans le cadre du test temporisé, pour former ce que l'on appelle des traces temporisées. Les systèmes distribués sont des collections de composants locaux communiquant entre eux et interagissant avec leur environnement via des interfaces physiquement distribuées. Différents événements survenant à ces différentes interfaces ne peuvent plus être ordonnés. Cette thèse concerne le test de conformité des systèmes distribués où un testeur est placé à chaque interface localisée et peut observer ce qui se passe à cette interface. Nous supposons qu'il n'y a pas d'horloge commune mais seulement des horloges locales pour chaque interface. La sémantique de tels systèmes est définie comme des tuples de traces temporisées. Nous considérons une approche du test dans le contexte de la relation de conformité distribuée *dtioco*. La conformité globale peut être testée dans une architecture de test en utilisant des testeurs locaux sans communication entre eux. Nous proposons un algorithme pour vérifier la communication pour un tuple de traces temporisées en formulant le problème de message-passing en un problème de satisfaction de contraintes (CSP). Nous avons mis en œuvre le calcul des verdicts de test en orchestrant à la fois les algorithmes du test centralisé off-line de chacun des composants et la vérification des communications par le biais d'un solveur de contraintes. Nous avons validé notre approche sur un cas étude de taille significative.

Title: Model-Based Testing of Timed Distributed Systems: A Constraint-Based Approach for Solving the Oracle Problem

Keywords: Timed Distributed Testing, Model-based Testing, Constraint-based Testing, Constraint Satisfaction Problem, Timed Input Output Symbolic Transition Systems, Off-line Testing.

Abstract : Model-based testing of reactive systems is the process of checking if a System Under Test (SUT) conforms to its model. It consists of handling both test data generation and verdict computation by using models. We specify the behaviour of reactive systems using Timed Input Output Symbolic Transition Systems (TIOSTS) that are timed automata enriched with symbolic mechanisms to handle data. When TIOSTSs are used to test systems with a centralized interface, the user may completely order events occurring at this interface (i.e., inputs sent to the system and outputs produced from it). Interactions between the tester and the SUT are sequences of inputs and outputs named traces, separated by delays in the timed framework, to form so-called timed traces. Distributed systems are collections of communicating local components which interact with their environment at physically distributed interfaces. Interacting with such a distributed system requires exchanging values with it by means of several interfaces in the same testing process. Different events occurring at different interfaces cannot be ordered any more. This thesis focuses on conformance testing for distributed systems where a separate tester is placed at each localized interface and may only observe what happens at this interface. We assume that there is no global clock but only local clocks for each localized interface. The semantics of such systems can be seen as tuples of timed traces. We consider a framework for distributed testing from TIOSTS along with corresponding test hypotheses and a distributed conformance relation called *dtioco*. Global conformance can be tested in a distributed testing architecture using only local testers without any communication between them. We propose an algorithm to check communication policy for a tuple of timed traces by formulating the verification of message passing in terms of Constraint Satisfaction Problem (CSP). Hence, we were able to implement the computation of test verdicts by orchestrating both localised off-line testing algorithms and the verification of constraints defined by message passing that can be supported by a constraint solver. Lastly, we validated our approach on a real case study of a telecommunications distributed system.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France