FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Observability and Controllability in Scenario-based Integration Testing of Time-Constrained Distributed Systems

Bruno Miguel Carvalhido Lima

PHD THESIS

Doctoral Program in Informatics Engineering

Supervisor: João Pascoal Faria

February 18, 2021

© Bruno Lima, 2021

Observability and Controllability in Scenario-based Integration Testing of Time-Constrained Distributed Systems

Bruno Miguel Carvalhido Lima

In partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Informatics Engineering

by the

Doctoral Program in Informatics Engineering, FEUP

Approved by:

President: Dr. Rui Filipe Lima Maranhão de Abreu
Referee: Dr. Shaukat Ali
Referee: Dr. Miguel Carlos Pacheco Afonso Goulão
Referee: Dr. Alberto Manuel Rodrigues da Silva
Referee: Dr. Ana Cristina Ramada Paiva
Referee: Dr. João Carlos Pascoal Faria (Supervisor)

February 18, 2021

Abstract

Evermore end-to-end digital services in several domains (e-health, smart cities, etc.) depend on the proper interoperation of multiple products, forming a distributed system, often subject to timing requirements. To ensure interoperability and the timely behavior of such systems, it is important to conduct integration tests that verify the interactions with the environment and between the system components in key scenarios. The automation of such integration tests requires that test components are also distributed, with local testers deployed close to the system components, coordinated by a central tester. However, test automation in this type of systems is a huge challenge, namely due to the difficulty to test the system as a whole due to the number and diversity of individual components; the difficulty to coordinate and synchronize the test participants and interactions, due to the distributed nature of the system (particularly, in integration testing); the difficulty to test the components individually, because of the dependencies on other components.

In this research work, we tackle the challenge of automating the scenario-based integration testing of time-constrained distributed systems showing that is possible to solve the coordination problems by performing a pre-processing of the test scenarios to determine if they can be executed in a purely distributed way or, if that is not possible, determining the minimum number of coordination messages or time constraints to be inserted, minimizing the communication overhead whilst maximizing the fault detection capability and the test harness responsiveness.

To achieve this goal, we perform a state of the art analysis on time-constrained distributed systems testing and a state of the practice analysis on testing distributed and heterogeneous systems by collecting the opinion of more than 140 professionals.

In response to the identified needs, we propose a novel testing approach and architecture for the integration testing of time-constrained distributed systems that provides a higher level of automation of the testing process because all phases of the test process are supported in an integrated fashion. To determine if a test scenario can be executed in a purely distributed way, we analyze two properties: local observability, defined as the ability of the local testers to detect conformance errors, without the need for exchanging coordination messages between them during test execution; and local controllability, defined as the ability of the local testers to decide test inputs locally, without the need for exchanging coordination messages between them during test execution. If such properties do not hold, we determine coordination messages and/or coordination time constraints that can be inserted to refine the given scenario and enforce local observability and/or local controllability. Even if the given scenario is not refined, the results of the analysis are helpful to influence the test execution strategy. Local observability and controllability analysis and enforcement algorithms are implemented in the DCO Analyzer tool, for test scenarios described by means of UML sequence diagrams. Since many local observability and controllability problems may be caused by design flaws or incomplete specifications, and multiple ways may exist to enforce local observability and controllability, the tool was designed as a static analysis assistant to be used before test execution.

The DCO Analyzer tool was validated in an industry case study and proved to be useful for

professionals who daily perform the modeling and testing of this type of systems. DCO Analyzer was able to detect local observability and controllability problems in different scenarios, as well as provide the user with corresponding solutions to overcome them, either through coordination messages, time constraints, or, in some cases, presenting the two alternatives.

Resumo

Cada vez mais os serviços digitais ponta-a-ponta em diversos domínios (e-saúde, cidades inteligentes, etc.) dependem da correta interoperação de múltiplos produtos, formando um sistema distribuído, muitas vezes sujeito a requisitos temporais. Para garantir a interoperabilidade e o correto comportamento destes sistemas, é importante realizar testes de integração que verifiquem as interações com o ambiente e entre os componentes do sistema em cenários chave. No entanto, a automatização de testes neste tipo de sistemas é um grande desafio, nomeadamente devido à dificuldade em testar o sistema como um todo devido ao número e diversidade de componentes envolvidos; a dificuldade de coordenar e sincronizar os participantes do teste e as suas interações, devido à natureza distribuída do sistema; e a dificuldade de testar os componentes individualmente, devido à dependência de outros componentes.

Neste trabalho, abordamos o desafio de automatizar o teste de integração de sistemas distribuídos baseado em cenários com restrições temporais, mostrando que é possível resolver os problemas de coordenação no teste deste tipo de sistemas através da realização de um pré-processamento dos cenários de teste para determinar se podem ser executados de forma puramente distribuída ou, caso não seja possível, determinar o número mínimo de mensagens de coordenação ou restrições temporais a inserir, minimizando a sobrecarga de comunicação enquanto se maximiza a capacidade de detecção de falhas e a responsividade do ambiente de teste.

Para atingir esse objetivo, realizamos uma análise do estado da arte em testes de sistemas distribuídos com restrições temporais e uma análise do estado da prática em testes de sistemas distribuídos e heterogéneos, recolhendo a opinião de mais de 140 profissionais.

Em resposta às necessidades identificadas, propomos nesta tese uma nova abordagem de teste e arquitetura para o teste de integração de sistemas distribuídos com restrições temporais que fornece um maior nível de automação do processo de teste uma vez que todas as fases do processo de teste são suportadas por um inovador sistema integrado. Para determinar se um cenário de teste pode ser executado de forma puramente distribuída, analisamos duas propriedades: observabilidade local, definida como a capacidade dos componentes de teste locais detetarem erros de conformidade, sem a necessidade de trocarem mensagens de coordenação entre eles durante a execução do teste; e controlabilidade local, definida como a capacidade dos componentes de teste locais decidirem os próximos dados de entrada localmente, sem a necessidade de trocarem mensagens de coordenação entre eles durante a execução do teste. Se tais propriedades não forem válidas, determinamos o conjunto de mensagens de coordenação e/ou restrições temporais que podem ser inseridas para refinar o cenário de forma a garantir a observabilidade local e/ou controlabilidade local. Mesmo que o cenário não seja refinado, os resultados da análise são úteis para selecionar a estratégia de execução dos testes. Os algoritmos para análise e garantia de observabilidade e controlabilidade local, para cenários de teste descritos por diagramas de sequência UML, foram implementados na ferramenta DCO Analyzer. Uma vez que muitos dos problemas de observabilidade e controlabilidade local podem ser causados por falhas de conceção ou especificações incompletas, e podem existir várias formas para impor a observabilidade e controlabilidade local, a ferramenta

foi projetada como um assistente de análise estática para ser usado antes da execução dos testes.

A ferramenta DCO Analyzer foi validada em um estudo de caso industrial, mostrando-se útil para profissionais que modelam e testam este tipo de sistemas. A DCO Analyzer foi capaz de detetar problemas de observabilidade e controlabilidade local em diferentes cenários, bem como fornecer ao utilizador as soluções correspondentes para corrigir os problemas encontrados, recorrendo a mensagens de coordenação, restrições temporais ou, em alguns casos, apresentando as duas alternativas.

Acknowledgements

Throughout the development of this thesis, I received a lot of support and assistance.

This thesis could not have been accomplished without the financial support of the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, under research grant SFRH/BD/115358/2016. INESC TEC, which supported me in the publication of some scientific papers, and the Department of Informatics Engineering, which welcomed me at its facilities and gave me the opportunity to teach during this time.

I would like to thank my advisor, Professor João Pascoal Faria, whose experience, professionalism and availability were invaluable throughout these years. During these years I have learned a lot from him, not only professionally but also personally. It was a pleasure to work with you.

I would like to thank my colleagues who have been through the software engineering laboratory over the years and with whom I had the opportunity to have several discussions about the development of this work.

I would also like to thank the people I have had the opportunity to meet over the years at the various conferences I have attended and who have given me important advice on this research work.

In addition, I would like to thank my parents for all their support. You have always been there for me. Finally, and no less important, I would like to thank Ni for having always been by my side during these years.

"Software testers succeed where others fail."

Anonymous

Contents

1 Introduction 1
1.1 Motivation . . . 1
1.2 Research Goals . . . 2
1.3 Research Contributions . . . 4
1.4 Document Structure . . . 8

2 Background and State of the Art 9
2.1 Software Testing Concepts . . . 9
2.1.1 Test Levels . . . 9
2.1.2 Test Case Design Strategies . . . 10
2.1.3 Test Automation . . . 11
2.2 Model-based Testing . . . 12
2.2.1 Model-based Testing Process . . . 12
2.2.2 Test Levels . . . 12
2.2.3 Test Generation . . . 13
2.2.4 Test Execution . . . 14
2.2.5 Input Models . . . 15
2.2.6 MBT Approaches for Integration Testing . . . 17
2.3 Testing Frameworks for Distributed Systems . . . 17
2.4 Test Architectures for Distributed Systems Testing . . . 20
2.5 Observability and Controllability in Distributed Systems Testing . . . 22
2.6 Other Testability Issues in Distributed Systems Testing . . . 23
2.7 Time Constraints in Distributed Systems Testing . . . 24
2.8 Summary . . . 25

3 State of the Practice on Testing Distributed and Heterogeneous Systems 27
3.1 Research Method and Scope . . . 27
3.1.1 Goal . . . 27
3.1.2 Survey Distribution and Sampling . . . 28
3.1.3 Survey Organization . . . 28
3.2 Results . . . 28
3.2.1 Participants Characterization . . . 28
3.2.2 Company Characterization . . . 29
3.2.3 Distributed and Heterogeneous Systems Testing . . . 29
3.2.4 Multivariate Analysis . . . 34
3.3 Discussion . . . 35
3.3.1 Relevance of respondents . . . 35
3.3.2 SRQ1: How relevant are DHS in the software testing practice? . . . 35

3.3.3 SRQ2: What are the most important features to be tested in DHS? 36
3.3.4 SRQ3: What is the current status of test automation and tool sourcing for testing DHS? 36
3.3.5 SRQ4: What are the most desired features in test automation solutions for DHS? 36
3.4 Case Based Analysis of Test Automation Obstacles 37
3.4.1 Case A 37
3.4.2 Case B 37
3.4.3 Case C 38
3.4.4 Synthesis 38
3.5 Other Surveys 39
3.6 Conclusions 40

4 Proposed Testing Approach and Architecture **43**
4.1 Test Architecture 43
4.2 Test Process 45
4.2.1 Visual Modeling 45
4.2.2 Local Observability and Controllability Analysis and Enforcement 46
4.2.3 Translation to Runtime Model 48
4.2.4 Test Initiation 49
4.2.5 Local Test Input Selection 50
4.2.6 Local Test Driving 51
4.2.7 Local Test Monitoring 51
4.2.8 Local Conformance Checking 51
4.2.9 Test Finalization 51
4.2.10 Test Results Mapping 52
4.3 Toolset Architecture 52
4.3.1 Visual Modeling, Analysis and Execution Environment 54
4.3.2 Distributed Test Input Selection and Conformance Checking Engine . . . 54
4.3.3 Distributed Test Monitoring and Control Infrastructure 54
4.4 Conclusions 55

5 Local Observability and Controllability Analysis and Enforcement Algorithms **57**
5.1 Motivating Examples 57
5.2 Semantics of Basic UML SDs 62
5.2.1 Valid Global Traces Defined by a UML Sequence Diagram 62
5.3 Semantics of Time-constrained UML SDs 66
5.3.1 Time-constrained Sequence Diagrams 66
5.3.2 Timed Traces 66
5.3.3 Time-constrained Traces 67
5.3.4 Valid Traces and Satisfiability Checking 67
5.3.5 Operators on Timed Traces and Time-constrained Traces 69
5.3.6 Conformance Checking Based on Distributed Observations 69
5.4 Local Observability Analysis 73
5.4.1 Definition 74
5.4.2 Local Observability Checking 74
5.4.3 Impact of Non-synchronized Clocks 75
5.5 Local Controllability Analysis 76
5.5.1 Definitions and Example 76

5.5.2 Symbolic Simulation 77
5.6 Local Observability and Controllability Enforcement 80
5.6.1 Determination of Error Locations 80
5.6.2 Generation of Coordination Messages 81
5.6.3 Generation of Coordination Time Constraints 82
5.6.4 Application and Evaluation of Candidate Fixes 84
5.7 Conclusions 85

6 Implementation **87**
6.1 Tool Architecture 87
6.2 Back-end 87
6.3 Front-end 89
6.4 Usage Examples 94

7 Validation **99**
7.1 Validation Tests 99
7.2 Industrial Case Study 109
7.2.1 Motorway Incident Detection Project 109
7.2.2 Test Scenario 110
7.2.3 Scenario Analysis - Local Controllability 110
7.2.4 Scenario Analysis - Local Observability 112
7.2.5 Scenario Refinement 112
7.2.6 Discussion 113
7.2.7 Threats to Validity 113
7.3 Conclusions 114

8 Conclusions **115**
8.1 Summary of Contributions 115
8.2 Research Questions Revisited 116
8.3 Future Work 117

A Testing Distributed and Heterogeneous Systems – State of the practice - Survey Form **119**
A.1 *Testing Distributed and Heterogeneous Systems – State of the practice Survey* . . 120

B Observability and Controllability in Scenario-based Integration Testing of Time- Constrained Distributed Systems: VDM++ Specifications **125**
B.1 *Observability and Controllability in Scenario- based Integration Testing of Time-Constrained Distributed Systems: VDM++ Specifications* 126

References **211**

List of Figures

2.1 Test harness . . . 11
2.2 Typical MBT test process (UML activity diagram) . . . 13
2.3 Application fields of model-based testing . . . 13
2.4 UML behavioral state machine diagram . . . 15
2.5 UML SD diagram of a cash withdrawal scenario at an ATM. . . . 16
2.6 A purely distributed test architecture. . . . 20
2.7 A purely centralized test architecture. . . . 21
2.8 A hybrid test architecture. . . . 22

3.1 Current Position and Time in Current Position. . . . 29
3.2 Time in Software Testing. . . . 29
3.3 Industry Sectors. . . . 30
3.4 Company Size. . . . 30
3.5 Company Roles. . . . 31
3.6 Test Levels. . . . 31
3.7 Test Roles DHS. . . . 32
3.8 Test Levels DHS. . . . 32
3.9 Features. . . . 33
3.10 Automation Level. . . . 33
3.11 Automation Tool. . . . 34
3.12 Tool Features. . . . 34
3.13 New Tool. . . . 35

4.1 Test architecture for the model-based integration testing of distributed systems. . 44
4.2 Dataflow view of the proposed test process. . . . 46
4.3 UML deployment diagram of a fall detection scenario. . . . 47
4.4 UML sequence diagram representing the interactions of the fall detection scenario 48
4.5 Location of the Local testers for the example represented in Figure 4.4. . . . 49
4.6 Toolset architecture. . . . 53

5.1 Interaction fragments with local observability and controllability problems and possible refinments. . . . 58
5.2 Interaction fragments with local observability and controllability problems and possible refinments (continued). . . . 60
5.3 Interactions . . . 63
5.4 Example of a fall detection scenario (simplified) from an ambient assisted living ecosystem (AAL4ALL). . . . 66
5.5 Satisfiability checking example (trace from Figure 5.4). . . . 70
5.6 Operators on timed traces and time-constrained traces. . . . 70

5.7 Examples of traces with different conformance checking verdicts in the fall detection scenario . . . 73
5.8 Example of local observability checking. . . . 75
5.9 Example of symbolic execution for the SD of Figure 5.4 . . . 79
5.10 Fixing race conditions with coordination time constraints. . . . 82
5.11 Causal dependencies and slicing operations (trace of Figure 5.1a). . . . 83

6.1 DCO Analyzer overview . . . 88
6.2 Structure of the DCO Analyzer Back-End (UML class diagram, simplified) . . . 90
6.3 DCO Analyzer Front-End Modules . . . 91
6.4 DCO Analyzer Front-End . . . 92
6.5 Example of Basic SD in Papyrus . . . 93
6.6 Example of Initial SD in Papyrus . . . 94
6.7 Example of DCO Analyzer output . . . 95
6.8 Refined SD in Papyrus . . . 96
6.9 Example of a scenario not locally observable . . . 97

7.1 TestRacePlusAlt UML SD. . . . 100
7.2 TesStrict UML SD Corrected. . . . 101
7.3 TesStrict UML SD with strict sequencing and a race condition. . . . 101
7.4 TestSendRecvEnabled UML SD. . . . 102
7.5 TestSendRecvEnabled UML SD. . . . 103
7.6 TestLoop UML SD. . . . 104
7.7 TestNonLocallyControlableTimed UML SD. . . . 105
7.8 TestStrangeControllableTimed UML SD. . . . 106
7.9 Traffic Control System. . . . 110
7.10 Initial scenario and problem locations. . . . 111
7.11 Refined locally controllable scenario . . . 114

List of Tables

2.1 MBT approaches supporting integration testing 18
2.2 Comparison of MBT approaches 19
2.3 Summary of comparison of related work in distributed systems testing. 26

Abbreviations

AAL	Ambient Assisted Living
CPN	Colored Petri Nets
CUT	Component Under Test
DHS	Distributed and Heterogeneous Systems
EMF	Eclipse Modelling Framework
FSM	Finite State Machine
IoT	Internet of Things
LTL	Labelled transition systems
MBT	Model-Based Testing
MSC	Message Sequence Charts
SD	Sequence Diagram
SUT	System Under Test
TEDCPN	Timed Event-Driven Colored Petri Nets
UML	Unified Modeling Language

Chapter 1

Introduction

1.1 Motivation

Due to the increasing ubiquity, complexity, criticality and need for assurance of software based systems (Boehm, 2011), testing is a fundamental lifecycle activity, with a huge economic impact if not performed adequately (Tassey, 2002). Such trends, combined with the needs for shorter delivery times and reduced costs, demand for the continuous improvement of software testing methods and tools, in order to make testing activities more effective and efficient. At the same time, the systems under test (SUT) are increasingly complex and interconnected, posing additional testing challenges.

As the recent Mind Commerce report (Commerce, 2020) shows, the number of interconnected devices will not stop growing considerably in the coming years, both at the industrial level and in solutions for the final consumer. It is also expected that the areas of application of these interconnected systems will be very comprehensive and that such systems are expected to be created in markets such as agriculture, advertising and media, automobiles, security management, energy management, healthcare, manufacturing, oil & gas, public safety, and telecommunication.

These new systems created from the interconnection of various devices are not more like simple applications but systems that have evolved to large and complex system of systems (DoD, 2008). A system of systems consists of a set of small independent systems that together form a new system. It can be a combination of hardware components (sensors, actuators, etc.) and software systems used to create big systems or ecosystems that can offer multiple services to their users. As the number of systems of systems has been growing, they have captured the interest of the software engineering research community (Ali et al., 2012).

This is particularly true for the end-to-end services that are being proposed in several domains (e-health, smart cities, etc.), taking advantage of recent advances in cloud computing, mobile computing, and Internet of Things (IoT) (Moutai et al., 2019; Kim et al., 2018; Hwang et al., 2020). Such services depend on the proper interoperation of multiple devices and applications, from different vendors, forming a distributed and heterogeneous system or system of systems, often subject to timing requirements. To ensure interoperability and the correct, secure and timely behavior of

such systems, it is important to conduct integration tests that verify not only the interactions with the environment but also between the system components in key scenarios. Integration test scenarios for that purpose may be conveniently specified by means of UML Sequence Diagrams OMG (2017) (SDs), because they are an industry standard well suited for describing and visualizing the interactions that occur between the components and actors of a distributed system, and may be enriched with control flow variants and time constraints.

However, test automation in this type of systems is a huge challenge (Bures et al., 2018), namely due to the difficulty to test the system as a whole due to the number and diversity of individual components; the difficulty to coordinate and synchronize the test participants and interactions, due to the distributed nature of the system; the difficulty to test the components individually, because of the dependencies on other components.

We experienced such challenges during our participation in the nationwide AAL4ALL project (AAL4ALL, 2015). During the ALL4ALL project it was prototyped an Ambient Assisted Living (AAL) ecosystem, comprising a set of interoperable AAL products and services (sensors, actuators, mobile and web based applications and services, middleware components, etc.), produced by different manufacturers using different technologies and communication protocols (web services, message queues, etc.). To ensure interoperability and the integrity of the ecosystem, it was developed and piloted a testing and certification methodology (Faria et al., 2014), encompassing the specification of 'standard' interfaces and component categories, the specification of unit (component) and integration test scenarios, and the test implementation and execution on candidate components by independent test laboratories. A major problem faced during test implementation and execution was related with test automation, due to the diversity of component types and communication interfaces, the distributed nature of the system, and the lack of support tools. Similar difficulties have been reported in other domains, such as the railway domain (Torens and Ebrecht, 2010). In fact, we found in the literature limited tool support for automating the whole process of specification-based (or model-based) testing of distributed and heterogeneous systems, as will be explained in Chapters 2 and 3.

1.2 Research Goals

In order to address the challenges and problems previously described, in this research work we tackle the challenge of automating the scenario-based integration testing of time-constrained distributed systems, using UML SDs as input models. Those diagrams describe the expected behavior of the SUT, with support for control flow variants, time constraints, and non-determinism.

To better target our research, we start by addressing the following research question.

> **RQ1** - *What are the main difficulties and needs in the integration testing of distributed systems listed in the state of the art and state of practice?*

The research study conducted confirmed the importance of automating the integration testing of distributed systems, with support for multiple platforms, time constraints, non-determinism,

UML SDs (as input models), among other features, as well as the limited tool support for that purpose.

In the state of the art, we found proposals of test architectures and associated conformance relations for testing distributed systems, with varying fault detection capabilities, but mostly focused on system testing and not integration testing.

This led us to the second research question.

> **RQ2** - *What is an adequate architecture and approach to conduct integration tests in these types of systems?*

The state of the art analysis showed that a *hybrid test architecture* (combining the characteristics of purely distributed and purely centralized architectures), is the one that maximizes the fault detection capability, in the context of system testing of distributed systems. Such a hybrid architecture comprises a central test component (*central tester*), that interacts asynchronously with local test components (*local testers*) located close to the ports of the distributed system (points of interaction with the environment).

In the context of integration testing, besides the need to check the interactions with the environment (users, other systems, or even the physical environment) and simulate inputs from the environment at multiple locations, there is also the need to check the interactions between the system components. This implies that test execution needs also to be distributed, with local testers deployed close to the system components (and not only the system ports), coordinated by a central tester.

To cope with non-determinism and response time constraints, test inputs may have to be selected at runtime in an adaptive and responsive way, based on the observed execution events and the behavioral specification, suggesting an adaptive and distributed test input selection approach. To facilitate fault localization, conformance errors should be detected as early as possible and as close as possible to the offending components, suggesting an incremental and distributed conformance checking approach.

However, depending on the test scenario under consideration, a purely distributed test execution (with purely distributed test input selection and conformance checking), without exchanging coordination messages between the test components during test execution, may not be possible or safe, which raises our next research question.

> **RQ3** - *How do we determine if a test scenario described by a UML SD can be executed safely in a purely distributed manner, without overlooking conformance faults (false negatives) or injecting conformance faults (false positives) by the test harness? In other words, how do we determine if a test scenario described by a UML SD is locally observable and locally controllable?*

Based on the state of the art analysis, we frame the problem in terms of observability and controllability. To determine if a test scenario can be executed in a purely distributed way, we analyze two properties: *local observability*, which we define as the ability of the local testers to

detect conformance errors, without the need for exchanging coordination messages between the test components during test execution; and *local controllability*, which we define as the ability of the local testers to decide test inputs locally, without the need for exchanging coordination messages between the test components during test execution.

The next question that arises is: what to do if those properties do not hold. Based on the state of the art analysis and several case studies, we conclude that, in many cases, violations to those properties are due to design flaws or incomplete specifications. So, having means to automatically recommend *refinements* to those scenarios, in order to *enforce* local observability and local controllability, would be helpful. In the state of the art, it are proposed refinements based on the addition of *coordination messages* to overcome some violations (such as race conditions), but refinements based on the addition of *coordination time constraints* can also be explored.

In other cases, violations of local observability and local controllability may be due to inherent characteristics of the SUT. In that case, the identification of minimal sets of coordination messages and coordination time constraints to be added to the given scenario to enforce local observability and local controllability is also of primary importance, as a means to solve the test coordination problem, in a way that minimizes the communication overhead and maximizes the fault detection capability and test responsiveness. Such coordination messages would be exchanged between the local testers, without affecting the SUT. Similarly, the coordination time constraints would affect the behavior of the local testers, and not the SUT.

This raises our last research question.

> **RQ4** - *Given a test scenario not locally controllable or locally observable, how can we automatically identify a minimal set of coordination messages and/or coordination time constraints to refine the test scenario and enforce local observability and/or local controllability?*

It also worth noting that, even if the given scenario is not refined, the results of the local observability and controllability analysis are helpful to influence the test execution strategy.

We can now state our main research hypothesis:

> *It is possible to solve the coordination problems in the scenario-based integration testing of distributed systems with time constraints by performing a preprocessing of the test scenarios to determine if they can be executed in a purely distributed way or, if that is not possible, determining the minimum number of coordination messages or time constraints to be inserted.*

1.3 Research Contributions

The contributions of this research are:

- **A state of the art on time-constrained distributed systems testing**. More details in Chapter 2.

- **A state of the practice on testing distributed and heterogeneous systems**. More details in Chapter 3.
- **A testing approach and architecture** for the integration testing of time-constrained distributed systems, requiring as only manual activity the creation of the input behavioral models (test scenarios). More details in Chapter 4.
- **Local observability and controllability analysis and enforcement algorithms**, for integration test scenarios described by means of UML SDs with time constraints and control flow variants (problem not solved before, to our knowledge). More details in Chapter 5.
- **DCO Analyzer**, the first tool to analyze and enforce local observability and controllability of integration test scenarios for time-constrained distributed systems described by UML SDs. More details in Chapter 6.
- **Industry case study validation** in the transportation domain. More details in Chapter 7.

During this research, the author contributed to 14 peer-reviewed publications directly related to this thesis:

- Lima, Bruno, and João Pascoal Faria. "An Approach for Automated Scenario-based Testing of Distributed and Heterogeneous Systems." In 10th International Conference on Software Engineering and Applications (ICSOFT-EA) 2015.

  This paper is about an initial approach and toolset architecture for automating the testing of end-to-end services in distributed and heterogeneous systems, comprising a visual modeling environment, a test execution engine, and a distributed test monitoring and control infrastructure. Part of Chapter 4 of the thesis is based on this paper.

- Lima, Bruno, and João Pascoal Faria. "Automated testing of distributed and heterogeneous systems based on UML sequence diagrams." In: Lorenz P., Cardoso J., Maciaszek L., van Sinderen M. (eds) Software Technologies. ICSOFT 2015. Communications in Computer and Information Science, vol 586. Springer, Cham, 2016.

  This paper is a extended version of ICSOF-EA 2015 paper. Part of Chapter 4 of the thesis is based on this paper.

- Lima, Bruno, and João Pascoal Faria. "Testing distributed and heterogeneous systems: State of the practice.", In 11th International Joint Conference on Software Technologies (ICSOFT), 2016.

  This paper is about an exploratory survey (responded by 147 software testing professionals that attended industry-oriented software testing conferences) on the current state of the practice concerning the testing of distributed and heterogeneous systems. Opportunities and priorities for research and innovation initiatives are also identified. Part of 3 of the thesis is based on this paper.

- Lima, Bruno. "Automated Scenario-based Testing of Distributed and Heterogeneous Systems.", In IEEE 9th International Conference on Software Testing, Verification and Validation (ICST), 2016

  This paper is about the outline of a Ph.D. research plan and a summary of preliminary results on test automation for distributed and heterogeneous systems.

- Lima, Bruno, and João Pascoal Faria. "Towards the Online Testing of Distributed and Heterogeneous Systems with Extended Petri Nets." In 2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC), pp. 230-235. IEEE, 2016.

  This paper is about a preliminary approach for the online model-based testing of end-to-end services in distributed and heterogeneous systems. The focus of this paper is on the test execution phase, in which sequence diagrams are converted to another more suitable formalism for execution. Part of Chapter 4 of the thesis is based on this paper.

- Lima, Bruno Carvalhido, and João Faria. "Conformance checking in integration testing of time-constrained distributed systems based on UML sequence diagrams." In 12th International Joint Conference on Software Technologies (ICSOFT), 2017.

  This paper is about decision procedures and criteria to check the conformance of observed execution traces against a specification set by a UML SD enriched with time constraints. Part of Chapter 5 of the thesis is based on this paper.

- Lima, Bruno Miguel Carvalhido, and João Carlos Pascoal Faria. "Towards decentralized conformance checking in model-based testing of distributed systems." In 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 356-365. IEEE, 2017.

  This paper is about conditions upon which conformance errors can be detected locally (local observability) and test inputs can be decided locally (local controllability) by the local testers, without the need for exchanging coordination messages between the test components during the test execution. Part of Chapter 5 of the thesis is based on this paper.

- Lima, Bruno, and João Pascoal Faria. "A survey on testing distributed and heterogeneous systems: The state of the practice." In International Conference on Software Technologies, pp. 88-107. Springer, Cham, 2016.

  This paper is an extended version of ICSOF 2016 paper. Some follow up interviews allowed us to further investigate drivers and barriers for distributed and heterogeneous systems test automation. Part of 3 of the thesis is based on this paper.

- Soares, Joao António Custódio, Bruno Lima, and Joao Pascoal Faria. "Automatic Model Transformation from UML Sequence Diagrams to Coloured Petri Nets." In MODELSWARD, pp. 668-679. 2018.

This paper is about an approach to automatically translate Sequence Diagrams to CPN ready for execution with CPN Tools, taking advantage of model-to-model transformation techniques provided by the Eclipse Modelling Framework (EMF).

- Lima, Bruno. "Automated scenario-based integration testing of distributed systems." In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 956-958. 2018.

  As part of the participation in the Microsoft student research competition that took place during the ESEC/FSE 2018 this paper is about algorithms for decentralized conformance checking and test input generation and for checking and enforcing the conditions (local observability and controllability) that allow decentralized test execution. Part of Chapter 5 of the thesis is based on this paper.

- Lima, Bruno. "Automated Scenario-Based Integration Testing of Time-Constrained Distributed Systems." In 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 486-488. IEEE, 2019.

  As part of the participation on the doctoral symposium of the ICST 2019, this paper is about algorithms for decentralized conformance checking and test input generation, and for checking and enforcing the conditions (local observability and controllability) that allow decentralized test execution. Part of Chapter 5 of the thesis is based on this paper.

- Lima, Bruno, João Pascoal Faria, and Robert Hierons. "Local Observability and Controllability Enforcement in Distributed Testing." In International Conference on the Quality of Information and Communications Technology, pp. 327-338. Springer, Cham, 2019.

  This paper is about an approach with tool support to automatically find coordination messages that, added to test scenarios, make it locally controllable and locally observable. Parts of the Chapters 5 and 6 of the thesis are based on this paper.

- Lima, Bruno, and João Pascoal Faria. 2020. DCO Analyzer: Local Controllability and Observability Analysis and Enforcement of Distributed Test Scenarios. In 42nd International Conference on Software Engineering Companion (ICSE '20 Companion), 2020.

  This paper is about DCO Analyzer, the first tool that checks if distributed test scenarios specified by means of UML sequence diagrams are locally observable and locally controllable, and automatically determines a minimum number of coordination messages to enforce these properties. Part of Chapter 6 of the thesis is based on this paper.

- Lima B, Faria JP, Hierons R. Local observability and controllability analysis and enforcement in distributed testing with time constraints. IEEE Access. 2020 Aug 17.

  This journal paper summarizes the work developed in this thesis and presents the main results. Parts of the Chapters 5, 6 and 7 of the thesis are based on this paper.

1.4 Document Structure

In addition to this chapter, this document is divided into seven more chapters:

- **Chapter 2** presents the state of the art on distributed systems testing. This state of the art analysis was carried out according to five different topics, namely:
  - Model-based testing for distributed systems
  - Testing architectures for distributed systems
  - Observability and controllability in distributed systems testing
  - Time constraints in distributed systems testing
  - Testing infrastructures for distributed systems testing

  This allowed us to identify existing gaps and focus our work on areas of research that were less explored.

- **Chapter 3** describes the analysis of the state of practice in distributed and heterogeneous systems testing from the perspective of practitioners in order to understand what the real difficulties and needs of companies that are testing these types of systems.

- **Chapter 4** presents our proposed hybrid architecture and overall approach for testing distributed systems.

- **Chapter 5** proposes novel local observability and controllability analysis and enforcement algorithms.

- **Chapter 6** describes DCO Analyzer, a novel controllability and observability analysis and enforcement tool for time-constrained distributed systems testing.

- **Chapter 7** presents the validation carried out not only through a set of test cases produced to cover various situations, as well as a business case.

- **Chapter 8** concludes this research work.

Chapter 2

Background and State of the Art

This chapter introduces the concepts and terminologies used in this thesis and presents a survey of the state of the art on time-constrained distributed systems testing. The research carried out was based on a systematic research seeking not only to find answers to the RQ1 and RQ2 presented in Chapter 1, but also to find principles and foundations that could be applied in the development of the work described in this thesis.

This chapter is organized as follows. Section 2.1 presents some software testing concepts; Model-based testing is described in section 2.2. Testing frameworks for distributed systems are presenting in Section 2.3; Section 2.4 presents the test architectures for distributed systems testing; The observability and controllability problems in distributed systems testing are described in Section 2.5; Other testability issues in distributed systems testing are presented in Section 2.6; Section 2.7 analyses the time constraints problems in distributed systems testing; Section 2.8 summarizes the main characteristics and features covered by the related work analyzed in this chapter and answers the RQ1 and RQ2 presented in Chapter 1.

2.1 Software Testing Concepts

Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended. Software should be predictable and consistent, offering no surprises to users (Myers et al., 2004). In this section it is presented some background on software testing concepts and terminology used in this chapter and throughout the thesis.

2.1.1 Test Levels

There are different levels during the software testing process (Beizer, 2003). Typically the levels considered are: unit testing, integration testing, system testing and acceptance testing.

Unit testing is the testing of individual hardware or software units or groups of related units (IEEE, 1990). The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.

In the case of a distributed and heterogeneous system, comprising a set of interconnected components running on different machines or execution environments, unit testing usually refers to the testing of individual components. In automated testing, if a component under test calls other components, such components need to be simulated by test stubs (see definition of test stub in the next section).

Integration testing is the testing in which software and/or hardware components are combined and tested to evaluate the interaction between them (IEEE, 1990).

In the case of integration testing of a distributed and heterogeneous system, besides checking the interactions of system components with the environment (users or external systems), it is also useful to check the interactions between components of the system, to improve fault detection and localization. Checking such interactions may be challenging, because of observability limitations.

System testing is the testing conducted on a complete, integrated system to evaluate the system's compliance with specified requirements (IEEE, 1990). System testing is concerned mainly with testing the interactions of the system with the environment (users or external systems), and evaluating extra-functional properties.

In the case of a distributed and heterogeneous system, interactions with the environment (users or external systems) typically occur at multiple locations. In automated testing, coordinating the test components that simulate those users or external systems is specially challenging, because of their distributed nature.

Acceptance testing is the formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable a customer, a user, or other authorized entity to determine whether or not to accept the system (IEEE, 1990). The challenges of acceptance testing in the context of distributed and heterogeneous systems are similar to those applied to system testing.

2.1.2 Test Case Design Strategies

Software testing methods are traditionally divided into white-box testing (Ostrand, 2002), gray-box testing (Linzhang et al., 2004) and black-box testing (Edwards, 2001).

In white-box testing, the internals of the system under test (SUT) are all visible. As a consequence, the knowledge about these internal matters can be used to create tests. Furthermore, white-box testing is not restricted to the detection of failures, but is also able to detect errors. Failures occur when there exist discrepancies between the specification and the behavior of the system (Mills et al., 1987). Errors are an incorrect internal state that is the manifestation of some fault that occur while running the test (Ammann and Offutt, 2016). Advantages are tests of higher quality because of the knowledge about system internals; however, there is also a big disadvantage, because looking into all aspects of a program requires a high effort.

In black-box testing, the SUT internal content is hidden from the tester. The tester only has knowledge about possible input and output values. The black-box testing only allows to test input-output functionality (functional testing). As an advantage, this technique is close to realistic conditions. One important disadvantage is the lack of internal information, which could be useful

to generate tests, because sometimes it is necessary to know the content to test boundary values, that are normally responsible for failures.

In gray-box testing the advantages of both previous techniques are combined. The test design is realized at white-box level but the tests are executed at black-box level. For the tester, this has the advantage of having access to the SUT internal information while designing tests; however, the tests are executed under realistic conditions, where only failures are detected. Gray-box testing techniques are used for commercial model-based testing (MBT), where, e.g., the test model contains information about the internal structure of the SUT, but the SUT internal matters themselves are not accessible (e.g. for reasons of non-disclosure).

2.1.3 Test Automation

Several testing activities can be automated, with varying costs and benefits (Ramler and Wolfmaier, 2006).

The testing activity that is most commonly automated is test execution. This requires that test cases are implemented as executable test scripts, test classes, etc.

To support the automation of the test activities the test harness may also have to be developed. In software testing, a test harness is a collection of software and test data configured to test a program unit by running it under varying conditions and monitoring its behavior and outputs.

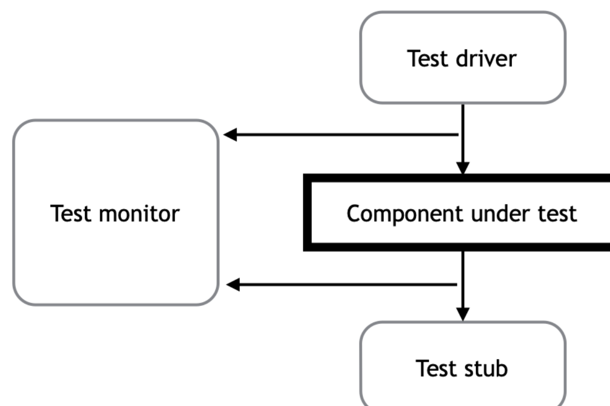


Figure 2.1: Test harness

As shown in the Figure 2.1, there are typically three main types of test components in a test harness: test driver, test stub and test monitor. The test drivers are responsible for calling the target code, simulating calling units or a user. In automatic testing they are also responsible for the implementation of test cases and procedures. The test stubs simulate modules, units or systems called by the target code; normally mock objects are used for this purpose. The test monitor is responsible to collect all the informations (or messages) sent and received by the component under test. This information is important for fault detection and diagnosis.

Automatic test execution is important to reduce the cost of regression testing, support iterative development approaches, enable load and performance testing, and avoid human errors that are common in manual test execution, among other reasons.

In the case of distributed and heterogeneous systems, automated test execution is specially challenging, because of the need to support multiple platforms and the need to coordinate the injection of test inputs and the monitoring of test outputs at distributed points, and very few testing frameworks exist for such systems (Lima and Faria, 2016).

Test coverage analysis is another testing activity that is commonly automated, usually in connection with automated test execution. Test coverage analysis is specially important in white-box testing, to determine parts of the code that are not being properly exercised, and help identifying additional test cases for increasing coverage.

Test case generation is usually performed manually. However, model-based testing (MBT) methods and tools have attracted increasing attention from industry, because of the ability to automatically generate test cases from system models. Based on behavioral models of the SUT, MBT tools are able to generate *abstract test cases*, which can be subsequently translated into *concrete test cases* ready for execution, based on mapping information between the model and the implementation.

In the case of distributed and heterogeneous systems, automated test case generation is specially challenging, because of the difficulty of modeling several characteristics inherent to such systems, such as timing aspects, concurrency aspects, and non-determinism, among other features, with very limited support provided by current MBT tools.

2.2 Model-based Testing

2.2.1 Model-based Testing Process

Model-based testing (MBT) usually means functional testing for which the test specification is given as a test model. The test model is derived from the system requirements. In model-based testing, test cases can be derived (semi-)automatically from the test model. Coverage criteria are often considered at the test model level. After we have the test cases definition, with some extra mapping information it is possible implement the test cases to obtain executable tests. With the executable tests we are prepared to run the test cases on the SUT. In the end of this process we obtain not only the errors but also the coverage (which part or interactions of the SUT was tested in each test). Figure 2.2 represents a typical MBT test process.

2.2.2 Test Levels

MBT can be applied to all levels from unit to system. Acceptance tests are usually not covered because user acceptance often also depends on many imprecise expectations. Figure 2.3 shows the application fields of model-based testing.

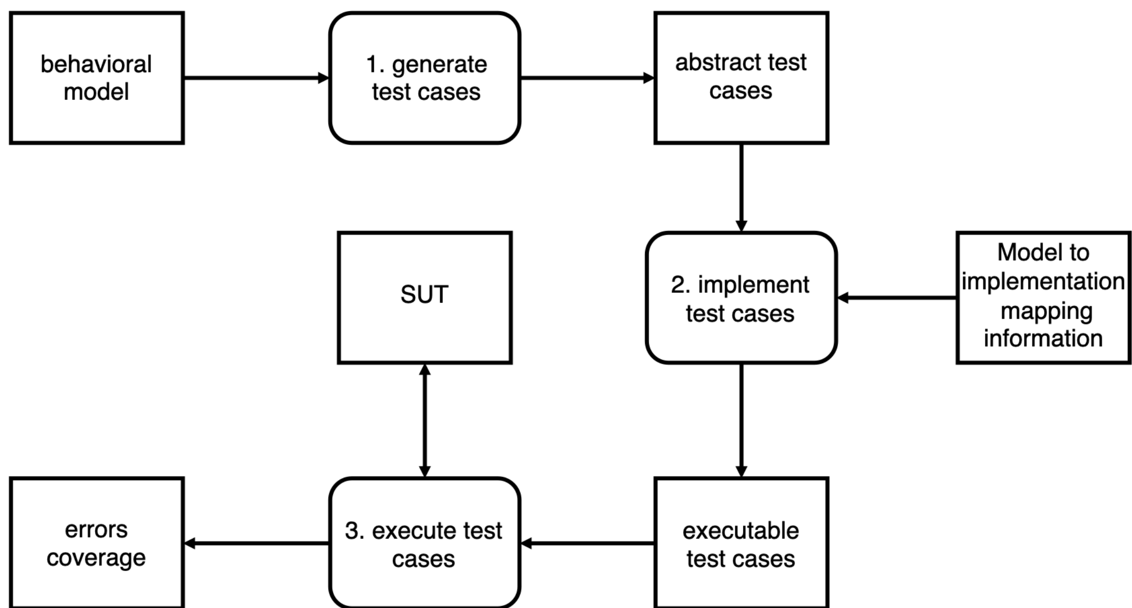


Figure 2.2: Typical MBT test process (UML activity diagram)

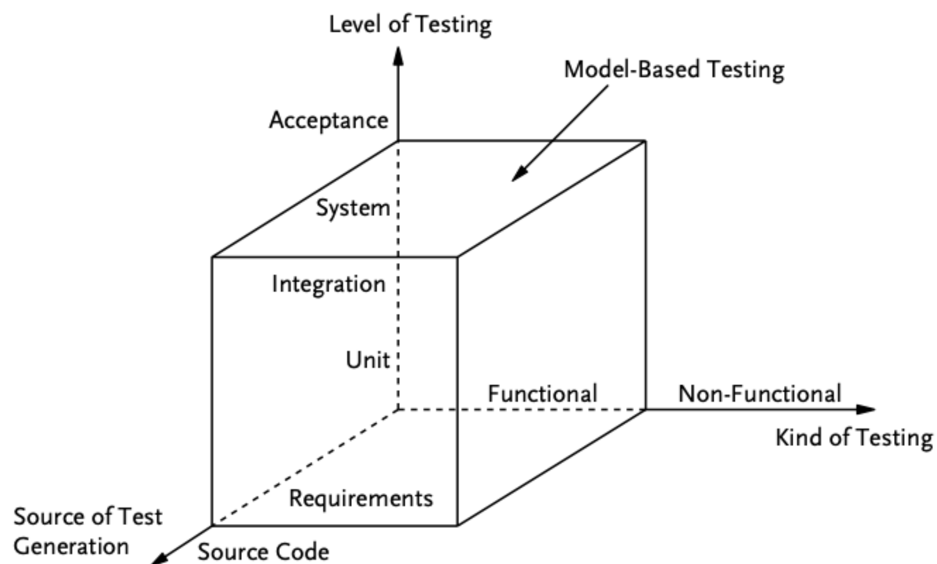


Figure 2.3: Application fields of model-based testing (Weißleder, 2010)

2.2.3 Test Generation

One of the most important characteristic of the MBT is automation. Given the test model and some test case specifications, test cases can be derived stochastically, or by using dedicated graph search algorithms and search-based techniques, model checking, symbolic execution, deductive theorem proving, or constraint solving (Utting et al., 2012).

Random generation of tests is performed by sampling the input space of a system. It is

straightforward to implement, but it takes an undefined period of time to reach a certain satisfying level of model coverage (Zander et al., 2017).

Search-based algorithms for model-based test generation include graph search algorithms such as node or arc coverage algorithms (e.g. the Chinese Postman algorithm, which covers each arc at least once), as well as other search-based algorithms such as metaheuristic search, evolutionary algorithms (e.g. genetic algorithms), and simulated annealing (Utting et al., 2012).

(Bounded) model checking is a technology for verifying or falsifying properties of a system. For certain classes of properties, model checkers can yield counter examples when a property is not satisfied. The general idea of the test case generation with model checkers is to first formulate test case specification as reachability properties, for instance, eventually, a certain state is reached, or a certain 'transition fires'. A model checker then, by searching for counter examples for the negation of the property, yields traces that reach the given state or that eventually make the transition fire. Other variants use mutations of models or properties to generate test suites (Utting et al., 2012).

Symbolic execution runs an (executable) model not with single input values but with sets of input values. These are represented as constraints. In this way, symbolic traces are generated: one symbolic trace represents many fully instantiated traces. The instantiation to concrete values must obviously be performed in order to get test cases for a SUT. Symbolic execution is guided by test case specifications. Often enough, these boil down to reachability statements as in the case of model checking. In other cases, test case specifications are given as explicit constraints, and the symbolic execution is guided by having to respect these constraints (Utting et al., 2012).

Deductive theorem proving can also be used for the generation of tests, particularly with provers that support the generation of witness traces or counterexamples. One variant is similar to the use of model checkers where a theorem prover replaces the model checker. Most often, however, theorem provers are used to check the satisfiability of formulas that directly occur as guards of transitions in state-based models. A theorem prover can compute assignments for the variables that occur in the guards and that, in turn, give rise to values of the respective input and output signals. A sequence of such sets of signals then becomes the test case (Utting et al., 2012).

Constraint solving is useful for selecting data values from complex data domains, e.g. in combinatorial n-wise testing. It is also often used in conjunction with other methods such as symbolic execution, graph search algorithms, model-checking or theorem proving where specific relationships between variables in guards or conditions are expressed as constraints and efficiently solved by dedicated constraint solvers (Utting et al., 2012).

In most cases the test tools use a combination of the previous technologies to generate test cases.

2.2.4 Test Execution

Regarding test execution, MBT approaches can be done offline or online (adaptive).

In offline testing, test generation and execution occur as separate phases, test cases are first generated and can then be executed (Schulz et al., 2007). The advantages of offline testing, when applicable, are directly connected to the generation of test repository. The tests repository can be

managed and executed using existing test management tools (e.g. TestLink (TestLink Development Team, 2020), MBTSuite (sepp.med gmbh, 2020)), which means that fewer changes to the test process are required. Tests can be generated and then executed multiple times on the SUT. The test generation and execution can be performed on different machines or environments and at different times (Utting et al., 2012).

In online testing, test generation and execution are intermixed, so that the test generation algorithms can react to how the SUT behaves (Mikucionis et al., 2004). This is sometimes necessary if the SUT is non-deterministic, because the test generator can see which path the SUT has taken, and follow the same path in the model (Utting et al., 2012).

2.2.5 Input Models

Regarding the input models there are two distinct approaches, state-based and scenario-based.

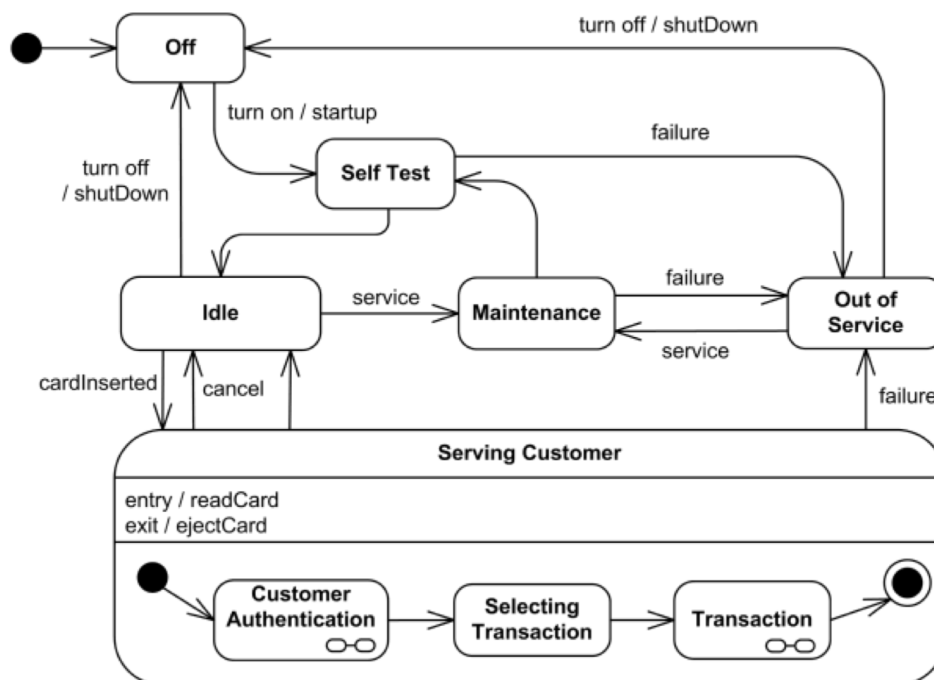


Figure 2.4: UML behavioral state machine diagram (Fakhroutdinov, 2013)

In state-based approaches UML state machines (Lilius and Paltor, 1999) or similar models (Veanes et al., 2008; Tani and Petrenko, 2013) are used for describing all possible behaviors of the SUT or its components.

When testing from a state-based model, there are several test strategies such as state coverage, transition coverage and path coverage that require the generation of a set of feasible test paths and associated constraints in order to produce a test suite. However, the automatic generation of test data is not simple due to the presence of infeasible paths. A given transition path can be infeasible due to the variable interdependencies among the actions and conditions. If an infeasible path is

chosen to exercise certain transitions, these transitions are not exercised. Problems arising from the existence of infeasible paths are generally undecidable. In addition, feasible transition paths are subject to different levels of traversal complexities; it can be hard to find a set of test data that can trigger a given feasible test path (Kalaji et al., 2009; Derderian et al., 2010; Rao et al., 2016).

Figure 2.4 presents a UML behavioral state machine diagram showing the top-level state machine for a Bank Automated Teller Machine (ATM). This diagram represents the states that the ATM can remain in as well as the valid transitions between those states.

In scenario-based approaches, UML SDs (Javed et al., 2007), message sequence charts (MSC) (Damm and Harel, 2001) or similar models (Grieskamp, 2006b) are used for describing interactions between the system components or with the environment in key scenarios, minimizing test case explosion (Grieskamp, 2006a).

When testing from scenario-based models the generation of test data is simpler, since the scenario can, in the end, represent a single test path simplifying the process of exercising certain actions. However, to test the entire system we will have to test all test scenarios, which implies an increase in the number of models or the creation of more complex models, with control flow variants.

Figure 2.5 presents a UML SD of an ATM usage scenario. In this scenario, interactions between the ATM, the user and the card issuing bank are represented in the cash withdrawal scenario.

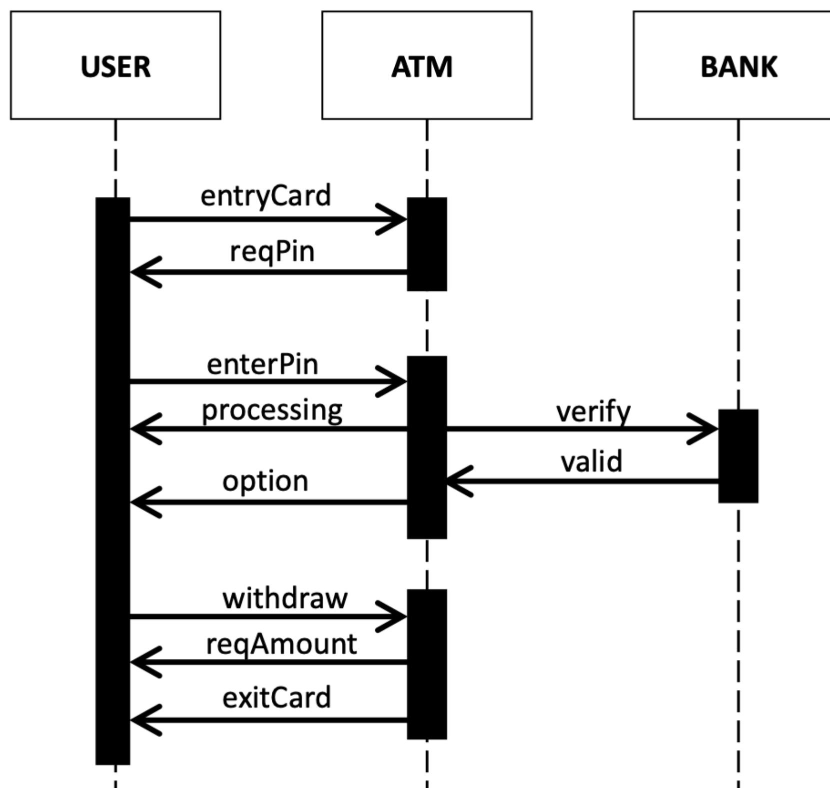


Figure 2.5: UML SD diagram of a cash withdrawal scenario at an ATM.

If we compare the two models present in Figures 2.4 and 2.5, we can see that the scenario represented in Figure 2.5 is just one of the possible operations that can be performed in the "Serving Customer" state represented at the bottom of Figure 2.4.

2.2.6 MBT Approaches for Integration Testing

Taking advantage of the systematic review prepared by (Dias Neto et al., 2007), we analyzed more than 100 MBT approaches. As some of the principles used in MBT approaches for integration testing can be applied in testing distributed systems, we have identified 25 whose scope is the integration testing, which are presented in Table 2.1.

To analyze which principles of these approaches could be applicable in our context, we analyzed each of them according to four different criteria. First, if the approach uses an industry standard input model, second if the entry model is a behavioral model, third, if they support the characteristics of distributed systems (time constraints, concurrency, no global clock) and, finally, if the approaches have support tools. The result of this analysis can be seen in Table 2.2.

Another limitation that we found is the lack of tool support; most MBT approaches don't have tools to support the whole test process. Even when there are tools to support the whole test process, several tools are needed with different input/output data formats which require the use of manual conversion steps that limit the usage of the MBT approaches.

2.3 Testing Frameworks for Distributed Systems

Regarding test concretization and execution for distributed systems, we found in the literature several frameworks that can be adapted and integrated for building a comprehensive test automation solution.

The Software Testing Automation Framework (STAF) (STAF, 2014) is an open source, multi-platform, multi-language framework designed around the idea of reusable components, called services (such as process invocation, resource management, logging, and monitoring). STAF removes the tedium of building an automation infrastructure, thus enabling the tester to focus on building an automation solution. The STAF framework provides the foundation upon which to build higher level solutions, and provides a pluggable approach supported across a large variety of platforms and languages.

Torens and Ebrecht (Torens and Ebrecht, 2010) proposed the RemoteTest framework as a solution for the testing of distributed systems and their interfaces. In this framework, the individual system components are integrated into a virtual environment that emulates the adjacent modules of the system. The interface details are thereby abstracted by the framework and there is no special interface knowledge necessary by the tester. In addition to the decoupling of components and interface abstraction, the RemoteTest framework facilitates the testing of distributed systems with flexible mechanisms to write test scripts and an architecture that can be easily adapted to different systems.

Table 2.1: MBT approaches supporting integration testing

Authors	Input Models	Testing Level
Ali et al. (2007)	UML State Diagram, Collaboration Diagram and SCOTEM	Integration Testing
Babić (2014)	Simulink models	Integration Testing
Basanieri and Bertolino (2000)	Use Case, Sequence and Class Diagrams	Integration Testing
Benz (2007)	Concurtasktrees (CTT)	Integration Testing
Bertolino et al. (2003)	Use Case, Sequence and Class Diagrams	Integration Testing
Bertolino et al. (2005)	UML Sequence and State Diagrams	Integration Testing
Beyer et al. (2003)	Annotated Sequence Diagram and MCUM	Integration Testing
Briand et al. (2001)	Class Diagram and Graph	Integration Testing
Chen et al. (2005)	Condition Data Flow Diagram	Integration Testing
Dai et al. (2004)	UML Interaction, Activity and State Diagrams	Integration Testing
Efkemann (2014)	ITML-B and ITML-A	Integration and System Testing
Gross et al. (2005)	U2TP and TTCN-3	Integration Testing
Hartmann et al. (2000)	Statechart Diagram and Graph	Integration and Unit Testing
Helle and Schamai (2014)	UML Testing Profile	Integration and System Testing
Nieminen and Raty (2015)	UML and state machine models	Integration and System Testing
Polgár et al. (2009)	Platform-specific models, workflow	Integration Testing
Reis et al. (2007)	UML Activity diagrams	Integration Testing
Richardson and Wolf (1996)	CHAM	Integration Testing
Scheetz et al. (1999)	UML Class and State Diagram	Integration Testing
Sinha and Smidts (2006)	WSDL-S, EFSM	Integration Testing
Sokenou et al. (2006)	UML Sequence and State Diagram	Integration and Unit Testing
Tretmans (2008)	Input/Output Automata	Integration and System Testing
Wieczorek et al. (2009)	Event-B models	Integration Testing
Wu et al. (2003)	UML Collaboration/Sequence or Statechart Diagram	Integration Testing
Xu and Xu (2006)	State Model (AOP)	Integration Testing

Zhang et al. (Zhang et al., 2009) developed a runtime monitoring tool called FiLM that can monitor the execution of distributed applications against labelled transition systems (LTL) specifications on finite traces. Implemented within the online predicate checking infrastructure D^3S (Liu et al., 2008), FiLM models the execution of distributed applications as a trace of consistent global snapshots with global timestamps, and it employs finite automata constructed from LTL specifications to evaluate the traces of distributed systems.

Table 2.2: Comparison of MBT approaches

Authors	**Industry standard input model**	**Behavioral input model**	**Time constraints, concurrency, no global clock**	**Tool Support**
Ali et al. (2007)	✓	✓	⊗	✓
Babić (2014)	⊗	⊗	⊗	✓
Basanieri and Bertolino (2000)	✓	⊗	⊗	⊗
Benz (2007)	⊗	⊗	⊗	✓
Bertolino et al. (2003)	✓	✓	⊗	✓
Bertolino et al. (2005)	✓	⚠	⊗	✓
Beyer et al. (2003)	✓	✓	⚠	✓
Briand et al. (2001)	⊗	⊗	⊗	⊗
Chen et al. (2005)	⊗	✓	⊗	⊗
Dai et al. (2004)	✓	⚠	⚠	⊗
Efkemann (2014)	⊗	⚠	✓	✓
Gross et al. (2005)	⚠	⚠	⊗	⊗
Hartmann et al. (2000)	✓	✓	⊗	✓
Helle and Schamai (2014)	✓	✓	⊗	✓
Nieminen and Raty (2015)	⚠	⚠	⊗	✓
Polgár et al. (2009)	⊗	⊗	⊗	✓
Reis et al. (2007)	✓	✓	⊗	⊗
Richardson and Wolf (1996)	⊗	⊗	⊗	⊗
Scheetz et al. (1999)	✓	⚠	⊗	✓
Sinha and Smidts (2006)	⊗	⊗	⊗	⊗
Sokenou et al. (2006)	✓	✓	⊗	⊗
Tretmans (2008)	⊗	⊗	⊗	✓
Wieczorek et al. (2009)	⊗	⊗	⊗	✓
Wu et al. (2003)	✓	✓	⊗	⊗
Xu and Xu (2006)	✓	✓	⊗	⊗
✓ - Yes, ⊗ - No, ⚠ - Partially				

Camini et al. (Canini et al., 2011) proposed DiCE, an approach that continuously and automatically explores the system behavior, to check whether the system deviates from its desired behavior. At a highlevel DiCE (i) creates a snapshot consisting of lightweight node checkpoints, (ii) orchestrates the exploration of relevant system behaviors across the snapshot by subjecting

system nodes to many possible inputs that exercise node actions, and (iii) checks for violations of properties that capture the desired system behavior. DiCE starts exploring from current system state, and operates alongside the deployed system but in isolation from it. In this way, testing can account for the current code, state and configuration of the system. DiCE reuses existing protocol messages to the extent possible for interoperability and ease of deployment.

2.4 Test Architectures for Distributed Systems Testing

In order to test distributed systems three test architectures have been proposed, with different conformance relations and fault detection capabilities:

- a purely distributed test architecture with independent local testers communicating synchronously with the SUT components (Ulrich and König, 1999);
- a purely centralized test architecture, in which a single central tester interacts asynchronously with the SUT components (Hierons, 2014);
- a hybrid test architecture that combines local testers and a central tester (Hierons, 2014).

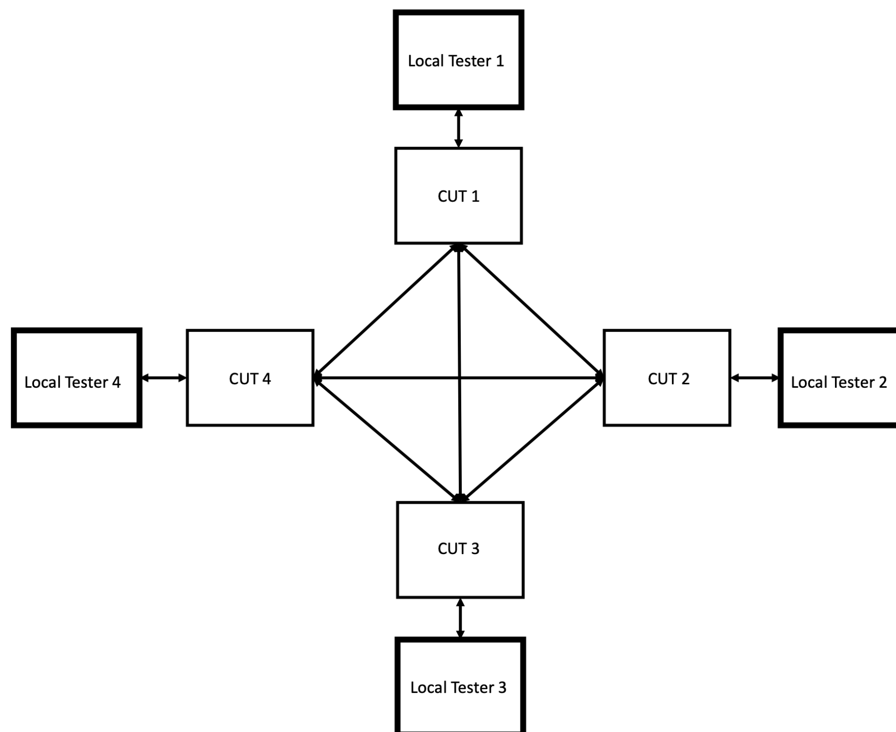


Figure 2.6: A purely distributed test architecture.

A purely distributed test architecture with independent local testers communicating synchronously with the SUT components is shown in Figure 2.6. In this case the system consists of 4

components that communicate with each other. In this type of architecture, a local tester is placed next to each component. The problem with this approach is the lack of coordination between local testers, which limits the ability to detect failures.

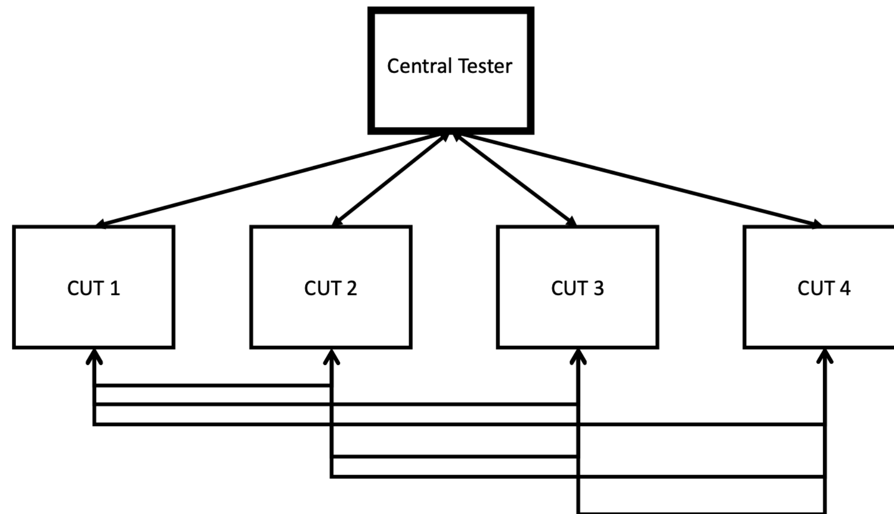


Figure 2.7: A purely centralized test architecture.

A purely centralized test architecture, in which a single central tester interacts asynchronously with the SUT components is shown in Figure 2.7. In this type of architecture and for the same system described above, there is only one central tester. The drawback of this approach is the communication overhead generated and the delays that this can cause since all the next inputs have to be validated by the central tester that is monitoring the messages exchanged by all components. If the system to be tested has time constraints, some errors can occur simply due to the delay caused by the central tester.

A hybrid test architecture that combines local testers and a central tester to achieve a higher fault detection capability is shown in Figure 2.8. In this hybrid approach, the central tester is responsible for deciding and sending test inputs to the SUT components, and local testers are responsible for observing the events (inputs and outputs) at each location; the SUT outputs are observed by the local testers and sent to the central tester. This way, the local testers are able to detect conformance faults associated with an incorrect combination or an incorrect ordering of events occurring in the same location, whilst the central tester is able to detect conformance faults associated with an incorrect combination of events or an incorrect ordering of pairs of input and output events occurring at different locations (e.g., an SUT output that is prematurely produced at one location before an input is injected at another location).

In our approach, we further decentralize test input generation and injection, minimizing the messages exchanged between the test components during test execution and increasing the responsiveness of the test harness, whilst keeping the same fault detection capability. Another difference

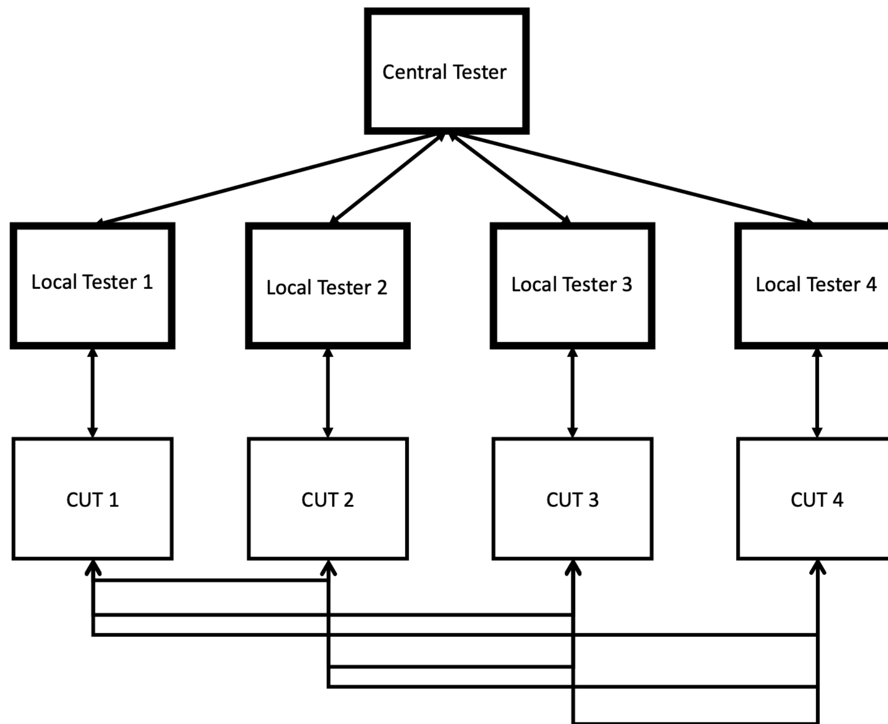


Figure 2.8: A hybrid test architecture.

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

2.5 Observability and Controllability in Distributed Systems Testing

One difficulty in distributed systems testing is observability, because communication delays and the lack of a global clock limit the conformance faults detectable. Another difficulty is controllability, i.e., the difficulty for the local testers to decide when and what test inputs to inject, without causing global conformance faults (e.g., in the presence of race conditions or non-local choices). Solutions proposed in the literature are based on the insertion of coordination messages between test components (Mitchell, 2005; Hierons, 2012; Boroday et al., 2009), but they do not handle timing constraints and, in most of the cases, they address only the "when" and not the "what" aspect (i.e., they don't consider control flow variants).

In (Mitchell, 2005), the author discusses the problems related to race conditions in scenarios described through MSCs or UML SDs, and presents solutions to these problems. The focus of their work is on analyzing scenario-based requirements specifications, but such scenarios can also be used for testing purposes. However, only basic scenarios are considered, without control flow variants and timing constraints.

In (Hierons, 2012), the author investigates the use of coordination messages to overcome controllability problems when testing from an input/output transition system (IOTS) and give an algorithm for introducing sufficient messages. The algorithm operates by identifying all of the controllability problems, and then resolving these one at a time. The author also characterizes the types of controllability problems that cannot be solved this way, and introduces the notion of *strongly uncontrollable* test cases. The author also proves that the problem of minimizing the number of coordination messages used is NP-hard. However, the approach is focused on system testing only and not integration testing, i.e., the messages exchanged between the system components are not considered (the observation of these messages by the local testers may reduce the need for introducing coordination messages). Other differences with our work are that they do not consider timing constraints, and assume that test inputs are deterministic (which we do not require).

In (Boroday et al., 2009), the authors propose algorithms to extend test scenarios for distributed systems represented by MSCs or UML SDs, in order to obtain race-free scenarios suitable for test implementation, by inserting coordination messages between test components and quiescence observation events (based on timeout events) in each test component. However, in their work, only the interactions with the environment are modeled, and they do not consider control flow variants and time constraints.

A common limitation of the above works (except (Mitchell, 2005)) is that they only consider the messages exchanged with the environment (system testing perspective), represented by a single input or output event, and not the messages exchanged between the system components (integration testing perspective), that need to be represented by pairs of send and receive events.

2.6 Other Testability Issues in Distributed Systems Testing

When testing a distributed system, it is sometimes necessary to test a running/deployed system (*runtime validation*), the additional challenge being that testing should not interfere with system use. In runtime validation, a component of a system is said to be *testable* if it has a separate test interface whose use reduces the potential for interference. *Isolation methods* have been proposed for components that are not testable. There is a line of work in which approaches to runtime validation have been developed using Testing and Test Control Notation Language Version 3 (TTCN-3) in order to enhance applicability (Lahami et al., 2012a). The proposed approach (TT4RT) includes a test management layer and a test isolation layer. A further development aimed to optimise the placement of test components that interact with system components, with this taking into account resource availability and network connectivity (Lahami et al., 2012b). It has also been noted that a system might have some components that are testable and some that are not, with a procedure being proposed to choose the appropriate test isolation approaches (Lahami and Krichen, 2013). The focus of this line of work is on the execution of abstract test cases that have already been provided, and it does not address coordination issues. However, there is potential for runtime

validation approaches to be integrated with techniques, such as those described in this thesis, that analyse test case and address coordination problems where they exist.

2.7 Time Constraints in Distributed Systems Testing

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

In (Zheng et al., 2002), the authors derive the valid traces for Timed Message Sequence Charts (T-MSCs), similar to UML SDs, but do not address the problem of conformance checking based on distributed observations. Timed traces are represented by incorporating special time events between normal events.

In (Akshay et al., 2007), the authors present a timed model of communicating finite-state machines, which communicate by exchanging messages through channels and use event clocks to generate collections of T-MSCs. In a more recent work (Akshay et al., 2015), the authors address model checking message-passing systems with real-time requirements. As behavioral specifications, they use TC-MSCs (time-constrained MSCs), in which lower and upper bounds on the time interval between certain pairs of events are added to plain MSCs. As system model, they use a network of communicating finite state machines with local clocks, whose global behavior can be regarded as a timed automaton. Their goal is to verify (by model checking) that all timed behaviors exhibited by the system model conform to the timing constraints imposed by the specification, and not to check the conformity of the implementation with the specification or system model.

In (Hierons et al., 2012), the authors derive conformance relations taking into account the event timestamps obtained with the local clocks present at each system port (point of interaction with the environment), assumed to differ up to a maximum clock skew, but only for system testing.

In (Gaston et al., 2013), the authors show that conformance checking in the presence of time constraints, within a distributed test architecture without a global clock, can be done in two phases: in the first phase, each local tester checks local conformance according to the *tioco* conformance relation; in the second phase, the local traces are brought together and it is checked if events are exchanged following some communication rules. Their results do not apply directly to UML SDs (OMG, 2017), since they assume internal multicast communications, among other differences.

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

The only previous work we found that relates the issue of observability and controllability to time constraints is (Khoumsi, 2002). In this article, the author demonstrates how to solve the problems of observability and controllability using coordination messages and time constraints. However, they do not support timing constraints or non-determinism in the input models, only consider interactions with the environment, and restrict their attention to SUT behaviors consisting of alternating sequences of inputs from the environment and outputs to the environment. In a more recent work (Azzouzi et al., 2020), the authors propose to solve controllability problems using so

called synchronization messages, for the same type of input models, but do not support timing constraints either in the input model.

2.8 Summary

Table 2.3 summarizes the main characteristics and features covered by the related work previously analyzed. Although some works address observability and controllability problems in distributed systems testing and design, none addresses the problem of observability and controllability analysis and enforcement for time-constrained distributed systems.

Regarding the research questions presented in Chapter 1, we can partially answer RQ1 and RQ2 as follows:

RQ1 - *What are the main difficulties and needs in the integration testing of distributed systems listed in the state of the art and state of practice?*

Looking at the state of the art, we can say that there is a lack of studies that solve the problem of observability and controllability analysis and enforcement for time-constrained distributed systems. This is a critical problem in this type of systems. However, the principles adopted by the approaches present in Table 2.3, namely at the level of solutions to the observability and controllability problems in distributed systems testing and design are a good starting point for this thesis.

RQ2 - *What is an adequate architecture and approach to conduct integration tests in these types of systems?*

Looking at the analysis performed, the hybrid test architecture that combines local testers and the central tester is one that appears to be the architecture that best suits this type of system, namely due to its failure detection capacity. For this reason it will be our starting point in the proposed solution for the integration testing in this type of systems.

Table 2.3: Summary of comparison of related work in distributed systems testing.

	Ulrich and König (1999)	Hierons (2014)	Mitchell (2005)	Hierons (2012)	Boroday et al. (2009)	Lahami et al. (2012a,b); Lahami and Krichen (2013)	Zheng et al. (2002)	Akshay et al. (2007)	Akshay et al. (2015)	Hierons et al. (2012)	Gaston et al. (2013)	Khoumsi (2002)	Azzouzi et al. (2020)
Scenario-based input model	-	-	X	-	X	X	X	X	X	-	-	-	-
Distributed test architecture	X	X	-	X	X	X	-	-	-	X	X	X	X
Internal interaction checking	X	-	X	-	-	X	X	X	X	-	X	-	-
Control-flow variants	X	X	-	X	-	X	X	X	X	X	X	X	X
Timing constraints	-	-	-	-	-	X	X	X	X	-	X	-	-
Non-determinism	-	X	-	X	-	X	X	X	X	X	X	-	-
Conformance checking	-	-	-	X	-	-	-	-	X	X	X	-	-
Observability analysis	-	-	-	-	-	-	-	-	-	-	-	X	X
Controllability analysis	-	-	X	X	X	-	-	-	-	-	-	X	X
Observability enforcement	-	-	-	-	-	-	-	-	-	-	-	X	X
Controllability enforcement	-	-	X	X	X	-	-	-	-	-	-	X	X
Tool implementation	-	-	-	-	-	X	-	-	-	-	-	-	-
Real-world case studies	-	-	X	-	-	X	-	-	-	-	-	-	-

Chapter 3

State of the Practice on Testing Distributed and Heterogeneous Systems

To explore the testing of distributed and heterogeneous systems (DHS) from the point of view of industry practitioners, in order to assess the current state of the practice and identify opportunities and priorities for research and innovation initiatives we conducted an survey and follow-up interviews with some survey respondents. The method, results and conclusions of this survey are presented in this chapter. Section 3.1 describe the research method and scope. Results are presented in Section 3.2. Section 3.3 presents a discussion of the results obtained from the survey. In order to analyze the drivers and barriers for DHS test automation in companies, some follow-up interviews with some survey respondents are present in Section 3.4. Section 3.5 point out other related surveys in this subject. Section 3.6 summarizes the main conclusions and answers the RQ1 presented in Chapter 1.

3.1 Research Method and Scope

The research method used in this work is the explanatory survey. Explanatory surveys aim at making explanatory claims about the population. For example, when studying how developers use a certain inspection technique (Wohlin et al., 2003).

3.1.1 Goal

The main goal of this survey is to explore the testing of DHS from the point of view of industry practitioners, in order to assess the current state of the practice and identify opportunities and priorities for research and innovation initiatives.

More precisely, we aim at responding to the following research questions:

- **SRQ1**: How relevant are DHS in the software testing practice?
- **SRQ2**: What are the most important features to be tested in DHS?

- **SRQ3**: What is the current status of test automation and tool sourcing for testing DHS?
- **SRQ4**: What are the most desired features in test automation solutions for DHS?

3.1.2 Survey Distribution and Sampling

Since our main goal was to collect the point of view of industry practitioners that were involved in the testing of DHS, we shared the survey to the participants of two industry-oriented conferences in the software testing area: TESTING Portugal 2015[1] and User Conference on Advanced Automated Testing (UCAAT) 2015[2]. In total we distributed 250 surveys and we obtained 167 answers. From these 167 answers, only 147 were complete and valid. Most of the invalid answers were related with respondents that did not complete the survey.

3.1.3 Survey Organization

The survey was composed of two main parts. The first part was an introduction, where we explained the goal of the survey and define the term "Distributed and Heterogeneous Systems". In the context of this survey we define a Distributed and Heterogeneous System as a set of small independent systems that together form a new distributed system, combining hardware components and software systems, possibly involving mobile and cloud-based platforms.

The second part of the questions is divided in three different groups. The first group is related with the professional characterization of the participants. The second group contains questions about the company characterization. The last group contains the questions related with the testing of DHS and the main research questions underlying the survey. The survey questionary can be found in Appendix A.

3.2 Results

3.2.1 Participants Characterization

Before drawing conclusions on the main questions of this survey it is important to realize the profile of the survey participants. The results show that most of the people (70%) that responded this survey work in software testing, verification & validation and 41% are in the current position for more than five years (see Figure 3.1).

Regarding the experience in software testing, the results show (see Figure 3.2) that the majority of the survey participants have more than 5 years of experience in software testing in general and 40% have more than 5 years of experience with DHS.

[1] https://testingportugal.pstqb.pt/2015/
[2] https://ucaat.etsi.org/2015/

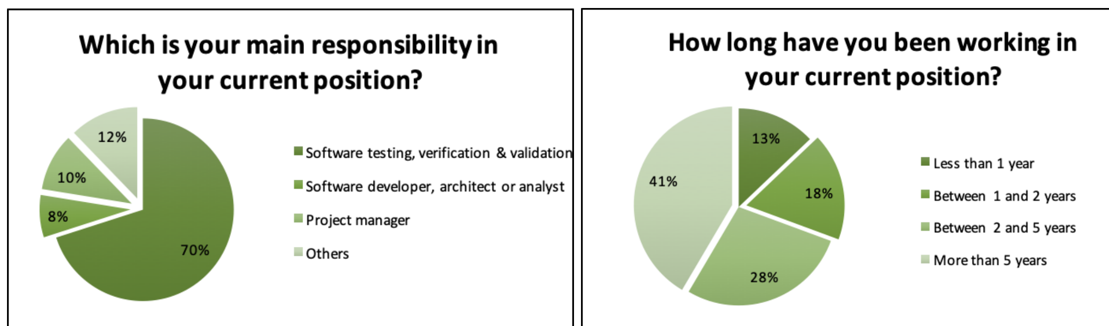


Figure 3.1: Current Position and Time in Current Position.

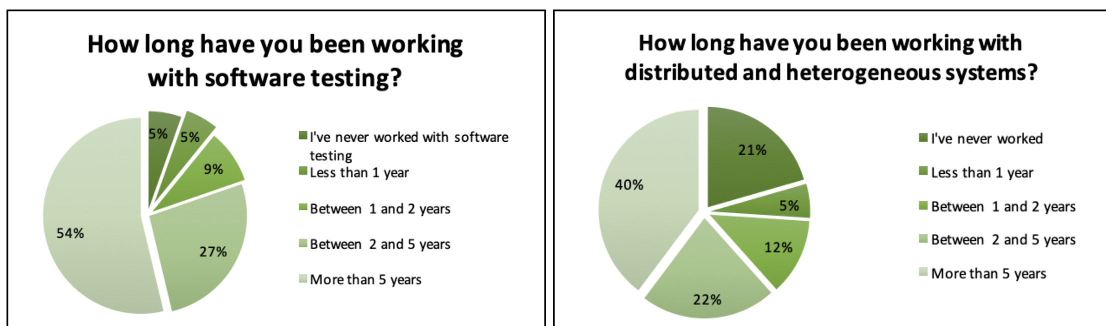


Figure 3.2: Time in Software Testing.

3.2.2 Company Characterization

The companies surveyed worked in a large range of industry sectors. The results represented in Figure 3.3 identify more than 10 different industry sectors.

We also analyzed the size of the companies according to their number of collaborators. Most of the companies are large companies, 37% have between 100 and 1,000 collaborators and 45% have more than 1,000 collaborators (Figure 3.4).

The answers to 'In what role(s) does your company conducts software test, if any?' show that half of the companies performs tests to the software developed by themselves (Figure 3.5).

Regarding the types of test levels performed, we realize from the answers (Figure 3.6) that the unit testing level is the less performed and the other three levels (integration, system and acceptance) are performed with the same frequency.

3.2.3 Distributed and Heterogeneous Systems Testing

Focusing now on the main questions of this survey, specifically related to the testing of DHS, the answers to 'In what role(s) does your company conducts software test (for DHS), if any?' show that a vast majority of 90% of the companies (all but 10%) conducts tests for DHS in at least one role, with 42% of the companies performing tests for DHS developed by themselves (Figure 3.7).

We also tried to understand what kinds of levels are most commonly used in the testing of such systems. Regarding the responses obtained (Figure 3.8), there is a higher emphasis on system

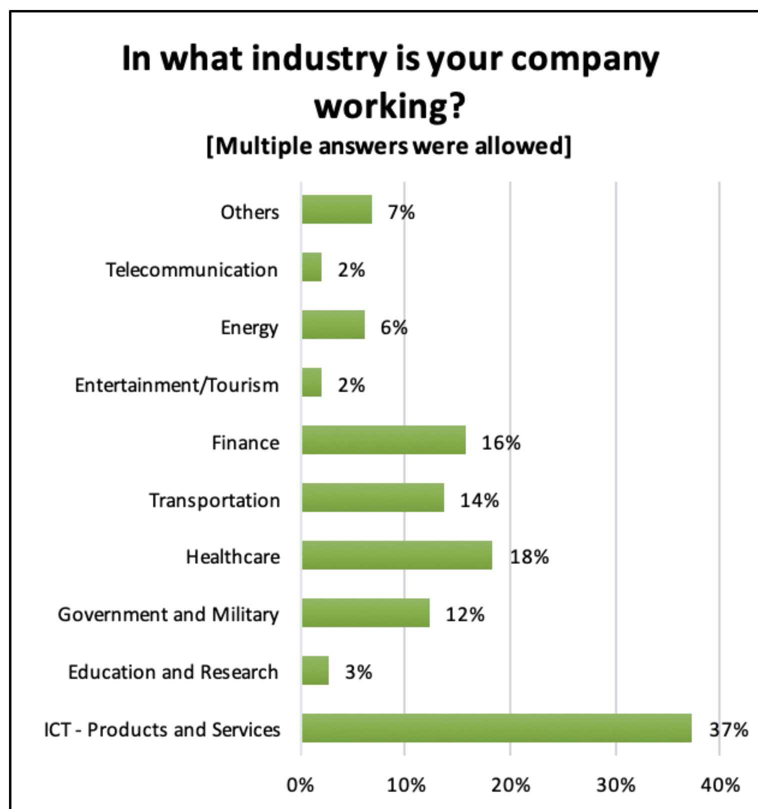


Figure 3.3: Industry Sectors.

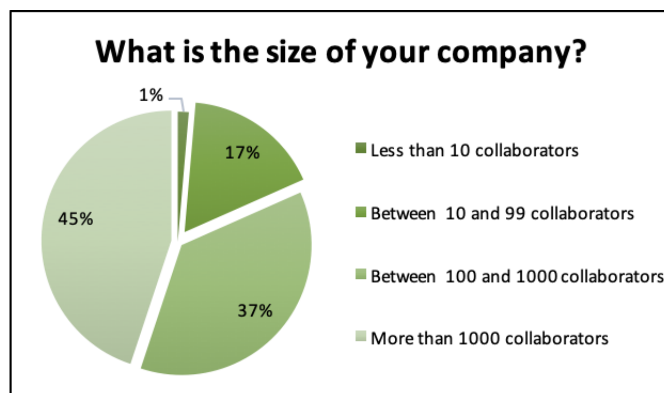


Figure 3.4: Company Size.

testing (71%) followed by integration testing (65%). Only 8% of the respondents did not mention any test level for DHS.

Regarding the most important features that need to be tested in DHS, the results in Figure 3.9 show that the feature that was considered the most important to be tested was 'Interactions between components of the system' (with 76% of responses high or very high), followed by 'Interactions between the system and the environment' (71%) and 'Multiple platforms' (66%). All the features have been considered of 'very high' or 'high' importance by a majority of respondents (50% or more).

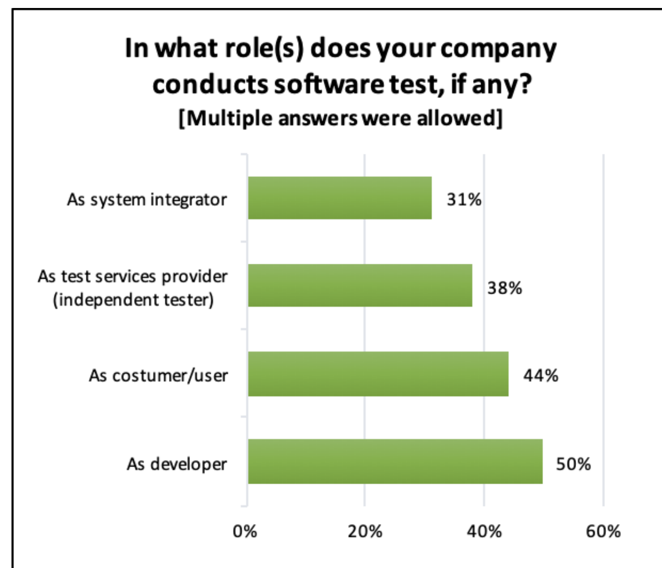


Figure 3.5: Company Roles.

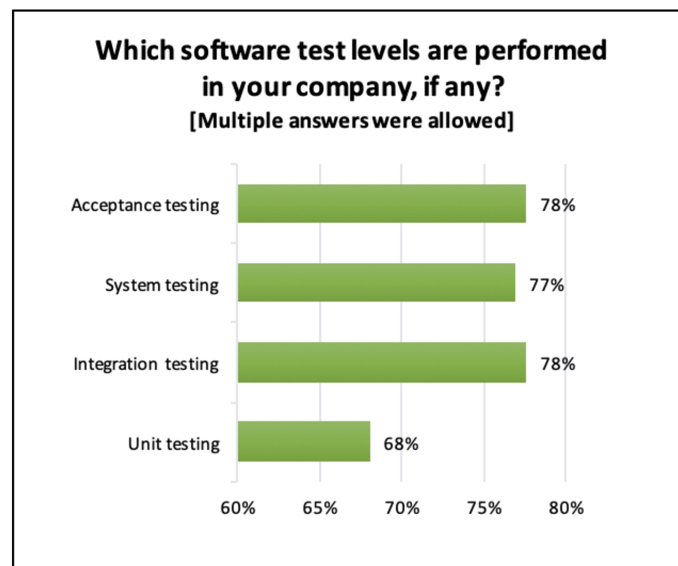


Figure 3.6: Test Levels.

Regarding the level of test automation for DHS, the results presented in Figure 3.10 show that 75% of the tests follow some automated process, however only 16% are fully automatic, which is lower than the 25% who claim to perform only manual testing.

For people who responded that there is at least some automatic process, we asked what kind of tool they use. With this question we can understand the level of effort required to automate the testing process. Looking at the results (Figure 3.11) we realize that only 31% use a commercial tool to automate the process, and the majority, 69%, use a tool developed in-house, reusable or not in different SUTs.

Regarding the desired features of a test automation solution for DHS, the results presented in

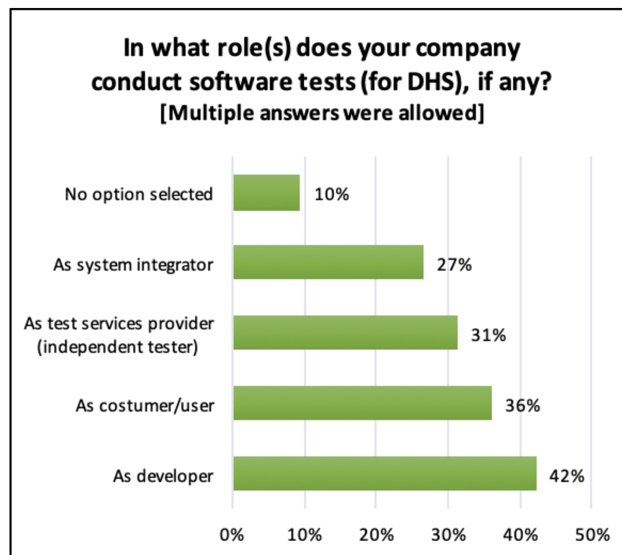


Figure 3.7: Test Roles DHS.

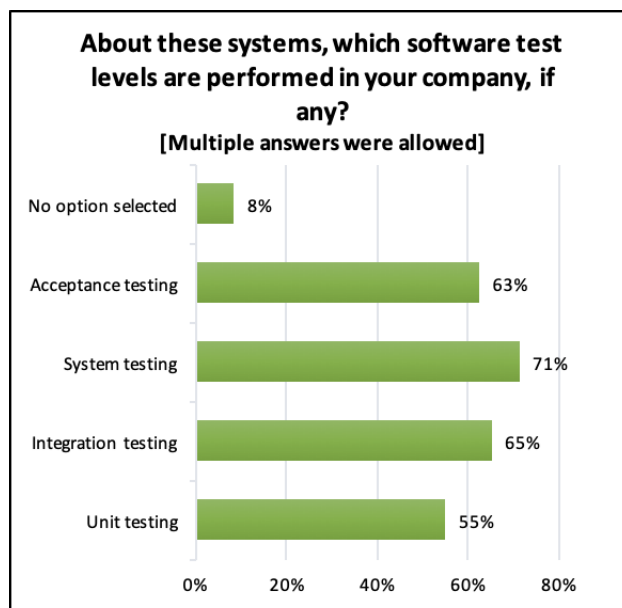


Figure 3.8: Test Levels DHS.

Figure 3.12 show that the most important features (based in the percentage of responses high or very high) in an automated testing tool for DHS are 'Support for automatic test case execution' (75%) and 'Support for multiple platforms' (71%).

As a possible solution to test DHS, we asked the participants in this survey if they would find useful a tool to test these systems that use only a model of interactions (UML sequence diagram) as an entry model. The results (Figure 3.13) show that 86% consider useful a tool with these characteristics.

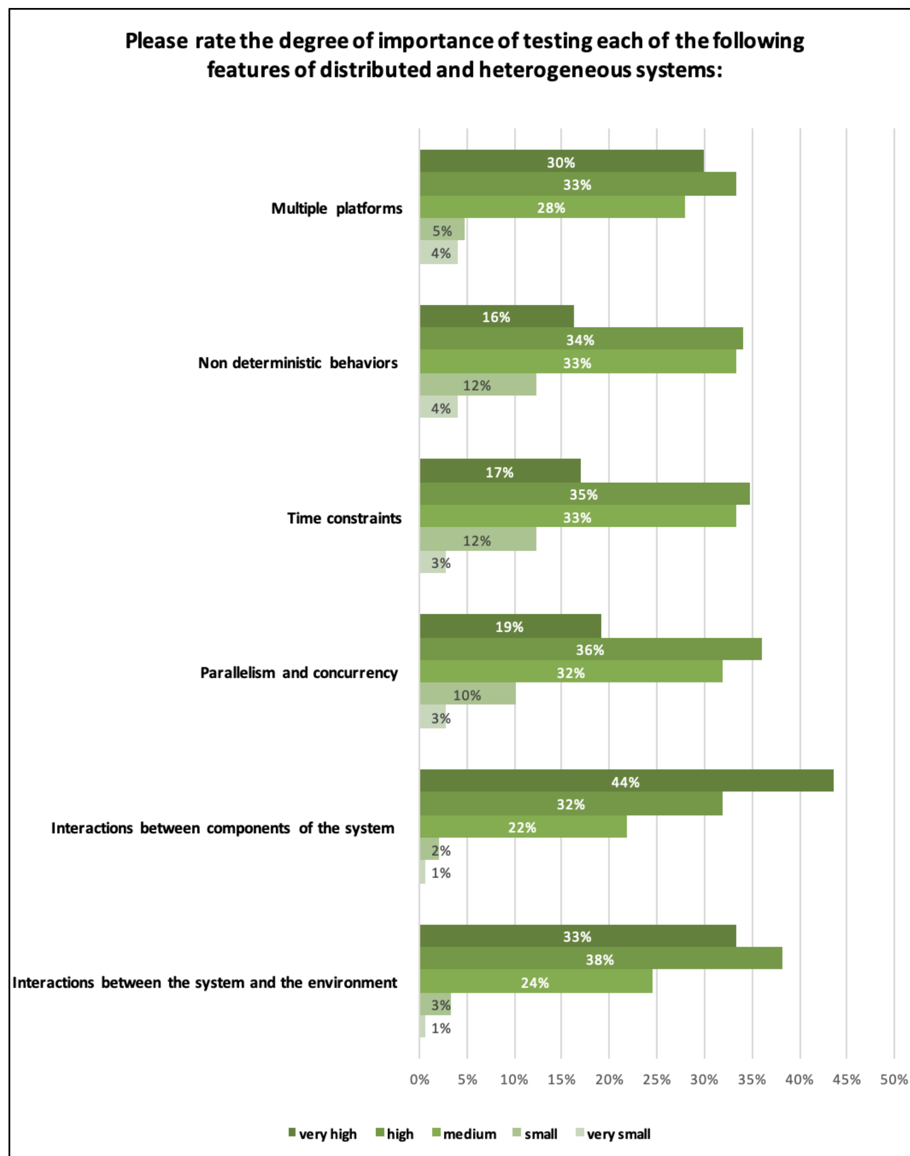


Figure 3.9: Features.

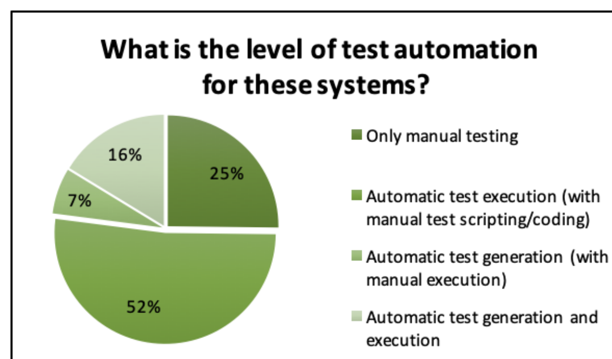


Figure 3.10: Automation Level.

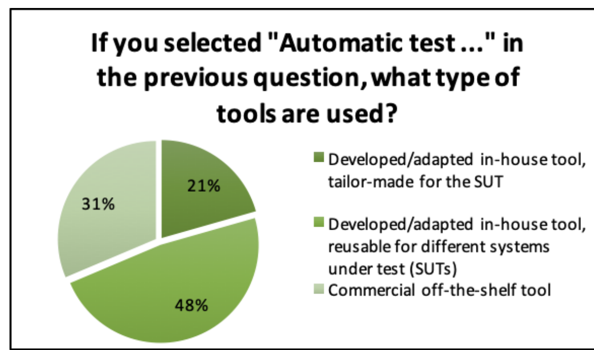


Figure 3.11: Automation Tool.

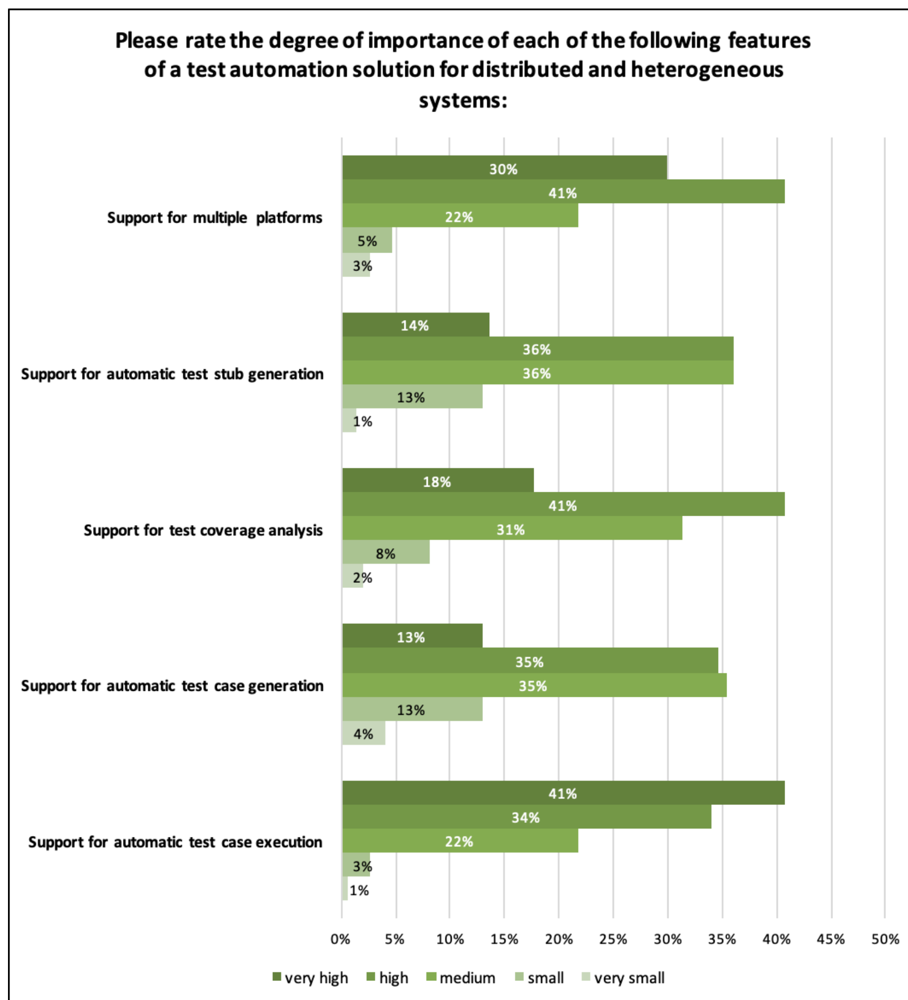


Figure 3.12: Tool Features.

3.2.4 Multivariate Analysis

For questions specifically related to the opinion of the participants, a multivariate analysis was held with the aim to determine whether the participants' responses depend on their current function (Software testing, verification & validation versus all the others).

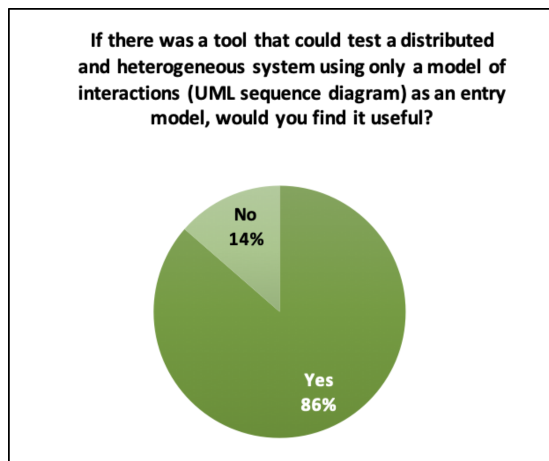


Figure 3.13: New Tool.

The results of the chi-square test for independence show that there is no statistically significant association (for a 95% significance level) between the current function (Software testing, verification & validation versus all others) and the answers to the questions shown in Figures 3.9, 3.12 or 3.13.

3.3 Discussion

3.3.1 Relevance of respondents

The results presented in the previous section show that this survey met the original purpose with regard to their target audience, since 70% of respondents' primary responsibility is related to 'Software testing, verification & validation'. With regard to their experience, the results showed that they are not only people who are mostly in their current position for several years, as work with software testing in general and specifically with DHS. With respect to the type of companies, the results show that this survey covers companies with diverse activity sectors and also large companies (45% have more than 1000 collaborators) which provides a great support to the conclusions reached.

Concerning the main conclusions we can draw from the results, they are next organized according to the initial research questions.

3.3.2 SRQ1: How relevant are DHS in the software testing practice?

The results (Figure 3.7) show that a vast majority of approximately 90% of the companies surveyed (all with software testing activities in general) conducts tests for DHS, in at least one role and at least one test level, hence confirming the high relevance of DHS in software testing practice.

3.3.3 SRQ2: What are the most important features to be tested in DHS?

Regarding the most important features that need to be tested in DHS, the results in Figure 3.9 show that the feature that was considered the most important to be tested was 'Interactions between components of the system' (with 76% of responses high or very high), followed by 'Interactions between the system and the environment' (71%) and 'Multiple platforms' (66%).

Nevertheless, all the features inquired were considered of high or very high importance by a majority of respondents (50% or more).

3.3.4 SRQ3: What is the current status of test automation and tool sourcing for testing DHS?

The results show that the current level of test automation for DHS is still very low, and there is large room for improvement, since 25% of companies in the survey claim that they only perform manual tests, against only 16% who claim to test DHS with a full automatic process.

If we look for companies that have some type of automation in its testing process, we realize that the automation process is requiring a high effort in the creation / adaptation of own tools, because only 31% of companies claim to use a commercial tool to test these types of systems.

3.3.5 SRQ4: What are the most desired features in test automation solutions for DHS?

Regarding the conclusions that can be drawn for future work, particularly at the level of creating tools that can reduce the effort required to test DHS, looking at Figure 3.12, we realize that companies identify as key aspects of a tool to test such systems the ability to automate test execution (75% of responses with high or very high importancte) and the support for multiple platforms (71%).

Nevertheless, all the features inquired were considered of medium, high or very high importance by a large majority of respondents (83% or more).

The comparison of the degree of importance attributed to automatic test case execution (96% of the responses mentioning a medium, high or very high importance in Figure 3.12) with the current status (78% of companies applying automatic text execution in Figure 3.10), show that there is a significant gap yet to be filled between the current status and the desired status of automatic test case execution.

The gap is even bigger regarding automatic test case generation, with 83% of the responses mentioning a medium, high or very high importance in Figure 3.12, and only 23% of the companies currently applying automatic text generation in Figure 3.10.

We realized even by the Figure 3.13, that companies are highly receptive to a test tool that has only a model of interactions as an input model for automatic test case generation and execution.

3.4 Case Based Analysis of Test Automation Obstacles

In order to analyze the drivers and barriers for DHS test automation in companies, we conducted follow-up interviews with some survey respondents. For the interviews we selected a sample of survey respondents from companies with different sizes (in terms of number of employees) and different test automation strategies.

3.4.1 Case A

Company A is a small company that develops software for external customers. The company performs manual tests on the software they develop. The justifications given by this company for only performing manual testing are:

- the low economic capacity to purchase commercial testing tools. Small companies and startups have limited budgets that have to be managed with many limitations and focusing on the rapid development of their products, leaving aside investments on test tools;
- the lack of human resources (in terms of availability and expertise) to allocate to test automation tasks. This type of company usually has a small number of employees, so all end up taking on various tasks. As the main focus is the rapid development of products, the use of any testing tool that requires additional learning time is immediately discarded;
- if they adopted a test automation solution, it would be costly to maintain automated test cases because of constant changes of product requirements and features, implying frequent changes in the test cases and test harness.

3.4.2 Case B

Company B is a large company that develops software for government and military areas. This company uses automation for test execution but still uses a manual process for test case generation (i.e., the creation of test scripts). For test automation, the company uses tools developed in-house, based on open-source frameworks. The justification given for not resorting to commercial tools is mainly due to the high costs charged by suppliers of these tools, often requiring the purchase of extra plug-ins for any additional feature needed.

Besides that reason, the following justifications were given for using tools developed in-house:

- to maintain knowledge within the company. Large companies prefer to develop their own tools because they are often unwilling to share information about their products with commercial tool vendors. For this reason they make use of open source tools that can be easily modified by the experts of the test area of the company itself;
- to be able to make adjustments to the tools more quickly, not being dependent on any vendor. Commercial tools leave companies dependent on their manufacturer, so large companies prefer to develop their own testing tools, since in the current market conditions it is increasingly important to have a quick reaction capability to modify the software.

With regard to manual test generation, the following justifications were given by the representative of the company for not using any automation process:

- the software developed by the company has very specific features that might be difficult to address with existing test generation tools;
- the lack of knowledge of company staff with regard to the creation of models needed as input for test case generation and the subsequent generation of test cases (this is one aspect that the company intends to improve in the near future).

3.4.3 Case C

Company C is a small to medium company that provides consulting services in the area of software testing. Most automation solutions that the company proposes and implements for its customers are related with test execution (and not test generation).

The reasons given for this have to do mainly with:

- the difficulty to automatically generate test cases;
- customers have no system model;
- the creation of system models requires a great effort;
- the system being tested is in a state of constant evolution and therefore not worth the effort in automatic test generating (or event automatic test execution).

As regards the tools that this company suggests to their customers for test execution automation, in most cases they recommend commercial testing tools. According to the representative of this company, this choice happens due to the following reasons:

- commercial tools are "ready to use";
- commercial tools do not require that the company has specialized human resources to adapt the test tool.

However, when the client has know-how in the test area, this company also indicates open source solutions that, in spite of requiring more maintenance, end up giving more flexibility and of course greater freedom to their users.

3.4.4 Synthesis

Analyzing the answers we have come to the conclusion that there are still several barriers and obstacles that prevent companies from adopting a fully automated test process.

Regarding the reasons for not adopting an automated test execution approach, we conclude that the main reasons are:

- cost of commercial testing tools (A);
- lack of human resources (availability and expertise) (A);
- frequent changes in the software under test (A, C).

For companies that have some level of automation in the testing tasks, the choice between commercial tools and in-house tools (usually based on open source tools) depends essentially on the type of company, since although the commercial tools are referred in the interviews as "ready to use" facilitating in this way the test automation process, they have several drawbacks, namely:

- are expensive, especially if extra functionalities and/or platforms are required (B);
- create too much dependence from vendors, and reduce flexibility for extensions and adaptations (B, C);
- know-how related with test automation is kept outside the company (B).

Regarding the reasons for choosing between a manual versus an automated test generation approach (with automatic test generation from models), we found:

- lack of human resources (availability and/or expertise) (A,B);
- frequent changes in the software under test (A,B,C);
- lack of system models (B,C);
- effort required for the creation of system models (B,C).

3.5 Other Surveys

We only found in literature one survey (Ghazi et al., 2015) that discuss some aspects related to the testing of heterogeneous systems. The survey conducted by Ghazi et al. (2015) explored the testing of heterogeneous systems with respect to the usage and perceived usefulness of testing techniques used for heterogeneous systems from the point of view of industry practitioners in the context of practitioners involved in heterogeneous system development reporting their experience on heterogeneous system testing. For achieving this goal the authors tried to answer two research questions:

- Which testing techniques are used to evaluate heterogeneous systems?
- How do practitioners perceive the identified techniques with respect to a set of outcome variables?

The authors concluded that the most frequently used technique is exploratory manual testing, followed by combinatorial and search-based testing, and that the most positively perceived technique for testing heterogeneous systems was manual exploratory testing. Our work has a different objective of the survey conducted by Ghazi. The Ghazi main goal was to identify testing techniques, our aim is to understand how distributed systems and heterogeneous are tested in companies realizing which test levels are performed and which are the automation levels for testing these systems. The Ghazi survey also involved a much smaller number of participants (27).

As regards the general software testing in the literature there are many surveys, however as the main aim of our work is to analyze the state of practice, we analyze surveys carried out in the industry by recognized standardization bodies as ISTQB (ISTQB, 2020). The most recent surveys of this organization (ISTQB, 2016) (conducted over more than 3,000 people from 89 countries) and (ISTQB, 2018) (conducted over more than 2,000 people from 91 countries), although it has a different purpose of our work because is related to the software test in general, provides results that meet the results presented in this article, namely that there are still significant improvement opportunities in test automation (was considered in this studies the area with highest improvement potential).

3.6 Conclusions

In order to assess the current state of the practice regarding the testing of DHS and identify opportunities and priorities for research and innovation initiatives, we conducted an exploratory survey that was responded by 147 software testing professionals that attended industry-oriented software testing conferences.

The survey allowed us to confirm the high relevance of DHS in software testing practice, confirm and prioritize the relevance of testing features characteristics of DHS, confirm the existence of a significant gap between the current and the desired status of test automation for DHS, and confirm and prioritize the relevance of test automation features for DHS. The survey results indicated a limited adoption of complete test automation processes by companies.

For better understanding what are the obstacles that companies face for not adopting complete test automation approaches, we conducted follow-up interviews with companies of different sizes and testing approaches. The conclusions drawn from the interviews allowed us to identify some common obstacles, such as the cost of acquisition and difficulty of adaptation of test automation tools, the cost of test suite maintenance (namely with frequent changes in the software under test), and the effort and expertise required for the creation of system models needed as input for automatic test suite generation.

Relatively to the research questions presented in Chapter 1, we can partially answer RQ1 as follows:

RQ1 - *What are the main difficulties and needs in the integration testing of distributed systems listed in the state of the art and state of practice?*

Looking at the conclusions of this survey, we can say that main difficulties and needs in the integration testing of distributed systems with time-constraints in the state of practice are expensive testing tools, so more open source solutions are needed in order to allow smaller companies to also be able to automate the testing process for these types of systems.

The results also highlights important needs in the context of DHS testing, namely: the need for checking interactions between the system components (Figure 3.9); the need for automated test execution (Figure 3.12); the need to support multiple platforms, among others.

Chapter 4

Proposed Testing Approach and Architecture

Taking into account the limitations identified in the analysis of the state of the art (see Chapter 2) and state of the practice (see Chapter 3), this chapter presents our proposal of an approach and a toolset to automate the whole process of model-based testing of distributed and heterogeneous systems in a seamless way, with a focus on integration testing, but supporting also unit (component) and system testing. The only manual activity (to be performed with tool support) should be the creation of the input model of the SUT.

This chapter is organized as follows. Section 4.1 presents the proposed architecture. The test process is explained in Section 4.2. Section 4.3 presents the toolset architecture. The chapter is concluded with a synthesis of novelties and benefits in Section 4.4.

4.1 Test Architecture

Our main goal is to support the scenario-based integration testing of distributed and heterogeneous systems, from the perspective of system integrators.

In integration testing, it is important to test not only the interactions with the environment (users, external systems, or the physical environment) but also the interactions between the system components.

In order to be able to check the interactions with the environment and between the system components, and simulate inputs from the environment at multiple locations, *local testers* have to be deployed close to the system components, coordinated by a *central tester*, as depicted in Figure 4.1. This test architecture is based on the hybrid test architecture presented in Chapter 2 that best suits this type of system, namely due to its fault detection capacity.

In this type of architecture the local testers may act as *test monitors* (observing the messages sent and received by each component), *test drivers* (simulating inputs from the environment), or even *test stubs* (simulating responses from emulated system components).

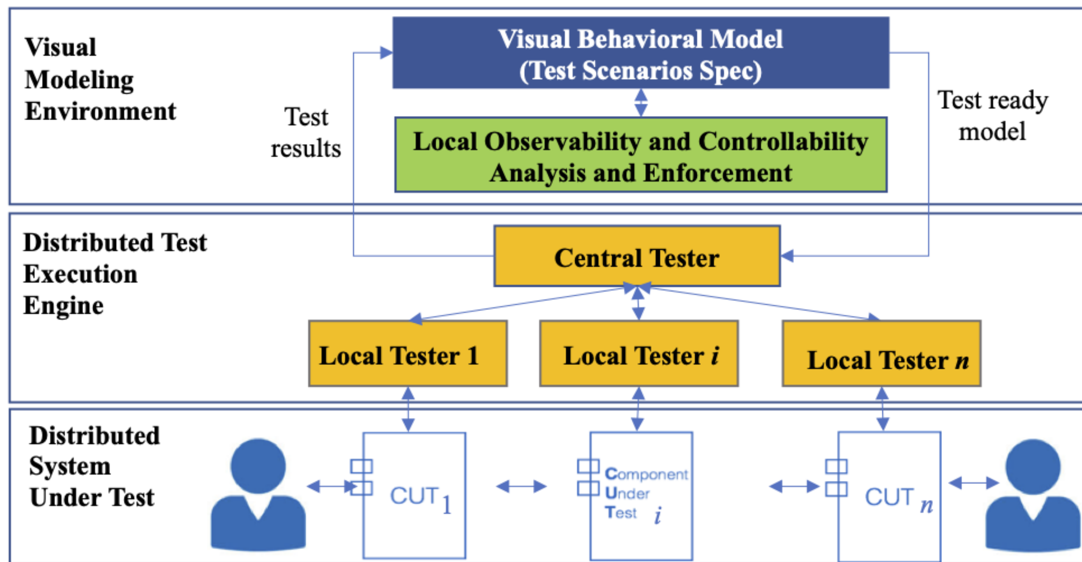


Figure 4.1: Test architecture for the model-based integration testing of distributed systems.

For the description of the test scenarios we advocate the usage of industry-standard UML sequence diagrams because they provide a convenient means to describe the interactions that occur between between the components and actors (environment) of a distributed system, with support for control flow variants, time constraints, and non-determinism.

To cope with non-determinism and response time constraints, test inputs may have to be selected at runtime in an adaptive and responsive way, based on the observed execution events and the behavioral specification, suggesting an *adaptive and distributed test input selection* approach. To facilitate fault localization, conformance errors should be detected as early as possible and as close as possible to the offending components, suggesting an *incremental and distributed conformance checking* approach.

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

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

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

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

4.2 Test Process

Figure 4.2 depicts the main activities and artifacts of the proposed test process based on the test architecture described in Section 4.1. The main activities are described in the next subsections and illustrated with a running example.

4.2.1 Visual Modeling

The behavioral model is created using an appropriate UML profile (OMG, 2017; Gross, 2005) and an existing modeling tool. We advocate the usage of UML 2 SDs, with a few restrictions and extensions, because they are well suited for describing and visualizing the interactions that occur between the components and actors of a distributed system. UML deployment diagrams can also be used to describe the distributed structure of the SUT. Mapping information between the model and the implementation, needed for test execution (such as the actual location of each component under test), may also be attached to the model with tagged values.

To illustrate the approach, we used a real world example from the AAL4ALL project, related with a fall detection and alert service. As illustrated in Figure 4.3, this service involves the interaction between different heterogeneous components running in different hardware nodes in different physical locations, as well as three users.

A behavioral model for a typical fall detection scenario is shown in Figure 4.4. In this scenario, a care receiver has a smartphone that has installed a fall detection application. When this person falls, the application detects the fall using the smartphone's accelerometer and provides the user a message which indicates that it has detected a drop giving the possibility for the user to confirm whether he/she needs help. If the user responds that he/she does not need help (the fall was slight, or it was just the smartphone that fell to the ground), the application does not perform any action; however, if the user confirms that needs help or does not respond within 5 seconds (useful if the person became unconscious due to the fall), the application raises two actions in parallel. On the one hand, it makes a call to a previously clearcut number to contact a health care provider (in this case can be a formal or informal caregiver); on the other hand, it sends the fall occurrence for a Personal Assistance Record database and sends a message to a portal that is used by a caregiver (e.g. a doctor or nurse) that is responsible for monitoring this care receiver. The last two actions are performed through a central component of the ecosystem called AALMQ (AAL Message Queue), which allows incoming messages to be forwarded to multiple subscribers, according to the publish-subscribe pattern (Gamma et al., 1994).

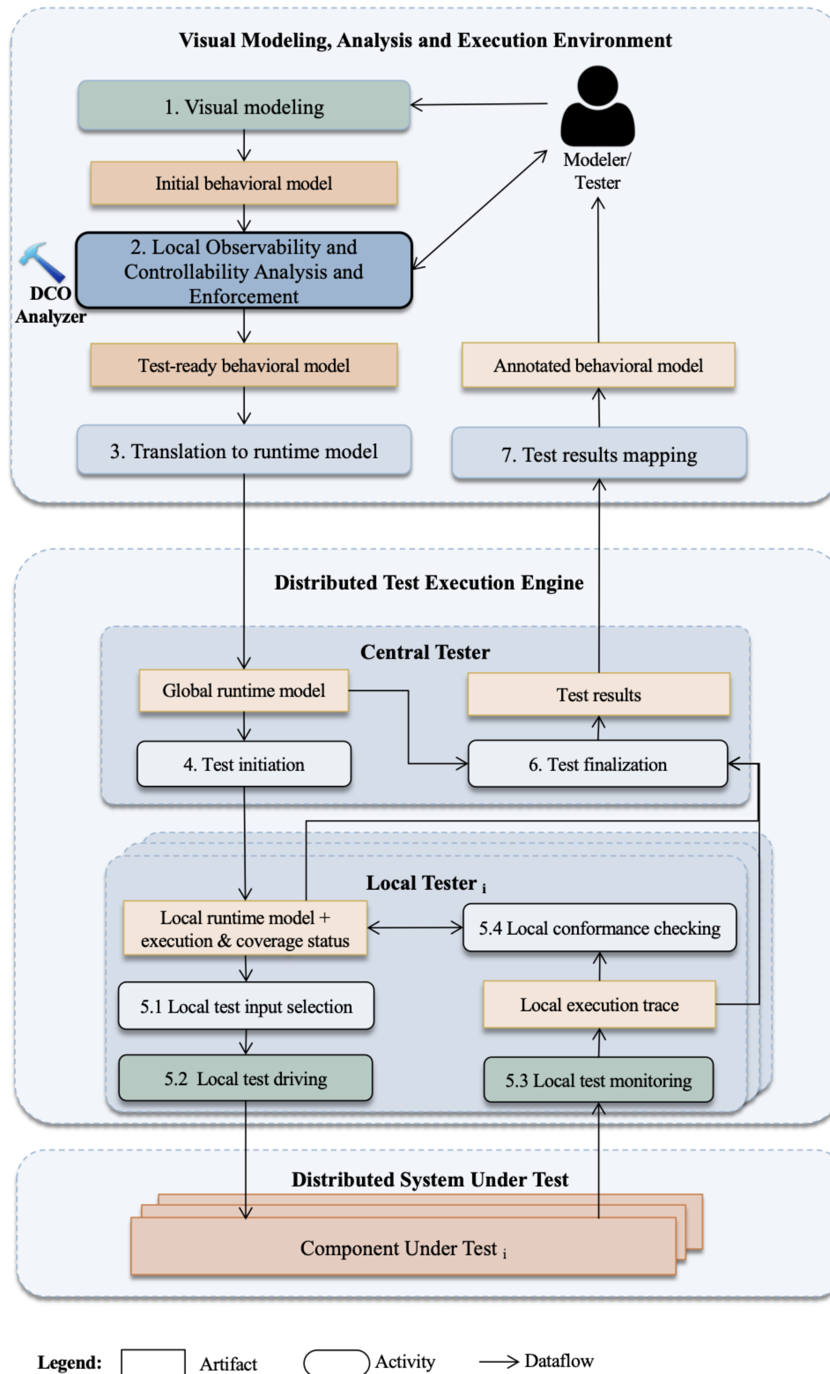


Figure 4.2: Dataflow view of the proposed test process.

4.2.2 Local Observability and Controllability Analysis and Enforcement

Implemented in the DCO Analyzer tool (described in Chapter 6) using the algorithms and procedures described in Chapter 5, this activity is responsible to check if conformance errors can be detected locally (*local observability*) and test inputs can be decided locally (*local controllability*) by the local testers for the test scenario under consideration, without the need for exchanging co-

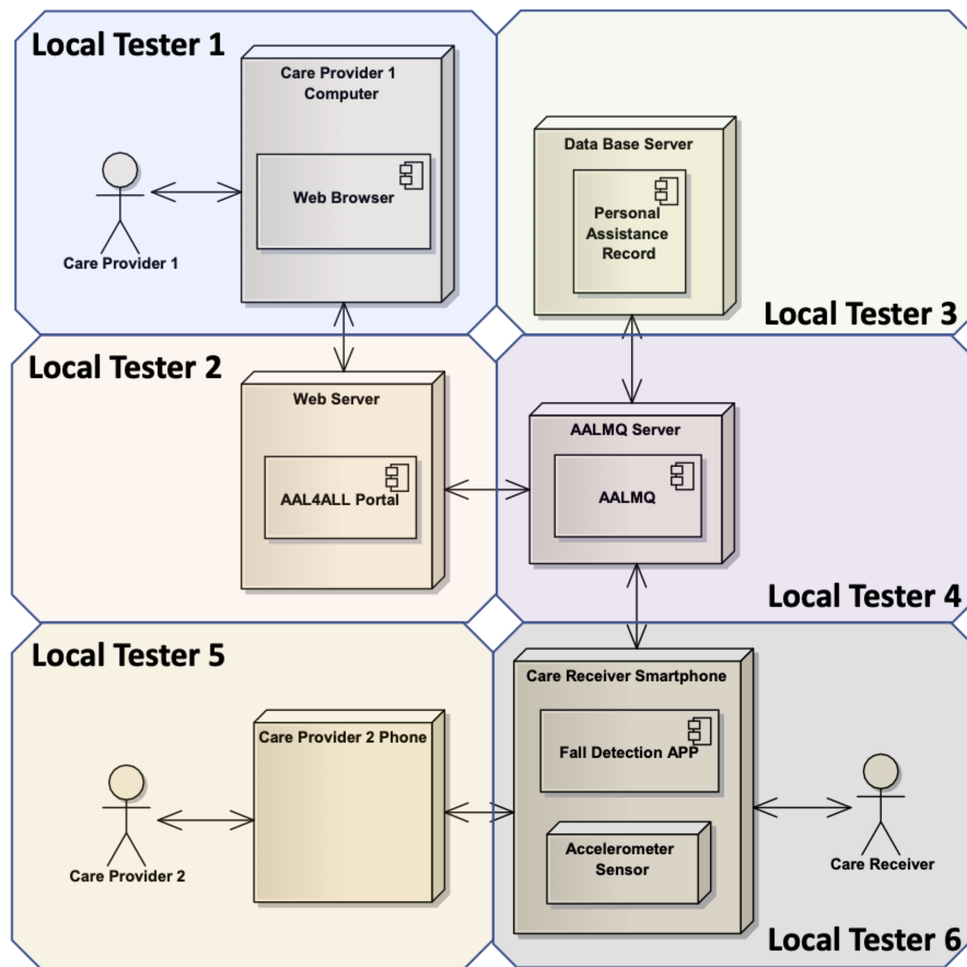


Figure 4.3: UML deployment diagram of a fall detection scenario.

ordination messages between the test components during test execution. If these two properties are not met a set of coordination messages or coordination time constraints are recommended.

Assuming that messages may be lost by the transmission channels, a common cause of local observability problems are optional messages without corresponding acknowledgment messages (to be detailed in Chapter 5). If we look at the running example (Figure 4.4), it is possible to realize that this scenario is not locally observable, as the local testers placed next to the CUTs, "AALMQ", "Personal Assistante Record", "AAL4ALL Portal", "Care Provider 1" and "Care Provider 2" are not able to detect the loss of messages, since receiving nothing is a valid behavior.

Regarding local controllability, a common cause of local controllability problems are mutually exclusive emission and reception events simultaneously enabled. In the running example, in the absence of time constraints, the local tester located next to the CUT "Fall Detection Mobile APP" would not know from which moment it could send the messages $emergency_call_on_possible_fail$ and $possible_fall_info$, which would make the scenario not locally controllable. A common solution is the insertion of appropriate timing constraints so that those events are not simultaneously enabled.

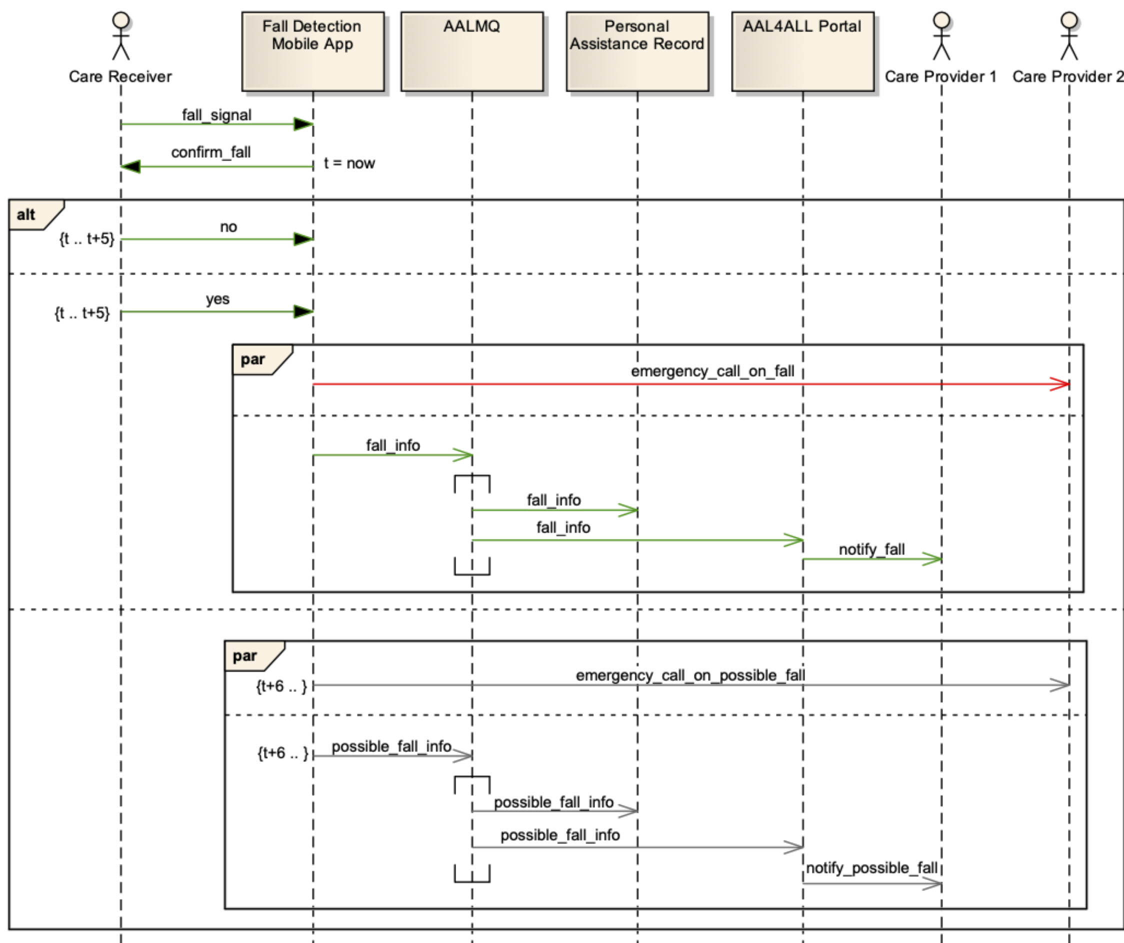


Figure 4.4: UML sequence diagram representing the interactions of the fall detection scenario. The diagram is already painted after a failed test execution in which the fall detection application didn't send an emergency call.

In cases where it is not possible to make the scenarios locally observable and locally controllable (or the modeler does not want to do so), it will be necessary to attach this information to the model, so that when starting the tests, the test strategy to be followed keeps in mind these characteristics of the model.

4.2.3 Translation to Runtime Model

Although the visual model is the most suitable for the user (modeler or tester), a more formal representation (runtime model) is necessary to be processed incrementally at run time. Different types of state-based models can be used for this function.

One possible option is to use the Timed Event-Driven Colored Petri Nets, or TEDCPN for short, proposed in (Lima and Faria, 2015), combining the characteristics of Event-Driven Colored Petri Nets proposed in (Faria and Paiva, 2014) for testing object-oriented systems, and Timed Petri Nets (Wang, 2012).

Petri Nets are well suited for describing in a rigorous and machine processable way the behavior of distributed and concurrent systems, usually requiring fewer places than the number of states of equivalent finite state machines. Translation rules from UML 2 SDs to Event-Driven Colored Petri Nets have been defined in (Faria and Paiva, 2014) and implemented in (Custódio Soares et al., 2018) using the Eclipse Modeling Framework (EMF).

4.2.4 Test Initiation

As previously explained in Section 4.1 we prefer an online, adaptive, strategy, in which the next test action is decided based on the current execution state. Whenever multiple alternatives can be taken by the test harness in an execution state, the test harness must choose one of the alternatives and keep track of unexplored alternatives (i.e., model coverage information) to be exercised in subsequent test repetitions.

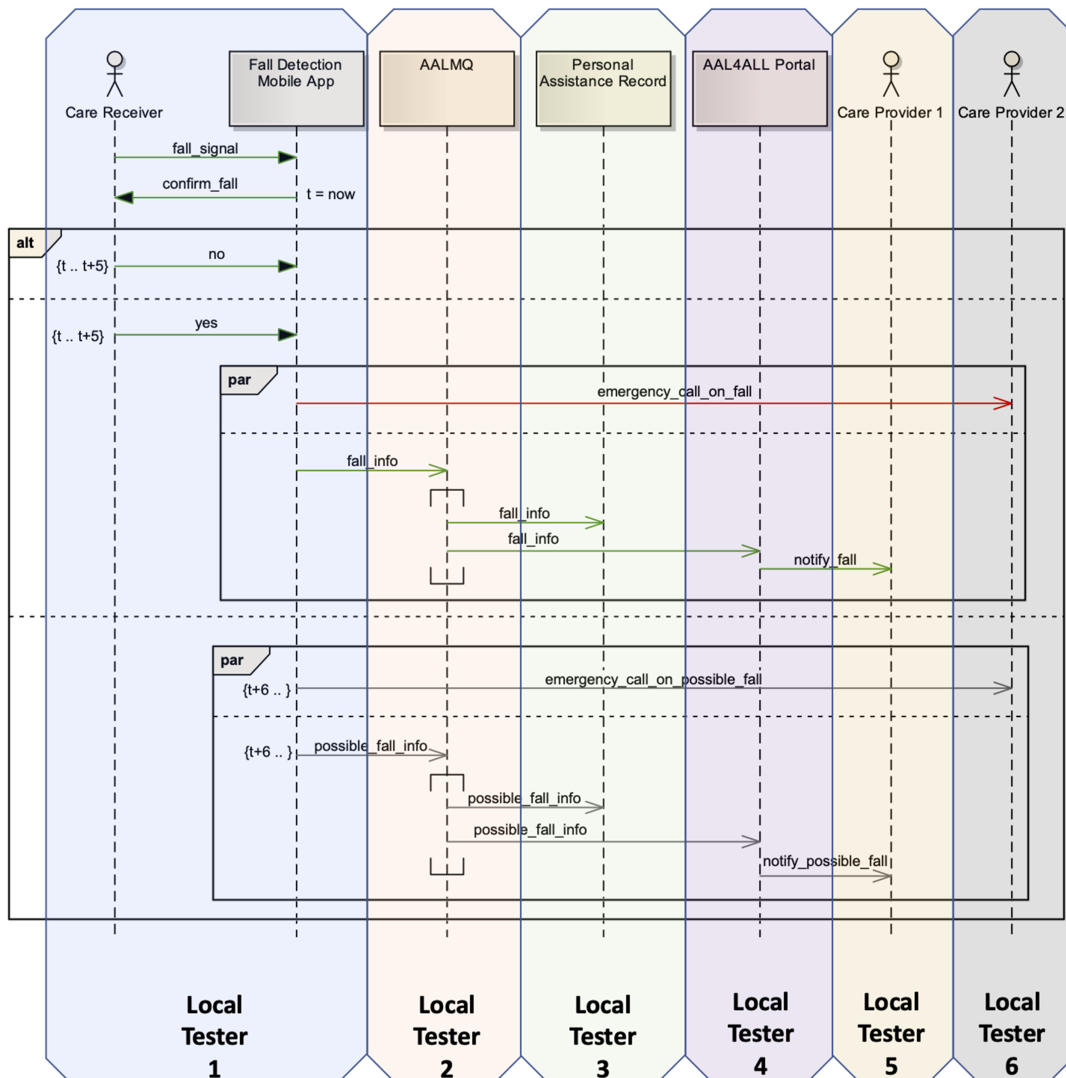


Figure 4.5: Location of the Local testers for the example represented in Figure 4.4.

To enable distributed test input generation and conformance checking, the central tester has to derive from the global runtime model local runtime models to be used by the local testers. Each local runtime model specifies the local traces valid at the location under consideration, in a format suitable for incremental processing. At the begin of test execution, the central tester has to deploy appropriate local testers to the different locations of the distributed SUT (if not done before), pass the local runtime models to the local testers, and initiate both the SUT and the local testers. In the test initiation stage, the central testers should also synchronize the clocks of the local testers. The same test scenario may be executed multiple times to improve coverage.

In the case of the running example, and as can be seen in Figure 4.5, the central tester needs to deploy six local testers, the first with the CUT "Fall Detection Mobile APP" that will be responsible for observing/generate the inputs of the "Care Reveiver ", as well as observing the messages sending by the application "Fall Detection Mobile APP". The second local tester will be placed in the CUT "AALMQ", the third next to the CUT "Personal Assistance Record", the fourth next to "AAL4ALL Portal", the fifth next to "Care Provider 1" and the sixth next to CUT "Care Provider 2".

4.2.5 Local Test Input Selection

Using the UML 2 interaction operators, a single SD, and hence the runtime model derived from it, may describe multiple control flow variants, that require multiple test cases for being properly exercised.

In the running example, from the reading of the set of interactions represented in Figure 4.4, one easily realizes that there are three test paths to be exercised (with at least one test case for each test path). The first test path (TP1) is the case where the care receiver responds negatively to the application and the application doesn't trigger any action. The second test path (TP2) is the situation where the user confirms to the application that he/she needs help and after that the application triggers the actions. The last test path (TP3) corresponds to the situation where the user doesn't answer within the defined time limit and the application triggers the remaining actions automatically. If one wants also to exercise the boundary values of allowed response time (close to 0 and close to 5 seconds), then two test cases can be considered for each of the test paths TP1 and TP2, resulting in a total of 5 test cases.

In automated test execution, the test harness (the local testers, in our case) has to simulate inputs from the environment, according to the test scenario.

In case multiple alternative inputs are possible at a given point of test execution (multiple input messages and/or timings), the test harness has to decide (select) the next input from the set of possible inputs.

In the running example, the local tester 1 deployed close to the Fall Detection App, has to simulate the inputs from the Care Receiver, namely the fall detection signal (to be read by the device's accelerometer) and the inputs through the application's GUI. After observing the confirmation request message in the GUI, the local tester may follow three courses of action, as previously explained.

4.2.6 Local Test Driving

This activity is responsible for translating the abstract (platform independent) test inputs decided by the test input selection activity into concrete test inputs, using model-to-implementation mapping information, and injecting those inputs on the corresponding CUT in a platform-dependent way. The idea is to produce a toolkit of test drivers for different platforms, and reuse them by multiple local testers.

If a CUT is to be emulated by a local tester acting as a test stub, the test harness has to inject the responses (outgoing messages) from the emulated CUT to the environment or other CUTs.

4.2.7 Local Test Monitoring

This activity is responsible for assuming the function of a test monitor, that is, it is responsible for observing all the iterations (messages exchanged) of a given CUT, with other CUTs or with the environment, thus producing the local execution trace.

The messages are observed in a platform dependent way and are mapped to an abstract (platform-independent) representation, using model-to-implementation mapping information whenever needed. This allows the local execution trace to be represented in a platform-independent way, that can be consumed for conformance checking purposes and for updating the execution status of the local runtime model.

4.2.8 Local Conformance Checking

Using the local runtime model (including the execution and coverage information) and the local execution trace, this activity is responsible for verifying locally if the local execution trace observed so far is a valid execution trace. If it detects that the execution trace is not valid or that it has reached the end of its execution, it reports that information to the central tester.

4.2.9 Test Finalization

This activity is responsible for aggregating the local testers' information and producing the tests' results, namely about coverage and possible errors detected, reporting this information to the "Test results mapping" activity.

Test execution terminates when a local testers sends a fail verdict to the central tester (because an error was encountered), or all local testers send a pass verdict (because they reached the end of execution) or a quiescence notification (with an associated pass or fail verdict). In practice quiescence may be detected using timeouts.

The central tester should then aggregate the verdicts and execution traces provided by the local testers to arrive at a final test verdict.

A fail verdict from one local tester implies a global fail verdict. If the test scenario is locally observable, a pass verdict from all local testers implies a global pass verdict. However, if the test scenario is not locally observable, a final conformance checking of the global execution trace

against the global runtime model has to be performed by the central tester to arrive at a final verdict. Because of the impossibility to ensure perfect clock synchronization in a distributed system, the resulting test verdict may be inconclusive (see Section 5.3.6).

In case the test scenario is not locally controllable, local testers may be the components to blame for failed test executions, because of injecting inputs that are locally valid but are not globally valid. Such executions need to be identified and discarded by the central tester, before reporting test results.

For each failed test execution (global execution trace with a fail verdict), it is also important to report where the global trace deviates from the valid traces and characterize the conformance error. This involves determining the valid trace(s) that best matches the observed execution trace (with the longest common prefix), to subsequently locate the missing or erroneous event in the observed trace. An event in this context is the emission or reception of a message by a scenario participant (component or actor). The participant to blame (component, actor or transmission medium) may also be identified.

In the running example, as the scenario is not locally observable, it is necessary for the central tester to perform a final conformance checking of the global execution trace against the global runtime model. Since, for example, if the message $emergency_call_on_fall$ is sent by the CUT "Fall Detection Mobile APP" but is lost and never reaches the CUT "Care Provider 2", all CUTs will report the PASS verdict despite an error. This error will then be detected by the Central Tester.

4.2.10 Test Results Mapping

At the end of test execution it is important to reflect the test results back in the visual behavioral model created by the user. With the help of a tool, this activity is responsible for mapping the test results (coverage and errors) in the visual behavioral model, thus ensuring that the results are easily perceived by the modeler/tester.

As an example, if during the test the Fall Detection App didn't send an emergency call (or that message was lost), a simple analysis of the final state of the formal model point out to the tester which messages in the source SD were covered and what was the cause of test failure (missing $emergency_call_on_fall$ message), as shown (in red) in Figure 4.4.

4.3 Toolset Architecture

Figure 4.6 depicts a layered architecture of a toolset for supporting the test process described in the previous section, promoting reuse and extensibility.

At the bottom layer in Figure 4.6, the SUT is composed by a set of components under test (CUT), executing potentially in different nodes (OMG, 2017). The CUTs interact with each other (usually asynchronously) and with the environment (users, external systems or physical environment) through well defined interfaces at defined interaction points or ports (Hierons, 2014; Gross, 2005).

The three layers of the toolset are described in the following sections.

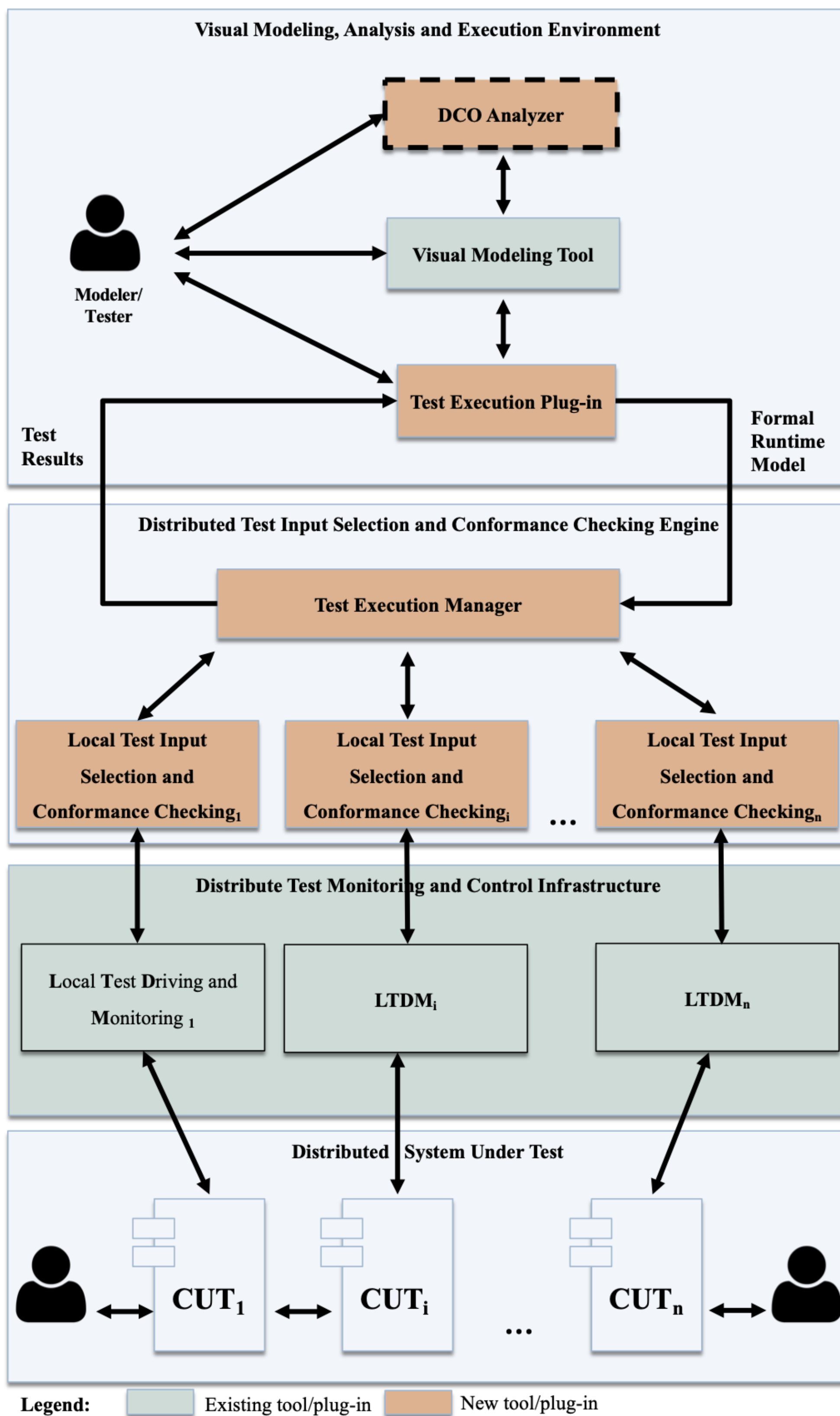


Figure 4.6: Toolset architecture.

4.3.1 Visual Modeling, Analysis and Execution Environment

At the top layer, we have a visual modeling environment, where the modeler/tester can create a *visual behavioral model* of the SUT, analyze observability and controllability properties, invoke test generation and execution, and visualize test results and coverage information back in the model.

This layer also includes a *translation plug-in* to automatically translate the visual behavioral models created by the user into the runtime format accepted by the test execution manager in the next layer, and a *mapping plug-in* to translate back the test results (coverage and error information) to annotations in the visual model.

The analyze of the observability and controllability proprieties of the visual behavioral model is performed by the DCO Analyzer tool (described in Chapter 6), the results of this analysis are presented to the model/test and include information of how to change the model to achieve this proprieties.

The model transformations can be implemented using existing MDA technologies and tools (Völter et al., 2013).

4.3.2 Distributed Test Input Selection and Conformance Checking Engine

At the next layer, the test execution engine is the core runtime engine of the toolset. It comprises a *model execution & conformance checking engine*, responsible for incrementally checking the conformance of observed execution traces in the SUT against the *runtime model* derived from the previous layer, and a *test execution manager*, responsible for initiating test execution (using the services of the next layer), decide next actions to be performed by the local test driving and monitoring components in the next layer of the system, and produce *test results* and diagnosis information for the layer above.

The model execution & conformance checking engine can be implemented by adapting existing Petri net engines, such as CPN Tools (Jensen et al., 2007).

4.3.3 Distributed Test Monitoring and Control Infrastructure

Existent software components and libraries can be reused or adapted for performing the test driving and test monitoring activities assigned to the local testers in a distributed manner.

Hence, the Distributed Test Monitoring and Control Infrastructure comprises a set of *local test driving and monitoring* (LTDM) components, each communicating (possibly synchronously) with a component under test (CUT), performing the roles of test monitor, driver and stub.

This infrastructure may be implemented by adapting and extending existing test frameworks for distributed systems, such as the ones described in Section 2.3.

Different LTDM components have to be implemented for different platforms and technologies under test, such as WCF (Windows Communication Foundation), Java EE (Java Platform, Enterprise Edition), Android, etc. However, a LTDM component implemented for a given technology

may be reused without change to monitor and control any CUT that uses that technology. For example, in our previous work for automating the scenario-based testing of standalone applications written in Java, we developed a runtime test library able to trace and manipulate the execution of any Java application, using AOP (aspect-oriented programming) instrumentation techniques with load-time weaving. In the case of a distributed Java application, we would need to deploy a copy of that library (or, more precisely, a modified library, to handle communication) together with each Java component under test. In the case of a distributed system implemented using other technologies (with different technologies for different components in case of heterogeneous systems), similar test monitoring components suitable for the technologies involved will have to be deployed.

4.4 Conclusions

In this chapter we presented our proposal of an approach and a toolset to automate the whole process of model-based testing of distributed systems, partially answering RQ2:

RQ2 - *What is an adequate architecture and approach to conduct integration tests in these types of systems?*

Our proposed approach is based on a process supported by tools that follow the following main ideas:

- the adoption of different ‘frontend’ and ‘backend’ modeling notations, with an automatic translation of the input behavioral models created by the user in an accessible ‘frontend’ notation (using industry standards such as UML (OMG, 2017)), to a formal ‘backend’ notation amenable for incremental execution at runtime;
- the adoption of an online and adaptive test strategy, where the next test input depends on the sequence of events that has been observed so far and the resulting execution state of the formal backend model, to allow for non-determinism in either the specification or the SUT (Hierons, 2014);
- preprocessing of test scenarios (local observability and controllability analysis and enforcement) to enable distributed test execution and solve the test coordination problem;
- the automatic mapping of test results (coverage and errors) to the ‘frontend’ modeling layer.

As compared to existing approaches (see Chapter 2), the approach proposed in this thesis provides the following novelties and benefits.

Our approach provides a higher level of automation of the testing process because all phases of the test process are supported in an integrated fashion. The only manual activity needed is the development in a user friendly notation of the model required as input for automatic test case generation and execution; there is no need to develop test components specific for each SUT.

This approach also provides a higher fault detection capability. The use of a hybrid test architecture allows the detection of a higher number of errors as compared to purely distributed or

centralized architectures. Interactions between components in the SUT are also monitored and checked against the specification, besides the interactions of the SUT with the environment. To facilitate fault diagnosis, it is used an incremental conformance checking algorithm allowing to capture the execution state of the SUT as soon as a failure occurs. Because of the support for temporal constraints, timing faults can also be detected. Our approach has the ability to test non-deterministic SUT behaviors, using an online, adaptive, test generation strategy.

The proposed approach provides easier support for multiple test levels because the same input model can be used to perform tests at different levels (unit, integration, and system testing), simply by changing the selection of observable and controllable events in the input model. A scenario-oriented approach simplifies the level of detail required in the input models.

With this approach the test execution process is more efficient. With a distributed conformance checking algorithm, communication overheads during test execution are minimized and the usage of a state-oriented runtime model allows a more efficient model execution and conformance checking.

Chapter 5

Local Observability and Controllability Analysis and Enforcement Algorithms

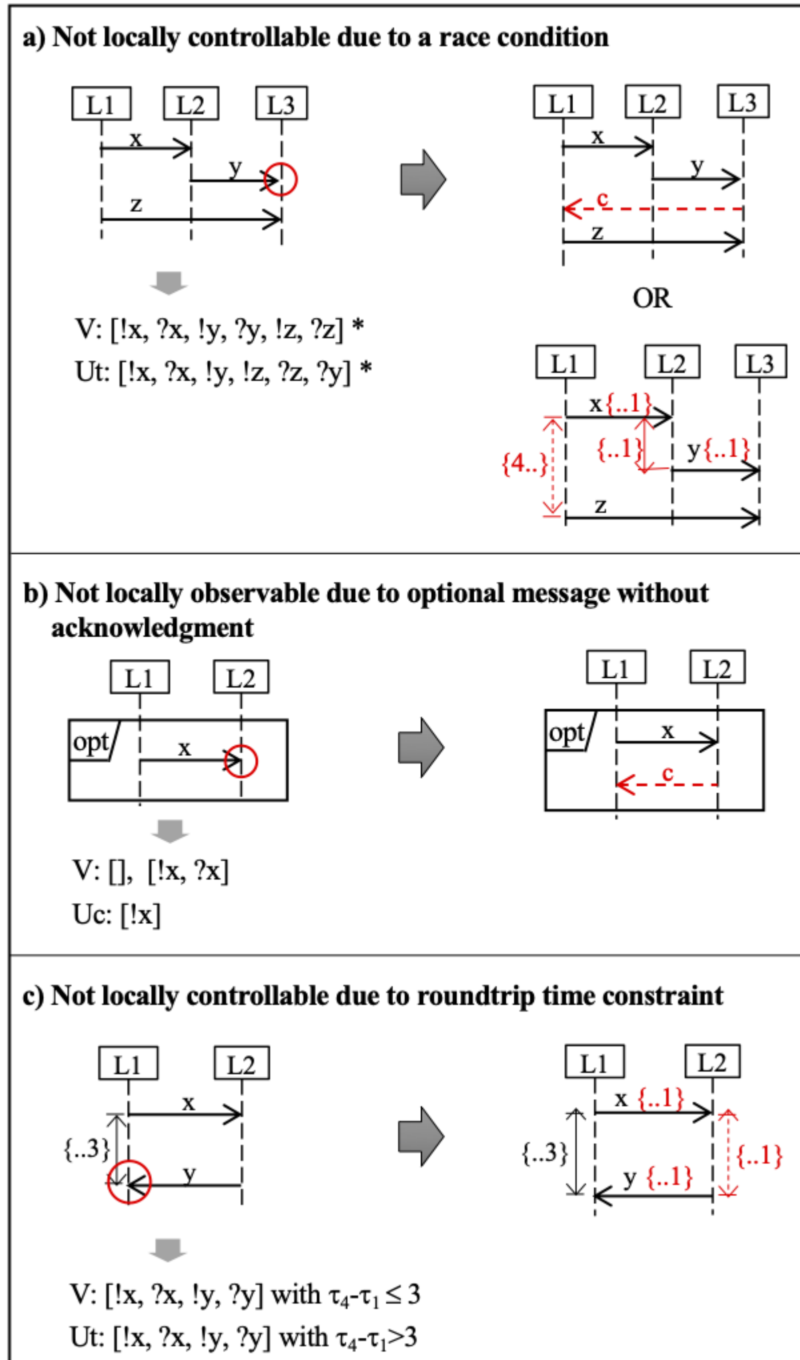
As described in Chapter 4, test coordination in a hybrid test architecture is a big challenge specifically if the properties of local (distributed) observability and controllability do not hold for the test scenario under consideration. In these cases, we need to try to determine a minimum set of coordination messages or coordination time constraints to be attached to the given test scenario to enforce those properties, whilst preserving the semantics of the test scenario. Then the refined test scenario can be executed as in the purely distributed approach. If only coordination time constraints are added, the whole testing approach is still purely distributed. But if coordination messages are added, the whole testing approach becomes a hybrid one (with some coordination messages exchanged during test execution, with minimal overhead and delays). In this chapter, we present the algorithms to automate the process of analyzing and enforcing the local observability and controllability properties.

This chapter is organized as follows. Section 5.1 presents some motivating examples. The semantics of basic UML SDs are present in Section 5.3. Section 5.3 presents the semantics of time-constrained UML SDs. Procedures for local observability analysis are presented in Section 5.4. Section 5.5 presents procedures for local controllability analysis. Procedures for local observability and controllability enforcement are presented in Section 5.6. Section 5.7 concludes the chapter by answering research questions 3 and 4.

5.1 Motivating Examples

Figures 5.1 and 5.2 show examples of simple scenarios to illustrate local observability and controllability problems and ways to overcome them.

Scenario a) illustrates a local controllability problem caused by a race condition. Based on local knowledge only, lifeline $L1$ doesn't know when to send z to ensure that it arrives at $L3$ after y, so it may generate invalid (*unintended*) traces with $?z$ before $?y$. On the right, are illustrated two ways to overcome this problem. In the first solution, a *coordination message* is transmitted from



<u>Notation</u>: !*x* – emission of message *x* ?*x* – reception of message *x*

τ_i – time instant of the *i*-th occurrence in the trace

* – and several permutations ◯ – error location

V – valid traces

Ut – unintended traces

Uc – locally uncheckable traces

Figure 5.1: Interaction fragments with local observability and controllability problems and possible refinments.

$L3$ to $L1$, so that $L1$ knows when to safely send z. From a testing perspective, assuming that $L1$ is simulated by a local tester (test driver) and $L3$ is monitored by another local tester, the coordination message would be exchanged between the local testers (without affecting the SUT). The communication overhead of this solution (1 message) is much smaller than the overhead incurred by a centralized testing approach, in which the events observed by the local testers are constantly communicated to the central tester (4 messages from the local testers at $L2$ and $L3$ to the central tester), that decides and communicates back to the local testers the next test inputs (2 messages from the central tester to the local tester at $L1$). The second solution relies on *coordination time constraints*. From a testing perspective, the maximum duration constraints could represent assumptions about the SUT behavior (lifelines and communication channels), and the minimum duration constraint could represent a constraint to be followed by the test driver at $L1$. If such assumptions can be made, this approach has the advantage of not implying any communication overhead during test execution (possibly at the cost of a pessimistic wait time at $L1$).

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

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

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

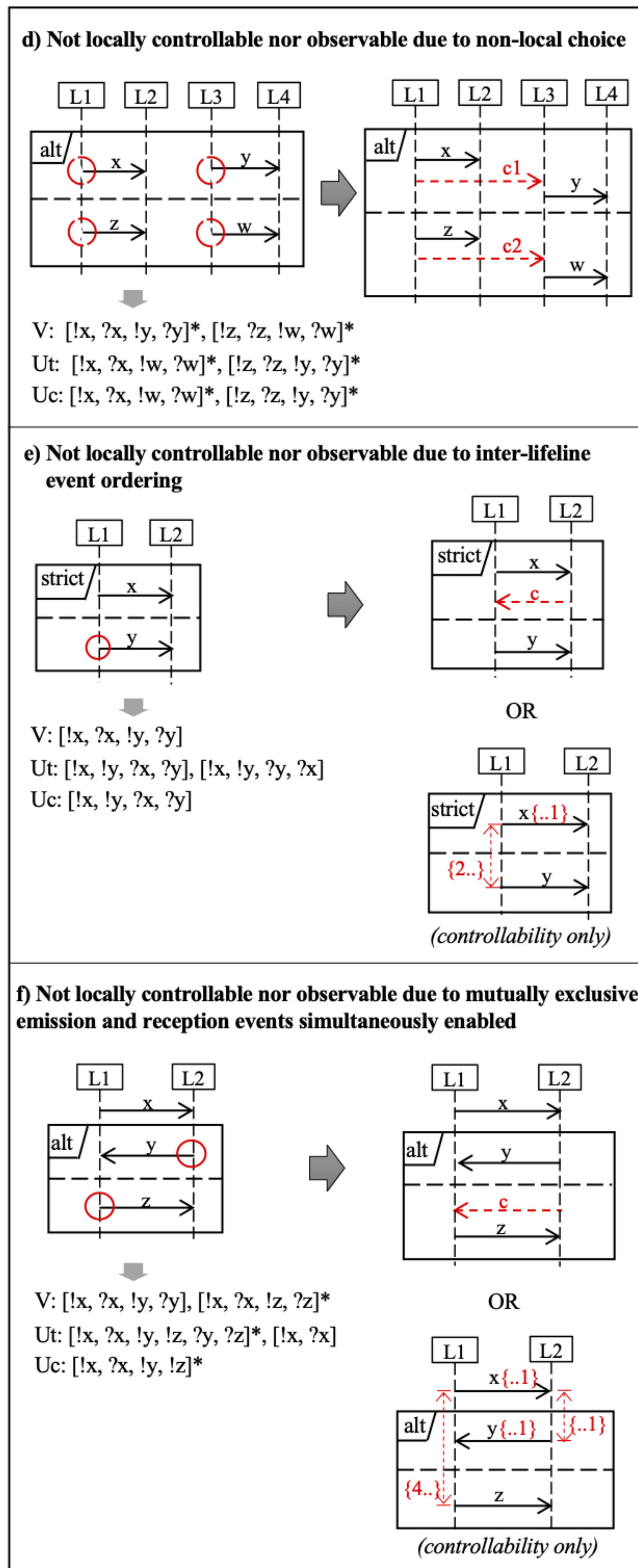


Figure 5.2: Interaction fragments with local observability and controllability problems and possible refinments (continued).

Scenario e) illustrates a local observability and local controllability problem caused by an inter-lifeline event ordering constraint. Based on local knowledge only, lifeline $L1$ does not know when to send y to ensure that this is done only after x has reached $L2$ (the `strict` interaction operator requires that all events in one interaction operand occur before all the events in the next operand). The early emission of y can then lead to invalid (*unintended*) traces with $!y$ before $?x$. On the other hand, the above error may not be locally observable, since, based on local knowledge only, the invalid execution trace $[!x, !y, ?x, ?y]$ is *locally uncheckable*. This problem may be overcome by adding a coordinating message c between $L2$ and $L1$, so that $L1$ knows when it can send message y. The communication overhead of this solution (1 message) is again smaller than the overhead of a centralized testing approach. Alternatively, the problem may be overcome by adding *coordination time constraints* in a way similar to scenario a) (with the difference that, in this case, the ordering we want to enforce is between events in different lifelines).

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

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

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

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

- if the analysis shows that a test scenario is locally observable and locally controllable, then it can be safely executed in a purely decentralized way;

- if the analysis shows that a test scenario is not locally controllable, a decentralized execution is not safe, because the local testers may inject inputs that are locally valid but not globally valid, leading to failed test executions. Failed test executions caused by such faulty inputs should be discarded by the central tester;

- if the analysis shows that a test scenario is locally controllable but not locally observable (as in scenario *b* above), then it can still be executed safely in a decentralized way, requiring only that events observed by the local testers are communicated to the central tester at the end of test execution to arrive at a final verdict (at the cost of delayed error detection, complicated by non-synchronized clocks);
- in many cases, local observability and controllability problems are associated with incomplete specifications or design flaws (Mitchell, 2005), so the analysis helps identifying the needed refinements;
- in other cases, the analysis helps identifying timing constraints or coordination messages to insert manually or automatically to enforce local observability or, at least, local controllability, with a lower communication overhead than a centralized testing approach, and hence solve the test coordination problem in a effective and efficient way.

5.2 Semantics of Basic UML SDs

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

In this section we handle basic UML SDs, without time constraints. Time-constrained UML SDs will be addressed in Section 5.3.

In this thesis, we formalize the structure and semantics of UML SDs using the VDM formal specification language (Fitzgerald et al., 2005; Larsen et al., 2016). This allows executing and validating the specification with a support tool, such as Overture (`http://overturetool.org/`). In this section, we start by presenting a higher-level description of the structure and semantics of UML SDs. The organization of the VDM++ specifications is described in Chapter 6; the complete specifications can be found in Appendix B.

5.2.1 Valid Global Traces Defined by a UML Sequence Diagram

In UML, an SD is a variant of an Interaction (OMG, 2017). Figure 5.3 describes the structure of Interactions based on the UML metamodel (OMG, 2017). For simplicity, we omit the applicable integrity constraints and the definition of some basic types. An interaction comprises a set of one or more lifelines (representing in our case CUTs or actors), a set of one or more messages, and a set of zero or more combined fragments (restricted in this thesis to the most common ones). A combined fragment covers a subset of lifelines and is divided in a sequence of one or more interaction operands; the semantics is determined by the interaction operator. Co-regions may be represented by `par` combined fragments covering the desired message ends, so we allow messages to cross boundaries of combined fragments. In each lifeline, the ordering of message ends (`sendEvent` and `receiveEvent` in Figure 5.3) and start/finish boundaries of combined fragments and interaction operands (startLocations and finishLocations in Figure 5.3) is represented by assigning

sequential natural numbers to their locations. Combined fragments with the `opt` and `loop` operator have only one operand. Operands of combined fragments with the `opt`, `loop` and `alt` operator can have a guard. Most messages exchanged in a distributed system are asynchronous, but synchronous messages may also be used to represent local interactions.

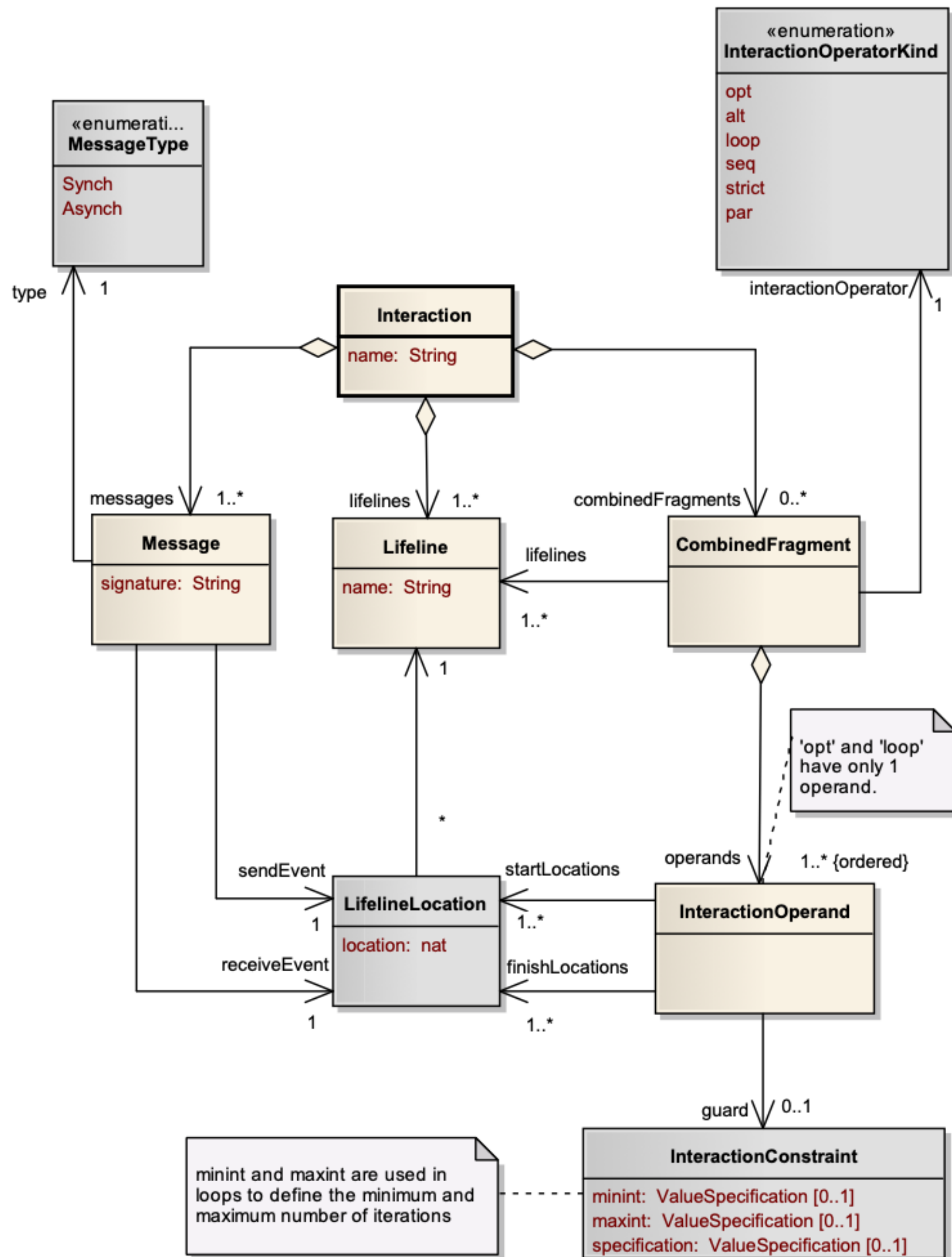


Figure 5.3: Interactions

In UML, the semantics of an Interaction is expressed in terms of sets of *valid* and *invalid*

traces (OMG, 2017). In this thesis, we do not handle the rarely used constructs for defining invalid traces (such as the *neg* interaction operator), so only the valid traces are relevant here.

In general, a trace is a sequence of *event occurrences* (OMG, 2017), corresponding to the sending or receiving of messages at lifelines. We represent an *event* by a tuple $\langle m, l, k \rangle$, where m is the message, l is the lifeline where the event occurs and k is the event kind (*Send* or *Receive*). For example, the event $e1$ shown in Figure 5.4 may be represented by the tuple $\langle$*"fall_signal", "Care Receiver", Send*$\rangle$.

We formalize the semantics of an Interaction I by a function $V(I)$ computed as describe in Procedure 5.1. The details of some auxiliary functions are omitted for simplifying the presentation.

Procedure 5.1 (Valid traces of an interaction)
Inputs:

- I - interaction (or interaction operand) with top-level messages $m_1, ..., m_k$ $(k \geq 0)$ and top-level combined fragments $F_1, ..., F_n$ $(n \geq 0)$;

Outputs:

- $V(I)$ - set of valid traces defined by I;

Computation:

- $V(I) = \bigcup \{w([s(m_1), r(m_1)], ..., [s(m_k), r(m_k)],\ t_1, ..., t_n) \mid t_1 \in V(F_1), ..., t_n \in V(F_n)\}$, where $s(m_i)$ and $r(m_i)$ denote the emission and reception events of message m_i, and w denotes the set of weak sequencing interleavings of the traces passed as arguments.

The semantics of each type of combined fragment is defined in Procedure 5.2. The expansion of each operand is similar to the expansion of the top-level diagram.

Procedure 5.2 (Valid traces of a combined fragment)
Inputs:

- F - combined fragment, with interaction operator f and interaction operands o_1, ..., o_n, also denoted $f(o_1, ..., o_n)$, with $n \geq 1$ ($n = 1$ for *opt* and *loop*);

Outputs:

- $V(F)$ - set of valid traces defined by F;

Computation:

- $V(\mathbf{alt}(o_1, ..., o_n)) = V(o_1) \cup ... \cup V(o_n)$;

- $V(\mathbf{opt}(o_1)) = V(o_1) \cup \{[]\}$, where [] is the empty trace;
- $V(\mathbf{strict}(o_1,...,o_n)) = \{t_1 \frown ... \frown t_n \mid t_1 \in V(o_1),...,t_n \in V(o_n)\}$;
- $V(\mathbf{par}(o_1,...,o_n)) = \bigcup\{p(t_1,...,t_n) \mid t_1 \in V(o_1),...,t_n \in V(o_n)\}$, where $p(t_1,...,t_n)$ denotes the set of parallel interleavings of the traces $t_1,...,t_n$;
- $V(\mathbf{seq}(o_1,...,o_n)) = \bigcup\{w(t_1,...,t_n) \mid t_1 \in V(o_1),...,t_n \in V(o_n)\}$, where $w(t_1,...,t_n)$ denotes the weak sequencing interleavings of the traces $t_1,...,t_n$.
- $V(\mathbf{loop_{m,n}}(o_1)) = V(seq(o_1(1),...,o_1(m))) \cup ... \cup V(seq(o_1(1),...,o_1(n)))$, where $o_1(i)$ denotes the i-th copy of o_1, and m and n denote the minimum and maximum number of iterations ($0 \leq m \leq n$);

The *parallel interleaving* $p(a,b)$ of two traces a and b, is given by the set of permutations of events from a and b that respect the order of events per trace. E.g., $p([a_1,a_2],[b_1,b_2]) = \{[a_1,a_2,b_1,b_2], [a_1,b_1,a_2,b_2], [a_1,b_1,b_2,a_2], [b_1,a_1,a_2,b_2], [b_1,a_1,b_2,a_2], [b_1,b_2,a_1,a_2]\}$.

The *weak sequencing interleaving* $w(a,b)$ of two traces a and b, is given by the set of permutations of events from a and b that, in addition, respect the order of events per lifeline (for events from different traces). E.g., assuming the ordering of events per lifeline as $\{l_1 \mapsto [a_1,b_1], l_2 \mapsto [a_2], l_3 \mapsto [b_2]\}$, then $w([a_1,a_2],[b_1,b_2]) = \{[a_1,a_2,b_1,b_2], [a_1,b_1,a_2,b_2], [a_1,b_1,b_2,a_2]\}$.

In the presence of loops or multiple messages with the same signature and lifelines, the same event may occur multiple times in a trace. To distinguish those occurrences when computing weak sequencing interleavings, it is necessary to keep extended information with each event, namely the event location, unique message identifier, and an iteration counter (besides the event type, message signature and lifeline).

Most messages exchanged in a distributed system are asynchronous, but synchronous messages may also be used to represent local interactions. We model the emission and reception of a synchronous message as a pair of events, but, since those events occur simultaneously, they need to be considered as an atomic pair when computing interleavings.

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

The operands of `alt` and `opt` combined fragments represent behaviors that are selected for execution non-deterministically and/or according to the values of guard conditions. In this thesis, we don't take into account possible guard conditions defined, because they shouldn't limit the set of possible traces but only the conditions upon which each trace may be selected. Hence, the sets of valid traces are given simply by the expressions described in Procedure 5.2.

To keep the model executable, in this thesis we restrict our attention to loops that have defined minimum and maximum numbers of iterations, so the operand of a loop represents a behavior that may be repeated a number of iterations chosen non-deterministically and/or according to a guard condition between the specified limits. Consecutive iterations are combined in a weak sequencing

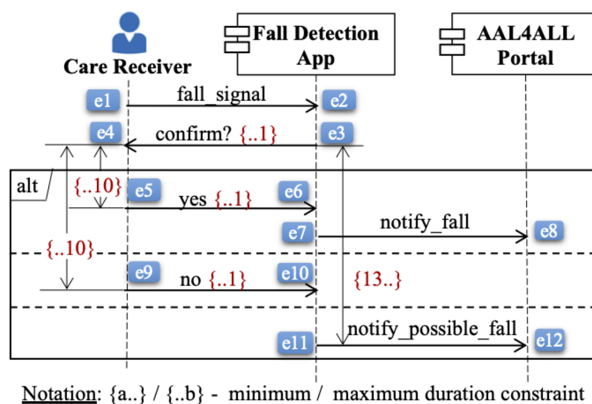


Figure 5.4: Example of a fall detection scenario (simplified) from an ambient assisted living ecosystem (AAL4ALL).

way as specified in the UML standard. However, the theoretical approach is also applicable for unbounded loops.

If we look at example d) in Figure 5.2, the complete set of global valid traces is (omitting the lifeline of each event):

$\{[!x, ?x, !y, ?y]$, $[!x, !y, ?x, ?y]$, $[!x, !y, ?y, ?x]$, $[!y, !x, ?y, ?x]$, $[!y, !x, ?x, ?y]$, $[!y, ?y, !x, ?x]$, $[!z, ?z, !w, ?w]$, $[!z, !w, ?z, ?w]$, $[!z, !w, ?w, ?z]$, $[!w, !z, ?w, ?z]$, $[!w, !z, ?z, ?w]$, $[!w, ?w, !z, ?z]\}$

5.3 Semantics of Time-constrained UML SDs

In this section we characterize the syntax and semantics of time-constrained UML SDs. First, we analyze how time is handled in UML SDs, and subsequently show how to compute the valid traces.

5.3.1 Time-constrained Sequence Diagrams

SDs may be annotated with *time constraints* (OMG, 2017), as illustrated by the SD of Figure 5.4. Although the UML standard allows the specification of more complex constraints, in this thesis we restrict our attention to the types of time constraints that are commonly addressed in the literature and are most relevant in practice: constraints that specify the minimum and maximum durations between two events (message sending or receiving) in the same lifeline, or between the sending and receiving of a message between two lifelines.

5.3.2 Timed Traces

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

in $[\langle e1,1\rangle,\langle e2,5\rangle,\langle e3,8\rangle]$ (Later on, in VDM++, we represent the time instant of an event as a field of the event, for convenience of implementation.).

5.3.3 Time-constrained Traces

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

A tc-trace is a pair of a trace and an associated Boolean expression on *time constraints between pairs of event occurrences*. In those constraints, the time instance of the i-th event occurrence is represented by the *time variable* τ_i. The time constraints are normalized as a conjunction of *difference constraints* Cormen et al. (2009) of the form $\tau_i - \tau_j \leq d$, where d is a time duration literal (positive or negative integer).

For example, the SD of Figure 5.4 defines the following valid tc-traces:

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

5.3.4 Valid Traces and Satisfiability Checking

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

Procedure 5.3 (Valid time-constrained traces).
Inputs:

- Time-constrained interaction ι;

Outputs:

- $\mathscr{V}(\iota)$ - set of valid tc-traces defined by ι;

Steps:

1. Compute the set $\mathscr{U}(\iota)$ of valid (untimed) traces defined by ι ignoring time constraints, following the procedure described described in Section 5.3 (this set gives all the possible event combinations and total orderings defined by ι);
2. Obtain the set $\mathscr{D}(t,\iota)$ of time constraints applicable to each trace t in $\mathscr{U}(\iota)$ (see Procedure 5.4);
3. Determine the satisfiability of those constraints (*sat*), and select the traces with satisfiable constraints (see Procedure 5.5).

Formally,

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

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

Procedure 5.4 (Applicable time constraints). Generates a conjunctive expression with time constraints between time instants of event occurrences in a (untimed) trace t of an interaction ι, based on the constraints defined between pairs of events in ι.

Inputs:

- Time-constrained interaction ι with a set of time constraints $timeConstr(\iota)$;
- (untimed) trace t;

Outputs:

- Conjunctive expression $D(t,\iota)$;

Formula:

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

A set of time constraints c is satisfiable for a trace t if there is an assignment of non-decreasing time instants to the event occurrences in t that satisfies all the constraints in c. Due to the special nature of the time constrains involved (a conjunction of difference constraints), satisfiability can be checked in polynomial time, following the procedure summarized in Procedure 5.5 (partly based on Cormen et al. (2009)) and illustrated in Figure 5.5. The example refers to a trace derived from the SD of Figure 5.4 that is valid when the time constraints are ignored but is invalid otherwise. In the case of a more general Boolean expression on difference constraints, as we will need later, we reduce the expression to disjunctive normal form (DNF), and apply the same procedure to each conjunctive term.

Procedure 5.5 (Satisfiability checking). Checks if a conjunctive expression E on time constraints is satisfiable ($sat(E)$), i.e., there is an assignment of non-decreasing values to the time variables referenced in E that makes the expression $true$, as follows:

Inputs:

- Conjunctive expression E;

Outputs:

- True - expression E is satisfiable
- False - expression E is not satisfiable

Steps:

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

The detection of cycles with negative weight can be performed by applying the Bellman-Ford algorithm (Bellman, 1958) for finding the shortest paths from a single source vertex to all of the other vertices in a directed graph with edges of negative weight. In the implementation, the implicit ordering constraints are not actually added to the set of constraints. Instead, the Bellman-Ford algorithm is slightly modified.

5.3.5 Operators on Timed Traces and Time-constrained Traces

The definitions and procedures for local observability and controllability analysis use the operators defined in Figure 5.6. For the sake of simplicity of presentation, implicit (instead of explicit) definitions are given for some operators, resorting to a function (*ext*) that gives the (possibly infinite) set of timed traces defined by a set of tc-traces. We also apply the *ext* function to complex structures (such as maps), in order to convert all occurrences of sets of tc-traces to corresponding sets of timed traces. By a *feasible* timed trace (see the join operator), we mean a timed trace with non-decreasing time instants that respects the fact that messages can be received only after being sent. Application examples can be found in Figure 5.8. Full details can be found in the Appendix B.

5.3.6 Conformance Checking Based on Distributed Observations

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

Are time constraints satisfiable for the trace [e1, e2, e3, e11, e4, e12]?

1. Determine the time constraints applicable to the event occurrences (D(t, ι)), add implicit ordering constraints between the referenced time variables, and normalize to a set of difference constraints of the form $\tau_i - \tau_j \leq d$.

$$\begin{cases} \tau_5 - \tau_3 \leq 1 \\ \tau_4 - \tau_3 \geq 13 \\ \tau_3 \leq \tau_4 \leq \tau_5 \end{cases} \Rightarrow \begin{cases} \tau_5 - \tau_3 \leq 1 \\ \tau_3 - \tau_4 \leq -13 \\ \cancel{\tau_3 - \tau_4 \leq 0} \\ \tau_4 - \tau_5 \leq 0 \end{cases}$$

τ_i - time instant of the *i*-th occurrence in the trace

2. Build the corresponding difference constraint graph G

3. Check if there are cycles with negative weight in G

There is a cycle with negative weight (-12), so the time constraints are not satisfiable for this trace.

Figure 5.5: Satisfiability checking example (trace from Figure 5.4).

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

Figure 5.6: Operators on timed traces and time-constrained traces.

However, in distributed testing, global traces are not directly observed, but only the local traces observed at each lifeline, which raises the need to infer global traces from the observed local traces, possibly leading to inconclusive verdicts.

In distributed testing, the time instant of each observed event occurrence is measured with the local clock of the respective lifeline. Although it is impossible to ensure perfect clock synchronization between lifelines, in practice the difference between the readings of any two clocks (*clock skew*) may be limited to a small value of the order of 10ms over Internet and below 1ms over LAN (Mills, 1991). This clock skew might not have practical impact for coarser time scales used in testing (e.g., seconds), but could be relevant for finer time scales (e.g., milliseconds). In any case, for test cases that run for short time spans, one can assume that there is no noticeable *clock drift* during test execution (i.e., clocks run at the same rate).

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

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

We present below the procedure to perform a final conformance check globally by the central tester.

Procedure 5.6 (Final conformance checking based on distributed observations).

Inputs:

- Time-constrained interaction ι;
- Observed trace $\mathscr{O}$, represented by a mapping from lifelines to the timed trace observed at each lifeline, with time instants measured with the local clocks;
- Maximum clock skew $\mathscr{M}$, indicating the maximum difference between the readings of any two clocks (at different lifelines).

Outputs:

One of the following verdicts, assuming the given maximum clock skew, and no noticeable clock drift:

- *Pass* - the observed trace conforms to the specification (as set by ι);
- *Fail* - the observed trace does not conform to the specification;

- *Inconclusive* - the observed trace may conform or not to the specification.

Steps:

1. Compute the set $\mathscr{V}(\iota)$ of *valid tc-traces* defined by ι (with time constraints defined with respect to an ideal global clock);
2. Denoting by δ_k the (unknown) difference between the readings of the clock at lifeline k and an ideal global clock, generate a conjunctive expression C for the *maximum clock skew constraints*, with a difference constraint $\delta_i - \delta_j \leq \mathscr{M}$ for each ordered pair i, j of lifelines;
3. Compute the set J of *feasible joins* of the observed timed traces per lifeline, considering the given max clock skew (formally, $\bowtie \mathscr{O}$);
4. For each possible join $t \in J$ and valid tc-trace $v \in \mathscr{V}(\iota)$, determine a *verdict* $ver(t,v)$ for the pair t,v as follows:
   (a) If the *untimed traces* corresponding to t and v are different, discard the pair t, v (the same as concluding a *Fail* verdict);
   (b) Denoting by
      - τ_k - time variable for the k-th event in t (defined in terms of an ideal global clock),
      - ψ_k - time instant of the k-th event in t (measured with the local clock), and
      - l_k - lifeline where the k-th event in t occurs,

      consider each *intra-lifeline constraint* $\tau_i - \tau_j \leq d$ in v with $l_i = l_j$, and check if $\psi_i - \psi_j \leq d$ holds. If one of these constraints does not hold, discard the pair t, v (the same as concluding a *Fail* verdict). If all the constraints in v are intra-lifeline constraints and all hold, conclude a *Pass* verdict for the pair t, v;
   (c) Generate a conjunctive expression D for the *inter-lifeline constraints* in v, by considering each constraint $\tau_i - \tau_j \leq d$ in v with $l_i \neq l_j$, and replacing τ_k by $\psi_k + \delta_{l_k}$, originating a modified difference constraint (with variables on the left-hand side and constants on the right-hand side) $\delta_{l_i} - \delta_{l_j} \leq d + \psi_j - \psi_i$;
   (d) If $sat(C \wedge D) \wedge \neg sat(C \wedge \neg D)$, then $ver(t,v)$ is *Pass*;
   (e) If $\neg sat(C \wedge D) \wedge sat(C \wedge \neg D)$, then $ver(t,v)$ is *Fail*;
   (f) Otherwise, $ver(t,v)$ is *Inconclusive*.
5. Determine the *final verdict* as follows:
   (a) If for every $t \in J$ and every $v \in \mathscr{V}(\iota)$, $ver(t,v) = Fail$, then the final verdict if also *Fail*;
   (b) If for every $t \in J$ there is at leat one $v \in \mathscr{V}(\iota)$ such that $ver(t,v) = Pass$, then the final verdict if also *Pass*;
   (c) Otherwise, the final verdict is *Inconclusive*.

Formally (for the outer steps only),

$FCC(\mathscr{O}, \iota, \mathscr{M}) \triangleq$
 if $\forall t \in J, \forall v \in \mathscr{V}(\iota) \cdot ver(t, v) = Fail$ then $Fail$
 else if $\forall t \in J, \exists v \in \mathscr{V}(\iota) \cdot ver(t, v) = Pass$ then $Pass$
 else $Inconclusive$.

Examples of traces yielding different verdicts are shown in Figure 5.7.

#	Observed Local Traces (locally valid)			Possible Joins * (with first problematic event underlined)	Verdict *
	Care Receiver	Fall Detection App	AAL4ALL Portal		
1	[(e1, 0), (e4, 4200), (e5, 14200)]	[(e2, 2000), (e3, 4000), (e6, 14500), (e7, 14600)]	[(e8,16000)]	[(e1, 0), (e2, 2000), (e3, 4000), (e4, 4200), (e5,14200), (e6,14500), (e7,14600), (e8,16000)]	Pass
2	[(e1, 0), (e4, 4200), (e5, 14200)]	[(e2, 2000), (e3, 4000), (e6, 15200), (e7, 15600)]	[(e8,16000)]	[(e1, 0), (e2, 2000), (e3, 4000), (e4, 4200), (e5,14200), (e6,15200), (e7,15600), (e8,16000)]	Inconclusive (a)
3	[(e1, 0), (e4, 4200), (e5, 14200)]	[(e2, 2000), (e3, 4000), (e6, 18000), (e7, 18600)]	[(e8,19000)]	[(e1, 0), (e2, 2000), (e3, 4000), (e4, 4200), (e5, 14200), (e6, 18000), (e7,18600), (e8,19000)]	Fail (b)
4	[(e1, 0), (e4, 16800)]	[(e2, 2000), (e3, 4000), (e11, 17000)]	[(e12,18000)]	[(e1, 0), (e2, 2000), (e3, 4000), (e4,16800), (e11,17000), (e12,18000)] [(e1, 0), (e2, 2000), (e3, 4000), (e11, 17000), (e4,16800), (e12,18000)]	Fail (c)

* Assuming *MaxClockSkew* = 500ms
(a) Time constraint between *e5* and *e6* (1000ms) possibly not respected.
(b) Time constraint between *e5* and *e6* (1000ms) not respected.
(c) Time constraint between *e3* and *e4* (1000ms) not respected.

Figure 5.7: Examples of traces with different conformance checking verdicts in the fall detection scenario

5.4 Local Observability Analysis

In this section, we present procedures to check if conformance checking of observed execution traces against the expectations set by a time-constrained SD under consideration can be performed by the local testers alone based on the events observed locally, without the need to communicate those events to the central tester to ensure that the final test verdict is correct (*local observability*).

A preliminary version of the procedures presented in this section were presented in (Lima and Faria, 2017) for SDs without time constraints.

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

5.4.1 Definition

Definition 5.1 (Local observability). We say that a test scenario specified by a time-constrained interaction ι is *locally observable* iff there are no feasible timed traces that are locally valid but are not globally valid (also called *locally uncheckable* traces). We say that a timed trace t is *globally valid* when $t \in ext(\mathscr{V}(\iota))$, and is *locally valid* when $\forall_{l \in \mathscr{L}(\iota)}, \pi_l t \in ext(\pi_l \mathscr{V}(\iota))$, where $\mathscr{L}(\iota)$ denotes the lifelines in ι.

5.4.2 Local Observability Checking

Procedure 5.7 (Local observability checking). We check the *local observability* of a test scenario described by a time constrained interaction ι in a constructive way (pinpointing violations), as follows:

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

Formally,

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

Theorem 5.1 (Correctness of Procedure 5.7). Procedure 5.7 correctly checks if an interaction ι is locally observable.

Proof. Follows from Definition 4.1 and from the definitions of the operators involved in Procedure 5.7. Based on the meaning of the difference operator (see Figure 5.6), the right-hand side of the formula in Procedure 5.7 can be rewritten:

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

Based on the definitions of the join operator (see Figure 5.6), the first term $(t \in ext(\bowtie (\pi_{\mathscr{L}(\iota)} \mathscr{V}(\iota))))$ can be rewritten:

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

This corresponds to the definition of local validity in Definition 5.1, whilst the second term $(t \notin ext(\mathscr{V}(\iota)))$ corresponds to the negation of global validity. Hence, we conclude that Procedure 5.7 correctly checks local observability.

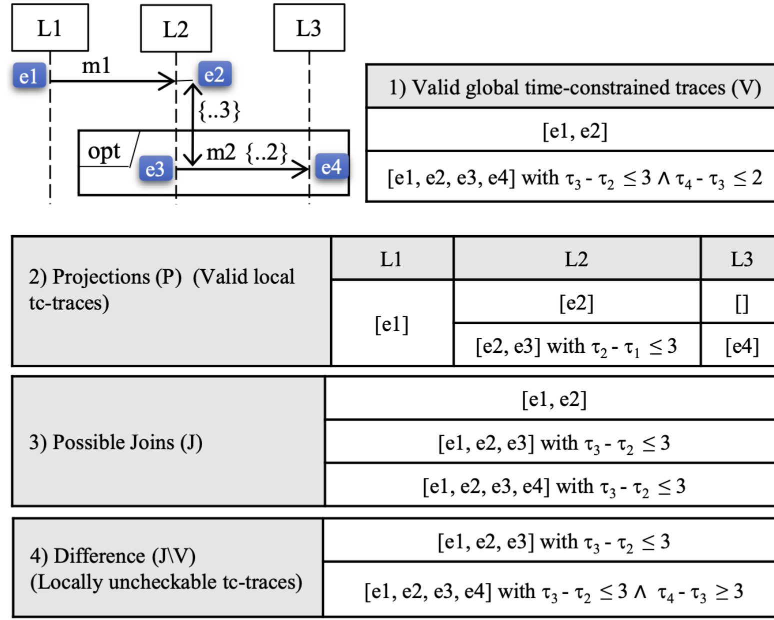


Figure 5.8: Example of local observability checking.

Example 5.1 Procedure 5.7 is illustrated in Figure 5.8. In this case, there are two tc-traces that are not locally checkable, so the scenario is not locally observable. The first one is due to an optional message without a corresponding acknowledgment message. The second one is due to an inter-lifeline time constraint (transmission constraint) that is not present in the projections onto the lifelines.

5.4.3 Impact of Non-synchronized Clocks

Next we show that imperfect clock synchronization in a distributed SUT does not affect local observability.

As previous explained in Section 5.3.6, although it is impossible to ensure perfect clock synchronization between lifelines, in practice the difference between the readings of any two clocks (clock skew) may be limited to a small value.

Under this assumption, we next prove our proposition.

Theorem 5.2 (Local observability and clock synchronization). If an interaction ι is locally observable with perfectly synchronized clocks, then it is also locally observable if clocks are not perfectly synchronized but run at the same rate.

Proof. Let us assume that ι is locally observable, i.e., all invalid feasible global traces are also locally invalid, in case the clocks are perfectly synchronized. Let us pick one arbitrary of those invalid feasible global traces t, and let us denote by t_k an invalid local trace observed at a lifeline k (based on our assumption, such lifeline and trace must exist). In case clocks are not perfectly synchronized but run at the same rate, the time instants of the corresponding observed local trace

t'_k in lifeline k will differ from the time instants in t_k by a constant amount (the clock skew δ_k of lifeline k). Since all the local time constraints we are considering are difference constraints, shifting time instants by the same amount will not affect the validity of those constraints. So t'_k will also be checked as invalid by lifeline k. Hence we conclude that invalid traces will also be detected locally.

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

5.5 Local Controllability Analysis

5.5.1 Definitions and Example

Definition 5.2 (Local controllability). We say that a time constrained interaction ι is locally controllable if no invalid timed traces are generated (i.e., there are no *unintended traces*) and all valid timed traces can be generated (i.e., there are no *missing traces*) when the lifelines and the communication channels behave in a locally correct way, using local knowledge only. Formally, denoting by $\mathscr{S}(\iota)$ the set of feasible timed traces that can be generated when the lifelines and the communication channels behave in a locally correct way, ι is locally controllable iff $\mathscr{S}(\iota) = ext(\mathscr{V}(\iota))$. Unintended traces are given by $\mathscr{S}(\iota) \setminus ext(\mathscr{V}(\iota))$. Missing traces are given by $ext(\mathscr{V}(\iota)) \setminus \mathscr{S}(\iota)$.

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

Example 5.2 The scenario of Figure 5.4 is locally controllable. In fact, the projection of the defined time constraints onto the “Fall Detection App” lifeline generates the derived local constraints $time(e6) \leq time(e3) + 12$ and $time(e10) \leq time(e3) + 12$. So, the lifeline knows that, after requesting confirmation from the user (event $e3$), it should wait for a response (events $e6$ or $e10$) up to 12 time units, and only send “notify_possible_fall” after at least one more time unit. Without the specified constraints, the scenario would not be locally controllable, because the lifeline would not know how much time to wait before sending “notify_possible_fall”. This could result in the generation of invalid traces such as:

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

We next clarify and formalize the notion of a locally correct behavior of lifelines, in Definition 5.3, and communication channels, in Definition 5.4. The set $\mathscr{S}(\iota)$ contains all feasible timed traces that satisfy the conditions of 5.3 and 5.4.

Definition 5.3 (Locally correct behavior of lifelines). A global timed trace t in an interaction ι demonstrates a locally correct (and complete) behavior of a lifeline $l \in \mathscr{L}(\iota)$ iff the local timed trace $p = \pi_l t$ observed at l satisfies the following conditions:

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

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

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

(b) l may remain in a *quiescent* state after p (i.e., not send any output, at least without first receiving an input (Hierons, 2012)), because one of the following holds (denoted $Q_l(p)$):

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

(ii) in case there are valid outputs that can be sent after p, there are also valid inputs that can be received with a deadline greater or equal than the deadline for the outputs (in this case, l may decide to wait for input, and, if it does not arrive up to the deadline, will no longer be able to send any output);[1] formally,

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

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

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

Definition 5.4 (Correct behavior of communication channels). A global timed trace t in an interaction ι demonstrates a correct (and complete) behavior of the communication channels iff the following conditions hold:

(a) messages are delivered within the specified transmission duration constraints (between pairs of related emission and reception events in t);

(b) all messages are delivered (i.e., for each emission event in t there is a corresponding reception event).

5.5.2 Symbolic Simulation

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

[1] This policy is important to guarantee that all valid traces can be generated, and hence prevent missing traces (at the possible cost of generating unintended traces).

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

Procedure 5.8 (Symbolic simulation). Computes a set $\mathscr{S}'(\iota)$ of tc-traces describing the timed traces that can be generated by the execution of an interaction ι when the lifelines and communication channels behave in a locally correct way, as follows:

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

where

- $\mathscr{T}_\iota(\langle u,c\rangle)$ is a *transition function* that gives the successors of tc-trace $\langle u,c\rangle$ (a pair of a trace u and a constraint c) in the symbolic execution tree of ι, by appending time-constrained emission or reception events generated according to conditions 5.2.a) or 5.3.a), in a proper temporal ordering. This ordering is determined by computing the earliest deadline D among all emission deadlines, for lifelines that are not in a quiescent state, and delivery deadlines, for messages in transit. When working with tc-traces, D is in fact a constraint on the time instants of the next event and previous events. For each candidate time-constrained event $\langle e,c'\rangle$ to append to $\langle u,c\rangle$, if the conjunction $c \wedge c' \wedge D$ is satisfiable, then the event is selected, generating the tc-trace $\langle u \frown [e], c \wedge c' \wedge D\rangle$.

- $\mathscr{T}_\iota^*(\langle [], true\rangle)$ denotes the set of tc-traces reachable from the empty tc-trace $\langle [], true\rangle$ by 0 or more applications of $\mathscr{T}_\iota$ (reflexive transitive closure);

- $Q_\iota(\langle u,c\rangle)$ denotes the condition (on the time variables of events in u) upon which the system may remain quiescent after the occurrence of $\langle u,c\rangle$, as set by conditions 5.2.b) and 5.3.b). If $Q_\iota(\langle u,c\rangle)$ is satisfiable, $\langle u,c\rangle$ is added to the result, further restricted by $Q_\iota(\langle u,c\rangle)$.

Theorem 5.2 (Correctness of Procedure 5.8). Procedure 5.8 correctly computes $\mathscr{S}'(\iota)$.

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

An example of an execution tree and possible quiescent tc-traces generated by the application of Procedure 5.8 is shown in Figure 5.9.

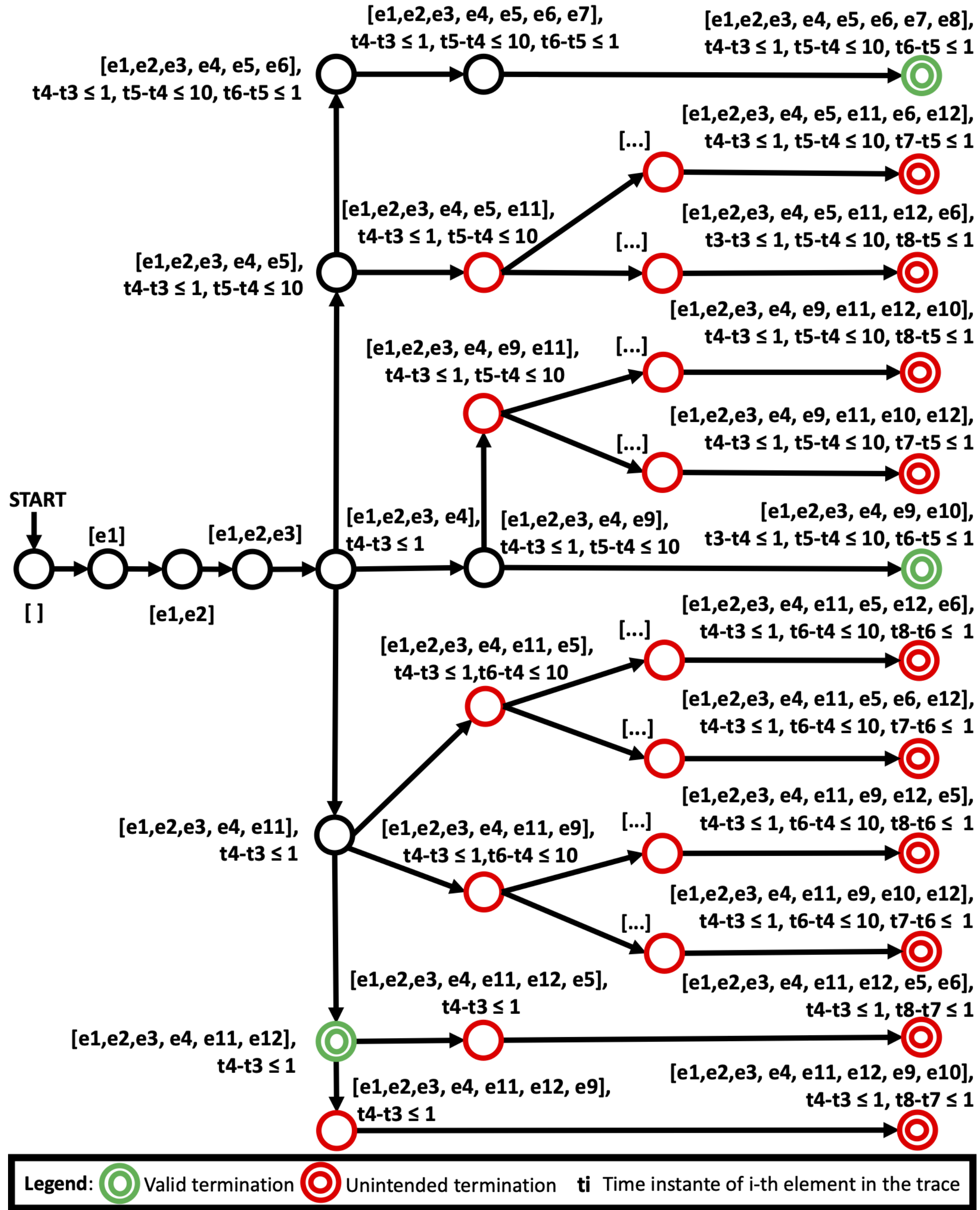


Figure 5.9: Example of symbolic execution for the SD of Figure 5.4 without the "{13..}" time constraint.

5.6 Local Observability and Controllability Enforcement

As illustrated by the examples in Section 5.1, many observability and controllability problems can be solved by the addition of coordination messages or coordination time constraints. Hence, in this section, we present algorithms to search for coordination messages or coordination time constraints to enforce local observability and/or local controllability of an interaction ι, whilst preserving the traces valid locally at each lifeline (apart possibly from timing constraints).

An heuristic used to guide the search is based on the intuition that the locations of local observability or controllability problems (locations where the locally uncheckable or unintended traces deviate from valid traces) might suggest points where coordination messages or time constraints need to be inserted.

Therefore, our main algorithm comprises 4 main steps.

Procedure 5.9 (Local observability and controllability enforcement).

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

In the next subsections we describe each of these steps.

5.6.1 Determination of Error Locations

Procedure 5.10 (Determination of error locations).

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

5.6.2 Generation of Coordination Messages

Candidate coordination messages are searched based on the error locations previously determined and common patterns of local observability and controllability violations and solutions.

Procedure 5.11 (Generation of coordination messages).

1. Determine a set $\mathscr{C}_1$ of coordination messages to fix local observability and controllability violations due to non-local choices, proceeding as follows for each `alt` combined fragment:
   1.1. For each operand and lifeline, check if the first event in the lifeline is an emission event and, if so, collect that event. Let $\mathscr{E}$ be the resulting set of events.
   1.2. From the set $\mathscr{E}$, pick a "master" lifeline l, by selecting the lifeline of an arbitrary event in $\mathscr{E}$ that occurs in the first operand.
   1.3. For each event e in $\mathscr{E}$ that occur in a lifeline other than l and corresponds to an error location, generate a coordination message having as target location the location immediately before e, and as source location a location in l selected as follows:
      - If there is an event e' in $\mathscr{E}$ that occurs in l in the same operand of e, pick the location immediately after e';
      - Otherwise, pick a location immediately after the start location of the operand of e in l.
2. Determine a set $\mathscr{C}_2$ of coordination messages to fix race conditions and other event ordering problems, proceeding as follows:
   2.1. Find triples of events $e1$, $e2$, and $e3$ such that: (i) $e3$ corresponds to a remaining error location; (ii) $e2$ is the emission event corresponding to $e3$ (possibly $e2 = e3$); (iii) the triple occurs by that order in a valid trace t; (iv) there are no other events between $e1$ and $e3$ in t in the lifeline of $e1$ or $e3$; (v) there is no causal chain of events in t between $e1$ and $e2$ (see discussion of causal chains of events in Section 5.6.3); and (vi) $e2$ does not occur after $e3$ in any valid trace.
   2.2. For each triple, generate a coordination message having as source location a location immediately after $e1$ and as target location a location immediately before $e2$.
3. Determine a set $\mathscr{C}_3$ of coordination messages to fix observability problems caused by optional messages without corresponding acknowledgment messages (typically inside `opt` or `loop` combined fragments), proceeding as follows:
   3.1. Find all asynchronous messages such that: (i) the reception event corresponds to a remaining error location; (ii) it has a variable number of occurrences among the set of all valid traces.
   3.2. For each selected message m, generate a coordination message (acknowledgment message) in the opposite direction, immediately after m.

4. The set $\mathscr{C}$ of coordination messages is the union of the three sets.

5.6.3 Generation of Coordination Time Constraints

As illustrated in Section 5.1, several controllability problems (such as race conditions and inter-lifeline event ordering constraints) may be solved by adding coordination time constraints that impose an ordering between pairs of events. In fact, lifelines may coordinate their actions by dynamically exchanging coordination messages or by statically 'agreeing' on an adequate timing for their actions.

The pattern of race conditions that our heuristic algorithm looks for and the fix strategy used are illustrated in Figure 5.10.

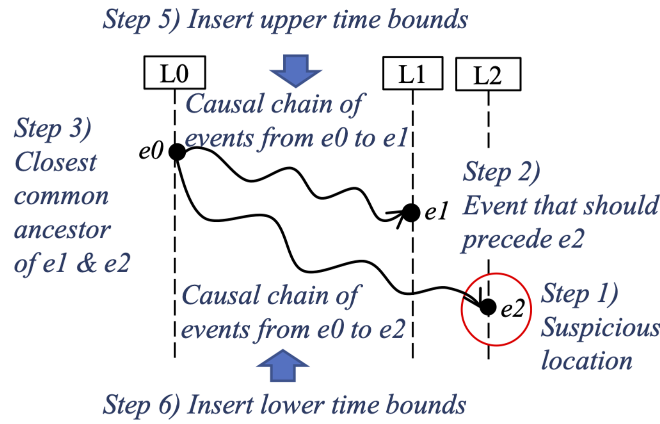


Figure 5.10: Fixing race conditions with coordination time constraints.

We use several trace slicing operations illustrated in Figure5.11, based on the causal dependencies that exist between pairs of emission and reception events, and between all the events that precede an emission event in a lifeline and the emission event itself (assuming the emission decision is taken based on the events previously observed in the lifeline).

Procedure 5.12 (Generation of coordination time constraints to fix race conditions).

1. Take as candidate instances for $e2$ the erroneous or missing events in $\mathscr{E}$, as computed by Procedure 5.10.

2. Take as candidate instances for $e1$ the events that immediately precede $e2$ (without intermediate events from the respective lifelines) in at least one valid trace $t \in \mathscr{V}$, and do not occur after $e2$ in any valid trace.

3. Take as candidate instances for $e0$ the closest common ancestors of $e1$ and $e2$ in the valid traces $t \in \mathscr{V}$ in which both occur (calculated as illustrated in Figure 5.11).

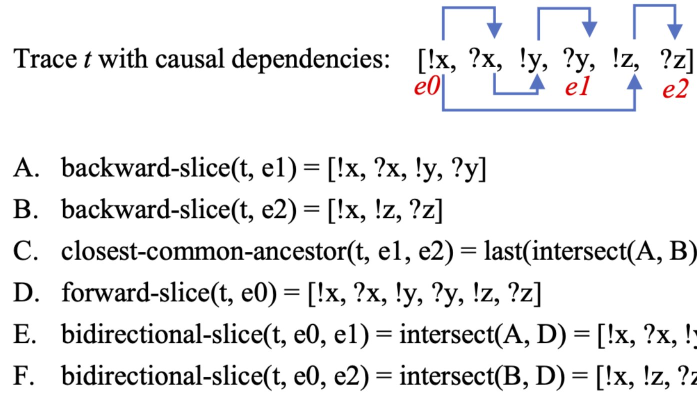


Figure 5.11: Causal dependencies and slicing operations (trace of Figure 5.1a).

4. Discard triples $(e0, e1, e2)$ where $e0 = e1$ or the maximum duration from $e0$ to $e1$ is less than the minimum duration from $e0$ to $e2$ (cases where $e1$ is guaranteed to precede $e2$).

5. Inject upper time bounds between pairs of events in the causal chain of events from $e0$ to $e1$ (bidirectional slice), based on default values for the maximum transmission time (between emission and reception events) and maximum reaction time (between an event in a lifeline and a subsequent emission event).

6. Determine the maximum duration τ from $e0$ to $e1$ that results from step 5, and inject a lower time bound τ+1 (wait time) in the chain of events from $e0$ to $e2$, between an event in a lifeline and a subsequent emission event (giving priority to emissions performed by actors as close as possible to $e0$). If a time bound cannot be injected, the triple $(e0, e1, e2)$ is discarded.

7. Return the set of candidate fixes found, where each candidate fix is a set of time constraints to enforce the ordering between a pair $(e1, e2)$ of events.

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

Procedure 5.13 (Generation of coordination time constraints to fix pairs of mutually exclusive reception and emission events simultaneously enabled).

1. Find pairs of events $e1$ and $e2$ that: (i) occur in the same lifeline, with $e1$ located before $e2$; (ii) are of different types (send and receive); (iii) are mutually exclusive (i.e., there is no valid trace in which both occur); and (iv) may be simultaneously enabled (from the perspective of their lifeline).

2. Perform step 3 as in Procedure 5.12, with the difference that distinct traces $t1$ and $t2$ have to be considered for $e1$ and $e2$, instead of a common trace t.

3. Perform steps 4, 5, 6 and 7 as in Procedure 5.12.

Procedure 5.14 (Generation of coordination time constraints to fix roundtrip constraints not implied by other constraints).

1. Find pairs of events $e0$ and $e1$ that: (i) occur in the same lifeline, with $e0$ located before $e1$; (ii) $e0$ is of type `send` and $e1$ is of type `receive`; (iii) there is a maximum duration constraint between $e0$ and $e1$ that is not begin met; and (iv) there is an error location in $e1$

2. Remove the maximum duration constraint between $e0$ and $e1$, and perform step 5 as in Procedure 5.12

3. If the resulting maximum duration between $e0$ and $e1$ exceed the desire duration discard this pair ($e0$ and $e1$)

5.6.4 Application and Evaluation of Candidate Fixes

Procedure 5.15 (Application and evaluation of candidate fixes).

1. Let $\mathscr{F}$ be the set of candidate fixes, where each fix is a set of coordination time constraints or set of coordination messages.

2. Search for single fix solutions, doing as follows for each candidate fix f (message-set or constraint-set) in $\mathscr{F}$:

   (a) apply the fix f (i.e., insert the message-set or constraint-set in ι), obtaining a new interaction ι';

   (b) determine the set $\mathscr{V}'$ of valid traces defined by ι';

   (c) if the projections of $\mathscr{V}$ and $\mathscr{V}'$ onto the lifelines of ι do not coincide (apart from coordination events and time constraints), discard f;

   (d) if the set $\mathscr{U}'$ of unintended and/or locally uncheckable traces of ι' is empty, find a minimal subset $f*$ of f that is still sufficient to enforce locally observability and/or local controllability, and return $f*$;

   (e) otherwise, if $\#\mathscr{U}'$ (with coordination events removed) is not smaller than $\#\mathscr{U}$, discard f;

3. If a single fix solution was not found, search for multiple fix solutions using a greedy heuristic as follows:

(a) pick the candidate fixes in $\mathscr{F}$ that were not discarded, and rank them by increasing values of $\#\mathscr{U}'$ (with coordination events removed), obtaining a new ordered set $\mathscr{F}'$ of candidate fixes;

(b) for each candidate fix $f \in \mathscr{F}'$, by the defined order, insert f onto ι and execute recursively Procedure 5.15; if a solution is found, return the inserted messages and/or time constraints.

4. If no single or multiple fix solution was found, fail.

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

5.7 Conclusions

In this chapter we presented several examples of simple scenarios to illustrate local observability and controllability problems and ways to overcome them. In order to automatically detect this type of problems, we also presented a set of algorithms that allow the automatic detection of local observability and controllability problems. In this way we can respond to RQ3 as follows:

RQ3 - *How do we determine if a test scenario described by a UML SD can be executed safely in a purely distributed manner, without overlooking conformance faults (false negatives) or injecting conformance faults (false positives) by the test harness? In other words, how do we determine if a test scenario described by a UML SD is locally observable and locally controllable?*

Using the algorithms proposed in Sections 5.4 and 5.5, it is possible to automatically determine whether a given SD is locally observable and/or locally controllable.

As we were able to automatically determine this type of problems, we also present in this chapter algorithms to overcome these problems, which allows us to respond to RQ4 as follows:

RQ4 - *Given a test scenario not locally controllable or locally observable, how can we automatically identify a minimal set of coordination messages and/or coordination time constraints to refine the test scenario and enforce local observability and/or local controllability?*

As we show through the algorithms proposed in Section 5.6, it is possible to overcome local observability and controllability problems, using coordination messages or time constraints. Our algorithms are able to determine this type of fixes automatically for common patterns of local observability and controllability violations.

Chapter 6

Implementation

The algorithms described in Chapter 5 for local observability and controllability analysis and enforcement were implemented in the DCO Analyzer tool.

The tool architecture is described in Section 6.1. Section 6.2 presents the tool back-end. The front-end of the tool is presented in Section 6.3. Section 6.4 presents some usage examples.

6.1 Tool Architecture

DCO Analyzer is an application developed in Java (Oracle, 2019) and VDM++ (Durr and Van Katwijk, 1992) to analyze UML SDs representing distributed systems test scenarios. As depicted in Figure 6.1, the user can use any visual editor of UML SDs (e.g. Papyrus[1]) and then upload the created diagrams to DCO Analyzer.

Internally, DCO Analyzer comprises a front-end, developed in Java, and a back-end, developed in VDM++. The front-end is responsible for receiving and parsing .UML files describing UML SDs, verifying their conformance with the UML metamodel (OMG, 2017), and converting them into the formal representation expected by the back-end (VDM++ data structures). It is also possible to directly provide a .VDMPP file containing the VDM++ data structures; this may be useful to overcome limitations of modeling tools.

6.2 Back-end

VDM++ is an extension of the VDM specification language (VDM-SL) that is one of the longest established model-oriented formal methods for the development of computer-based systems and software. VDM++ supports both the functional style (with types, values and functions' definitions) and the object-oriented style (with classes, instance variables and operations) in a single language. Since the functional style is usually more declarative, most of the data structures and procedures are defined following the functional style (with types and functions). Some procedures are defined using the imperative style (with pure static operations), for performance reasons. Classes are used

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

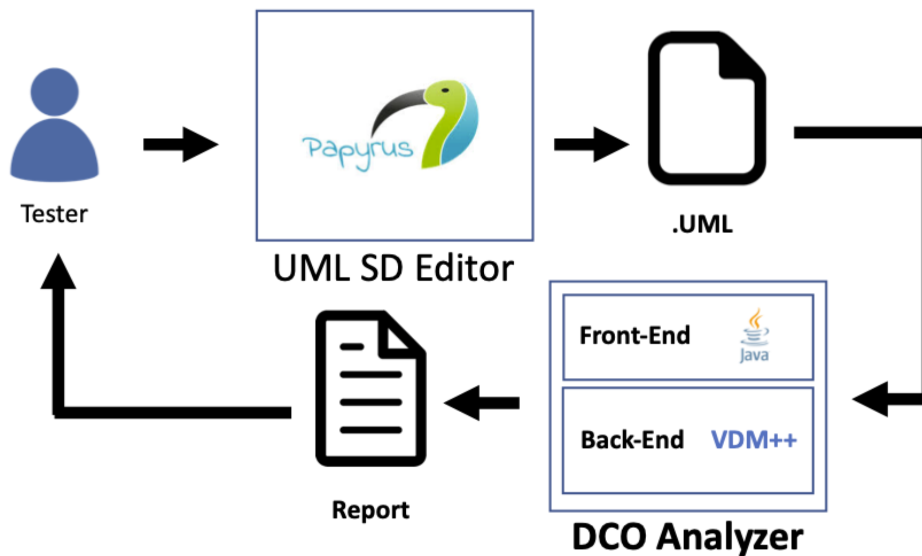


Figure 6.1: DCO Analyzer overview

simply as modules (without instance variables or non-static operations), to group together type definitions, function definitions and some pure static operations. Invariants and pre-conditions are used to formalize applicable constraints.

An advantage of VDM ++ is that it can be converted to Java, so it can be more easily integrated with other tools (Maimaiti, 2011). However, we found that this conversion is not fully compatible, so we decided to keep the backend in VDM++ and solve the problem of integration with Java in the tool's front-end.

As we can see in Figure 6.2, the DCO Analyzer back-end is composed of three packages, `DefinitionAndSemantics`, `ObservabilityAndControlabilityAnalysisAndEnforcement` and `ConformanceCheckingAndInputSelection`.

The `DefinitionAndSemantics` package is responsible for the definitions and semantics of the diagrams and includes four modules, `SequenceDiagrams`, `ValidTraces`, `Traces` and `DifferenceConstraints`. The `SequenceDiagrams` module formalizes the structure and semantics of the SDs that are composed of interactions, lifelines, messages and time constraints. The `ValidTraces` module contains the procedures to determine the set of valid traces of a given SD. Trace operations are implemented in the `Traces` module. Constraint satisfiability procedures are implemented in the `DifferenceConstraints` module.

The `ObservabilityAndControllabilityAnalysisAndEnforcement` package is the main package since it is there that the procedures for the analysis of observability and controllability are implemented, as well as the the procedures to enforce those properties. This package consists of four modules, `SimulatedExecution`, `Observability`, `Controllability` and `Enforcement`. The `SimulatedExecution` module implements the procedure responsible for the simulated execution. The procedures for analyzing local observability are implemented in the `Observability` module. The `Controllability` module is responsible for implementing the

procedures for analyzing local controllability. The procedure for the enforcement of observability and controllability properties is implemented in the `Enforcement` module.

The last package is the `ConformanceCheckingAndInputSelection` package, which consists of only one module, module `ConformanceChecking`. This package contains the procedures for conformance checking and test input selection.

The various methods specified in VDM ++, are capable of analyzing the following properties of the UML SDs:

- **validTraces:** Set of valid global traces defined by the given SD;
- **unintendedTraces:** Set of invalid global traces caused by locally valid decisions (representing violations of local controllability);
- **uncheckableLocally:** Set of invalid global traces that cannot be verified locally (representing violations of local observability);
- **isLocallyControllable:** The diagram is locally controllable if there are no unintended traces;
- **isLocallyObservable:** The diagram is locally observable if there are no locally uncheckable traces;
- **genCoordinationFeatures:** Set of coordination messages and/or time constraints to enforce local controllability and/or local observability.

The full specifications in VDM++ can be found in Appendix B. These specifications can be directly executed with the Overture tool (Overture community, 2020).

6.3 Front-end

Taking into account the limitations that we found when converting VDM++ specifications to Java code and the common difficulty of the end-user in dealing with formal specifications, we decided to implement an interface that facilitates the use of the developed methods by people without the knowledge of formal methods.

The DCO analyzer frontend was developed in Java and as shown in Figure 6.3, it is composed of four modules, `GUI`, `UML Parser`, `VDM Generator` and `VDM Caller`.

The `GUI` module is the main module, where the user interface is implemented. In `UML Parser` module, parser functions for UML files are implemented. This module takes advantage of the `DocumentBuilder` class to obtain instances from an XML document. In the `VDM Generator` module, the functions that allow the generation of VDM models are implemented. Finally the `VDM Caller` module is responsible for the connection between Java and VDM++, which is where the invocations of the VDM++ methods are carried out and the results that are later presented to the user are interpreted.

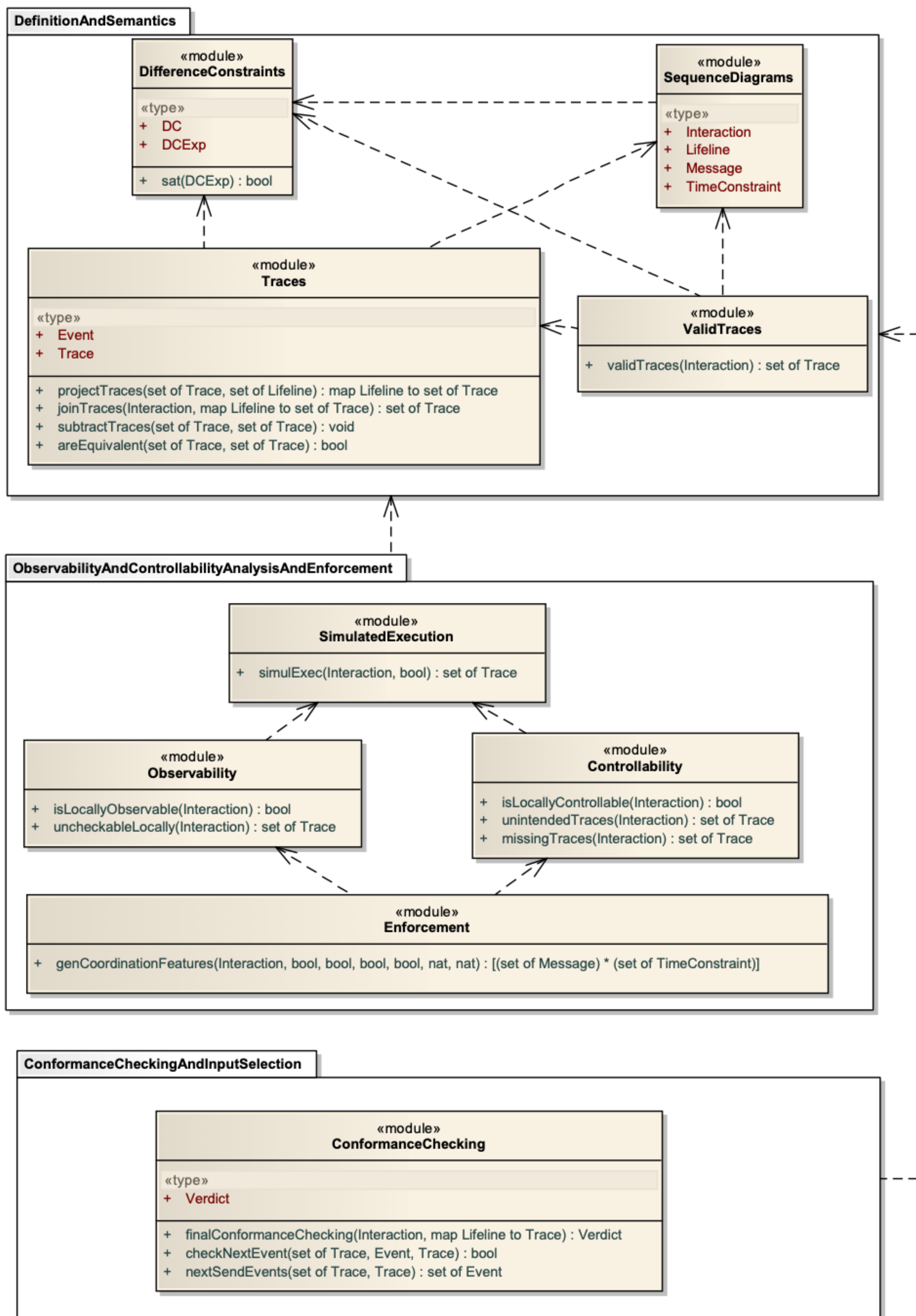


Figure 6.2: Structure of the DCO Analyzer Back-End (UML class diagram, simplified)

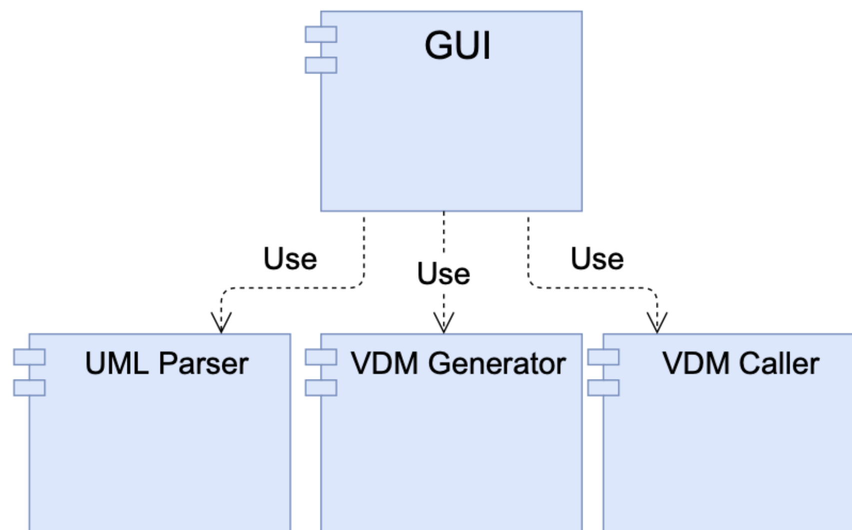


Figure 6.3: DCO Analyzer Front-End Modules

The DCO Analyzer tool provides a simple to use interface. All its features are accessible in a single window that is divided into three areas as shown in Figure 6.4.

The first area is the area where the user selects the type of input file. The tool accepts two types of files, the first one is the .UML files that represent the UML standard and which is the typical export format for UML SD visual editors. Listing 6.1 presents an example of a .UML file that represents a simple UML SD with two lifelines, a `opt` combined fragment, and a message exchanged between the lifelines.

When the user submits a .UML file, the tool starts by doing a pre-analysis in order to check if the file respects the UML standard. The parser verifies the structure of the document according to the UML standard for representing SDs. If the structure are correct, the tool converts the .UML file to the corresponding formal specification (.VDMPP file). The .VDMPP file corresponding to the example shown in Listing 6.1, can be seen in Listing 6.2. These files correspond to the Papyrus diagram presented in Figure 6.5, which consists of two lifelines and a message that is contained in a `opt` combined fragment.

However, given the current limitations of the visual editing tools, namely with regard to the modeling of time constraints the user can, if he/she so wishes, directly supply the .VDMPP file to be analyzed by the tool. For that, he/she only has to select the option .VDMPP in the interface.

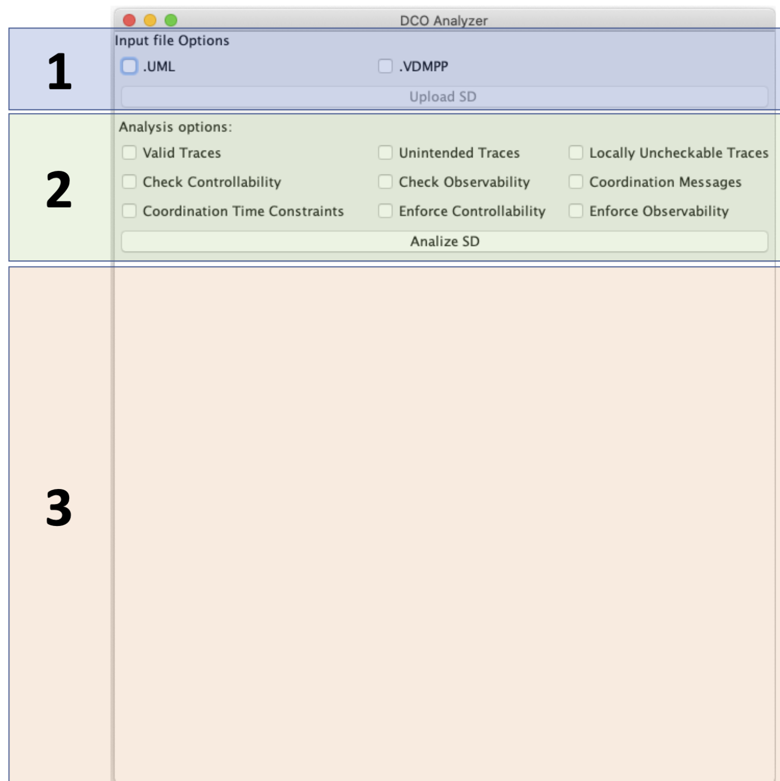


Figure 6.4: DCO Analyzer Front-End

```xml
<?xml version="1.0" encoding="UTF−8"?>
<uml:Model xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001" xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML" xmi:id="
        _YP03oJJ0EemokoQUb−fCSA" name="ExampleB">
  <packageImport xmi:type="uml:PackageImport" xmi:id="_YTIDQJJ0EemokoQUb−fCSA">
    <importedPackage xmi:type="uml:Model" href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#_0"/>
  </packageImport>
  <packagedElement xmi:type="uml:Interaction" xmi:id="_YQIZoJJ0EemokoQUb−fCSA" name="Interaction1">
    <lifeline xmi:type="uml:Lifeline" xmi:id="_aJYaQJJ0EemokoQUb−fCSA" name="L1" coveredBy="_FFxBEJJ1EemokoQUb−fCSA _FFyPMJJ1EemokoQUb−fCSA
          _kMIcIJJ1EemokoQUb−fCSA"/>
    <lifeline xmi:type="uml:Lifeline" xmi:id="_bULZcJJ0EemokoQUb−fCSA" name="L2" coveredBy="_FFxBEJJ1EemokoQUb−fCSA _FFyPMJJ1EemokoQUb−fCSA
          _kMIcIZJ1EemokoQUb−fCSA"/>
    <fragment xmi:type="uml:CombinedFragment" xmi:id="_FFxBEJJ1EemokoQUb−fCSA" name="CombinedFragment1" covered="_bULZcJJ0EemokoQUb−fCSA
          _aJYaQJJ0EemokoQUb−fCSA" interactionOperator="opt">
      <operand xmi:type="uml:InteractionOperand" xmi:id="_FFyPMJJ1EemokoQUb−fCSA" name="InteractionOperand0" covered="_bULZcJJ0EemokoQUb−fCSA
            _aJYaQJJ0EemokoQUb−fCSA">
        <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_kMIcIJJ1EemokoQUb−fCSA" name="Message1SendEvent" covered="
              _aJYaQJJ0EemokoQUb−fCSA" message="_kMGm8JJ1EemokoQUb−fCSA"/>
        <fragment xmi:type="uml:MessageOccurrenceSpecification" xmi:id="_kMIcIZJ1EemokoQUb−fCSA" name="Message1ReceiveEvent" covered="
              _bULZcJJ0EemokoQUb−fCSA" message="_kMGm8JJ1EemokoQUb−fCSA"/>
        <guard xmi:type="uml:InteractionConstraint" xmi:id="_FFyPMZJ1EemokoQUb−fCSA" visibility="package">
          <specification xmi:type="uml:LiteralString" xmi:id="_JgtF4JJ2EemokoQUb−fCSA" value="x&lt;2"/>
        </guard>
      </operand>
    </fragment>
    <message xmi:type="uml:Message" xmi:id="_kMGm8JJ1EemokoQUb−fCSA" name="m1" messageSort="asynchCall" receiveEvent="_kMIcIZJ1EemokoQUb−fCSA"
          sendEvent="_kMIcIJJ1EemokoQUb−fCSA"/>
  </packagedElement>
</uml:Model>
```

Listing 6.1: Example of an .UML file

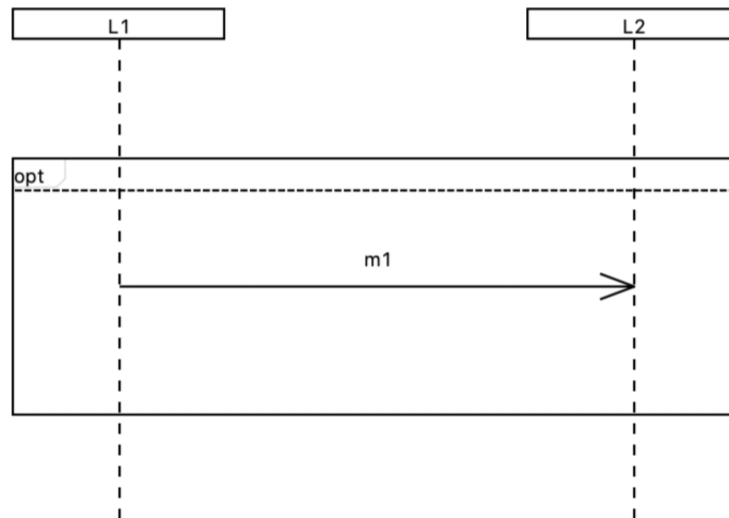


Figure 6.5: Example of Basic SD in Papyrus

```
class DiagramT is subclass of SequenceDiagrams

 operations

 public makeDiagram() res: Interaction ==
(
  let
 l1 = mk_Lifeline("L1"),
 l2 = mk_Lifeline("L2"),
 m1 = mkMessage(1, mk_(l1, 3), mk_(l2, 3), "m1"),
 o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2)}, {mk_(l1, 4), mk_(l2, 4)
     }),
 f1 = mk_CombinedFragment(<opt>, [o1], {l2 ,l1}),
 sd1 = mkInteraction({l1, l2}, {m1}, {f1})
in
 return sd1)

 end DiagramT
```

Listing 6.2: Example of a generated .VDMPP file

In the second area, and after the tool checks whether that the user has uploaded a valid file, the user can select the type of analysis he wants to perform to the UML SD, being able to choose from the 9 options previously described.

For each analysis, and in order to connect the front-end in java to the back-end developed in VDM++, an invocation of the respective VDM++ method is performed by running the Overture tool. For example, to perform the analysis that allows identifying Valid Traces, the tool executes

the command `java -jar Overture-2.7.0.jar -vdmpp -w -e new ValidTraces().validTracesStr(new DiagramT().makeDiagram()) ./ ./VDDJAR ./VDDJAR/lib`. The result is then processed automatically so that it can be presented to the user.

The third area is the area where the messages are presented to the user, either error messages (when, for example, the inserted files do not respect the standard format) or results of the analysis requested by the user.

6.4 Usage Examples

As previously described, the tool is designed so that the user can analyze diagrams obtained from his favorite visual modeling tool. Figure 6.6 shows an example of an UML SD test scenario, drawn with the Papyrus tool, for an online driving license renewal system (greatly simplified for illustration purposes).

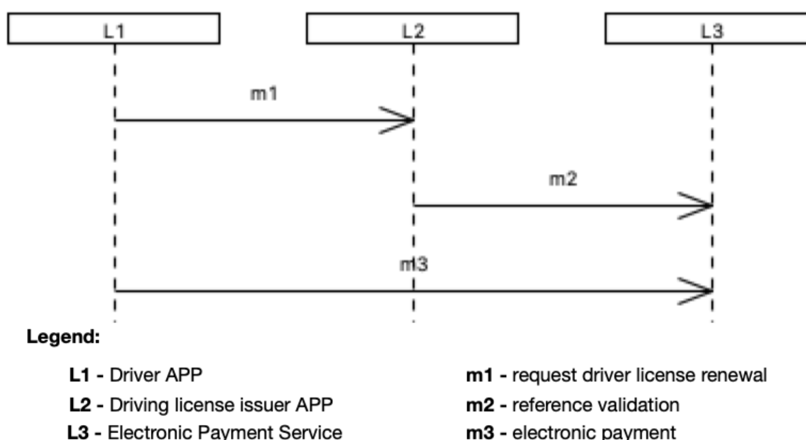


Figure 6.6: Example of Initial SD in Papyrus

After uploading the respective .UML file and selecting the analysis of all properties, the user obtains the output represented in Figure 6.7.

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

In this example, the tool identified a set of four valid traces and was able to detect that the given diagram is not locally controllable, indicating six unintended traces.

These unintended traces are related to the possibility of the electronic payment message ($m3$) being received by the electronic payment service ($L3$) before the reference validation message ($m2$).

In order to help the user to make this diagram locally controllable, our tool suggests adding a coordination message ($Ctrl1$) between the Electronic Payment Service ($L3$) and the Driver APP ($L2$), after $m2$ (suffix "Am3") " and $m1$ (suffix "Am1"), respectively. In practice, such message

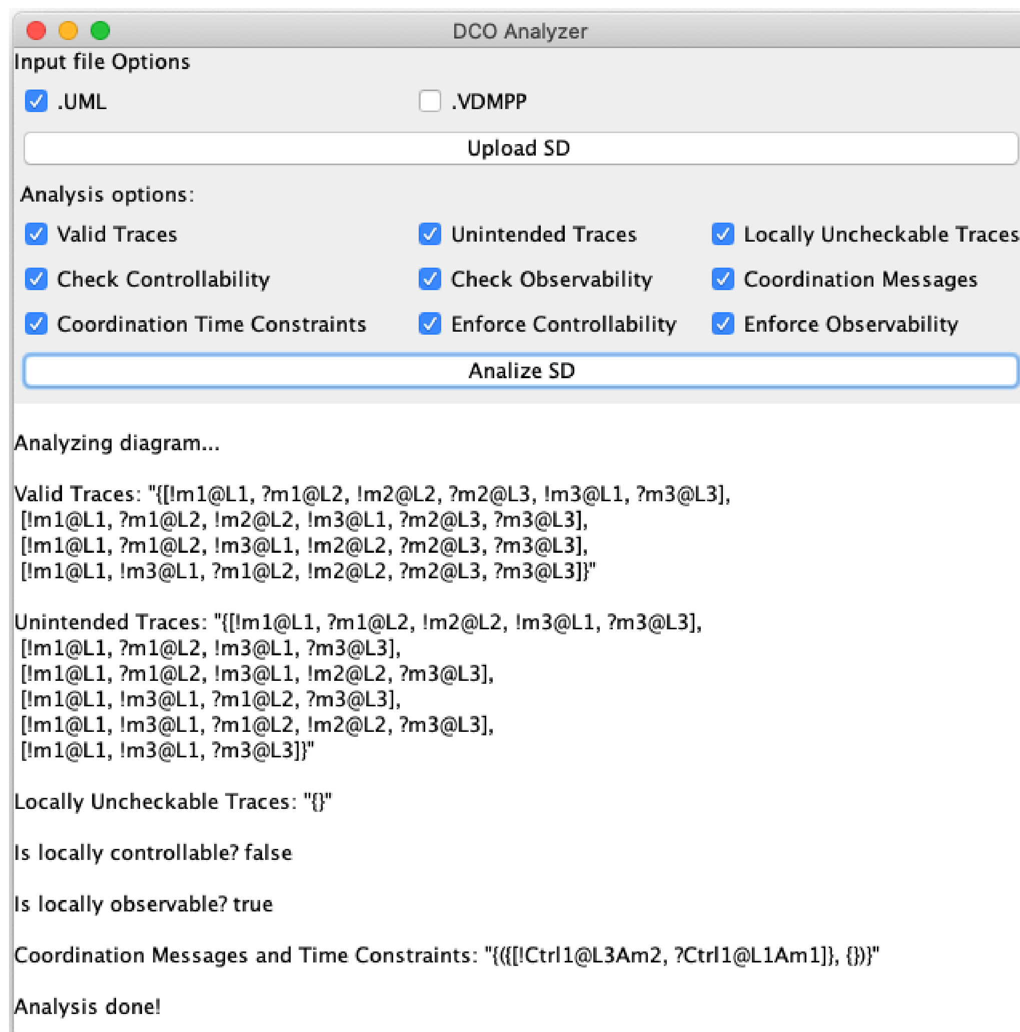


Figure 6.7: Example of DCO Analyzer output

might represent a payment authorization confirmation message, thereby ensuring that payment can only be made after the payment reference has been validated.

With this suggestion, the user can then refine the SD as shown in Figure 6.8.

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

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

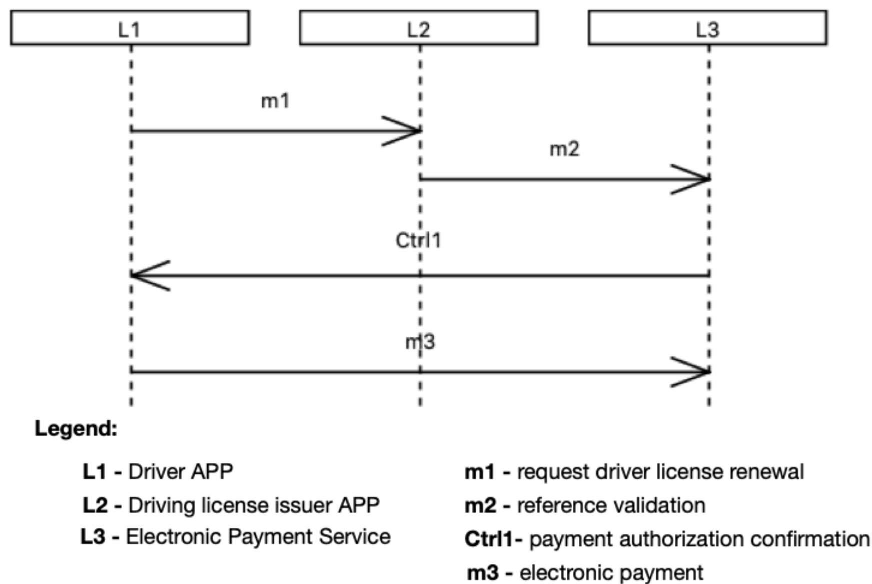


Figure 6.8: Refined SD in Papyrus

Another example, illustrating a local observability problem, is shown in Figure 6.9. This diagram represents the login scenario of a mobile application, where the user, after login, can receive pending notifications since the last time the application was connected to the server. By analyzing the diagram with our tool it is possible to detect that local testers are unable to locally detect the execution trace $[!m1@L1, ?m1@L2, !m2@L2]$, which corresponds to the case where the message $m2$ is sent but lost. Such loss will not be detected as an error at $L1$ because not receiving $m2$ is also a valid behavior at $L1$. The solution to this problem recommended by the DCO Analyzer is to place a coordination message between $L1$ and $L2$ upon receipt of $m2$ in $L1$. Such message can be interpreted as an acknowledgment message; if $m2$ is lost (or the acknowledgment message is lost), then a problem will be detected at $L2$.

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

DCO Analyzer executable files, some test scenarios in UML, and a demo video can be found at https://brunolima.info/DCOANALYZER/.

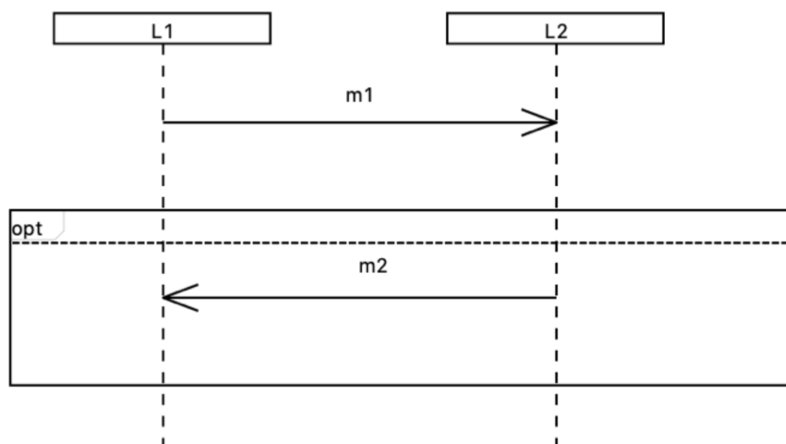


Figure 6.9: Example of a scenario not locally observable

Chapter 7

Validation

In this chapter we describe the validation of the algorithms and the tool developed in this thesis. Our validation process was carried out at two different times. First, we carried out a series of validation tests that tried to cover different real-life situations, covering different types of observability and controllability problems as well as different types of UML SDs (with and without time constraints) using different combined fragments. The second part of the validation consisted in an industrial case study whose purpose was to prove the functionality of our tool and algorithms in real world test scenarios.

Section 7.1 presents the validation tests. Section 7.2 presents the industrial case study. Section 7.3 concludes the chapter.

7.1 Validation Tests

To validate the specifications, we encoded in VDM++ real-world test scenarios coming from a nation-wide project in the ambient-assisted living domain AAL4ALL (2015). Additionally test scenarios were designed and encoded to ensure full coverage of the specification. In total, 38 test scenarios (test cases) were defined, divided in three different sets.

In the first set of 21 test cases we used scenarios without time constraints. We focused only on covering the most common observability and controllability problems (for example race conditions, non-local choices, etc.) in diagrams with different types of combined fragments.

Figure 7.1 shows one of the test cases in this set. This diagram consists of three lifelines, five messages and an `alt` combined fragment, and represents an example of multiple violations of local observability and controllability. The encoding of this test case in VDM++ can be seen in Listing 7.1.

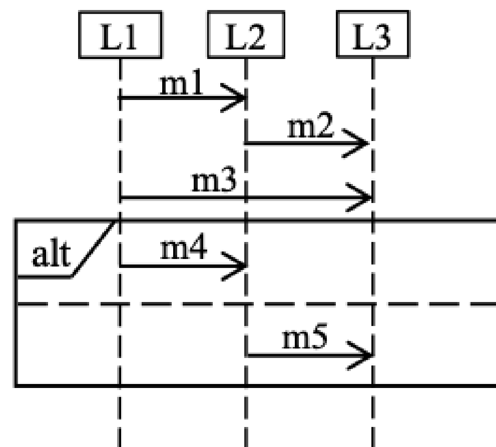


Figure 7.1: TestRacePlusAlt UML SD.

```
public testRacePlusAlt() ≜
(
 let l1 = mkLifeline("L1"),
    l2 = mkLifeline("L2"),
    l3 = mkLifeline("L3"),
    m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
    m2 = mkMessage(2, mk_(l2, 2), mk_(l3, 2), "m2"),
    m3 = mkMessage(3, mk_(l1, 3), mk_(l3, 3), "m3"),
    m4 = mkMessage(4, mk_(l1, 5), mk_(l2, 5), "m4"),
    m5 = mkMessage(5, mk_(l2, 7), mk_(l3, 7), "m5"),
    o1 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)}, {mk_
        (l1, 6), mk_(l2, 6), mk_(l3, 6)}),
    o2 = mk_InteractionOperand(nil, {mk_(l1, 6), mk_(l2, 6), mk_(l3, 6)}, {mk_
        (l1, 8), mk_(l2, 8), mk_(l3, 8)}),
    f1 = mk_CombinedFragment(<alt>, [o1, o2], {l1, l2, l3}),
    sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4, m5}, {f1})
 in
 (
  assertEqual("{({}, {!m2@L2 - ?m1@L2 <= 1, ?m1@L2 - !m1@L1 <= 2, 6 <= !m3@L1 -
       !m1@L1 <= 7, ?m2@L3 - !m2@L2 <= 2, 11 <= !m5@L2 - !m2@L2, ?m3@L3 - !
     m3@L1 <= 2, !m4@L1 - !m3@L1 <= 1, ?m4@L2 - !m4@L1 <= 2})}",
   genCoordinationFeaturesStr(sd1, false, true, false, true, 2, 1));
 )
);
```

Listing 7.1: The testRacePlusAlt() test case

Our goal with this test case is to prove that our tool is able to identify and solve the controllability problems by suggesting to the user the set of coordination time constraints that enforce local controllability.

Figure 7.2 presents the diagram from Figure 7.1, modified to incorporate the time constraints

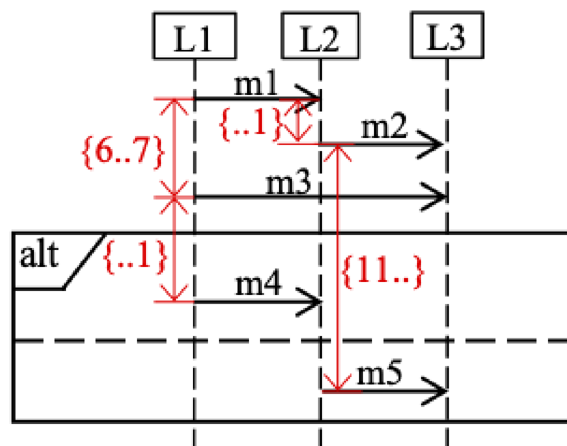


Figure 7.2: TesStrict UML SD Corrected.

suggested by the tool (the time constraint for messages delivery <2, which should be applied to all messages, have been omitted from the diagram for better understanding).

Another example of this set of test cases is theUML SD represented in Figure 7.3. This diagram is composed of three lifelines, two messages and the `strict` combined fragment, and represents an example of a not locally controllable nor observable scenario. The `strict` combined fragment imposes an exact order between the events that occur within each part of the `strict`, this means that in this case the sending of m2 only occurs after m1 has been received. The encoding of this test case in VDM++ can be seen in Listing 7.2.

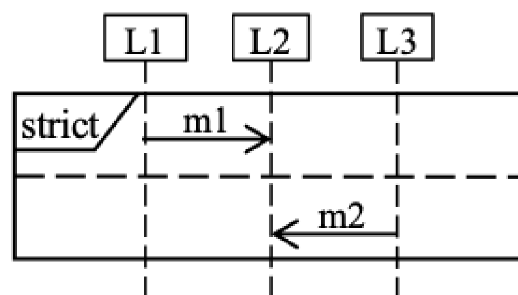


Figure 7.3: TesStrict UML SD with strict sequencing and a race condition.

```
public testStrict() ≜
(
 let l1 = mkLifeline("L1"),
     l2 = mkLifeline("L2"),
     l3 = mkLifeline("L3"),
     m1 = mkMessage(1, mk_(l1, 3), mk_(l2, 3), "m1"),
     m2 = mkMessage(2, mk_(l3, 5), mk_(l2, 5), "m2"),
     o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2), mk_(l3, 2)},
                              {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)}),
     o2 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)},
```

```
                    {mk_(l1, 6), mk_(l2, 6), mk_(l3, 6)}),
    f1 = mk_CombinedFragment(<strict>, [o1, o2], {l1, l2, l3}),
    sd1 = mkInteraction({l1, l2, l3}, {m1, m2}, {f1})
 in
 (
  assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
  assertEqual(<Inconclusive>, finalConformanceChecking(sd1,
            {l1 ↦ [s(m1)], l2 ↦ [r(m1), r(m2)], l3 ↦ [s(m2)]}));
  assertEqual({[s(m1), s(m2), r(m1), r(m2)], [s(m2), s(m1), r(m1), r(m2)]},
          uncheckableLocally(sd1));
  assertEqual({[s(m1), s(m2)], [s(m2)]}, unintendedTraces(sd1));
  assertEqual(∅, missingTraces(sd1)) ;
  assertEqual("{{[!Ctrl1@L2Am1, ?Ctrl1@L3Bm2]}}",
         genCoordinationMessagesStr2(sd1, false));
 )
);
```

Listing 7.2: The testStrict() test case

Our goal with this test case is to prove that our tool is able to identify and solve the local observability and controllability problems by suggesting to the user the set of coordination messages that enforce these proprieties.

One more example of this first set of test cases can be seen in Figure 7.4. This diagram consists of three lifelines, three messages and an `alt` combined fragment. This UML SD represents a scenario with mutually exclusive emission and reception events simultaneously enabled, making the scenario not locally controllable. The encoding of this test case in VDM++ can be seen in Listing 7.3.

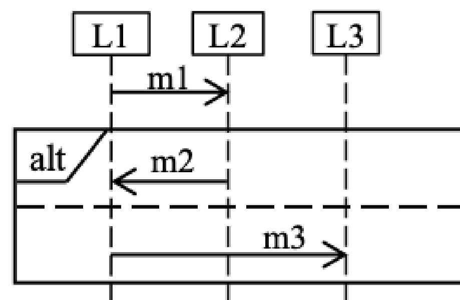


Figure 7.4: TestSendRecvEnabled UML SD.

```
public testSendRecvEnabled() ≜
(
 let l1 = mkLifeline("L1"),
    l2 = mkLifeline("L2"),
    l3 = mkLifeline("L3"),
    o11 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2), mk_(l3, 2)},
                       {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)}),
```

```
    o12 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)},
                                   {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8)}),
    f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3}),
    m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
    m2 = mkMessage(2, mk_(l2, 3), mk_(l1, 3), "m2"),
    m3 = mkMessage(3, mk_(l1, 6), mk_(l3, 6), "m3"),
    sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3}, {f1})
  in
  (
   assertFalse(isLocallyObservable(sd1));
   assertFalse(isLocallyControllable(sd1));
   assertEqual("{({[!Ack_m3@L3Am3, ?Ack_m3@L1Am3], [!Ctrl1@L2Am2, ?Ctrl1@L1Am2]},
       {})}",
       genCoordinationFeaturesStr(sd1, true, true, true, false, 2, 1));
   assertEqual("{({}, {!m2@L2 - ?m1@L2 <= 1, ?m1@L2 - !m1@L1 <= 2, 6 <= !m3@L1 - !
       m1@L1, ?m2@L1 - !m2@L2 <= 2})}",
       genCoordinationFeaturesStr(sd1, false, true, false, true, 2, 1));
  )
);
```

Listing 7.3: The testSendRecvEnabled() test case

Our goal with this test case is to prove that our tool is able to identify and solve the controllability problems by suggesting to the user two alternatives to enforce local controllability, a set of coordination messages and a set of time constraints. Figure 7.5 presents two alternative diagrams that correspond to the implementation of the solutions proposed by the tool when analyzing the diagram from Figure 7.4. The diagram a) incorporate the solution based on acknowledgement and control messages, and diagram b) incorporate the solution based in time constraints.

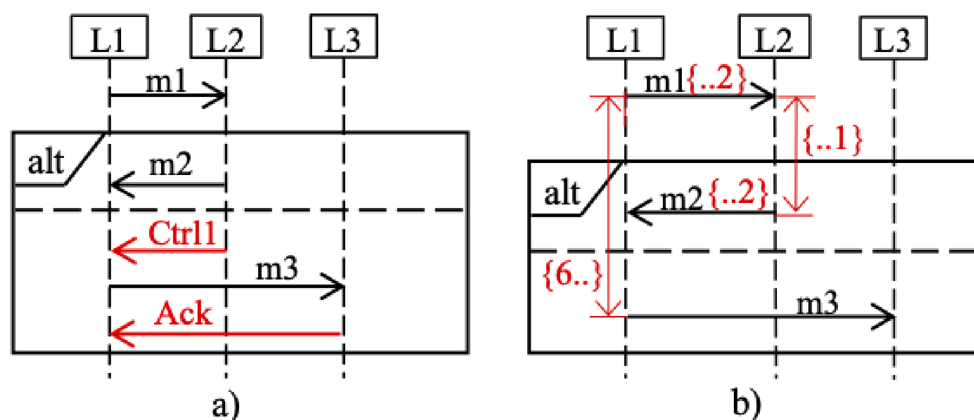


Figure 7.5: TestSendRecvEnabled UML SD.

A last example of this first set of test cases can be seen in Figure 7.6. This diagram consists of two lifelines, with a `loop` combined fragment with a finite number of iterations and a single message in each iteration. Although this is a simple diagram according to the UML standard,

there is a weak sequencing between iterations, since it is not guaranteed that the sending of the message m1 in an iteration $n+1$ occurs only after the reception of the message m1 sent in iteration n. Because last iteration is optional and the message has no acknowledgment, the scenario is not locally observable, but can be fixed with an acknowledgment message. The encoding of this test case in VDM++ can be seen in Listing 7.4.

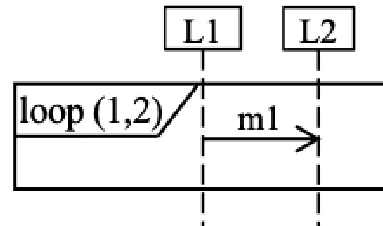


Figure 7.6: TestLoop UML SD.

```
public testLoop() ≙
(
 let l1 = mkLifeline("L1"),
    l2 = mkLifeline("L2"),
    m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
    ctrl = mkMessage(2, mk_(l2, 3), mk_(l1, 3), "m2"),
    o1 = mk_InteractionOperand(mk_InteractionConstraint(1, 2, nil),
         {mk_(l1, 1), mk_(l2, 1)}, {mk_(l1, 4), mk_(l2, 4)}),
    f1 = mk_CombinedFragment(<loop>, [o1], {l1, l2}),
    sd1 = mkInteraction({l1, l2}, {m1}, {f1})
 in
 (
  assertEqual({[s(m1), r(m1)], [s(m1), r(m1), s(m1), r(m1)],
             [s(m1), s(m1), r(m1), r(m1)]}, validTraces(sd1));
  assertEqual({[s(m1), s(m1), r(m1)], [s(m1), r(m1), s(m1)]},
             uncheckableLocally(sd1));
  assertTrue(isLocallyControllable(sd1));
  assertEqual("{{[!Ctrl1@L2Am1, ?Ctrl1@L1Am1]}}",
           genCoordinationMessagesStr2(sd1, false));
 )
);
```

Listing 7.4: The testLoop() test case

With this test case we intend to verify that our tool can handle well the particular cases of UML like this one.

The second set of 14 test cases focuses on scenarios with time constrains. In this set, we focus on the validation of scenarios where time constraints cause problems of local observability and controllability, cause possible inconclusive verdicts, or even cannot be satisfied.

An example of this set of test cases can be seen in Figure 7.7. This UML SD consists of two lifelines, two messages and a time constraint between the event of sending message m1 and the

reception of message m2. The time constraint determines that a maximum of 1000ms can pass between these two events. This diagram is not locally controllable, due to its roundtrip constraint as explained in Section 5.1. The encoding of this test case in VDM++ can be seen in Listing 7.5.

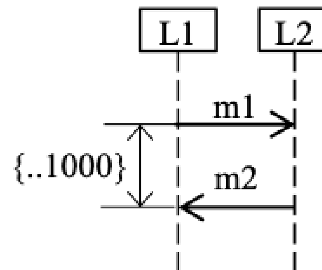


Figure 7.7: TestNonLocallyControlableTimed UML SD.

```
public testNonLocallyControlableTimed() ≜
(
 let l1 = mkLifeline("L1"),
     l2 = mkLifeline("L2"),
     m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
     m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
     sd1 = mkInteraction({l1, l2}, {m1, m2}, ∅,
                {mk_TimeConstraint(t(s(m1)), t(r(m2)), 0, 1000)})
 in
 (
  assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
  assertTrue(isLocallyObservable(sd1));
  assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, unintendedTraces(sd1));
  assertEqual(∅, missingTraces(sd1))
 )
);
```

Listing 7.5: The testNonLocallyControlableTimed() test case

Our goal with this test is to prove that our tool is capable of detecting local controllability problems in diagrams with time constraints.

Another example of this set of test cases can be seen in Figure7.8. This diagram is composed of two lifelines, two messages, two `opt` combined fragments, and four time constraints. The diagram represents a scenario that, despite containing two optional combined fragments, they end up becoming mutually exclusive due to time constraints, thus making the scenario locally controllable. The encoding of this test case in VDM++ can be seen in Listing 7.6.

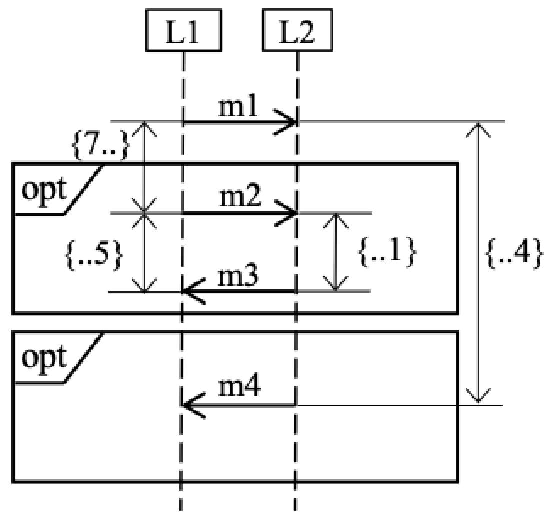


Figure 7.8: TestStrangeControllableTimed UML SD.

```
public testStrangeControllableTimed() ≜
(
 let l1 = mkLifeline("L1"),
     l2 = mkLifeline("L2"),

     m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
     m2 = mkMessageTimed(2, mk_(l1, 3), mk_(l2, 3), "m2"),
     m3 = mkMessageTimed(3, mk_(l2, 4), mk_(l1, 4), "m3"),
     m4 = mkMessageTimed(4, mk_(l2, 7), mk_(l1, 7), "m4"),

     o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2)}, {mk_(l1, 5), mk_(l2,
          5)}),
     o2 = mk_InteractionOperand(nil, {mk_(l1, 6), mk_(l2, 6)}, {mk_(l1, 8), mk_(l2,
          8)}),
     f1 = mk_CombinedFragment(<opt>, [o1], {l1, l2}),
     f2 = mk_CombinedFragment(<opt>, [o2], {l1, l2}),

     sd1 = mkInteraction({l1, l2}, {m1, m2, m3, m4}, {f1, f2},
       mkMsgTimeConstraints({m1, m2, m3, m4}, 0, 1) ∪
       {mk_TimeConstraint(t(s(m1)), t(s(m2)), 7, nil),
        mk_TimeConstraint(t(r(m2)), t(s(m3)), 0, 1),
        mk_TimeConstraint(t(r(m1)), t(s(m4)), 0, 4),
        mk_TimeConstraint(t(s(m2)), t(r(m3)), 0, 5)}),
     e1 = s(m1), e2 = r(m1),
     e3 = s(m2), e4 = r(m2),
     e5 = s(m3), e6 = r(m3),
     e7 = s(m4), e8 = r(m4)
 in
 (
  assertEqual({[e1, e2], [e1, e2, e3, e4, e5, e6], [e1, e2, e7, e8]},
            validTraces(sd1));
```

```
    assertTrue(isLocallyControllable(sd1));
    assertFalse(isLocallyObservable(sd1));
  )
);
```

Listing 7.6: The testStrangeControllableTimed() test case

Our goal with this test case was to prove that the tool is able to correctly analyze more complex diagrams with multiple combined fragments and multiple time constraints.

The third and last set of test cases is composed of three real scenarios taken from different projects. One of the tested scenarios in this set is the fall detection scenario previously presented in Figure 5.4. The scenario is locally controllable with the specified time constraints but would not in the absence of such constraints. The encoding of this test case in VDM++ can be seen in Listing 7.7.

```
public testFallDetection() ≜
(
 let l1 = mkLifeline("Care_Receiver"),
     l2 = mkLifeline("Fall_Detection_App"),
     l3 = mkLifeline("AAL4ALL_Portal"),
     m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "fall_signal"),
     m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "confirm?"),
     m3 = mkMessageTimed(3, mk_(l1, 4), mk_(l2, 4), "yes"),
     m4 = mkMessageTimed(4, mk_(l2, 5), mk_(l3, 5), "notify_fall"),
     m5 = mkMessageTimed(5, mk_(l1, 7), mk_(l2, 7), "no"),
     m6 = mkMessageTimed(6, mk_(l2, 9), mk_(l3, 9), "notify_possible_fall"),
     o1 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)},
                          {mk_(l1, 6), mk_(l2, 6), mk_(l3, 6)}),
     o2 = mk_InteractionOperand(nil, {mk_(l1, 6), mk_(l2, 6), mk_(l3, 6)},
                          {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8)}),
     o3 = mk_InteractionOperand(nil, {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8)},
                          {mk_(l1, 10), mk_(l2, 10), mk_(l3, 10)}),
     f1 = mk_CombinedFragment(<alt>, [o1, o2, o3], {l1, l2, l3}),
     tcs = {mk_TimeConstraint(t(s(m2)), t(r(m2)), 0, 1000),
           mk_TimeConstraint(t(s(m3)), t(r(m3)), 0, 1000),
           mk_TimeConstraint(t(s(m5)), t(r(m5)), 0, 1000),
           mk_TimeConstraint(t(r(m2)), t(s(m3)), 0, 10000),
           mk_TimeConstraint(t(r(m2)), t(s(m5)), 0, 10000),
           mk_TimeConstraint(t(s(m2)), t(s(m6)), 13000, nil)},
     sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4, m5, m6}, {f1}, tcs),
     sd2 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4, m5, m6}, {f1}, ∅),
     e1 = s(m1), e2 = r(m1),
     e3 = s(m2), e4 = r(m2),
     e5 = s(m3), e6 = r(m3),
     e7 = s(m4), e8 = r(m4),
     e9 = s(m5), e10 = r(m5),
     e11 = s(m6), e12 = r(m6),
```

```
    e1a = mk_Event(<Send>, "fall_signal", l1, 0),
    e2a = mk_Event(<Receive>, "fall_signal", l2, 2000),
    e3a = mk_Event(<Send>, "confirm?", l2, 4000),
    e4a = mk_Event(<Receive>, "confirm?", l1, 4200),
    e5a = mk_Event(<Send>, "yes", l1, 14200),
    e6a = mk_Event(<Receive>, "yes", l2, 14500),
    e7a = mk_Event(<Send>, "notify_fall", l2, 14600),
    e8a = mk_Event(<Receive>, "notify_fall", l3, 16000),

    e6b = mk_Event(<Receive>, "yes", l2, 15200),
    e7b = mk_Event(<Send>, "notify_fall", l2, 15600),

    e6c = mk_Event(<Receive>, "yes", l2, 18000),
    e7c = mk_Event(<Send>, "notify_fall", l2, 18600),
    e8c = mk_Event(<Receive>, "notify_fall", l3, 19000),

    e4d = mk_Event(<Receive>, "confirm?", l1, 16800),
    e11d = mk_Event(<Send>, "notify_possible_fall", l2, 17000),
    e12d = mk_Event(<Receive>, "notify_possible_fall", l3, 18000)

 in
 (
  assertEqual({[e1, e2, e3, e4, e5, e6, e7, e8],
           [e1, e2, e3, e4, e9, e10],
           [e1, e2, e3, e4, e11, e12]},
          validTraces(sd1));

  MaxClockSkew := 500;
  assertEqual(<Pass>, timedFinalConformanceChecking(sd1,
           {l1 ↦ [e1a, e4a, e5a], l2 ↦ [e2a, e3a, e6a, e7a], l3 ↦ [e8a]}));
  assertEqual(<Inconclusive>, timedFinalConformanceChecking(sd1,
           {l1 ↦ [e1a, e4a, e5a], l2 ↦ [e2a, e3a, e6b, e7b], l3 ↦ [e8a]}));
  assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
          {l1 ↦ [e1a, e4a, e5a], l2↦ [e2a, e3a, e6c, e7c], l3 ↦ [e8c]}));
  assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
          {l1 ↦ [e1a, e4d], l2 ↦ [e2a, e3a, e11d], l3 ↦ [e12d]}));
  MaxClockSkew := 10;

  assertTrue(isLocallyControllable(sd1));
  assertFalse(isLocallyControllable(sd2));
  assertFalse(isLocallyObservable(sd1)); -- because of optional messages without
      ack
 )
);
```

Listing 7.7: The testFallDetection() test case

With this test case we intend to verify that the tool has the behavior expected in the analysis of

this type of diagrams, analyzing the same scenario with and without time constraints. As already described in Figure 5.7 a particular case in this scenario occurs when the message `yes` is sent and the times recorded in the local observation of events `e5` and `e6`, are 14200 and 15200 respectively. Applying the procedure described in Section 5.3.6, we obtain an inconclusive verdict since the difference between the observed times is exactly coincident with the maximum time allowed by the inter lifeline time constraint (1000). If the local clocks were perfectly synchronized, the test would pass, however admitting a `clock skew` of 500, the test may have violated this time constraint, so the verdict is inconclusive.

The VDM++ specification for all test cases can be found in Appendix B (TestCases class), these specifications can be directly executed with the Overture tool (Overture community, 2020). The complete set of test cases executes in approximately 23 seconds in Overture, with all run-time checks enabled (dynamic type checks, invariant checks, pre-condition checks, post-condition checks, and measure run-time checks) and test coverage instrumentation activated. The test cases were executed in a MacBook Pro 2018 with an Intel Core i7 CPU @2.7 GHZ, 16GB RAM, and the macOS Catalina.

7.2 Industrial Case Study

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

1. to check if our analysis tool is able to correctly identify local controllability and/or local observability issues in real-world test scenarios;

2. to check if the analysis is performed in an adequate time;

3. to check if the output results produced by the tool help the users to understand the root causes of the detected problems and refine the input test scenarios accordingly.

7.2.1 Motorway Incident Detection Project

The project of our industrial partner (here described in a simplified way for privacy reasons), illustrated in Figure 7.9, consists of the placement of sensors on the motorways that interact with each other and are able (among other features) to detect incidents automatically.

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

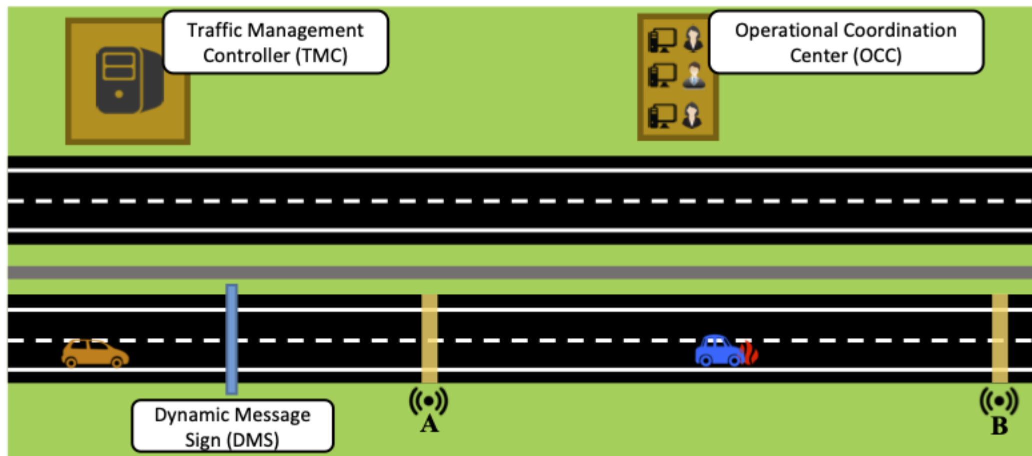


Figure 7.9: Traffic Control System.

7.2.2 Test Scenario

We asked our partner to describe the system interactions (including temporal constraints) using UML SDs. Figure 7.10 shows one of the scenarios that was provided.

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

7.2.3 Scenario Analysis - Local Controllability

We analyzed the local controllability of the previous test scenario (Figure 7.10) with our tool, which took 1.1 s to run in the machine previously described and reported 3 unintended tc-traces (with lifeline indicated only when needed to disambiguate):

1. $[!id_signal, ?id_signal@A, !notify_id, !id_signal, ?id_signal@B, ?notify_id, \ldots]$, with $\tau_4 - \tau_1 \leq 72$;

2. $[!id_signal, ?id_signal@A, !id_signal, ?id_signal@B, !notify_id, ?notify_id, \ldots]$, with $\tau_3 - \tau_1 \leq 72$;

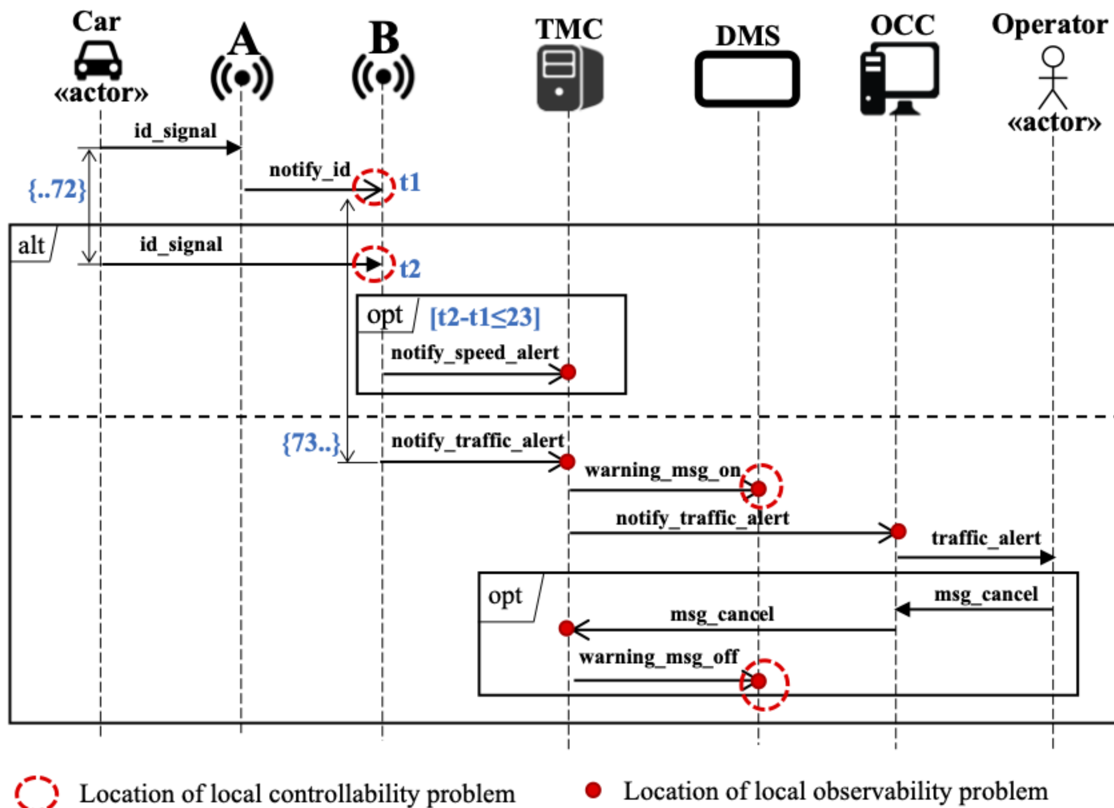


Figure 7.10: Initial scenario and problem locations.

3. [$!id_signal$, $?id_signal@A$, $!notify_id$, $?notify_id$, $!notify_traffic_alert$, $\ldots$, $?warning_msg_off$, $?warning_msg_on$], with $\tau_5 - \tau_1 \geq 73$.

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

1. Unexpected reception of *id_signal* at sensor B before reception of $notify_id$ (unintended traces 1 and 2). As delays can occur in the transmission of the $notify_id$ message between sensor A and sensor B, the message $notify_id$ may arrive at sensor B before the message *id_signal*. As a consequence, the system may be unable to use the sensor data or may incorrectly conclude that a vehicle is moving against the flow of traffic; this suggests a design flaw or an incomplete specification.

2. Unexpected late reception of *warning_msg_on* at DMS after reception of $warning_msg_off$ (unintended trace 3). As delays can occur in the transmission of the *warning_msg_on* message between TMC and DMS, the message $warning_message_off$ may arrive at DMS before the message *warning_message_on*. This case shows that in some situations an alert message can remain visible in the DMS even after it has been removed by the operator, thereby transmitting erroneous information to the drivers.

7.2.4 Scenario Analysis - Local Observability

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

1) [$!id_signal$, $?id_signal@A$, $!notify_id$, $?notify_id$, $!id_signal$, $?id_signal@B$, $!notify_speed_alert$], with $\tau_5 - \tau_1 \leq 72 \wedge \tau_6 - \tau_4 \leq 23$ (message $notify_speed_alert$ lost);

...

22) [$!id_signal$, $?id_signal@A$, $!notify_id$, $?notify_id$, $!notify_traffic_alert$, ..., $!warning_msg_off$], with $\tau_5 - \tau_4 \geq 73$ (message $warning_message_off$ lost).

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

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

As explained in Section 5.1, if any of these messages is lost, the problem will go undetected by the target lifeline, because not receiving a message is also a locally valid behavior. The solution recommended by our tool to enforce local observability consists of the addition of 6 corresponding acknowledgment (coordination) messages.

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

7.2.5 Scenario Refinement

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

The tool recommended the addition of 3 upper time bounds and 2 lower time bounds as indicated in red in Figure 7.11, solving both controllability problems. The analysis took 1.8 s to run in the machine previously described.

Our partner accepted the suggestions, but opted to further refine the test scenario as indicated by the solid arrows in Figure 7.11. Considering that a maximum car speed of $450\,\mathrm{km\,h^{-1}}$ could be safely assumed, the minimum time for a car to travel between sensors A and B was changed from 3 to 8 s. Our partner also decided to redesign the operator user interface, so that traffic alert messages can only be canceled after 5 s; hence, the minimum operator response time was changed from 2 to 5 s.

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

Those scenarios are related to other traffic anomalies that can be detected and notified using the same road infrastructure (see Figure 7.9), namely:

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

Those scenarios differ from the scenario in Figure 7.10 in the initial sensor activation sequence, but share a similar traffic alert notification sequence, and present similar types of observability and controllability problems.

7.2.6 Discussion

Regarding the goals of the experiment, we concluded that:

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

7.2.7 Threats to Validity

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

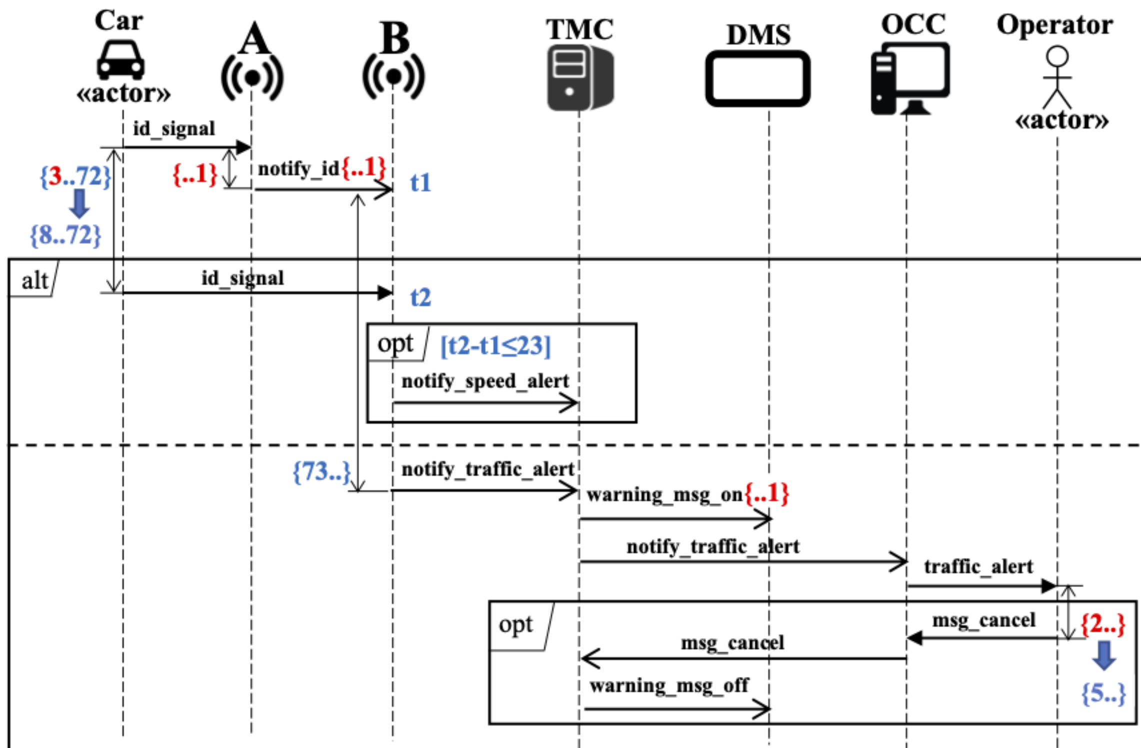


Figure 7.11: Refined locally controllable scenario (automatic refinement in red, followed by manual refinement indicated with solid arrows).

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

7.3 Conclusions

The validation tests and the industrial case study allowed an extended validation of the local observability and controllability analysis and enforcement algorithms, and the DCO Analyzer tool.

We believe that with validation tests we are able to cover not only the most common causes of local observability and controllability problems, but also the different types of UML SD (with or without time restrictions, different combined fragments, etc.). The results also showed that the tool was always able to suggest the respective corrections to the user, either in the form of coordination messages, time constraints or in some cases the two alternatives.

The validation in the industrial case study, allowed us to confirm that the tool is capable of dealing with real scenarios, as well as giving suggestions that the user is able to understand, thus proving to be a useful tool for professionals interested in modeling and testing this type of systems.

Chapter 8

Conclusions

8.1 Summary of Contributions

The main contributions of the research work are a state of the art on time-constrained distributed systems testing, a state of the practice on testing distributed and heterogeneous systems, a testing approach and architecture for the integration testing time-constrained distributed systems, local observability and controllability analysis and enforcement algorithms implemented in the DCO Analyzer tool and validated in an industry case study.

The state of the art analysis performed in this thesis allowed us to conclude that although some works address observability and controllability problems in distributed systems testing and design, none addresses the problem of observability and controllability analysis and enforcement for time-constrained distributed systems. This open research problem requires community attention given its serious importance in this type of systems.

The state of the practice analysis on testing DHS that we carry out through an exploratory survey that was responded by 147 software testing professionals that attended industry-oriented software testing conferences allowed us to confirm the high relevance of DHS in software testing practice, confirm and prioritize the relevance of testing features characteristics of DHS, confirm the existence of a significant gap between the current and the desired status of test automation for DHS, and confirm and prioritize the relevance of test automation features for DHS. The survey results indicated a limited adoption of complete test automation processes by companies. For better understanding what are the obstacles that companies face for not adopting complete test automation approaches, we conducted follow-up interviews with companies of different sizes and testing approaches. The conclusions drawn from the interviews allowed us to identify some common obstacles, such as the cost of acquisition and difficulty of adaptation of test automation tools, the cost of test suite maintenance (namely with frequent changes in the software under test), and the effort and expertise required for the creation of system models needed as input for automatic test suite generation.

The proposed testing approach and architecture for the integration testing time-constrained distributed systems that provides a higher level of automation of the testing process because all

phases of the test process are supported in an integrated fashion. The only manual activity needed is the development in a user friendly notation of the model required as input for automatic test case generation and execution, and there is no need to develop test components specific for each SUT. This approach also provides a higher fault detection capability. The use of a hybrid test architecture allows the detection of a higher number of errors as compared to purely distributed or centralized architectures. The proposed approach provides easier support for multiple test levels because the same input model can be used to perform tests at different levels, simply by changing the selection of observable and controllable events in the input model. A scenario-oriented approach simplifies the level of detail required in the input models. The test execution process is also more efficient. With a distributed conformance checking algorithm, communication overheads during test execution are minimized and the usage of a state-oriented runtime model allows a more efficient model execution and conformance checking.

We also proposed an approach to assess if test scenarios are ready for distributed execution, and, if not, refine them to become test ready with minimal overhead. This approach is based on the notions of local (or distributed) observability and controllability, that is, the ability to perform conformance checking (observability) and test input selection (controllability) in a purely distributed way, without exchanging coordination messages between the test components during test execution or overlooking conformance faults or causing incorrect test inputs. All the algorithms were implemented in the DCO Analyzer tool, for test scenarios specified by means of UML sequence diagrams.

To validate the algorithms and the tool we first conducted a series of validation tests that tried to cover different real-life situations, covering different types of observability and controllability problems as well as different types of UML SDs (with and without time constraints) using different combined fragments. Secondly we conducted an evaluation experiment with real-world test scenarios from an industrial partner. In that experiment, our tool was able to correctly identify local observability and controllability issues and recommend possible fixes; the outputs reported helped the users to understand and fix the root causes of the detected problems.

8.2 Research Questions Revisited

The results achieved allow us to answer the four research questions set in Section 1.2.

RQ1 - *What are the main difficulties and needs in the integration testing of distributed systems listed in the state of the art and state of practice?*

Looking at the state of the art presented in Chapter 2, we can say that there is a lack of studies that solve the problem of observability and controllability analysis and enforcement for time-constrained distributed systems. This is a critical problem in this type of systems. However, the principles adopted by the approaches present in Table 2.3, namely at the level of solutions to the observability and controllability problems in distributed systems testing and design are a good starting point for this thesis.

Looking at the conclusions of this survey presented in Chapter 3, we can say that main difficulties and needs in the integration testing of distributed systems with time-constraints in the state of practice are expensive testing tools, so more open source solutions are needed in order to allow smaller companies to also be able to automate the testing process for these types of systems. The results also highlights important needs in the context of DHS testing, namely: the need for checking interactions between the system components; the need for automated test execution; the need to support multiple platforms, among others.

RQ2 - *What is an adequate architecture and approach to conduct integration tests in these types of systems?*

Looking at the analysis performed in Chapter 2, the hybrid test architecture that combines local testers and the central tester is one that appears to be the architecture that best suits this type of system, namely due to its failure detection capacity. For this reason it was our starting point in the proposed solution for the integration test in this type of systems.

RQ3 - *How do we determine if a test scenario described by a UML SD can be executed safely in a purely distributed manner, without overlooking conformance faults (false negatives) or injecting conformance faults (false positives) by the test harness? In other words, how do we determine if a test scenario described by a UML SD is locally observable and locally controllable?*

Using the algorithms proposed in Sections 5.4 and 5.5, it is possible to automatically determine whether a given SD is locally observable and/or locally controllable.

RQ4 - *Given a test scenario not locally controllable or locally observable, how can we automatically identify a minimal set of coordination messages and/or coordination time constraints to refine the test scenario and enforce local observability and/or local controllability?*

As we show through the algorithms proposed in Section 5.6, it is possible to overcome local observability and controllability problems, using coordination messages or time constraints. Our algorithms are able to determine this type of fixes automatically.

8.3 Future Work

Currently, DCO Analyzer is available as a standalone Java application. As future work, we intend to make it available as a plug-in for some visual modeling editors so it can be accessed in a more integrated way in the modeling environment. We believe that this way we will reach more possible users of the tool and enable further validation experiments in industrial settings. We also intend to explorer a way to present enforcement recommendations visually; we believe that this will facilitate the interpretation of the recommendations given by the tool and thereby allow users with less experience in modeling to use it. It is also in our plans integrate the tool in a full-edged toolset for model-based distributed systems testing since at this moment the testing execution part of our proposed testing approach is not implemented.

Regarding the features of the DCO Analyzer we intend to improve the enforcement heuristics and algorithms, for common patterns of problems and solutions.

Appendix A

Testing Distributed and Heterogeneous Systems – State of the practice - Survey Form

To explore the testing of distributed and heterogeneous systems from the point of view of industry practitioners, in order to assess the current state of the practice and identify opportunities and priorities for research and innovation initiatives we conducted an survey in two industry-oriented conferences in the software testing area. The form used in these questionnaires is presented below.

Testing Distributed and Heterogeneous Systems – State of the practice

Dear Sir/Madam,

We are contacting you to participate in this survey due to your invaluable knowledge and experience in software development and testing.

This survey is part of a research project carried out at the Software Engineering Research Group (http://softeng.fe.up.pt/) at the Faculty of Engineering, University of Porto, Portugal.

In the context of this survey we define a Distributed and Heterogeneous System as a set of small independent systems that together form a new distributed system, combining hardware components and software system, possibly involving mobile and cloud-based platforms.

By sharing your expertise with us, you will get access to the expert opinions of others (in terms of aggregated results), see how your opinion relates to theirs, and better understand the current state of the art in testing Distributed and Heterogeneous Systems.

We sincerely appreciate your experience and expert opinion. Your identity and individual answers will be kept anonymous, only aggregated results will be presented.

Please do not hesitate to contact us if you need any additional information.

Thank you!

Yours faithfully,
Bruno Lima (bruno.lima@fe.up.pt)
João Pascoal Faria (jpf@fe.up.pt)

If you prefer, can also answer this survey in:

https://goo.gl/GExS2w

Conducted by: U.PORTO FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO INESCTEC

Supported by: PSTQB ASSOCIAÇÃO PORTUGUESA DE TESTES DE SOFTWARE Partilhar o conhecimento para evoluir

*Mandatory

1- Professional Characterization

1.1 - Which is your main responsibility in your current position? *

- Software testing, verification & validation
- Software developer, architect or analyst
- Project manager
- Other: ____

1.2 - How long have you been working in your current position? *

- Less than 1 year
- Between 1 and 2 years
- Between 2 and 5 years
- More than 5 years

1.3 - How long have you been working with software testing? *

- I've never worked with software testing
- Less than 1 year
- Between 1 and 2 years
- Between 2 and 5 years
- More than 5 years

1.4 - How long have you been working with distributed and heterogeneous systems? *

- I've never worked with testing of distributed and heterogeneous systems
- Less than 1 year
- Between 1 and 2 years
- Between 2 and 5 years
- More than 5 years

1.5 - In case you want to receive the aggregated results of this questionnaire, please provide us an email address (which will be kept private):

____

2 - Company Characterization

2.1 - In what industry is your company working? *

- ICT - Products and Services
- Education and Research
- Government and Military
- Healthcare
- Transportation
- Finance
- Entertainment/Tourism
- Energy
- Other. ____

2.2 - What is the size of your company? *

- Less than 10 collaborators
- Between 10 and 99 collaborators
- Between 100 and 1000 collaborators
- More than 1000 collaborators

2.3 - In what role(s) does your company conducts software test, if any?

- As developer
- As costumer/user
- As test services provider (independent tester)
- As system integrator

2.4 - Which software test levels are performed in your company, if any?

- Unit testing
- Integration testing
- System testing
- Acceptance testing

3 - Testing Distributed and Heterogeneous Systems

3.1 - In what role(s) does your company conduct software tests, if any?

- As developer
- As costumer/user
- As test services provider (independent tester)
- As system integrator

3.2 - About these systems, which software test levels are performed in your company, if any?

- Unit testing
- Integration testing
- System testing
- Acceptance testing

3.3 - What is the level of test automation for these systems?

- Only manual testing
- Automatic test execution (with manual test scripting/coding)
- Automatic test generation (with manual execution)
- Automatic test generation and execution

3.4- If you selected "Automatic test ..." in the previous question, what type of tools are used?

- Commercial off-the-shelf tool
- Developed/adapted in-house tool, reusable for different systems under test (SUTs)
- Developed/adapted in-house tool, tailor-made for the SUT

3.5 - Please rate the degree of importance of testing each of the following features of distributed and heterogeneous systems: *

	VERY SMALL	SMALL	MEDIUM	HIGH	VERY HIGH
INTERACTIONS BETWEEN THE SYSTEM AND THE ENVIRONMENT	○	○	○	○	○
INTERACTIONS BETWEEN COMPONENTS OF THE SYSTEM	○	○	○	○	○
PARALLELISM AND CONCURRENCY	○	○	○	○	○
TIME CONSTRAINTS	○	○	○	○	○
NON DETERMINISTIC BEHAVIORS	○	○	○	○	○
MULTIPLE PLATFORMS	○	○	○	○	○

3.6 - Please rate the degree of importance of each of the following features of a test automation solution for distributed and heterogeneous systems: *

	VERY SMALL	SMALL	MEDIUM	HIGH	VERY HIGH
SUPPORT FOR AUTOMATIC TEST CASE EXECUTION	○	○	○	○	○
SUPPORT FOR AUTOMATIC TEST CASE GENERATION	○	○	○	○	○
SUPPORT FOR TEST COVERAGE ANALYSIS	○	○	○	○	○
SUPPORT FOR AUTOMATIC TEST STUB GENERATION	○	○	○	○	○
SUPPORT FOR MULTIPLE PLATFORMS	○	○	○	○	○

3.7 - If there was a tool that could test a distributed and heterogeneous system using only a model of interactions (UML sequence diagram) as an entry model, would you find it useful?

- ○ Yes
- ○ No

4 – In case you have any comments, suggestions or further information please tell us in the following box:

Appendix B

Observability and Controllability in Scenario-based Integration Testing of Time- Constrained Distributed Systems: VDM++ Specifications

This report presents the complete specification in VDM++ of the local observability and local controllability analysis and enforcement procedures and associated test cases described in this thesis.

Observability and Controllability in Scenario-based Integration Testing of Time-Constrained Distributed Systems: VDM++ Specifications

Bruno Lima, João Pascoal Faria

FEUP, December 2020

Contents

1. Introduction 3
2. Class Utils 5
3. Class DifferenceConstraints 6
4. Class SequenceDiagrams 13
5. Class Traces 18
6. Class ValidTraces 24
7. Class ConformanceChecking 30
8. Class SimulatedExecution 32
9. Class Observability 38
10. Class Controllability 40
11. Class Enforcement 42
12. Class TestCases 62
13. References 85

1. Introduction

This report presents the complete specification in VDM++ [1] of the local observability and local controllability analysis and enforcement procedures and associated test cases described in the thesis [2].

The specification follows a combination of the functional and imperative styles supported by VDM++. Classes are used simply as modules. The imperative style is used in some cases for performance reasons.

The test cases can be executed with the Overture interpreter [1].

In this document, it used to the extent possible the mathematical notation of VDM++ supported by Overture [1], as indicated in the next table.

[1] http://overturetool.org/

Category	ASCII notation	Mathematical notation
Type definition	set of T	T-set
	map U to T	$U \xrightarrow{m} T$
	seq of T	T*
	seq1 of T	T+
	nat	ℕ
	int	ℤ
	bool	𝔹
	real	ℝ
Set operators and literals	in set	∈
	not in set	∉
	inter	∩
	union	∪
	subset	⊆
	dinter	⋂
	dunion	⋃
	{}	∅
	iota	℩
	card	#
Sequence operators	in seq	∈
	^	↷
Map operators	munion	∪
	|->	↦
	++	†
	:>	▷
	<:	◁
	:->	⩥
	<-:	⩤
Record operations	mu	μ
Logical and comparison operators	forall	∀
	exists	∃
	not exists	∄
	and	∧
	or	∨
	not	¬
	<=>	⇔
	=>	⇒
	<>	≠
	<=	≤
	>=	≥
Function and operation definition	->	⟶
	==>	⟶
	==	≜
	*	×
	lambda	λ
	&	•

2. Class Utils

```
/**
 * Common utilities.
 */

class Utils

types
public String = char*;

functions

-- Obtains the minimum of a non-empty set (s) of elements of type @T.
public min[@T]: @T-set ⟶ @T
min(s) ≜ ι x ∈ s • ∄ y ∈ s • y < x
pre s ≠ ∅;

-- Obtains the maximum of a non-empty set (s) of elements of type @T.
public max[@T]: @T-set ⟶ @T
max(s) ≜ ι x ∈ s • ∄ y ∈ s • y > x
pre s ≠ ∅;

-- Sorts a set (s) of elements of type @T given a comparision function (compFunc).
public sort[@T]: @T-set × (@T × @T ⟶ ℤ) ⟶ @T*
sort(s, compFunc) ≜
  if s = ∅ then []
  else let x ∈ s be st ∄ y ∈ s • compFunc(y, x) < 0
       in [x] ↷ sort[@T](s \ {x}, compFunc);

-- Computes the transitive reflexive closure of a set (S) by a binary relation (R).
public getTransitiveReflexiveClosure[@T]: @T-set × (@T × @T)-set ⟶ @T-set
getTransitiveReflexiveClosure(S, R) ≜
  let next = {y | mk_(x, y) ∈ R • x ∈ S} ∪ S
  in if next = S then S else getTransitiveReflexiveClosure[@T](next, R);

-- Converts a set of values to a string.
public set2str[@T]: @T-set × (@T ⟶ String) ⟶ String
set2str(s, elem2str) ≜
  if s = ∅ then "{}"
  else let t ∈ s in
         if # s = 1 then "{" ↷ elem2str(t) ↷ "}"
         else "{" ↷ elem2str(t) ↷ ", " ↷ tl set2str[@T](s \ {t}, elem2str);

-- Converts a sequence of values to a string.
public seq2str[@T]: @T* × (@T ⟶ String) ⟶ String
seq2str(t, elem2str) ≜
  if t = [] then "[]"
  else if len t = 1 then "[" ↷ elem2str(hd t) ↷ "]"
  else "[" ↷ elem2str(hd t) ↷ ", " ↷ tl seq2str[@T](tl t, elem2str);

-- Converts a pair of values to a string.
public pair2str[@T1, @T2]: (@T1 × @T2) × (@T1 ⟶ String) × (@T2 ⟶ String)⟶ String
pair2str(mk_(first, second), first2str, second2str) ≜
  "(" ↷ first2str(first) ↷ ", " ↷ second2str(second) ↷ ")";

end Utils
```

3. Class DifferenceConstraints

```
/**
 * Manipulation of difference constraints (DC) on integer time variables
 * and boolean combinations in disjunctive normal form (DNF).
 */

class DifferenceConstraints

/*** Representation of difference constraints and expressions in DNF ***/

types
public VariableId = ℕ;   -- time variable identifier
public TimeValue  = ℕ;   -- time value in any desired scale (sec, mili, etc.)
public Duration   = ℤ;   -- difference between time values

-- Difference constraint, meaning vi - vj <= d
public DC :: i: VariableId
             j: VariableId
             d: Duration;

-- Expressions in Disjunctive-Normal-Form (DNF)
public OrExp  :: args: (AndExp | DC)-set;
public AndExp :: args: DC-set;
public DCExp = DC | AndExp | OrExp;

values
public FalseExp: DCExp = mk_OrExp(∅); -- existential quantifier on empty set
public TrueExp: DCExp = mk_AndExp(∅); -- unversal quantifier on empty set


/*** Creation and simplificatin of difference constraint expressions ***/

-- Given a set of expressions in DNF, returns the conjunction normalized in DNF
-- (partially simplified).
operations
public static pure mkAndExp: DCExp-set ⟶ DCExp
mkAndExp(args) ≜ (
  dcl left : DCExp;

  -- special case (absorbing element)
  if FalseExp ∈ args then
    return FalseExp;

  -- process conjunctive or terminal arguments
  left := mk_AndExp(simplifyDC(⋃{if is_AndExp(e) then e.args else {e} |
                                      e ∈ args • ¬ is_OrExp(e)}));

  -- process one disjunctive argument at a time
  for all right ∈ args do
    if is_OrExp(right) then
      if left = TrueExp then
             left := right
      else (
        -- applies distributive property
        left := mkOrExp({mkAndExpAux(e1, e2) |
                           e1 ∈ (if is_OrExp(left) then left.args else {left}),
                           e2 ∈ right.args});
```

```
        -- aborts with absorbing element, i.e., FalseExp
        if left = FalseExp then
          return FalseExp
    );

    return left
 );

-- Produces the conjunction (partially simplified) of two non-disjunctive expressions.
-- Does a partial check for contradictions.
functions
mkAndExpAux: (AndExp | DC) × (AndExp | DC) ⟶ DCExp
mkAndExpAux(exp1, exp2) ≜
  let args1 = if is_AndExp(exp1) then exp1.args else {exp1},
      args2 = if is_AndExp(exp2) then exp2.args else {exp2}
  in
    if ∃ mk_DC(i, j, d1) ∈ args1 •
        ∃ mk_DC((j), (i), d2) ∈ args2 • d1 + d2 < 0 then
      FalseExp
    else
      mk_AndExp(simplifyDC(args1 ∪ args2));

-- Creates a conjunctive expression (partially simplified) based on a set of difference
constraints.
public mkAndExpDC: DC-set  ⟶ DCExp
mkAndExpDC(args) ≜ mk_AndExp(simplifyDC(args));

-- Given an expression in DNF, returns the negation in DNF (partially simplified).
public mkNotExp: DCExp ⟶ DCExp
mkNotExp(exp) ≜
  if is_AndExp(exp) then mkOrExp({mkNotExp(arg) | arg ∈ exp.args})
  else if is_OrExp(exp) then mkAndExp({mkNotExp(arg) | arg ∈ exp.args})
  else mk_DC(exp.j, exp.i, -(exp.d + 1));

-- Creates a DNF expression (partially simplified) for the disjunction
-- of a set of expressions in DNF.
public mkOrExp: DCExp-set ⟶ DCExp
mkOrExp(args) ≜
  let flatten = ⋃ {arg.args | arg ∈ args • is_OrExp(arg)}
                ∪ {arg | arg ∈ args • ¬ is_OrExp(arg)},
      non_redundant = {a | a ∈ flatten • ∄ b ∈ flatten •
                           a ≠ b ∧ impliesDC(if is_AndExp(a) then a.args else {a},
                                             if is_AndExp(b) then b.args else {b})}
  in
     -- in case of a single term, doesn't need or'ing
    if # non_redundant = 1 then (let arg ∈ non_redundant in arg)
    else mk_OrExp(non_redundant); -- normal case

-- Partially simplifies a set of difference constraints by removing redundant
-- constraints (because they hold trivially or are implied directly by others).
-- Assumes implicit ordering constraints.
simplifyDC: DC-set ⟶ DC-set
simplifyDC(C) ≜
  {mk_DC(i, j, d) | mk_DC(i, j, d) ∈ C •
      ¬ (i ≤ j ∧ d ≥ 0) ∧
      ¬ (∃ mk_DC(i2, j2, d2) ∈ C • mk_DC(i, j, d) ≠ mk_DC(i2, j2, d2)
            ∧ i ≤ i2 ∧ j ≥ j2 ∧ d2 ≤ d)};-- because of implicit ordering
```

```
-- Given expressions 'a' and 'b', obtains the expression corresponding to 'a and not b'.
public mkExceptExp: DCExp × DCExp ⟶ DCExp
mkExceptExp(a, b) ≜
  if a = b then FalseExp  -- optimization
  else if b = FalseExp then a -- optimization
  else mkAndExp({a, mkNotExp(b)});

-- Given expressions 'a' and 'b', obtains the expression corresponding to 'not a or b'.
public mkImpliesExp: DCExp × DCExp ⟶ DCExp
mkImpliesExp(a, b) ≜
  if a = TrueExp ∨ b = TrueExp then b  -- optimization
  else mkOrExp({mkNotExp(a), b});

-- Checks if a set (conjunction) of difference constraints, implies another set
-- (conjunction). Assumes implicit ordering constraints.
public impliesDC: DC-set × DC-set ⟶ 𝔹
impliesDC(args1, args2) ≜
  ∀ mk_DC(i2, j2, d2) ∈ args2 •
       ∃ mk_DC(i1, j1, d1) ∈ args1 •
         i2 ≤ i1 ∧ j2 ≥ j1 ∧ d2 ≥ d1;


/*** Satisfiability checking ***/

-- Checks the satisfiability of an expression in DNF.
-- The first version assumes implicit ordering constraints between (numbered) variables,
-- so the problem  is to check if the there is an assignment of non-decreasing values to
-- the variables that satisfy the given expression.
-- The second version doesn’t assume implicit ordering constraints.
operations
public static pure sat: DCExp ⟶ 𝔹
sat(exp) ≜ satMain(exp, true);

public static pure satRaw: DCExp ⟶ 𝔹
satRaw(exp) ≜ satMain(exp, false);

static pure satMain: DCExp  × 𝔹 ⟶ 𝔹
satMain(exp, implicitOrdering) ≜ (
  dcl vertices: VariableId*;
  dcl dist : VariableId →(m) Duration;
  dcl changed : 𝔹;

  -- special cases
  if is_OrExp(exp) then
    return ∃ arg ∈ exp.args • satMain(arg, implicitOrdering);

  if is_DC(exp) then
    return exp.i ≠ exp.j ∨ exp.d ≥ 0;

  if ∀ mk_DC(-,-,d) ∈ exp.args • d ≥ 0 then
     return true; -- satisfiable by assigning 0 to all variables

  -- ordered vertex set
  vertices := [i | i ∈ {c.i | c ∈ exp.args} ∪ {c.j | c ∈ exp.args} ];

  -- Bellman-Ford algorithm to find shortest paths from a (artificial) source vertex to
  -- all vertices in the presence of edges of negative weight
  dist := {v ↦ 0 | v ∈ vertices}; -- start with 0 because of implicit ordering edges
```

```
  for i = 1 to len vertices - 1 do (
    changed := false;
    -- process the given edges
    for all mk_DC(u, v, d) ∈ exp.args do
      if dist(v) > dist(u) + d then (
        dist(v) := dist(u) + d;
        changed := true;
      );

    -- optimization
    if ¬ changed then
      return true;

    -- propagate changes using implicit ordering constraints, in order
    if implicitOrdering then (
      dcl min: Duration := 0;
      for v in vertices do
        if dist(v) > min then
          dist(v) := min
        else
          min := dist(v)
    );
  );

  -- If didn't converge, there are negative loops, so exp is not satisfiable
  return ∄ mk_DC(u, v, d) ∈ exp.args • dist(v) > dist(u) + d
);

-- Reduces an expression by eliminating non-satisfiable terms.
functions
public red: DCExp  ⟶ DCExp
red(exp) ≜ (
  if is_OrExp(exp) then
      let feasible = {arg | arg ∈ exp.args • sat(arg)}
      in if # feasible = 1 then let arg ∈ feasible in arg
         else mk_OrExp(feasible)
  else if sat(exp) then exp else FalseExp
);


/*** Variable elimination in difference constraint expressions ***/

-- Eliminates variables after a given one in a given expression (i.e., projects the
-- expression onto the variables that remain).
-- Assumes implicit ordering constraints between consecutively numbered vertices.
operations
public static pure elimVarsAfter:  VariableId × DCExp ⟶ DCExp
elimVarsAfter(maxV, c) ≜
  return if is_OrExp(c) then mkOrExp({elimVarsAfter(maxV, arg) | arg ∈ c.args})
         else mk_AndExp(elimVarsAfter(maxV, if is_AndExp(c) then c.args else {c}));

-- Eliminates variables after a given one (v) in a set of difference constraints (C).
-- Assumes implicit ordering constraints between consecutively numbered vertices.
public static pure elimVarsAfter: VariableId × DC-set ⇨ DC-set
elimVarsAfter(v, C) ≜ (
   dcl C2: DC-set;
   dcl vars : VariableId*;

   -- special cases
```

```
   if ∄ mk_DC(i, j, -) ∈ C • i < v ∨ j < v then
     return ∅;

   -- relevant vertices in constraint graph (referenced vertices plus v), sorted
   vars := [v] ⁀ [k | k ∈ {i | mk_DC(i,-,-) ∈ C • i>v} ∪ {j | mk_DC(-,j,-) ∈ C • j>v}];
   if len vars = 1 then
     return C;

   -- removes one variable/vertex at a time from right to left
   -- (shortcircuiting constraints/edges)
   C2 := C;
   for idx = len vars to 2 by -1 do
     let e = vars(idx) in
        C2 := {mk_DC(i1, j2, d1 + d2) | mk_DC(i1, (e), d1),
                     mk_DC((e), j2, d2) ∈ C2 • i1 ≠ e ∧ j2 ≠ e}
             ∪ {mk_DC(i, j, d) | mk_DC(i, j, d) ∈ C2 • i ≠ e ∧ j ≠ e}
             ∪ {mk_DC(vars(idx-1), j, d) | mk_DC((e), j, d) ∈ C2 • j ≠ e};

    -- simplifies
    C2 := simplifyDC(C2);

    return C2
 );

-- Projects an expression (satisfiable) onto a set of variables (eliminating other
-- variables), and renumbers the variables to sequential numbers starting in 1.
public static pure projectToVars: DCExp × VariableId* ⟶ DCExp
projectToVars(c, V) ≜ (
  dcl C : DC-set;
  dcl vars : VariableId*;

  if is_OrExp(c) then
    return mkOrExp({projectToVars(arg, V) | arg ∈ c.args});

  -- optimization
  if len V ≤ 1 ∨ c = TrueExp then
    return TrueExp;

  -- set of difference constraints
  C:= if is_AndExp(c) then c.args else {c};

  -- sorted list of relevant variables (mentioned in constraints and range)
  vars := [i | i ∈ elems V ∪ {i | mk_DC(i, -, -) ∈ C} ∪ {j | mk_DC(-, j, -) ∈ C}];

  -- add implicit ordering constraints
  C := C ∪ {mk_DC(vars(i), vars(i+1), 0) | i ∈ {1,..., len vars-1}
                 • ∄ mk_DC((vars(i)), (vars(i+1)), d) ∈ C • d ≤ 0};

  -- remove unwanted variables (shortcircuiting constraints/edges)
  for all v ∈ elems vars \ elems V do
    C :=  {mk_DC(i1, j2, d1 + d2) |
              mk_DC(i1, (v), d1), mk_DC((v), j2, d2) ∈ C • i1 ≠ v ∧ j2 ≠ v}
          ∪ {mk_DC(i, j, d) | mk_DC(i, j, d) ∈ C • i ≠ v ∧ j ≠ v};

  -- simplify and remove implicit ordering constraints, and then
  -- renumber (pack) variables sequentially, mapping old to new numbers
  return mk_AndExp(renumVars({V(i) ↦ i | i ∈ inds V}, simplifyDC(C)))
);
```

```
/*** Variable renumbering in difference constraint expressions ***/

-- Renumbers the variables in a set C of difference constraints, based on a given
-- map or sequence from old ids to new ids.
functions
public renumVars: (VariableId →ᵐ VariableId | VariableId*) × DC-set ⟶ DC-set
renumVars(renum, C) ≜ {mk_DC(renum(i), renum(j), d)| mk_DC(i, j, d) ∈ C};

-- Renumbers a single variable.
public renumVar: DCExp × VariableId × VariableId ⟶ DCExp
renumVar(exp, oldNum, newNum) ≜
  if is_OrExp(exp) then
    mk_OrExp({renumVar(arg, oldNum, newNum) | arg ∈ exp.args})
  else if is_AndExp(exp) then
    mk_AndExp({renumVar(arg, oldNum, newNum) | arg ∈ exp.args})
  else let mk_DC(i, j, d) = exp in
    mk_DC(if i = oldNum then newNum else i, if j = oldNum then newNum else j, d);


/*** Maximum and minimum difference between two variables ***/

-- Obtains the maximum difference between two variables (j - i <= ?) as
-- determined by a difference constraint expression (nil if there is no upper limit).
public getMaxDiff: DCExp × VariableId × VariableId ⟶ [Duration]
getMaxDiff(exp, i, j) ≜ getMaxDiffAux(projectToVars(exp, [i, j]))
pre i < j;

getMaxDiffAux: DCExp ⟶ [Duration]
getMaxDiffAux(exp) ≜
  if is_OrExp(exp) then getMax({getMaxDiffAux(e) | e ∈ exp.args})
  else if is_AndExp(exp) then getMin({getMaxDiffAux(e) | e ∈ exp.args})
  else if exp.i = 2 ∧ exp.j = 1 then exp.d
  else nil;

-- Obtains the minimum difference between two variables (j - i >= min or i - j <= -min)
-- as determined by a difference constraint expression (0 if there is no lower limit).
public getMinDiff: DCExp × VariableId × VariableId ⟶ Duration
getMinDiff(exp, i, j) ≜
  getMinDiffAux(projectToVars(exp, [i, j]))
pre i < j;

getMinDiffAux: DCExp ⟶ [Duration]
getMinDiffAux(exp) ≜
  if is_OrExp(exp) then getMin({getMinDiffAux(e) | e ∈ exp.args})
  else if is_AndExp(exp) then getMax({getMinDiffAux(e) | e ∈ exp.args})
  else if exp.i = 1 ∧ exp.j = 2 then -exp.d
  else 0;

-- Obtains the maximum of a set of (non-negative) durations, possibly including nil
-- (treated as infinity).
-- The maximum of an empty set is the minimum of the scale, i.e., zero.
operations
static pure getMax: [Duration]-set ⟶ [Duration]
getMax(s) ≜ (
   dcl m : [Duration] := 0; -- start with min (0)
   for all d ∈ s do
      if d = nil then
```

```
         return nil
      else if d > m then
         m := d;
   return m;
);

-- Obtains the minimum of a set of durations, possibly with nil (treated as infinity).
-- The minimum of an empty set is the maximum of the scale, i.e., infinity (nil).
static pure getMin: [Duration]-set ⟶ [Duration]
getMin(s) ≜ (
   dcl m : [Duration] := nil; -- start with max (infinity)
   for all d ∈ s do
      if d ≠ nil ∧ (m = nil ∨ d < m) then
         m := d;
   return m
);

end DifferenceConstraints
```

4. Class SequenceDiagrams

```
/**
* Specification of UML Sequence Diagrams (UML Interactions) used for describing
* integration test scenarios of time-constrained distributed systems,
* primitives for conformance checking and test input selection,
* primitives for local observability and local controlability analysis and enforcement,
* and examples.
**/

class SequenceDiagrams is subclass of DifferenceConstraints, Utils

/*** Configuration parameters ***/

-- Semantic variation point: FIFO channel between each pair of lifelines
values
public static FIFO_CHANNELS = false;

-- Maximum difference between clocks associated with different lifelines
instance variables
public static MaxClockSkew : TimeValue := 10;  -- e.g., 10 ms

types

/*** Values, Value Specifications, and Timing (based on UML metamodel) ***/

public Value = ℕ | 𝔹 | ℝ | String;

public ValueSpecification = Value | Variable | Expression | <Unknown>;
public Variable :: name: String;
public Expression :: symbol: ExpSymbol
                     operands: [ValueSpecification]*;
public ExpSymbol = <Neg> | <Eq> | <Plus> | <Minus> | <Lt> | <Lte> | <Gt> | <Gte> | <And>
| <Or>;

public TimeInterval = [TimeValue] × [TimeValue];
public DurationInterval = [Duration] × [Duration];


/*** UML Interactions (based on UML meta-model) ***/

public Interaction ::
  lifelines          : Lifeline-set
  messages           : Message-set
  combinedFragments : CombinedFragment-set
  timeConstraints    : TimeConstraint-set
                                m
  messageMap         : MessageId → Message
inv i ≜
  -- message ids and send and receive locations are unique
  (∀ m1, m2 ∈ i.messages • m1 ≠ m2 ⇒
       m1.id ≠ m2.id
       ∧ m1.sendEvent ≠ m2.sendEvent
       ∧ m1.receiveEvent ≠ m2.receiveEvent)

  -- lifeline names are unique
  ∧ (∀ l1, l2 ∈ i.lifelines • l1 ≠ l2 ⇒ l1.name ≠ l2.name)
```

```
  -- referenced lifelines exist
  ∧  (∀ m ∈ i.messages • {m.sendEvent.#1, m.receiveEvent.#1} ⊆ i.lifelines)
  ∧  (∀ c ∈ i.combinedFragments • c.lifelines ⊆ i.lifelines)

  -- time variables are unique
  ∧ (∀ m1, m2 ∈ i.messages • m1 ≠ m2 ⇒
       let l = [m1.sendTimestamp, m1.recvTimestamp, m2.sendTimestamp, m2.recvTimestamp]
       in ∄ i, j ∈ inds l • i ≠ j ∧ l(i) ≠ nil ∧ l(j) ≠ nil ∧ l(i) = l(j))
  ∧ (∀ m ∈ i.messages • m.sendTimestamp ≠ nil ∧ m.recvTimestamp ≠ nil ⇒
                          m.sendTimestamp ≠ m.recvTimestamp);

public Lifeline :: name : String
                   actor : 𝔹;

public MessageType = <Synch> | <Asynch>;

public Message ::
  id            : MessageId
  sendEvent     : LifelineLocation
  receiveEvent  : LifelineLocation
  signature     : MessageSignature
  sendTimestamp : [Variable]
  recvTimestamp : [Variable]
  type          : MessageType
  guard         : [TimeConstraint]
inv m ≙ m.sendEvent ≠ m.receiveEvent;

public MessageSignature = String;
public MessageId = ℕ;
public Location = ℕ;
public LifelineLocation = Lifeline × Location;

public CombinedFragment ::
  interactionOperator : InteractionOperatorKind
  operands            : InteractionOperand+
  lifelines           : Lifeline-set
inv f ≙
  -- number of operands and guards allowed depend on the interaction operator
  cases f.interactionOperator:
    <loop>, <opt>, <sloop> ⟶ len f.operands = 1,
    <alt>, <par>, <strict>, <seq> ⟶
        len f.operands > 1 ∧ ∀ op ∈ f.operands • op.guard = nil
  end
  -- the lifelines covered by the combined fragment must be the same as the ones
  -- covered by its operands
  ∧ (∀ o ∈ f.operands •
         {lf | mk_(lf, -) ∈ o.startLocations} = f.lifelines
         ∧ {lf | mk_(lf, -) ∈ o.finishLocations} = f.lifelines)
  -- the finish locations of an operand must equal the start locations of the next
  -- operand
  ∧ (∀ i ∈ {1, ..., len f.operands - 1} •
          f.operands(i+1).startLocations = f.operands(i).finishLocations);

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

public InteractionOperand ::
  guard            : [InteractionConstraint]
  startLocations   : LifelineLocation-set
```

```
  finishLocations  : LifelineLocation-set;

public InteractionConstraint ::
  minint       : [ValueSpecification]  -- loop
  maxint       : [ValueSpecification]  -- loop
  specification: [ValueSpecification] | <else>;

public TimeConstraint ::
  firstEvent : Variable
  secondEvent: Variable
  min        : [Duration]
  max        : [Duration]
inv tc ≜ tc.min ≠ nil ∨ tc.max ≠ nil;

functions

/*** Helper functions for creating Interactions, Messages, and Lifelines ***/

-- Creates an interaction.
public mkInteraction : Lifeline-set × Message-set × CombinedFragment-set ×
                       TimeConstraint-set ⟶ Interaction
mkInteraction(lifelines, messages, combinedFragments, timeConstraints) ≜
      mk_Interaction(lifelines, messages, combinedFragments, timeConstraints, {↦}, ∅);

public mkInteraction : Lifeline-set × Message-set × CombinedFragment-set ⟶ Interaction
mkInteraction(lifelines, messages, combinedFragments) ≜
      mkInteraction(lifelines, messages, combinedFragments, ∅);

protected mkMessage: MessageId × LifelineLocation × LifelineLocation ×
                     MessageSignature ⟶ Message
mkMessage(id, sendEvent, receiveEvent, signature) ≜
  mk_Message(id, sendEvent, receiveEvent, signature, nil, nil, <Asynch>, nil);

protected mkMessageTimed: MessageId × LifelineLocation × LifelineLocation ×
                          MessageSignature ⟶ Message
mkMessageTimed(id, sendEvent, receiveEvent, signature) ≜
  mk_Message(id, sendEvent, receiveEvent, signature,
             mk_Variable("s_" ⁀ signature ⁀ VDMUtil`val2seq_of_char[MessageId](id)),
             mk_Variable("r_" ⁀ signature ⁀ VDMUtil`val2seq_of_char[MessageId](id)),
<Asynch>,nil);

protected mkMessageTimedGuarded: MessageId × LifelineLocation × LifelineLocation ×
                          MessageSignature × TimeConstraint ⟶ Message
mkMessageTimedGuarded(id, sendEvent, receiveEvent, signature, guard) ≜
  mk_Message(id, sendEvent, receiveEvent, signature,
    mk_Variable("s_" ⁀ signature ⁀ VDMUtil`val2seq_of_char[MessageId](id)),
    mk_Variable("r_" ⁀ signature ⁀ VDMUtil`val2seq_of_char[MessageId](id)),
<Asynch>,guard);

protected mkMessageTimedSynch: MessageId × LifelineLocation × LifelineLocation ×
                               MessageSignature ⟶ Message
mkMessageTimedSynch(id, sendEvent, receiveEvent, signature) ≜
  mk_Message(id, sendEvent, receiveEvent, signature,
             mk_Variable("s_" ⁀ signature ⁀ VDMUtil`val2seq_of_char[MessageId](id)),
             mk_Variable("r_" ⁀ signature ⁀ VDMUtil`val2seq_of_char[MessageId](id)),
<Synch>, nil);

public mkLifeline: String ⟶ Lifeline
```

```
mkLifeline(name) ≜ mk_Lifeline(name, false);

public mkActor: String ⟶ Lifeline
mkActor(name) ≜ mk_Lifeline(name, true);

-- Builds the messageMap index from message ids to messages in an interaction.
protected buildMessageMap: Interaction ⟶ Interaction
buildMessageMap(sd) ≜ μ(sd, messageMap ↦ {m.id ↦ m | m ∈ sd.messages});


/*** Containment checking functions ***/

protected contains: CombinedFragment × CombinedFragment ⟶ 𝔹
contains(f1, f2) ≜
  contains(f1.operands(1).startLocations, f1.operands(len f1.operands).finishLocations,
           f2.operands(1).startLocations, f2.operands(len f2.operands).finishLocations);

protected contains: InteractionOperand × CombinedFragment ⟶ 𝔹
contains(o, c) ≜
  contains(o.startLocations, o.finishLocations,
          c.operands(1).startLocations, c.operands(len c.operands).finishLocations);

protected contains: InteractionOperand × InteractionOperand ⟶ 𝔹
contains(o1, o2) ≜
  contains(o1.startLocations, o1.finishLocations,
           o2.startLocations, o2.finishLocations);

protected contains: InteractionOperand × LifelineLocation ⟶ 𝔹
contains(o, lfloc) ≜ contains(o.startLocations, o.finishLocations, lfloc);

protected contains: CombinedFragment × LifelineLocation ⟶ 𝔹
contains(f, lfloc) ≜ contains(f.operands(1).startLocations,
                               f.operands(len f.operands).finishLocations, lfloc);

protected contains: LifelineLocation-set × LifelineLocation-set × LifelineLocation ⟶ 𝔹
contains(startLocs, endLocs, mk_(lf, loc)) ≜
  (∃ mk_(lf1, loc1) ∈ startLocs • lf1 = lf ∧ loc1 < loc)
  ∧ (∃ mk_(lf2, loc2) ∈ endLocs • lf2 = lf ∧ loc2 > loc);

protected contains: LifelineLocation-set × LifelineLocation-set ×
                    LifelineLocation-set × LifelineLocation-set ⟶ 𝔹
contains(startLocs1, endLocs1, startLocs2, endLocs2) ≜
  (∀ mk_(lf2, loc2) ∈ startLocs2 •
     ∃ mk_(lf1, loc1) ∈ startLocs1 • lf1 = lf2 ∧ loc1 < loc2)
  ∧ (∀ mk_(lf2, loc2) ∈ endLocs2 •
         ∃ mk_(lf1, loc1) ∈ endLocs1 • lf1 = lf2 ∧ loc1 > loc2);


/*** Miscelaneous functions on time constraints ***/

-- Checks if a difference constraint is a maximum duration constraint.
protected isMaxDuration: DC ⟶ 𝔹
isMaxDuration(mk_DC(i, j, d)) ≜ i > j;

-- Checks if there is a maximum duration constraint between two time variables.
public hasMaxDurationConstraint: Interaction × Variable × Variable ⟶ 𝔹
hasMaxDurationConstraint(sd, e1_timestamp, e2_timestamp) ≜
  ∃ c ∈ sd.timeConstraints •
```

```
      c.firstEvent = e1_timestamp ∧ c.secondEvent = e2_timestamp ∧ c.max ≠ nil;

-- Checks if there is a minimum duration constraint between two time variables.
public hasMinDurationConstraint: Interaction × Variable × Variable ⟶ 𝔹
hasMinDurationConstraint(sd, e1_timestamp, e2_timestamp) ≜
  ∃ c ∈ sd.timeConstraints •
    c.firstEvent = e1_timestamp ∧ c.secondEvent = e2_timestamp ∧ c.min ≠ nil;

-- Obtain the minimum duration between two time variables.
public getMinDurationConstraint: Interaction × Variable × Variable ⟶ [ℕ]
getMinDurationConstraint(sd, e1_timestamp, e2_timestamp) ≜
  let s = {c.min | c ∈ sd.timeConstraints •
             c.firstEvent =  e1_timestamp ∧ c.secondEvent = e2_timestamp ∧ c.min ≠ nil}
  in if s = ∅ then nil else max[Duration](s);

-- Obtain the maximum duration between two time variables.
public getMaxDurationConstraint: Interaction × Variable × Variable ⟶ [ℕ]
getMaxDurationConstraint(sd, e1_timestamp, e2_timestamp) ≜
  let s = {c.max | c ∈ sd.timeConstraints •
             c.firstEvent =  e1_timestamp ∧ c.secondEvent = e2_timestamp ∧ c.max ≠ nil}
  in if s = ∅ then nil else min[Duration](s);

-- Simplifies the set of time constraints in an interaction.
public simplifyTimeConstraints: Interaction ⟶ Interaction
simplifyTimeConstraints(sd) ≜
  let pairs = {mk_(e1, e2) | mk_TimeConstraint(e1, e2, min, max) ∈ sd.timeConstraints},
      newTimeConstraints = {mk_TimeConstraint(e1, e2,
                              getMinDurationConstraint(sd, e1, e2),
                              getMaxDurationConstraint(sd, e1, e2))
      | mk_(e1, e2) ∈ pairs }
  in μ(sd, timeConstraints ↦ newTimeConstraints);

end SequenceDiagrams
```

5. Class Traces

```
/**
 * Representation and manipulation of traces (plain, timed and time-constrained).
 */

class Traces is subclass of SequenceDiagrams


/*** Representation and construction of traces ***/

Types

public Trace = Event*;
protected TraceExt = EventExt*; -- Trace with extra info

public TCTrace = Trace × DCExp; -- Time constrained trace
protected TCTraceExt = TraceExt × DCExp; -- Time constrained trace with extra info

public Event ::
      type        : EventType
      signature   : MessageSignature
      lifeline    : Lifeline
      timestamp   : [Variable | TimeValue]; --Var. in event; Value in event occurrence

public EventType = <Send> | <Receive> | <Stop>;

protected EventExt ::
      type        : EventType
      signature   : MessageSignature
      lifeline    : Lifeline
      timestamp   : [ValueSpecification]
      location    : Location
      messageId   : ℕ
      itercounter: ℕ*
      messageType: MessageType;

functions

protected mkEvent: EventType × MessageSignature × Lifeline ⟶ Event
mkEvent(type, signature, lifeline) ≜ mk_Event(type, signature, lifeline, nil);

protected mkEvent: EventType × MessageSignature × Lifeline × [ValueSpecification] ⟶
Event
mkEvent(type, signature, lifeline, timestamp) ≜
  mk_Event(type, signature, lifeline, timestamp);

public mkStopEvent: Lifeline ⟶ Event
mkStopEvent(l) ≜ mk_Event(<Stop>, [], l, nil);


/*** Project (sets of) traces and tc-traces onto lifelines ***/

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

```
-- Projects a set of traces (T) onto a lifeline (l).
protected projectTraces: Trace-set × Lifeline ⟶ Trace-set
projectTraces(T, l) ≜ {projectTrace(t, l) | t ∈ T} ;

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

-- Projects a set T of tc-traces onto a set L of lifelines.
public projectTCTraces: TCTrace-set × Lifeline-set ⟶ Lifeline →(m) TCTrace-set
projectTCTraces(T, L) ≜ {l ↦ projectTCTraces(T, l) | l ∈ L};

-- Projects a set T of tc-traces onto a lifeline (l).
public projectTCTraces: TCTrace-set × Lifeline ⟶ TCTrace-set
projectTCTraces(T, l) ≜
  let P = {projectTCTrace(t, l) | t ∈ T}
  in {mk_(t, c) | mk_(t, c) ∈ P •
          ∄ mk_((t), c2) ∈ P • c2 ≠ c ∧ impliesDC(c.args, c2.args)};

-- Projects a tc-trace (t, c) onto a lifeline (l).
public projectTCTrace: TCTrace × Lifeline ⟶ TCTrace
projectTCTrace(mk_(t,c), l) ≜
  mk_(projectTrace(t, l), projectToVars(c, lifelineInds(l, t)));


/*** Subtraction of sets of tc-traces ***/

-- Subtracts two sets of tc-traces (S1 - S2).
public subtractTimedTraces: TCTrace-set × TCTrace-set ⟶ TCTrace-set
subtractTimedTraces(S1, S2) ≜
  ⋃ {let c2 = mkOrExp({c2 | mk_(t2, c2) ∈ S2 • eqIgnTimestamps(t1, t2)}),
         c3 = red(mkExceptExp(c1, c2))
     in if c3 ≠ FalseExp then {mk_(t1, c3)} else ∅
    | mk_(t1, c1) ∈ S1};


/*** Equality of sets of tc-traces and traces ***/

-- Checks if two sets of tc-traces are equivalent, i.e., represent the same set of
-- timed traces.
public areEquivalent: TCTrace-set × TCTrace-set ⟶ 𝔹
areEquivalent(S1, S2) ≜
  subtractTimedTraces(S1, S2) = ∅ ∧ subtractTimedTraces(S2, S1) = ∅;

-- Checks if two traces are equal, ignoring timestamps
protected eqIgnTimestamps: Trace × Trace ⟶ 𝔹
eqIgnTimestamps(t1, t2) ≜
  len t1 = len t2 ∧ ∀ i ∈ inds t1 • eqIgnTimestamps(t1(i), t2(i));

-- Check if two events are equal, ignoring timestamps
protected eqIgnTimestamps: Event × Event ⟶ 𝔹
eqIgnTimestamps(e1, e2) ≜ μ(e1, timestamp ↦ 0) = μ(e2, timestamp ↦ 0);


/*** Join (sets of) traces and tc-traces from different lifelines ***/

functions
```

```
-- Gives the feasible joins of traces from different lifelines (one per lifeline),
-- respecting the order of events per trace and message.
protected joinTraces: Interaction × Lifeline →(m) Trace ⟶ Trace-set
joinTraces(sd, localTraces) ≜
  joinTraces(sd, [], {l ↦ {localTraces(l)} | l ∈ dom localTraces});

-- Recursive version, with an accumulator for already processed events.
protected joinTraces: Interaction × Trace × Lifeline →(m) Trace-set ⟶ Trace-set
joinTraces(sd, left, m) ≜
  if m = {↦} then {left}
  else ⋃ { ⋃ {if t = [] then joinTraces(sd, left, {l} ⩤ m)
                else joinTraces(sd, left ⁀ [hd t], m † {l ↦ {tl t}})
               | t ∈ m(l) • t = [] ∨ isFeasibleAddition(sd, left, hd t)}
          | l  ∈ dom m};

-- Checks if an event occurrence is a feasible addition to a trace, i.e., respects the
-- fact that messages can only be received after being sent, and respects timestamp
-- ordering (for the given MaxClockSkew).
protected isFeasibleAddition: Interaction × Trace × Event ⟶ 𝔹
isFeasibleAddition(sd, t, e) ≜
  (e.type = <Receive> ⇒
     len [ 0 | mk_Event(<Send>, sig, -, -) ∈ t • sig = e.signature] >
     len [ 0 | mk_Event(<Receive>, sig, -, -) ∈ t • sig = e.signature])
  ∧ (len t > 0 ∧ t(len t).type = <Send>
      ⇒ ∃ m ∈ sd.messages • eqIgnTimestamps(s(m), t(len t)) ∧
          (m.type=<Asynch> ∨ eqIgnTimestamps(r(m), e)))
  ∧ (e.timestamp ≠ nil ∧ ¬ is_Variable(e.timestamp) ⇒
      ∀ f ∈ t • f.timestamp ≠ nil ⇒
          if f.lifeline = e.lifeline then f.timestamp ≤ e.timestamp
          else f.timestamp ≤ e.timestamp + MaxClockSkew);

-- Joins sets of tc-traces per lifeline, i.e., gives the set of all the possible
-- combinations of tc-traces from different lifelines, preserving the order of
-- events per lifeline and message (send before receive, and synchronous messages
-- respected), and such that the joined time constraints are satisfiable.
operations
protected static pure joinTimedTraces2: Interaction × Lifeline →(m) TCTrace-set ⟶
TCTrace-set
joinTimedTraces2(sd, M) ≜
(
  dcl res : TCTrace-set := ∅;
  let J = joinTimedTraces(sd,  M) in
    if ∃ m ∈ sd.messages • m.type = <Synch> then
    (
      for all mk_(t, c) ∈ J do
        if checkSyncMessagesPresent(sd, t) then
          let c2 = getSyncMessagesConstr(sd, t),
              c3 = red(mkAndExpDC(c.args ∪ c2))
          in if sat(c3) then
                res := res ∪ {mk_(t,c3)};
      return res
    )
    else
      return J;
);

-- Checks if the send and receive events of sync messages are present and contiguous
```

```
functions
checkSyncMessagesPresent: Interaction × Trace ⟶ 𝔹
checkSyncMessagesPresent(sd, t) ≜
  ∀ i ∈ inds t • t(i).type = <Send> ⇒
        let m = msg(sd, t(i)) in
           m.type = <Synch> ⇒ i < len t ∧ t(i+1) = r(m);

-- Obtains the time constraints corresponding to the synchronous messages in a trace t.
getSyncMessagesConstr: Interaction × Trace ⟶ DC-set
getSyncMessagesConstr(sd, t) ≜
  {mk_DC(i+1, i, 0) | i ∈ inds t • t(i).type = <Send> ∧ msg(sd, t(i)).type = <Synch>};

-- Joins sets of tc-traces from different lifelines.
protected joinTimedTraces: Interaction × Lifeline →m TCTrace-set ⟶ TCTrace-set
joinTimedTraces(sd, m) ≜ joinTimedTracesAux(sd, mk_([], TrueExp), m);

-- Recursive version, with an accumulator for a tc-trace constructed so far.
operations
static pure joinTimedTracesAux: Interaction × TCTrace × Lifeline →m TCTrace-set ⟶
TCTrace-set
joinTimedTracesAux(sd, mk_(t, c), m) ≜
(
  dcl result : TCTrace-set := ∅;
  dcl terminated : Lifeline-set := ∅;

  for all l ∈ dom m do
    for all mk_(t2, lc) ∈ m(l) do
      if t2 = [] then
          terminated := terminated ∪ {l}
      else if isFeasibleAddition(sd, t, hd t2) then
             let e = hd t2,
                 rt = tl t2,
                 newT = t ⁀ [e]
             in if lc.args = ∅ then
                  let newM = m † {l ↦ {mk_(rt, lc)}} -- restricts to this trace in l
                  in result := result ∪ joinTimedTracesAux(sd, mk_(newT, c), newM)
                else
                 let r = lifelineInds(l, newT),
                     C2 = elimVarsAfter(len r, lc.args),
                     newC = mkAndExp({c} ∪ renumVars(r, C2))
                  in if sat(newC) then
                     let newM = m † {l ↦{mk_(rt, lc)}} --restricts to this trace in l
                     in result := result ∪ joinTimedTracesAux(sd, mk_(newT,newC), newM);

   if terminated = sd.lifelines then
     result := result ∪ {mk_(t,c)};

  return result
);


/*** Utility functions ***/

functions

-- Obtains the sequence of indices of events in a trace t that occur at a lifeline l.
protected lifelineInds: Lifeline × Trace  ⟶ ℕ*
lifelineInds(l, t) ≜ [i | i ∈ inds t • t(i).lifeline = l];
```

```
-- Similar, with extra trace info .
protected lifelineInds: Lifeline × TraceExt ⟶ ℕ*
lifelineInds(l, t) ≜ [i | i ∈ inds t • t(i).lifeline = l];


/*** Auxiliary query functions ***/

-- Obtains the set of traces in a set of tc-traces.
public untimed: TCTrace-set ⟶ Trace-set
untimed(T) ≜ {t | mk_(t, -) ∈ T};

-- Get timestamp of an event.
protected t: Event ⟶ [Variable | TimeValue]
t(e) ≜ e.timestamp;

-- Get 'send' event of a message.
protected s: Message ⟶ Event
s(m) ≜ mk_Event(<Send>, m.signature, m.sendEvent.#1, m.sendTimestamp);

-- Get 'receive' event of a message.
protected r: Message ⟶ Event
r(m) ≜ mk_Event(<Receive>, m.signature, m.receiveEvent.#1, m.recvTimestamp);

-- Gets the message corresponding to an event.
protected msg: Interaction × Event ⟶ Message
msg(sd, e) ≜ let m ∈ sd.messages be st e = s(m) ∨ e = r(m) in m;

-- Checks if an event is of type Send.
public isSend: Event ⟶ 𝔹
isSend(e) ≜ e.type = <Send>;

-- Checks if an event is of type Receive.
public isReceive: Event ⟶ 𝔹
isReceive(e) ≜ e.type = <Receive>;

-- Obtains the event corresponding to a time variable
operations
public static pure getEventByTimeVariable: Interaction × Variable ⟶ [Event]
getEventByTimeVariable(sd, v) ≜
(
   for all m ∈ sd.messages do
     if m.sendTimestamp = v then return s(m)
     else if m.recvTimestamp = v then return r(m);
   return nil
);


/*** Auxiliary functions for converting Events and Traces to String ***/

functions
public event2str: Event ⟶ String
event2str(mk_Event(type, msg, lifeline, -)) ≜
   (if type = <Send> then "!" else "?") ⁀ msg ⁀ "@" ⁀ lifeline.name;

public event2strSimple: Event ⟶ String
event2strSimple(mk_Event(type, msg, lifeline, -)) ≜
   (if type = <Send> then "!" else "?") ⁀ msg;
```

```
public traces2str: Trace-set ⟶ String
traces2str(s) ≜ set2str[Trace](s, trace2str);

public traces2strSimple: Trace-set ⟶ String
traces2strSimple(s) ≜ set2str[Trace](s, trace2strSimple);

public tctraces2str: TCTrace-set ⟶ String
tctraces2str(s) ≜ set2str[TCTrace](s, tctrace2str);

public trace2str: Trace ⟶ String
trace2str(t) ≜ seq2str[Event](t, event2str);

public trace2strSimple: Trace ⟶ String
trace2strSimple(t) ≜ seq2str[Event](t, event2strSimple);

public tctrace2str: TCTrace ⟶ String
tctrace2str(mk_(t, c)) ≜ "";

-- Represents a time constraint in a string
public timeConstraint2str: Interaction × TimeConstraint ⟶ String
timeConstraint2str(sd, mk_TimeConstraint(firstEvent, secondEvent, min, max)) ≜
   (if min = nil then "" else VDMUtil`val2seq_of_char[Duration](min) ↷ " <= ")
   ↷ event2str(getEventByTimeVariable(sd, secondEvent))
   ↷ " - "
   ↷ event2str(getEventByTimeVariable(sd, firstEvent))
   ↷ (if max = nil then "" else " <= " ↷ VDMUtil`val2seq_of_char[Duration](max));

end Traces
```

6. Class ValidTraces

```
/**
 * Computation of valid traces defined by an Interaction
 * (without and with time constraints).
 */

class ValidTraces is subclass of SequenceDiagrams, Traces

/*** Configuration parameters ***/

values
-- Maximum number of iterations to consider when expanding unconstrained loops.
public MAX_LOOP_ITER = 4;

functions

/*** Main functions ***/

-- Valid traces represented in a string
public validTracesStr: Interaction ⟶ String
validTracesStr(sd) ≜ traces2str(validTraces(sd));

-- Valid traces defined by an interaction (sd), with timing info removed.
public validTraces: Interaction ⟶ Trace-set
validTraces(sd) ≜ untimed(validTimedTraces(sd));

-- Valid traces defined by an interaction (sd), without considering time constraints.
public validTracesUntimed: Interaction ⟶ Trace-set
validTracesUntimed(sd) ≜  removeExtraInfo(validTracesExt(sd));

-- Valid timed constrained traces (tc-traces) defined by an interaction (sd).
public validTimedTraces: Interaction ⟶ TCTrace-set
validTimedTraces(sd) ≜ removeExtraInfo(validTimedTracesExt(sd));

-- Valid tc-traces with extra event information.
protected validTimedTracesExt: Interaction ⟶ TCTraceExt-set
validTimedTracesExt(sd) ≜
  let cand =  {mk_(t, constraintExp(t, sd)) | t ∈ validTracesExt(sd)}
  in {mk_(t,c) | mk_(t,c) ∈ cand • sat(c)};

-- Incremental evaluation of valid tc-traces with extra event information, after adding
time constraints.
protected validTimedTracesExtIncr: Interaction × TCTraceExt-set ⟶ TCTraceExt-set
validTimedTracesExtIncr(sd, oldV) ≜
  let cand =  {mk_(t, constraintExp(t, sd)) | mk_(t, -) ∈ oldV}
  in {mk_(t, c) | mk_(t, c) ∈ cand • sat(c)};

-- Truncates an invalid tc-trace (t, c) to the shortest invalid sub-trace,
-- to facilitate error diagnosis, given the set V of valid tc-traces.
operations
protected static pure truncateOnError: TCTrace-set × TCTrace ⟶ Trace
truncateOnError(V, mk_(t, c)) ≜ (
  dcl t1 : Trace := t;
  dcl c1 : DCExp := c;
  dcl res : Trace := t;
  while t1 ≠ [] do (
```

```
    -- truncate removing last event
    t1 := t1(1,..., len t1 - 1);
    c1 := elimVarsAfter(len t1, c1);

    -- if this is a valid subtrace, at least partially, then stop
    if ∃ mk_(vt, vc) ∈ V •
          len t1 ≤ len vt ∧ eqIgnTimestamps(t1, vt(1,...,len t1))
          ∧ sat(mkAndExp({c1} ∪ elimVarsAfter(len t1, vc.args)))
    then
      return res;
    res := t1
  );
  return res;
);

/*** Computation of valid tc-traces ***/

-- Obtains a conjunctive expression with difference constraints applicable to a trace t.
operations
protected static pure constraintExp: TraceExt × Interaction ⟶ DCExp
constraintExp(t, sd) ≜ (
  dcl res : DC-set := ∅;

  -- Normal constraints
  for i = 1 to len t - 1 do
    for all c ∈ sd.timeConstraints do
      if c.firstEvent = t(i).timestamp then
        for j = i + 1 to len t do
          if c.secondEvent = t(j).timestamp ∧ t(i).itercounter = t(j).itercounter then
             res := res ∪ ev2ocConstr(i, j, c);

  -- Message guard constraints
  for all m ∈ sd.messages do
    if m.guard ≠ nil then let c = m.guard in
      for k = 3 to len t do
        if t(k).type = <Send> ∧ t(k).messageId = m.id then
          for i = 1 to k - 2 do
            if c.firstEvent = t(i).timestamp ∧ t(i).itercounter = t(k).itercounter then
              for j = i+1 to k - 1 do
                if c.secondEvent=t(j).timestamp ∧ t(i).itercounter=t(j).itercounter then
                  res := res ∪ ev2ocConstr(i, j, c);

  return mkAndExpDC(res)
);


functions
-- Generates the difference constraints corresponding to the application of a time
-- constraint 'c' to a pair of event occurences at positions 'i' and 'j' of a trace.
protected ev2ocConstr: ℕ × ℕ × TimeConstraint ⟶  DC-set
ev2ocConstr(i, j, c) ≜
   if c.max = nil then {mk_DC(i, j, -c.min)}
   else if c.min = nil then {mk_DC(j, i, c.max)}
   else {mk_DC(i, j, -c.min), mk_DC(j, i, c.max)};


/*** Conversion from traces with extra info to simple traces. ***/

protected removeExtraInfo: EventExt ⟶ Event
```

```
removeExtraInfo(e) ≜ mkEvent(e.type, e.signature, e.lifeline, e.timestamp);

protected removeExtraInfo: TraceExt ⟶ Trace
removeExtraInfo(t) ≜ [removeExtraInfo(e) | e ∈ t];

protected removeExtraInfo: TraceExt-set ⟶ Trace-set
removeExtraInfo(s) ≜ {removeExtraInfo(t) | t ∈ s};

protected removeExtraInfo: TCTraceExt-set ⟶ TCTrace-set
removeExtraInfo(s) ≜ {mk_(removeExtraInfo(t), c) | mk_(t, c) ∈ s};


/*** Computation of valid traces without time constraints (with extra event info) ***/

-- Gets the valid traces with extra event info in an Interaction (sd).
validTracesExt: Interaction ⟶ TraceExt-set
validTracesExt(sd) ≜
  freeComb({{s} | s ∈ topLevelEvents(sd) • s ≠ []}
           ∪ {expandCombinedFragment(sd, c) | c ∈ topLevelCombFrag(sd)});

-- Gets the top level event sequences in an Interaction (sd).
topLevelEvents: Interaction ⟶ TraceExt-set
topLevelEvents(sd) ≜
  {(if ∄ c ∈ sd.combinedFragments • contains(c, m.sendEvent) then
     [mk_EventExt(<Send>, m.signature, m.sendEvent.#1, m.sendTimestamp, m.sendEvent.#2,
m.id, [], m.type)]
    else [])
   ↷
   (if ∄ c ∈ sd.combinedFragments • contains(c, m.receiveEvent) then
      [mk_EventExt(<Receive>, m.signature, m.receiveEvent.#1, m.recvTimestamp,
m.receiveEvent.#2, m.id, [], m.type)]
     else [])
  | m ∈ sd.messages};

-- Gets the top level combined fragments in an Interaction (sd).
topLevelCombFrag: Interaction ⟶ CombinedFragment-set
topLevelCombFrag(sd) ≜
  {c | c ∈ sd.combinedFragments • ∄ c2 ∈ sd.combinedFragments • contains(c2, c)};

-- Given several sets of traces, obtains all possible trace interleavings, picking one
-- trace from each set.
freeComb: TraceExt-set-set ⟶ TraceExt-set
freeComb(s) ≜
  if s = ∅ then {[]}
  else let s1 ∈ s in ⋃ {freeComb2(t1, t2) | t1 ∈ s1, t2 ∈ freeComb(s \ {s1})}
measure # s;

-- Obtains all interleavings of two traces.
freeComb2: TraceExt × TraceExt ⟶ TraceExt-set
freeComb2(t1, t2) ≜
  if t1 = [] then {t2}
  else if t2 = [] then {t1}
  else freeComb2Aux(t1, t2) ∪ freeComb2Aux(t2, t1)
 measure len t1 + len t2;

-- Gets all interleavings of traces t1 and t2 that start with the first event of t1.
freeComb2Aux:  TraceExt × TraceExt ⟶ TraceExt-set
freeComb2Aux(t1, t2) ≜
```

```
  let e1 = hd t1, r1 = tl t1 in
    if precedes(t2, e1) then ∅
    else
       if e1.messageType = <Synch> ∧ e1.type = <Send>
          ∧ r1 ≠ [] ∧ (hd r1).messageId = e1.messageId ∧ (hd r1).type = <Receive>
       then
          if precedes (t2, hd r1) then ∅
          else {[e1, hd r1] ↷ r | r ∈ freeComb2(tl r1, t2)}
       else
          {[e1] ↷ r | r ∈ freeComb2(r1, t2)};

-- Checks if there exists any event in a trace t1 that precedes an event e2.
precedes: TraceExt × EventExt ⟶ 𝔹
precedes(t1, e2) ≜ ∃ e1 ∈ t1 • precedes(e1, e2);

-- Checks if an event e1 precedes an event e2.
precedes: EventExt × EventExt ⟶ 𝔹
precedes(e1, e2) ≜
  (e1.messageId = e2.messageId ∧ e1.itercounter = e2.itercounter ∧ e1.type = <Send> ∧
e2.type = <Receive>)
   ∨ (e1.lifeline = e2.lifeline
      ∧ (e1.location < e2.location
         ∨ e1.location = e2.location ∧ precedesIter(e1.itercounter, e2.itercounter)));

-- Checks if an iteration counter s1 precedes another iteration counter s2.
precedesIter: ℕ* × ℕ* ⟶ 𝔹
precedesIter(s1, s2) ≜
  s1 ≠ [] ∧ s2 ≠ [] ∧ (hd s1 < hd s2 ∨ hd s1 = hd s2 ∧ precedesIter(tl s1, tl s2))
pre len s1 = len s2
measure len s1 + len s2;

-- Gets the valid traces defined by a combined fragment 'c' in an interaction 'sd'.
expandCombinedFragment: Interaction × CombinedFragment ⟶ TraceExt-set
expandCombinedFragment(sd, c) ≜
  cases c.interactionOperator:
    <seq>     ⟶ expandNary(sd, c.operands, seqComb),
    <strict>  ⟶ expandNary(sd, c.operands, strictComb),
    <par>     ⟶ expandNary(sd, c.operands, parComb),
    <alt>     ⟶ expandAlt(sd, c.operands),
    <opt>     ⟶ expandOpt(sd, c.operands(1)),
    <loop>    ⟶ expandLoop(sd, c.operands(1), false),
    <sloop>   ⟶ expandLoop(sd, c.operands(1), true)
  end;

-- Gets the valid traces defined by a combined fragment of type seq, strict or par.
expandNary: Interaction × InteractionOperand* × (TraceExt × TraceExt ⟶ TraceExt-set) ⟶
TraceExt-set
expandNary(sd, args, comb) ≜
  if args = []  then {[]}
  else ⋃ {comb(t1, t2) | t1 ∈ expandOperand(sd, hd args),
                          t2 ∈ expandNary(sd, tl args, comb)};

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

```
      ∪ if ∃ e ∈ t1 • (hd t2).lifeline = e.lifeline then ∅
        else {[hd t2] ↷ r | r ∈ seqComb(t1, tl t2)};

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

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

-- Gets the valid traces defined by an 'alt' combined fragment, which is
-- simply the union of traces defined by its operands (args).
expandAlt: Interaction × InteractionOperand* ⟶ TraceExt-set
expandAlt(sd, args) ≜ ⋃ {expandOperand(sd, arg) | arg ∈ args};

-- Gets the valid traces defined by an 'opt' combined fragment.
expandOpt: Interaction × InteractionOperand ⟶ TraceExt-set
expandOpt(sd, arg) ≜ expandOperand(sd, arg) ∪ {[]};

-- Gets the valid traces defined by a 'loop' combined fragment (limiting the number of
-- loop iterations to MAX_LOOP_ITER).
operations
pure static expandLoop: Interaction × InteractionOperand × 𝔹 ⟶ TraceExt-set
expandLoop(sd, arg, strict) ≜
  let argExpansions = expandOperand(sd, arg),
      max = if arg.guard ≠ nil ∧ arg.guard.maxint ≠ nil then arg.guard.maxint
            else MAX_LOOP_ITER,
      min = if arg.guard ≠ nil ∧ arg.guard.minint ≠ nil then arg.guard.minint else 0
  in (
    dcl arg2n: TraceExt-set := {[]}; -- arg ^ n
    dcl res: TraceExt-set := if min = 0 then arg2n else {};
    for n = 1 to max do (
      arg2n :=  ⋃{if strict then strictComb(t1, addIterNumber(t2, n))
                  else seqComb(t1, addIterNumber(t2, n))
                 | t1 ∈ arg2n, t2 ∈ argExpansions};
      if n ≥ min then res := res ∪ arg2n
    );
    return res
  );

-- Adds an interation number to an iteration counter of events in a trace.
functions
addIterNumber: TraceExt × ℕ ⟶ TraceExt
addIterNumber(t, iter) ≜ [μ(e, itercounter ↦ [iter] ↷ e.itercounter) | e ∈ t];

-- Expands an interaction operand to a set of traces.
expandOperand: Interaction × InteractionOperand ⟶ TraceExt-set
expandOperand(i, o) ≜
  freeComb({{s} | s ∈ nestedEvents(i, o) • s ≠ []}
           ∪ {expandCombinedFragment(i, c) | c ∈ nestedCombFrag(i, o)});

-- Gets the topmost event sequences contained in a given intreaction operand.
nestedEvents: Interaction × InteractionOperand ⟶ TraceExt-set
nestedEvents(sd, o) ≜
```

```
  let cf = {c | c ∈ sd.combinedFragments • contains(o, c)} in
    {(if contains(o, m.sendEvent) ∧ ∄ c ∈ cf • contains(c, m.sendEvent)
      then [mk_EventExt(<Send>, m.signature, m.sendEvent.#1, m.sendTimestamp,
m.sendEvent.#2, m.id, [], m.type)]
      else [])
     ↷
     (if contains(o, m.receiveEvent) ∧ ∄ c ∈ cf • contains(c, m.receiveEvent)
      then [mk_EventExt(<Receive>, m.signature, m.receiveEvent.#1, m.recvTimestamp,
m.receiveEvent.#2, m.id, [], m.type)]
      else [])
    | m ∈ sd.messages};

-- Gets the topmost combined fragments contained in a given intreaction operand.
nestedCombFrag: Interaction × InteractionOperand ⟶ CombinedFragment-set
nestedCombFrag(sd, o) ≜
  let cf = {c | c ∈ sd.combinedFragments • contains(o, c)}
  in {c | c ∈ cf • ∄ c2 ∈ cf • c2 ≠ c ∧ contains(c2, c)};
end ValidTraces
```

7. Class ConformanceChecking

```
/**
 * Local and global conformance checking primitives, as well as local input
 * selection primitives.
 */

class ConformanceChecking is subclass of SequenceDiagrams, Traces, ValidTraces

types
public Verdict = <Pass> | <Fail> | <Inconclusive>;


/*** Primitives for local conformance checking, without and with time-constraints ***/

-- Checks if the next observed event occurence (e) in a lifeline is valid, given the
-- local  trace previously observed (t), and the set V of valid traces for the lifeline.
functions
public checkNextEvent: Trace × Event × Trace-set ⟶ 𝔹
checkNextEvent(t, e, V) ≜ ∃ (t) ↷ [(e)] ↷ - ∈ V • true;

-- Checks if the next observed event occurrence (e) in a lifeline is valid, given the
-- local timed trace previously observed (t), and the set of valid local tc-traces (V).
public timedCheckNextEvent: Trace × Event × TCTrace-set ⟶ 𝔹
timedCheckNextEvent(t, e, V) ≜
  ∃ mk_(v, c) ∈ V • len v > len t
     ∧ matchesTimed(t ↷ [e], mk_(v(1,...,len t + 1), elimVarsAfter(len t + 1, c))) =
        <Pass>;


/*** Primitives for global conformance checking, without and with time-constraints ***/

-- Final conformance checking in the absence of time constraints, given the observed
-- local traces.
public finalConformanceChecking: Interaction × Lifeline →(m) Trace ⟶ Verdict
finalConformanceChecking(sd, localTraces) ≜
  let V = validTraces(sd),
      J = joinTraces(sd, localTraces)
  in if J ∩ V = ∅ then <Fail>
     else if J ⊆ V then <Pass>
     else <Inconclusive>;

-- Final conformance checking in the presence of time constraints, given the observed
-- local traces.
public timedFinalConformanceChecking: Interaction × Lifeline →(m) Trace ⟶ Verdict
timedFinalConformanceChecking(sd, localTraces) ≜
  let V = validTimedTraces(sd),
      J = joinTraces(sd, localTraces)
  in if ∀ j ∈ J • ∀ v ∈ V • matchesTimed(j, v) = <Fail> then <Fail>
     else if ∀ j ∈ J • ∃ v ∈ V • matchesTimed(j, v) = <Pass> then <Pass>
     else <Inconclusive>;

-- Checks if an actual timed trace t matches a valid tc-trace (v, c).
matchesTimed: Trace × TCTrace ⟶ Verdict
matchesTimed(t, mk_(v, c))  ≜
  if ¬ eqIgnTimestamps(t, v) then <Fail>
```

```
  else if ∃ mk_DC(i, j, d) ∈ c.args • t(i).lifeline = t(j).lifeline ∧ t(i).timestamp -
t(j).timestamp > d then <Fail>
  else if ∄ mk_DC(i, j, d) ∈ c.args • t(i).lifeline ≠ t(j).lifeline then <Pass>
  else let lfOrd = VDMUtil`set2seq[Lifeline]({e.lifeline | e ∈ t}),
           vars = {lfOrd(i) ↦ i | i ∈ inds lfOrd},
           C = {mk_DC(i, j, MaxClockSkew) | i ∈ inds lfOrd, j ∈ inds lfOrd • i ≠ j},
           D = {mk_DC(vars(t(i).lifeline), vars(t(j).lifeline),
                d + t(j).timestamp - t(i).timestamp) | mk_DC(i, j, d) ∈ c.args},
           s1 = satRaw(mk_AndExp(C ∪ D)), -- C and D
           s2 = satRaw(mk_OrExp({mk_AndExp(C ∪ {mk_DC(j, i, -d-1)}) | mk_DC(i, j, d) ∈
D})) -- C and not D
       in if s1 ∧ ¬ s2 then <Pass>
          else if ¬ s1 ∧ s2 then <Fail>
          else <Inconclusive>;


/*** Primitives for local test input selection, without and with time-constraints  ***/

-- Gives the next events that can be sent by a lifeline, given the local trace
-- trace observed so far (t), and the set of valid local traces (V).
public nextSendEvents: Trace × Trace-set ⟶ Event-set
nextSendEvents(t, V) ≜ {e | (t) ↷ [e] ↷ - ∈ V • e.type = <Send>};

-- Gives the next events that can be sent by a lifeline, and  the time interval
-- for sending each event, given the local timed trace observed so far (t),
-- and the set of valid local tc-traces (V).
public nextSendEventsTimed: Trace × TCTrace-set ⟶ (Event × TimeInterval)-set
nextSendEventsTimed(t, V) ≜
  {mk_(v(len t +1), nextEventInterval(t, mk_(v, c))) | mk_(v, c) ∈ V •
     len v > len t ∧ v(len t + 1).type = <Send>
     ∧ matchesTimed(t, mk_(v(1, ..., len t), elimVarsAfter(len t, c))) = <Pass>
     ∧ nextEventInterval(t, mk_(v, c)) ≠ nil};

-- Determines the TimeInterval for the next valid event in a lifeline, given the
-- trace observed so far (t), and a valid local tc-trace (v, c).
-- Returns nil if impossible.
nextEventInterval: Trace × TCTrace ⟶ [TimeInterval]
nextEventInterval(t, mk_(v, c)) ≜
  if t = [] then mk_(nil, nil)
  else let c2 = elimVarsAfter(len t + 1, c),
           mn = min[TimeValue]({t(i).timestamp - d | mk_DC(i, j, d) ∈ c2.args •
                                j = len t + 1} ∪ {t(len t).timestamp}),
           smx = {t(j).timestamp + d | mk_DC(i, j, d) ∈ c2.args • i = len t + 1},
           mx = if smx = ∅ then nil else max[TimeValue](smx)
       in if mx ≠ nil ∧ mn > mx then nil else mk_(mn, mx);

end ConformanceChecking
```

8. Class SimulatedExecution

```
/**
 * Simulated symbolic execution of time-constrained sequence diagrams.
 */

class SimulatedExecution is subclass of SequenceDiagrams, Traces, ValidTraces


/******************** Configuration parameters ******************/

values

protected EMISSION_TIMESTAMP_TRANSMITTED = false;
-- If true, interlifeline constraints don't cause local observability problems.

protected MAY_LOOSE_MESSAGES = true;
-- If true, optional messages without a corresponding ackowledgment message,
-- cause local observability problems

protected MAX_TRACE_LEN = 1000000;
-- Use to generate event timestamp ids that are beyond valid ones.


/********************* Auxiliary structures **********************/

types

-- Transmission channel for each pair of lifelines (if FIFO_CHANNELS = true)
-- or pair of lifelines and message signature (if FIFO_CHANNELS = false).
Channel = (Lifeline × Lifeline) | (Lifeline × Lifeline × MessageSignature);

-- Queue of index of emission event and corresponding reception event.
ChannelStatus = (ℕ × Event)*;

-- Each extension of a tc-trace is a pair of an added event and added time constraints.
Extension = Event × DC-set;

-- Structure return by traceExtLf
LifelineTraceExtensions = Extension-set × TCTrace-set × DCExp × DCExp;


/********************* Main procedures **********************/

-- Computes the set of tc-traces that can be generated by the simulated execution
-- of an interaction, when lifelines behave local knowledge only.
-- If the goal is to analyse controllability, it is assumed that transmission channels
-- behave correctly (without losing messages and respecting transmission time
-- constraints), and lifelines are input enabled (accept all messages delivered by
-- the transmission channels).
-- If the goal is to analyse observability, transmission channels need no to behave
-- correctly, but lifelines only accept valid reception messages.
functions
public simulExec: Interaction × 𝔹 ⟶ TCTrace-set
simulExec(sd, observability) ≜
  simulExec(sd, projectTCTraces(validTimedTraces(sd), sd.lifelines), observability);

-- Similar with the set of valid tc-traces per lifeline pre-computed.
```

```
public simulExec: Interaction × Lifeline →(m) TCTrace-set × 𝔹  ⟶ TCTrace-set
simulExec(sd, P, observability) ≜
  simulExec(sd, P, mk_([], TrueExp), {↦}, observability, mk_([], TrueExp),
      {l ↦ traceExtLf(mk_([], TrueExp), l, P(l), observability) | l ∈ sd.lifelines});

-- Recursively computes the time constrained traces that can be generated by the
-- execution of an interaction (sequence diagram), if each lifeline behaves according
-- to local knowledge only (traces observed locally and traces valid locally) and the
-- transmission chanel respects transmission constraints (in case of controlability).
-- Parameters:
--  sd - interaction (sequence diagram)
--  P - valid local time constrained traces per lifeline (with trace advanced to current
--  (t, c) - time constrained trace generated so far (initially empty)
--  m - map from channel identifier to queue of messages in transit
--  observability - if true, is for observability (instead of controlability) analysis.
--  oldT, oldE - parameters for incremental calculations.
static simulExec: Interaction × Lifeline →(m) TCTrace-set × TCTrace ×
            Channel →(m) ChannelStatus × 𝔹 × TCTrace × Lifeline →(m) LifelineTraceExtensions
            ⟶ TCTrace-set
simulExec(sd, P, mk_(t, c), m, observability, oldT, oldE) ≜
(
 -- Handle different cases in disjunction separately
 if is_OrExp(c) then
   ⋃ {simulExec(sd, P, mk_(t, arg), m, observability, oldT, oldE) | arg ∈ c.args}

 -- Handle normal cases
 else
   let -- Compute possible trace extensions, from the perspective of each lifeline,
       -- as well as termination condition and action deadline for each lifeline.
       E = {l ↦ updTraceExtLf(mk_(t, c), l, P(l), observability, oldT, oldE(l))
            | l ∈ sd.lifelines},

       -- Flat set of possible next events (including stop events) and respective
       -- constraints from all lifelines.
       E1 =  ⋃ {E(l).#1 | l ∈ sd.lifelines},

       -- Emission candidates and respective constraints
       S = {mk_(e, C) | mk_(e, C) ∈ E1 • e.type = <Send>},

       -- Updated status of projections per lifeline
       newP = {l ↦ E(l).#2 | l ∈ sd.lifelines},

       -- Compute reception candidates, based on messages in transit,
       -- transmission constraints (only for controllability analysis),
       -- and reception constraints (only for observability analysis)
       R0 = candFromChannels(sd, mk_(t, c), m, observability),
       R = if ¬ observability then R0
           else let R1 = {mk_(e, C) | mk_(e, C) ∈ E1 • e.type = <Receive>}
                in ⋃ {{mk_(i, e, C0 ∪ C1) | mk_((e), C1) ∈ R1} | mk_(i, e, C0) ∈ R0},

        -- Compute constraint for lifeline actions
       cS = {E(l).#4 | l ∈ sd.lifelines},

       -- Compute constraint for channel actions
       cR = if observability then ∅ -- already included in Cs
            else ⋃ {{ct | ct ∈ C • isMaxDuration(ct)} | mk_(-, -, C) ∈ R}

   in
```

```
     -- Reception
     ⋃ {let newC = renumVar(red(mkAndExp({c} ∪ cR ∪ cS ∪ C)), MAX_TRACE_LEN, len t + 1)
        in if newC = FalseExp then ∅
           else simulExec(sd, consumeEvent(e, newP), mk_(t ⁀ [e], newC),
                                 updChannelsRecv(e, t(i), m), observability, mk_(t,c), E)
        | mk_(i, e, C) ∈ R}

     -- Emission
     ∪
     ⋃ {let me = msg(sd, e) in
          if me.type = <Synch> then
            let r = r(me),
                C2 = if ¬ observability then TrueExp
                     else mkOrExp({mkAndExp(C2)| mk_((r),C2) ∈ E(r.lifeline).#1}),
                C3 = mkAndExp({c, C2, mk_DC(MAX_TRACE_LEN + 1, MAX_TRACE_LEN, 0)}
                              ∪ cR ∪ cS ∪ C)
                C4 = renumVar(red(C3), MAX_TRACE_LEN, len t + 1),
                newC = renumVar(C4, MAX_TRACE_LEN + 1, len t + 2)
            in if newC = FalseExp then ∅
               else simulExec(sd, consumeEvent(r, consumeEvent(e, newP)),
                              mk_(t ⁀ [e, r], newC), m, observability, mk_(t,c), E)
          else
            let newC = renumVar(red(mkAndExp({c}∪cR∪cS ∪ C)), MAX_TRACE_LEN, len t + 1)
            in if newC = FalseExp then ∅
               else simulExec(sd, consumeEvent(e, newP), mk_(t ⁀ [e], newC),
                              updChannelsSend(len t + 1, e, r(msg(sd, e)), m),
                              observability, mk_(t,c), E)
       | mk_(e, C) ∈ S}

     -- Termination (quiescense)
     ∪ (if R = ∅ ∨ observability /*may loose messages*/then
           let cQ = red(mkAndExp({c} ∪ {E(l).#3 | l ∈ sd.lifelines}))
           in if cQ = FalseExp then ∅ else {mk_(t, cQ)}
        else ∅)
);


functions
-- Update status (m) of transmission channels after an emission event (s) in position
-- 'i' of a trace (being 'r' the corresponding reception event).
updChannelsSend: ℕ × Event × Event × Channel →(m) ChannelStatus
                 ⟶ Channel →(m) ChannelStatus
updChannelsSend(i, s, r, m) ≜
   let channel = if FIFO_CHANNELS then mk_(s.lifeline, r.lifeline)
                 else mk_(s.lifeline, r.lifeline, s.signature)
   in if channel ∈ dom m then m † {channel ↦ m(channel) ⁀ [mk_(i, r)]}
      else m ∪ {channel ↦ [mk_(i, r)]};

-- Update status (m) of transmission channels after reception event (r) at the head of
-- the queue (being 's' the corresponding emission event).
updChannelsRecv: Event × Event × Channel →(m) ChannelStatus ⟶ Channel →(m) ChannelStatus
updChannelsRecv(r, s, m) ≜
  let channel = if FIFO_CHANNELS then mk_(s.lifeline, r.lifeline)
                else mk_(s.lifeline, r.lifeline, s.signature)
  in if len m(channel) = 1 then {channel} ⩤ m
     else m † {channel ↦ tl m(channel)};
```

```
-- Determines candidate reception events from FIFO transmission channels (m),
-- after a given tc-trace.
candFromChannels: Interaction × TCTrace × Channel ⇸ ChannelStatus × 𝔹
                  ⟶ (ℕ × Event × DC-set)-set
candFromChannels(sd, mk_(t, -), m, observability) ≜
  {let mk_(i, r) = hd m(channel),
       C = if observability ∧ ¬ EMISSION_TIMESTAMP_TRANSMITTED then ∅
           else ⋃ {ev2ocConstr(i, MAX_TRACE_LEN, c2) | c2 ∈ sd.timeConstraints •
                         c2.firstEvent = t(i).timestamp ∧ c2.secondEvent = r.timestamp}
   in mk_(i, r, C)
   | channel ∈ dom m};

-- Updates the status (P) of lifelines after an event (e)
consumeEvent: Event × Lifeline ⇸ TCTrace-set ⟶ Lifeline ⇸ TCTrace-set
consumeEvent(e, P) ≜
   P † {e.lifeline ↦ {mk_(tl t, c) |
                        mk_(t, c) ∈ P(e.lifeline) • t ≠ [] ∧ eqIgnTimestamps(hd t, e)}};

-- Obtains the possible extensions of a global time constrained trace (t,c), from the
-- perspective of a lifeline l with a set V of locally valid time constrained traces.
-- Each extension is a pair of an added event and added time constraints.
-- Return a tuple with:
--   - set of extensions
--   - updated V (restricting to satisfiable tc-traces).
--   - acceptance/quiescence/termination condition after the given tc-trace
--   - action deadline condition after the given tc-trace.
operations
static pure traceExtLf: TCTrace × Lifeline × TCTrace-set × 𝔹 ⟶ LifelineTraceExtensions
traceExtLf(mk_(t, c), l, V, observability) ≜ (
  dcl E : Extension-set := ∅;
  dcl newV : TCTrace-set := ∅;
  dcl newE : Event;
  dcl newC : DC-set;
  dcl r1 : ℕ* := lifelineInds(l, t);
  dcl r2 : ℕ* := r1 ↷  [MAX_TRACE_LEN]; -- len t + 1
  dcl hasSend : 𝔹 := false;
  dcl hasUnrestrictedSend : 𝔹 := false;
  dcl hasUnrestrictedStop : 𝔹 := false;
  dcl hasUnrestrictedRecv : 𝔹 := false;

  for all mk_(lt, lc) ∈ V do (
    if lt = [] then (
      newC := renumVars(r1, lc.args);
      newE := mkStopEvent(l)
    )
    else (
      newC := renumVars(r2, elimVarsAfter(len r2, lc.args));
      newE := hd lt
    );

    if newC ⊆ c.args ∨ sat(mkAndExpDC(c.args ∪ newC)) then (
      E := E ∪ {mk_(newE, newC)};
      newV := newV ∪ {mk_(lt, mk_AndExp(lc.args))};
      cases newE.type:
         <Send>    ⟶ (hasSend := true;
                         if newC = ∅ then hasUnrestrictedSend := true),
         <Stop>    ⟶ if newC = ∅ then hasUnrestrictedStop := true,
         <Receive> ⟶ if newC = ∅ then hasUnrestrictedRecv := true
```

```
      end
    )
  );

  -- cases in which may remain quiescent (terminate) for sure
  if hasUnrestrictedStop ∨ (¬ observability ∧ (¬ hasSend ∨ hasUnrestrictedRecv)) then
    return mk_(E, newV, TrueExp, TrueExp);

  -- other cases
  let n = MAX_TRACE_LEN, -- len t + 1,
      preE = {C ↦ elimVarsAfter(len t, C) | mk_(e, C) ∈ E • e.type ≠ <Stop>},
      maxE = {C ↦ {mk_DC(n, j, d) | mk_DC((n), j, d) ∈ C • j < n} -- max dur. constr.
              | mk_(e, C) ∈ E • e.type ≠ <Stop>},

      -- termination
      A = mkOrExp({mk_AndExp(C) | mk_(e,C) ∈ E • e.type = <Stop>}),

      -- for all emission candidates 's', if 's' is enabled, then there is at least
      -- on reception event 'r' such that 'r' is enabled and deadline(r) <= deadline(s)
      B = if observability then FalseExp
          else
            mkAndExp({
              mkImpliesExp(
                mk_AndExp(preE(Cs)),
                mkOrExp({
                   mkAndExp({
                     mk_AndExp(preE(Cr)),
                     mkAndExp({mkOrExp({mk_DC(js,jr,dr-ds)| mk_DC(-,js,ds) ∈ maxE(Cs)})
                                | mk_DC(-, jr, dr) ∈ maxE(Cr)})
                   })
                 | mk_(r, Cr) ∈ E • r.type = <Receive>})
              )
            | mk_(s, Cs) ∈ E • s.type = <Send>}),

     -- for at least one emission (or reception) event, it may be enabled and deadline
     -- is met
     C = if observability then
           mkOrExp({mk_AndExp(preE(C) ∪ maxE(C)) | mk_(e, C) ∈ E • e.type ≠ <Stop>})
         else
           mkOrExp({mk_AndExp(preE(C) ∪ maxE(C)) | mk_(e, C) ∈ E • e.type = <Send>})
  in
    if observability then
      return mk_(E, newV, A, mkOrExp({A, C}))
    else
      return mk_(E, newV, mkOrExp({A, B}), mkOrExp({A, B, C}));
);

-- Incremental calculation of traceExtLf for otpmization purposes.
functions
updTraceExtLf: TCTrace × Lifeline × TCTrace-set × 𝔹 × TCTrace × LifelineTraceExtensions
                  ⟶ LifelineTraceExtensions
updTraceExtLf(mk_(t, c), l, V, observability, mk_(old_t, old_c), oldE) ≜
  if (len t = len old_t ∨ ∀ i ∈ {len old_t + 1, ..., len t} • t(i).lifeline ≠ l)
     ∧ (c.args = old_c.args ∨
          ∀ mk_(-, newC) ∈ oldE.#1 • newC = ∅ ∨ sat(mkAndExpDC(c.args ∪ newC)))
  then oldE
  else traceExtLf(mk_(t, c), l, V, observability);
```

```
end SimulatedExecution
```

9. Class Observability

```
/**
 * Analysis of local observability.
 */

class Observability is subclass of SequenceDiagrams, Traces, ValidTraces,
SimulatedExecution

functions

-- Determines if an interaction sd is locally observable.
public isLocallyObservable: Interaction ⟶ 𝔹
isLocallyObservable(sd) ≜ uncheckableLocallyTimed(sd) = ∅;

-- Gives the set of global traces that are uncheckable locally in an interaction sd,
-- i.e., the global traces that are locally valid but are not globally valid.
public uncheckableLocally: Interaction ⟶ Trace-set
uncheckableLocally(sd) ≜
  if sd.timeConstraints = ∅ then uncheckableLocallyUntimed(sd)
  else untimed(uncheckableLocallyTimed(sd));

-- Gives a string representation of the uncheckable locally traces in an interaction sd.
public uncheckableLocallyStr: Interaction ⟶ String
uncheckableLocallyStr(sd) ≜ traces2str(uncheckableLocally(sd));

-- Gives the set of global tc-traces that are uncheckable locally in an interaction sd.
public uncheckableLocallyTimed: Interaction ⟶ TCTrace-set
uncheckableLocallyTimed(sd) ≜ uncheckableLocallyTimed(sd, validTimedTraces(sd));

-- Same as above, but receives the set V of valid tc-traces pre-computed.
public uncheckableLocallyTimed: Interaction × TCTrace-set ⟶ TCTrace-set
uncheckableLocallyTimed(sd, V) ≜
  let P = projectTCTraces(V, sd.lifelines),
      J = joinTimedTraces2(sd, P)
  in subtractTimedTraces(J, V);

-- Gives the set of global traces that are locally uncheckable in an interaction sd
-- without time constraints (or ignoring time constraints).
public uncheckableLocallyUntimed: Interaction ⟶ Trace-set
uncheckableLocallyUntimed(sd) ≜
  let V = validTraces(sd),
      P = projectTraces(V, sd.lifelines)
  in joinTraces(sd, [], P) \ V;


-- Faster calculation of local observability violations using simulated execution.
-- Truncated on error.
public uncheckableTracesTimed: Interaction ⟶ Trace-set
uncheckableTracesTimed(sd) ≜
  let V = validTimedTraces(sd),
      U = uncheckableTracesTimedRaw(sd, V)
  in {truncateOnError(V, tc) | tc ∈ U};

-- Faster calculation of local observability violations using simulated execution.
public uncheckableTracesTimedRaw: Interaction ⟶ TCTrace-set
uncheckableTracesTimedRaw(sd) ≜
```

```
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P, true),
      U = subtractTimedTraces(S, V)
  in U;

-- Similar, with the set of valid tc-traces (V) pre-computed.
public uncheckableTracesTimedRaw: Interaction × TCTrace-set ⟶ TCTrace-set
uncheckableTracesTimedRaw(sd, V) ≜
  let P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P,  true)
  in subtractTimedTraces(S, V);


end Observability
```

10. Class Controllability

```
/**
 * Analysis of local controllability and identification of violations.
 */

class Controllability is subclass of SequenceDiagrams, Traces, ValidTraces,
SimulatedExecution


/************************** Main public functions  ***********************/

-- Determines if an interaction is locally controllable, i.e., no invalid traces are
-- generated and all valid traces are generated when lifelines behave using local
-- knowledge only (traces observed locally and traces valid locally), without exchanging
-- coordination messages between them, and transmission channels behave correctly.
functions
public isLocallyControllable: Interaction ⟶ 𝔹
isLocallyControllable(sd) ≜
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P, false)
  in areEquivalent(V, S);

-- Determines the invalid traces that can be generated when lifelines
-- behave using local knowledge only (traces observed locally and traces valid locally).
-- The invalid traces are truncated up to the first invalid event.
public unintendedTraces: Interaction ⟶ Trace-set
unintendedTraces(sd) ≜
      if sd.timeConstraints = ∅ then unintendedTracesUntimed(sd)
      else unintendedTracesTimed(sd);

-- String representation of the set unintended traces.
public unintendedTracesStr: Interaction ⟶ String
unintendedTracesStr(sd) ≜ traces2str(unintendedTraces(sd));

-- Determines the invalid traces that can be generated when lifelines
-- behave using local knowledge only (sets of traces valid locally),
-- in the presence of time constraints.
public unintendedTracesTimed: Interaction ⟶ Trace-set
unintendedTracesTimed(sd) ≜
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P,  false),
      U = subtractTimedTraces(S, V)
  in {truncateOnError(V, tc) | tc ∈ U};

-- Similar, but tc-traces, not truncated.
public unintendedTracesTimedRaw: Interaction ⟶ TCTrace-set
unintendedTracesTimedRaw(sd) ≜
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P, false),
      U = subtractTimedTraces(S, V)
  in U;

-- Similar, with set of vald tc-traces (V) pre-computed.
```

```
public unintendedTracesTimedRaw: Interaction × TCTrace-set ⟶ TCTrace-set
unintendedTracesTimedRaw(sd, V) ≜
  let P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P,  false),
      U = subtractTimedTraces(S, V)
  in U;

-- Determines the valid traces that are not generated when lifelines behave using
-- local knowledge only (traces observed locally and traces valid locally).
public missingTraces: Interaction ⟶ Trace-set
missingTraces(sd) ≜
  let V = validTimedTraces(sd),
      P = projectTCTraces(V, sd.lifelines),
      S = simulExec(sd, P, false)
  in untimed(subtractTimedTraces(V, S));


/*********** Auxiliary functions for untimed interactions **********/

-- Calculation of unintended traces, in the absence of time constraints.
-- Gives subtraces that can be generated according to causality rules,
-- but end in an unintended send (us), receive (ur) or termination (ut).
protected unintendedTracesUntimed: Interaction ⟶ Trace-set
unintendedTracesUntimed(sd) ≜ unintendedTracesUntimed(sd, validTracesUntimed(sd));

-- Idem, with set of valid traces (V) pre-computed.
protected unintendedTracesUntimed: Interaction × Trace-set ⟶ Trace-set
unintendedTracesUntimed(sd, V) ≜
  let T = prefixes(V),
      L = sd.lifelines,
      P = projectTraces(V, L),
   us = {q ⁀ [t2(len t2)] | q ∈ T, t2 ∈ T • len t2 > 0 ∧
            let p = t2(1, ..., len t2-1), e = t2(len t2) in
             e.type = <Send>
             ∧ projectTrace(q, e.lifeline) = projectTrace(p, e.lifeline)} \ T,
   ur =   ∪ {{q  ⁀ [t2(len t2)] | q ∈ prefixes({t2(1,...,len t2-1)}) •
                                    isFeasibleAddition(sd, q, t2(len t2))}
               | t2 ∈ T • len t2 > 0 ∧ t2(len t2).type = <Receive>} \ T,
   ut = {p | p ∈ T • allMsgsReceived(p) ∧ mayRemainQuiescentUntimed(sd, p, P)} \ V
  in us ∪ ur ∪ ut;

-- Computes the set of prefixes of a set of traces
prefixes: Trace-set ⟶ Trace-set
prefixes(T) ≜ {[]} ∪ ∪{{t(1, ..., i) | i ∈ inds t} | t ∈ T};

-- Checks (in a simplified way) if all message have been received in a trace (t).
allMsgsReceived: Trace ⟶ 𝔹
allMsgsReceived(t) ≜
  # {i | i ∈ inds t • isSend(t(i))} = # {i | i ∈ inds t • isReceive(t(i))};

-- Determines if a lifeline may remain quiescent after a valid global trace (t).
                                                       m
mayRemainQuiescentUntimed: Interaction × Trace × Lifeline → Trace-set ⟶ 𝔹
mayRemainQuiescentUntimed(sd, t, P) ≜
  ∀ l ∈ sd.lifelines • let p = projectTrace(t, l) in
    p ∈ P(l)
    ∨ (∄ (p) ⁀ [e] ⁀ - ∈ P(l) • e.type = <Send>)
    ∨ (∃ (p) ⁀ [e] ⁀ - ∈ P(l) • e.type = <Receive>);
end Controllability
```

11. Class Enforcement

```
/*
 * Algorithms for the enforcement of local observability and controllability by
 * the addition of coordination messages and coordination time constraints.
 */

class Enforcement is subclass of Controllability, Observability

values

/*********************** Configuration parameters *********************************/

public CoordinationMessagePrefix = "Ctrl"; -- prefix for generated coordination messages
public AcknowledgmentMessagePrefix = "Ack"; -- prefix for generated ackowledgment
messages
public LocationMultiplier = 10000; -- used to insert new messages between given
locations


/******* Interaction manipulation functions and operations *************/

-- Compares messages by their ids (used for sorting purposes)
functions
compByMessageId: Message × Message ⟶ ℤ
compByMessageId(m1, m2) ≜ m1.id - m2.id;

-- Compares pairs (event, message) by the message id (used for sorting purposes)
compByMessageId2: (Event × Message) × (Event × Message) ⟶ ℤ
compByMessageId2(mk_(-,m1), mk_(-,m2)) ≜ m1.id - m2.id;

-- Obtains a message corresponding to a given extended event
-- (assumes messageMap is updated).
getMessageOfEventExt: Interaction × EventExt ⟶ Message
getMessageOfEventExt(sd, e) ≜ sd.messageMap(e.messageId);

-- Makes sure that all messages in a sequence diagram 'sd' have defined
-- timestamp variables and messageMap is updated. Returns a new SD with such property.
mkMsgsTimedAndIndexed: Interaction ⟶ Interaction
mkMsgsTimedAndIndexed(sd) ≜
  let ms = {μ(m, sendTimestamp ↦
                 if m.sendTimestamp ≠ nil then m.sendTimestamp
                 else mk_Variable("s_" ↷ m.signature ↷ "_" ↷
VDMUtil`val2seq_of_char[MessageId](m.id)),
               recvTimestamp ↦
                 if m.recvTimestamp ≠ nil then m.recvTimestamp
                 else mk_Variable("r_" ↷ m.signature ↷ "_" ↷
VDMUtil`val2seq_of_char[MessageId](m.id)))
          | m ∈ sd.messages}
   in μ(sd, messages ↦ ms, messageMap ↦ {m.id ↦ m | m ∈ ms});

-- Renumber the location numbers in an Interaction (sd), multiplying all numbers by a
-- given factor, to permit subsequent insertion of coordination messages in
-- intermmediate locations.
renumberInteraction: Interaction × ℕ ⟶ Interaction
renumberInteraction(sd, factor) ≜
  let ms =  {μ(m, sendEvent ↦ renumberLocation(m.sendEvent, factor),
```

```
                    receiveEvent ↦ renumberLocation(m.receiveEvent, factor))
             | m ∈ sd.messages}
  in μ(sd,
      messages ↦ ms,
      messageMap ↦ {m.id ↦ m | m ∈ ms},
      combinedFragments ↦
        {μ(c, operands ↦
                [μ(o, startLocations ↦ {renumberLocation(l, factor) | l ∈
o.startLocations},
                      finishLocations ↦ {renumberLocation(l, factor) | l ∈
o.finishLocations})
               | o ∈ c.operands])
         | c ∈ sd.combinedFragments});

-- Renumbers a lifeline location,  multiplying by a given factor.
renumberLocation: LifelineLocation × ℕ ⟶ LifelineLocation
renumberLocation(mk_(lifeline, location), factor) ≜ mk_(lifeline, location × factor);


/******** Finding coordination time constraints  *********/

-- Determines the events in a trace t that have a causal link to the i-the event in the
-- trace.  Returns a boolean sequence, with the same length as t, indicating those
-- events (including the i-th event).
operations
static pure getCausalChainBackwards: TraceExt × ℕ ⟶ 𝔹*
getCausalChainBackwards(t, i) ≜
(
  dcl select: 𝔹* := [k = i | k ∈ inds t];
  for k = i to 1 by -1 do
    if select(k) then
      if t(k).type = <Send> then
      (
         -- select all the preceeding events in the same lifeline
         -- (because they may influence the decision to emit this event)
         for all j ∈ {1, ..., k-1} do
           if t(j).lifeline = t(k).lifeline then
             select(j) := true
      )
      else
      (
        -- <Receive>: select the corresponding emission event
        let j ∈ {1, ..., k-1} be st
            t(j).messageId = t(k).messageId ∧ t(j).type = <Send>
            ∧ t(j).itercounter = t(k).itercounter
        in select(j) := true
      );
  return select
)
pre i ∈ inds t;

-- Similar, but in forward direction.
static pure getCausalChainForward:TraceExt × ℕ ⟶ 𝔹*
getCausalChainForward(t, i) ≜
(
  dcl select: 𝔹* := [k = i| k ∈ inds t];
  for k = i to len t by 1 do
     if select(k) then
```

```
    (
      -- select all the succeeding emission events in the same lifeline
      for all j ∈ {k+1, ..., len t} do
        if t(j).lifeline = t(k).lifeline ∧ t(j).type = <Send> then
          select(j) := true;

      -- select the corresponding reception event
      if t(k).type = <Send> then
        let j ∈ {k+1, ..., len t} be st
           t(j).messageId = t(k).messageId ∧ t(j).type = <Receive>
           ∧ t(j).itercounter = t(k).itercounter
        in select(j) := true
    );
  return select
)
pre i ∈ inds t;

-- Select the events in a trace that have a causal relation regarding two other events.
functions
getCausalEventsBetween: TraceExt × ℕ × ℕ ⟶ 𝔹*
getCausalEventsBetween(t, i1, i2) ≜
(
  let select1 = getCausalChainForward(t, i1),
      select2 = getCausalChainBackwards(t, i2)
  in [select1(i) ∧ select2(i) | i ∈ inds t]
) pre i1 < i2 ∧ i1 ≥ 1 ∧ i2 ≤ len t;

-- Obtains the nearest common ancestor in the causal event chains that lead
-- to events in positions i1 and i2 in a trace t.
-- Returns 0 if no such common ancestor can be found.
operations
static pure getNearestCommonAncestor2: TraceExt × ℕ × ℕ ⟶ ℕ
getNearestCommonAncestor2(t, i1, i2) ≜
(
   let chain1 = getCausalChainBackwards(t, i1),
       chain2 = getCausalChainBackwards(t, i2)
   in
   (
      for res = len t to 1 by -1 do
        if chain1(res) ∧ chain2(res) then
           return res;
      return 0
    )
)
pre i1 ∈ inds t ∧ i2 ∈ inds t;

-- Similar, but with different traces for i1 and i2
static pure getNearestCommonAncestor3: TraceExt × ℕ × TraceExt × ℕ ⟶ [ℕ × ℕ]
getNearestCommonAncestor3(t1, i1, t2, i2) ≜
(
   let chain1 = getCausalChainBackwards(t1, i1),
       chain2 = getCausalChainBackwards(t2, i2)
   in
   (
      for res1 = i1 to 1 by -1 do
         if chain1(res1) then
            for res2 = i2 to 1 by -1 do
               if chain2(res2) ∧ t1(res1) = t2(res2) then
                  return mk_(res1, res2);
```

```
      return nil
   )
)
pre i1 ∈ inds t1 ∧ i2 ∈ inds t2;


-- Obtains the set of events (and corresponding traces) that can occur immediatly before
-- a given event, without intermediate events.
static pure getAdjacentEvents: TCTraceExt-set × Event × 𝔹 × 𝔹⟶ Event →ᵐ (TCTraceExt ×
ℕ × ℕ × ℕ)
getAdjacentEvents(T, e2, ancestorMandatory, findAll) ≜
(
  dcl res : Event →ᵐ (TCTraceExt × ℕ × ℕ × ℕ) := {↦};
  dcl occurAfter : Event-set := ∅;
  for all mk_(t, c)  ∈ T do
    for k = len t to 1 by -1 do
      if removeExtraInfo(t(k)) = e2 then
      (
        dcl lfs : Lifeline-set := ∅;
        for i = k-1 to 1 by -1 do
          if t(i).lifeline ∉ lfs then
            let e1 = removeExtraInfo(t(i)) in
            (
              if e1 ∉ occurAfter ∧ e1 ∉ dom res then
                let i0 = getNearestCommonAncestor2(t, i, k) in
                  if (i0 > 0 ∨ ¬ ancestorMandatory) ∧ i0 ≠ i then
                    if mayOccurAfter(T, e1, e2) then
                      occurAfter := occurAfter ∪ {e1}
                    else
                    (
                      res(e1) := mk_(mk_(t, c), i0, i, k);
                      if ¬ findAll then
                        return res
                    );
              lfs := lfs ∪ {e1.lifeline}
            )
      );
  return res
);

-- Checks if an event e1 may occur after an event e2 in a set T of tc-traces.
functions
mayOccurAfter: TCTraceExt-set × Event × Event ⟶ 𝔹
mayOccurAfter(T, e1, e2) ≜
  ∃ mk_(t,c) ∈ T •
    ∃ i1, i2 ∈ inds t •
      i1 > i2 ∧ removeExtraInfo(t(i1)) = e1 ∧ removeExtraInfo(t(i2)) = e2;

-- Checks if two (mutually exclusive) events (in the same lifeline) may be
-- simultaneously enabled.
operations
static pure simultaneouslyEnabled: TCTraceExt-set × Event × Event ⟶ [TCTraceExt × ℕ ×
ℕ × TCTraceExt × ℕ × ℕ]
simultaneouslyEnabled(T, e1, e2) ≜
(
  for all mk_(t1, c1) ∈ T do
    for i1 = 1 to len t1 by 1 do
      if removeExtraInfo(t1(i1)) = e1 then
```

```
          for all mk_(t2, c2) ∈ T do
            if e1 ∉ elems removeExtraInfo(t2) then
              for i2 = 1 to len t2 by 1 do
                if removeExtraInfo(t2(i2)) = e2 then
                  let p1 = [t1(i) | i ∈ {1, ..., i1-1} • t1(i).lifeline = e1.lifeline],
                      p2 = [t2(i) | i ∈ {1, ..., i2-1} • t2(i).lifeline = e2.lifeline]
                  in
                     if p1 = p2 then
                       let I1 =  lifelineInds(e1.lifeline, t1(1, ..., i1)),
                           I2 =  lifelineInds(e2.lifeline, t2(1, ..., i2)),
                           cp1 = projectToVars(c1, I1),
                           cp2 = projectToVars(c2, I2)
                       in if sat(mkAndExp(cp1.args ∪ cp2.args)) then
                            let nca = getNearestCommonAncestor3(t1, i1, t2, i2) in
                              if nca ≠ nil then
                                let mk_(i01, i02) = nca in
                                  return mk_(mk_(t1, c1), i01, i1, mk_(t2, c2), i02,
i2);
   return nil
);

-- Tries to add time constraints to a sequence diagram (sd) to enforce local
-- controllability.
-- Assumes all the messages are timed.
-- Receives parameters with maximum transmission time and maximum response time that
-- can be assumed.
-- The first refers to the duration between between emission and reception of a message.
-- The second refers to the duration between the event that precedes an emission event
-- in a lifeline, and the emission event itself.
-- Returns a set of candidates, where each candidate is a set of time constraints to
-- solve a specific problem.
static pure genCoordTimeConstraints: Interaction × TCTraceExt-set × Event-set × Duration
× Duration  ⟶ TimeConstraint-set-set
genCoordTimeConstraints(sd, VExt, E0, transmissionTime, responseTime) ≜
(
   dcl sol : TimeConstraint-set-set := ∅;
   dcl sortedMessages: Message* := sort[Message](sd.messages, compByMessageId);
   dcl E : Event-set :=  ⋃ {let m = msg(sd, e) in
                                 if m.type = <Synch> then {s(m), r(m)} else {e}
                                 | e ∈ E0};

   -- search for patterns of potential race conditions:  triples of events e0, e1, e2,
   -- such that e2 is a discrepant event, e1 immeditaly precedes e2,
   -- and e0 is a common ancestor of e1 and e2 (distinct from e1).
   let pairs = sort[Event × Message]({mk_(e, msg(sd, e)) | e ∈ E}, compByMessageId2) in
     for mk_(e2, -) in pairs do
       let map1 = getAdjacentEvents(VExt, e2, true, true) in
         for all e1 ∈ dom map1 do
           let mk_(tc, i0, i1, i2) = map1(e1),
               tcs = insertUpperLowerBounds(
                         μ(sd, timeConstraints ↦ sd.timeConstraints ∪ ⋃ sol),
                         tc, i0, i1, tc, i0, i2, responseTime, transmissionTime)
           in if tcs ≠ ∅ then
                sol := sol ∪ {tcs};

   -- search for pairs (e1, e2) of mutually exclusive emission and reception events
   -- on the same lifeline simultaneously enabled, with e1 before e2 in the layout,
   -- and e2 an error location
```

```
   let sd2 = if sol = ∅ then sd else μ(sd, timeConstraints ↦ sd.timeConstraints ∪ ⋃
sol),
       VExt2 = if sol = ∅ then VExt else validTimedTracesExt(sd2)
   in for m2 in sortedMessages do
        for mk_(e2, loc2) in [mk_(r(m2), m2.receiveEvent), mk_(s(m2), m2.sendEvent)] do
          for m1 in sortedMessages do
            for all mk_(e1, loc1) ∈ {mk_(r(m1), m1.receiveEvent), mk_(s(m1),
m1.sendEvent)} do
              if e1.lifeline = e2.lifeline ∧ loc1.#2 < loc2.#2 ∧ e1.type ≠ e2.type
                 ∧ (e1 ∈ E ∨ e2 ∈ E)
                 then
                   let res = simultaneouslyEnabled(VExt2, e1, e2) in
                     if res ≠ nil then
                         let mk_(tc1, i01, i1, tc2, i02, i2) = res,
                             tcs = insertUpperLowerBounds(sd2,
                                      tc1, i01, i1, tc2, i02, i2, responseTime,
transmissionTime)
                         in if tcs ≠ ∅ then
                              sol := sol ∪ {tcs};

   -- search for patterns of roundtrip constraints not satisfied: pairs of events e0 and
   -- e1, such that e1 is associated with an error location, e0 is a reception even that
   -- precedes e1 in the same lifeline, and there is a maximum duration constraint from
   -- e0 to e1 that is not met.
   let sd3 = if sol = ∅ then sd else μ(sd, timeConstraints ↦ sd.timeConstraints ∪ ⋃
sol),
       VExt3 = if sol = ∅ then VExt else validTimedTracesExt(sd3)
   in  for all e1 ∈ E do
         if e1.type = <Receive> then
           for all mk_(t, c) ∈ VExt do
             for all j ∈ inds t do
               if removeExtraInfo(t(j)) = e1 then
                 for all i ∈ {1, ..., j-1} do
                   if t(i).lifeline = t(j).lifeline ∧ t(i).type = <Send> then
                     for all mk_DC((j), (i), d) ∈ c.args do
                       let tcs = insertUpperBounds(
                                   μ(sd, timeConstraints ↦ sd.timeConstraints ∪ ⋃ sol),
                                   mk_(t, mk_AndExp(c.args \ {mk_DC(j, i, d)})), i, j,
d, responseTime, transmissionTime)
                       in if tcs ≠ ∅ then
                            sol := sol ∪ {tcs};

   return sol;
);

-- Inserts time constraints for delaying one of the paths.
static pure insertWaitTime: Interaction × TimeConstraint-set × TraceExt × ℕ × ℕ ×
TraceExt × ℕ × ℕ ⟶ TimeConstraint-set
insertWaitTime(sd, tcs, t1, i01, i1, t2, i02, i2) ≜
(
    let c21 = constraintExp(t1, μ(sd, timeConstraints ↦ sd.timeConstraints ∪ tcs)),
        c22 = constraintExp(t2, μ(sd, timeConstraints ↦ sd.timeConstraints ∪ tcs)),
        d1 = getMaxDiff(c21, i01, i1),
        d2 = getMinDiff(c22, i02, i2),
        m02 = getMessageOfEventExt(sd, t2(i02))
    in if d1 ≠ nil ∧ d1 ≥ d2 then
         let E2 = getCausalEventsBetween(t2, i02, i2) in
            for restrict2actors in [true, false] do
```

```
            for chgMin in [false, true] do
             -- find the first emission event in the causal chain without a lower time
             -- bound with priority for events performed by actors
             -- (and no upper time bound or compatible time bound)
             for k = i02 + 1 to i2 by 1 do
               if E2(k) ∧ t2(k).type = <Send> ∧ (¬ restrict2actors ∨
t2(k).lifeline.actor) then
               (
                 dcl j : ℕ := k-1;
                 dcl found: 𝔹 := false;
                 while j ≥ i02 ∧ ¬ found do
                 (
                   if E2(j) ∧ t2(j).lifeline = t2(k).lifeline then
                   (
                      let m = getMessageOfEventExt(sd, t2(k)),
                          m1 = getMessageOfEventExt(sd, t2(j)),
                          ts = if t2(j).type = <Send> then m1.sendTimestamp else
m1.recvTimestamp
                      in
                           if (¬ hasMaxDurationConstraint(sd, ts, m.sendTimestamp)
                               ∨ getMaxDurationConstraint(sd, ts, m.sendTimestamp) >
d1+1)
                              ∧ (¬ hasMinDurationConstraint(sd, ts, m.sendTimestamp)
                                  ∨ chgMin ∧ getMinDurationConstraint(sd, ts,
m.sendTimestamp) < d1+1)
                           then
                             return {mk_TimeConstraint(ts, m.sendTimestamp, d1 + 1,
nil)};
                      found := true
                   );
                   j := j-1
                 )
               );
    return ∅;
);


-- Inserts time constraints for accelerating one of path and delaying the other.
functions
insertUpperLowerBounds: Interaction × TCTraceExt × ℕ × ℕ × TCTraceExt × ℕ × ℕ × ℕ × ℕ
⟶ TimeConstraint-set
insertUpperLowerBounds(sd, mk_(t1, c1), i01, i1, mk_(t2, c2), i02, i2, responseTime,
transmissionTime) ≜
   let d1 = getMaxDiff(c1, i01, i1),
       d2 = getMinDiff(c2, i02, i2)
   in if d1 = nil ∨ d1 ≥ d2 then
        let u = insertUpperBounds2(sd, mk_(t1, c1), i01, i1,  responseTime,
transmissionTime),
            w = insertWaitTime(sd, u, t1, i01, i1, t2, i02, i2)
        in if w = ∅ then ∅ else u ∪ w
      else ∅;

-- Inserts time constraints for accelerating a path between two events in a trace
-- bewlow a certain limit.
insertUpperBounds: Interaction × TCTraceExt × ℕ × ℕ × ℕ × ℕ × ℕ ⟶ TimeConstraint-set
insertUpperBounds(sd, mk_(t, c), i, j, upperBound, responseTime, transmissionTime) ≜
   let d = getMaxDiff(c, i, j) in
     if d = nil ∨ d > upperBound then
```

```
      let tcs = insertUpperBounds2(sd, mk_(t, c), i, j, responseTime, transmissionTime)
in
        if tcs ≠ ∅  then
          let newc = constraintExp(t, μ(sd, timeConstraints ↦ sd.timeConstraints ∪
tcs)),
              newd = getMaxDiff(newc, i, j)
          in if newd ≠ nil ∧  newd ≤ upperBound then tcs else ∅
        else ∅
     else ∅;

-- Inserts time constraints (upper bounds) for trying to accelerate a path between two
-- events in a trace.
operations
static pure insertUpperBounds2: Interaction × TCTraceExt × ℕ × ℕ × ℕ × ℕ ⟶
TimeConstraint-set
insertUpperBounds2(sd, mk_(t, c), i, j,  responseTime, transmissionTime) ≜
(
   dcl tcs : TimeConstraint-set := ∅;
   let E1 = getCausalEventsBetween(t, i, j) in
     for k = j to i + 1 by - 1 do
     (
        if E1(k) then
           if t(k).type = <Send> then
           (
              dcl l : ℕ := k - 1;
              dcl found: 𝔹 := false;
              while l ≥ i ∧ ¬ found do
              (
                 if E1(l) ∧ t(l).lifeline = t(k).lifeline then
                 (
                    if ¬ hasMaxDurationConstraint(sd, t(l).timestamp, t(k).timestamp)
then
                       if hasMinDurationConstraint(sd, t(l).timestamp, t(k).timestamp)
then
                          tcs := tcs ∪ {mk_TimeConstraint(t(l).timestamp,
t(k).timestamp, nil,
                                       getMinDurationConstraint(sd, t(l).timestamp,
t(k).timestamp) + responseTime)}
                        else
                          tcs := tcs ∪ {mk_TimeConstraint(t(l).timestamp,
t(k).timestamp, nil, responseTime)};
                    found := true;
                 );
                 l := l-1;
             );
           )
           else if getMessageOfEventExt(sd, t(k)).type = <Asynch> then
           (
              dcl l : ℕ := k - 1;
              dcl found: 𝔹 := false;
              while l ≥ i ∧ ¬ found do
              (
                 if E1(l) ∧ t(l).type = <Send> ∧ t(l).messageId = t(k).messageId  then
                 (
                    if ¬ hasMaxDurationConstraint(sd, t(l).timestamp, t(k).timestamp)
then
                       tcs := tcs ∪ {mk_TimeConstraint(t(l).timestamp, t(k).timestamp,
nil, transmissionTime)};
```

```
                  found := true;
               );
               l := l-1;
            );
         );
      );

   return tcs

);


/**************************** Finding error locations *************************/

-- Finds invalid or missing events in a set I of invalid tc-traces,
-- by comparison with V a set of valid tc-traces.
static pure findDiscrepantEvents2: TCTrace-set × TCTrace-set ⟶ Event-set
findDiscrepantEvents2(I, V) ≜
(
   dcl res : Event-set := ∅;
   for all mk_(t, c) ∈ I do
      let missing = {v(len t + 1) | mk_(v, vc) ∈ V • len v > len t
                              ∧ eqIgnTimestamps(t, v(1,...,len t))
                              ∧ sat(mkAndExp({c} ∪ elimVarsAfter(len t, vc.args)))} in
         if missing ≠ ∅ then
            res := res ∪ missing
         else
         (
            dcl t1 : Trace := t;
            dcl c1 : DCExp := c;
            while t1 ≠ [] ∧ ∄ mk_(v, vc) ∈ V • len v ≥ len t1
                              ∧ eqIgnTimestamps(t1, v(1,...,len t1))
                              ∧ sat(mkAndExp({c1} ∪ elimVarsAfter(len t1, vc.args)))
            do (
               t1 := t1(1, ..., len t1 - 1);
               c1 := elimVarsAfter(len t1, c1);
            );
            if len t1 < len t then
               res := res ∪ {t(len t1 + 1)}
         );
  return res;
);


/****************** Insertion of coordination messages *************************/

functions

-- Generates a valid message id for a new coordination message to insert in an
Interaction (sd).
newMessageId: Interaction ⟶ ℕ
newMessageId(sd) ≜ hd reverse [id | id ∈ {m.id | m ∈ sd.messages}] + 1
pre sd.messages ≠ ∅;

-- Inserts a set of coordination messages (msgs) in an Interaction (sd).
insertMessages: Interaction × Message-set ⟶ Interaction
insertMessages(sd, msgs) ≜
   μ(sd, messages ↦ sd.messages ∪ msgs,
```

```
            combinedFragments ↦ sd.combinedFragments,
            messageMap ↦ sd.messageMap ∪ {m.id ↦ m | m ∈ msgs});

-- Removes all events associated with coordination messages in a set of traces (T).
removeCoordinationEvents: Trace-set × Message-set ⟶ Trace-set
removeCoordinationEvents(T, newMsgs) ≜
  let sigs = {m.signature | m ∈ newMsgs} in
     {[e | e ∈ t • e.signature ∉ sigs] | t ∈ T};

-- Removes all events associated with coordination messages in a set of traces (T).
-- Assumes that such messages start with a common prefix and order number.
removeCoordinationEvents: Trace-set × ℕ ⟶ Trace-set
removeCoordinationEvents(T, ordnum) ≜
     {[e | e ∈ t •
            ¬ (e.signature(1,..., len CoordinationMessagePrefix) =
CoordinationMessagePrefix
                  ∨ e.signature(1,..., len AcknowledgmentMessagePrefix) =
AcknowledgmentMessagePrefix)]
     | t ∈ T};

-- Checks is a modified Interaction (sd), with coordination features added,
-- preserves the valid local traces of the initial Interaction (appart from timing
constraints).
preserveValidLocalTraces3: Interaction ×  (Lifeline →m Trace-set) × (Lifeline →m Trace-
set)  ⟶ 𝔹
preserveValidLocalTraces3(sd, oldP, newP) ≜
  ∀ l ∈ sd.lifelines • oldP(l) = newP(l);

-- Checks is a modified Interaction (sd), with coordination features added,
-- preserves the valid local traces (P) of the initial Interaction (appart from timing
constraints).
preserveValidLocalTraces2: Interaction ×  Lifeline →m Trace-set × Message-set × TCTrace-
set ⟶ 𝔹
preserveValidLocalTraces2(sd, P, newMsgs, V) ≜
  let newV = removeCoordinationEvents({t | mk_(t,-) ∈ V}, newMsgs),
      newP = {l ↦ projectTraces(newV, l) | l ∈ sd.lifelines},
      changed = {l | l ∈ sd.lifelines • P(l) ≠ newP(l)}
  in changed = ∅;


/*********** Generation of candidate coordination messages ******************/

-- Generates coordination messages to overcome problems with non-local choices in an
-- interaction (sd).
-- Assumes locations in the given Interaction have been renumbered to permit insertion
-- of coordination messages.
-- Receives the set (E) of suspicious events and parameters for message and location
-- numbering.
--  startId - next (added) message identifier
--  startSuffix - next control message sequence number
--  delta - location delta for insertion of new message ends
operations
static pure genCoordMessagesForNonLocalChoices: Interaction × Event-set × ℕ × ℕ × ℕ ⟶
Message-set
genCoordMessagesForNonLocalChoices(sd, E, startId, startSuffix, delta) ≜
(
  dcl sol : Message-set := ∅;
  dcl sortedMessages: Message* := sort[Message](sd.messages, compByMessageId);
```

```
  dcl id : ℕ := startId;
  dcl suffix : ℕ := startSuffix;

  for all c ∈ sd.combinedFragments do
    if c.interactionOperator = <alt> then
    (
      dcl firstDecider: [InteractionOperand × LifelineLocation × Event]  := nil;
      dcl otherDeciders: (InteractionOperand × LifelineLocation × Event)-set := ∅;
      for o in c.operands do
        for all mk_(lf, loc1) ∈ o.startLocations do
        (
          dcl firstEv : [LifelineLocation]  := nil;
          let loc2 = let mk_((lf), loc2) ∈ o.finishLocations in loc2 in
            for m in sortedMessages do
              if firstEv = nil ∧ m.sendEvent.#1 = lf ∧ m.sendEvent.#2 > loc1 ∧
m.sendEvent.#2 < loc2 then
              (
                firstEv := m.sendEvent;
                if firstDecider = nil then
                  firstDecider := mk_(o, m.sendEvent, s(m))
                else
                  otherDeciders := otherDeciders ∪ {mk_(o, m.sendEvent, s(m))}
              )
        );

        if firstDecider ≠ nil then
          let mk_(o1, mk_(lf1, loc1), -) = firstDecider in
            for all mk_(o2, mk_(lf2, loc2), e2) ∈ otherDeciders do
              if lf2 ≠ lf1 ∧ e2 ∈ E then
              (
                let sourceLoc = if o2 = o1 then loc1
                                else if ∃ mk_((o2), mk_((lf1), -), -) ∈ otherDeciders •
true then
                                        let mk_((o2), mk_((lf1), loc), -) ∈
otherDeciders in loc
                                else let mk_((lf1), loc) ∈ o2.startLocations in loc,
                          m = mkMessageTimed(id, mk_(lf1, sourceLoc + delta), mk_(lf2, loc2 -
delta),
                                  CoordinationMessagePrefix ^
VDMUtil`val2seq_of_char[ℕ](suffix))
                in
                (
                  sol := sol ∪ {m};
                  suffix := suffix + 1;
                  id := id + 1
                )
              )
    );
  return sol;
);

-- Find position of emission event corresponding to another event in an extended trace.
operations
pure static findEmissionPos: TraceExt × ℕ ⟶ ℕ
findEmissionPos(t, k) ≜
(
  for i = k to 1 by -1 do
```

```
    if t(i).messageId = t(k).messageId ∧ t(i).type = <Send> ∧ t(i).itercounter =
t(k).itercounter then
      return i;
  return 0
);

-- Generates coordination messages to overcome race conditions and other event ordering
-- problems.
-- Besides the parameters received by genCoordMessagesForNonLocalChoices,
-- also receives the set of valid tc-traces with extra info (VExt).
operations
static pure genCoordMessagesForEventOrdering: Interaction × TCTraceExt-set × Event-set ×
ℕ × ℕ × ℕ ⟶ Message-set
genCoordMessagesForEventOrdering(sd, VExt, E0, startId, startSuffix, delta) ≜
(
  dcl sol : Message-set := ∅;
  dcl sortedMessages: Message* := sort[Message](sd.messages, compByMessageId);
  dcl id : ℕ := startId;
  dcl suffix : ℕ := startSuffix;

  -- augment set of error events to consider both sides of synchronous messages
  dcl E : Event-set :=  ⋃ {let m = msg(sd, e) in
                                   if m.type = <Synch> then {s(m)} else {e}
                                   | e ∈ E0};

   -- Search for triples of events e0, e1, e2, such that e2 is a discrepant event,
   -- e1 immeditaly precedes e2 (in the e1's lifeline), e0 is the emission event
   -- corresponding to e2 (possibly e0 = e2), and there is no causal chain of events
between e1 and e0.
   -- Generates a coordination message to be sent immeditaly after e1 to the lifeline of
e2.
   let pairs = sort[Event × Message]({mk_(e, msg(sd, e)) | e ∈ E}, compByMessageId2) in
     for mk_(e2, -) in pairs do
       let map1 = getAdjacentEvents(VExt, e2, false, false) in
         for all e1 ∈ dom map1 do
           let mk_(tc, -, i1, i2) = map1(e1),
               mk_(t, c) = tc,
               i0 = findEmissionPos(t, i2)
           in if i0 ≠ 0 ∧ t(i0).lifeline ≠ t(i1).lifeline then
                let m = mkMessageTimed(id, mk_(t(i1).lifeline, t(i1).location + delta),
mk_(t(i0).lifeline, t(i1).location + delta),
                             CoordinationMessagePrefix ↷
VDMUtil`val2seq_of_char[ℕ](suffix))
                in
                (
                  sol := sol ∪ {m};
                  suffix := suffix + 1;
                  id := id + 1
                );
  return sol;
);

-- Generates a set of acknowledgment messages.
operations
static pure genAckMessages: Interaction × TCTrace-set × Event-set × ℕ × ℕ × ℕ ⟶
Message-set
genAckMessages(sd, VT, E, startId, startSuffix, delta) ≜
(
  dcl sol : Message-set := ∅;
```

```
  dcl id : ℕ := startId;
  for all e ∈ E do
    if e.type = <Receive> then
      let m = msg(sd, e) in
        if m.type = <Asynch>
           ∧ # {len [i | i ∈ inds t • t(i) = e] | mk_(t,-) ∈ VT} > 1 then -- opt or loop
           (
             sol := sol ∪ {mkMessageTimed(id,
                                         mk_(m.receiveEvent.#1, m.receiveEvent.#2 + delta),
                                         mk_(m.sendEvent.#1, m.sendEvent.#2 + delta),
                                         AcknowledgmentMessagePrefix ↷ "_" ↷ m.signature)};
             id := id + 1;
           );
  return sol
);



/** Core algorithm for the generation of coordination messages and time constraints ***/

-- Auxiliary structures to represent solutions and intermmediate results
types
public EnforcementSolution = Interaction × Message-set × TimeConstraint-set;

InteractionEval1 ::
        validTCTracesExt    : TCTraceExt-set
        validTCTraces       : TCTrace-set
        validTraces         : Trace-set
        validLocalTraces    : Lifeline →m Trace-set;

InteractionEval2 ::
        validTCTracesExt    : TCTraceExt-set
        validTCTraces       : TCTrace-set
        validTraces         : Trace-set
        validLocalTraces    : Lifeline →m Trace-set
        unintendedTCTraces  : TCTrace-set
        uncheckableTCTraces : TCTrace-set
        unintendedTraces    : Trace-set
        uncheckableTraces   : Trace-set;

-- Computation of several properties of the interaction under analysis
functions
evalInteraction1: Interaction × [TCTraceExt-set] ⟶ InteractionEval1
evalInteraction1(sd, oldVExt) ≜
  let
     VExt = if oldVExt = nil then validTimedTracesExt(sd) else
validTimedTracesExtIncr(sd, oldVExt),
     VT = {mk_(removeExtraInfo(t), c) | mk_(t,c) ∈ VExt}, -- valid tc-traces
     V = untimed(VT), -- valid traces
     P = projectTraces(V, sd.lifelines) -- valid traces per lifeline
  in
    mk_InteractionEval1(VExt, VT, V, P);

evalInteraction2: Interaction × 𝔹 × 𝔹 ⟶ InteractionEval2
evalInteraction2(sd, observability, controllability ) ≜
  let
     VExt = validTimedTracesExt(sd), -- valid tc-traces extended
     VT = {mk_(removeExtraInfo(t), c) | mk_(t,c) ∈ VExt}, -- valid tc-traces
```

```
     V = untimed(VT), -- valid traces
     P = projectTraces(V, sd.lifelines), -- valid traces per lifeline
     U1_timed = if ¬ controllability then ∅
                else if sd.timeConstraints ≠ ∅ then unintendedTracesTimedRaw(sd, VT)
                else {mk_(t, TrueExp) | t ∈ unintendedTracesUntimed(sd, V)}, --
truncated
     U2_timed = if ¬ observability then ∅
                else if sd.timeConstraints ≠ ∅ then uncheckableTracesTimedRaw(sd, VT)
                else {mk_(t, TrueExp) | t ∈  joinTraces(sd, [], P) \ V} -- truncated
  in
    mk_InteractionEval2(VExt, VT, V, P, U1_timed, U2_timed, untimed(U1_timed),
untimed(U2_timed));

evalInteraction2: Interaction × 𝔹 × 𝔹 × InteractionEval1 ⟶ InteractionEval2
evalInteraction2(sd, observability, controllability, mk_InteractionEval1(VExt, VT, V,
P)) ≜
  let
     U1_timed = if ¬ controllability then ∅
                else if sd.timeConstraints ≠ ∅ then unintendedTracesTimedRaw(sd, VT)
                else {mk_(t, TrueExp) | t ∈ unintendedTracesUntimed(sd, V)}, --
truncated
     U2_timed = if ¬ observability then ∅
                else if sd.timeConstraints ≠ ∅ then uncheckableTracesTimedRaw(sd, VT)
                else {mk_(t, TrueExp) | t ∈  joinTraces(sd, [], P) \ V} -- truncated
  in
    mk_InteractionEval2(VExt, VT, V, P, U1_timed, U2_timed, untimed(U1_timed),
untimed(U2_timed));

-- Minimizes a given set of coordination messages (msgs) for a given interaction (sd),
-- with a given set of valid traces per lifeline (P).
operations
static pure minimizeCoordMsgs: Interaction × Lifeline →(m) Trace-set × Message-set × 𝔹 × 𝔹
⟶ Message-set
minimizeCoordMsgs(sd, P, msgs, enforceObservability, enforceControllability) ≜
(
  dcl best : Message-set := msgs;
  for all m ∈ msgs do
    if # best > 1 then
      let msgs2 = msgs \ {m},
           sd2 = insertMessages(sd, msgs2),
           eval1_sd2 = evalInteraction1(sd2, nil)
       in if preserveValidLocalTraces2(sd2, P, msgs2, eval1_sd2.validTCTraces) then
             let eval2_sd2 = evalInteraction2(sd2, enforceObservability,
enforceControllability, eval1_sd2),
                 count1 = # removeCoordinationEvents(eval2_sd2.uncheckableTraces,
msgs2),
                 count2 = # removeCoordinationEvents(eval2_sd2.unintendedTraces, msgs2)
              in if count1 + count2 = 0 then
                   let msgs3 = minimizeCoordMsgs(sd, P, msgs2, enforceObservability,
enforceControllability)
                   in if # msgs3 < # best then
                        best := msgs3;
  return best
);

functions
findDiscrepantEvents: Interaction × 𝔹 × 𝔹 ⟶ TCTraceExt-set × TCTrace-set × Event-set
findDiscrepantEvents(sd, enforceObservability, enforceControllability) ≜
```

```
(
   let mk_InteractionEval2(VExt, VT, V, P, U1_timed, U2_timed, U1, U2) =
         evalInteraction2(sd, enforceObservability, enforceControllability)
   in mk_(VExt, VT, findDiscrepantEvents2(U2_timed ∪ U1_timed, VT))
);


-- Core internal procedure (recursive).
-- Generates and checks possible coordination messages and time constraints to make an
-- interaction locally observable and locally controllable (assuming it initially is not
-- so).
-- Returns a set of solutions, where each solution is a set of coordination messages and
-- time constraints.
operations
static pure genCoordinationFeatures: Interaction × 𝔹 × 𝔹 × 𝔹 × 𝔹 × Duration × Duration
× ℕ × ℕ × [InteractionEval2] ⟶ EnforcementSolution-set
genCoordinationFeatures(sd0, enforceObservability, enforceControllability,
genCoordMessages, genCoordConstraints, transmissionTime, responseTime, suffix,
multiplier, eval) ≜
(
  dcl partialSolWithCoordMsgs : [EnforcementSolution] := nil;
  dcl partialSolWithCoordTc : [EnforcementSolution] := nil;
  dcl cost : ℕ := 0; -- number of candidates explored

  let
     -- makes sure all messages are timed (i.e., all message ends have timestamp
     -- variables)
     sd = if suffix = 1 then mkMsgsTimedAndIndexed(sd0) else sd0,

     -- renumbers the locations in the given Interaction to permit insertion of
     -- coordination messages
     sd2 = if suffix = 1 then renumberInteraction(sd, multiplier) else sd,

     -- pre-computes several sets (valid traces, projected, unintended, uncheckable)
     mk_InteractionEval2(VExt, VT, V, P, U1_timed, U2_timed, U1, U2) =
       if eval ≠ nil then eval else evalInteraction2(sd2, enforceObservability,
enforceControllability),

     U1_truncated = {truncateOnError(VT, tc) | tc ∈ U1_timed},
     count1 = # U1_truncated,
     count2 = # U2,
     UT = U2_timed ∪ U1_timed,
     E = findDiscrepantEvents2(UT, VT)

  in (
    if count1 + count2 = 0 then
      return ∅;

    -- Generate and check candidate coordination messages
    if genCoordMessages then
    (
       dcl coordMsgs : Message-set := ∅;
       dcl newId : ℕ := newMessageId(sd2); -- start id for new messages
       dcl delta : ℕ := multiplier div 2;
       dcl n : ℕ := suffix;

       -- coordination messages for non-local choices
       let nlcMsgs = genCoordMessagesForNonLocalChoices(sd2, E, newId, n, delta) in
         if nlcMsgs ≠ ∅ then
```

```
          (
            coordMsgs := coordMsgs ∪ nlcMsgs;
            n := n + # nlcMsgs;
            newId := newId + # nlcMsgs
          );

        -- coordination messages for races and other event ordering problems remaining
        delta := delta div 2;
        let sd3 = if coordMsgs = ∅ then sd2 else insertMessages(sd2, coordMsgs),
            mk_(VExt3, -, E3) = if coordMsgs = ∅ then mk_(VExt, VT, E)
                                  else findDiscrepantEvents(sd3, enforceObservability,
enforceControllability),
            eoMsgs = genCoordMessagesForEventOrdering(sd3, VExt3, E3, newId, n, delta)
        in if eoMsgs ≠ ∅ then
           (
             coordMsgs := coordMsgs ∪ eoMsgs;
             n := n + # eoMsgs;
             newId := newId + # eoMsgs
           );

        -- acknowledgment messages, for observability problems remaining
        if count2 > 0 then
        (
          delta := delta div 2;
          let sd3 = if coordMsgs = ∅ then sd2 else insertMessages(sd2, coordMsgs),
              mk_(-, VT3, E3) = if coordMsgs = ∅ then mk_(VExt, VT, E)
                                  else findDiscrepantEvents(sd3, enforceObservability,
enforceControllability)
          in let ackMsgs = genAckMessages(sd3, VT3, E3, newId, suffix, delta) in
               if ackMsgs ≠ ∅ then
               (
                 coordMsgs := coordMsgs ∪ ackMsgs;
                 n := n + # ackMsgs;
                newId := newId + # ackMsgs
               )
        );

        -- evaluate the set of candidates
        if coordMsgs ≠ ∅ then
           let sd3 = insertMessages(sd2, coordMsgs),
               eval1_sd3 = evalInteraction1(sd3, nil)
           in
           (
              cost := cost+1;
              if preserveValidLocalTraces2(sd3, P, coordMsgs, eval1_sd3.validTCTraces)
then
                 let eval2_sd3 = evalInteraction2(sd3, enforceObservability,
enforceControllability, eval1_sd3),
                     count4 = # removeCoordinationEvents(eval2_sd3.uncheckableTraces,
coordMsgs),
                     count3 = # removeCoordinationEvents(eval2_sd3.unintendedTraces,
coordMsgs)
                 in
                   -- full solution
                   if count3 + count4 = 0 then
                   (
                      -- minimze solution and terminate
                      let minMsgs = minimizeCoordMsgs(sd2, P, coordMsgs,
enforceObservability, enforceControllability)
```

```
                  in return {mk_(sd3, minMsgs, ∅)};
                 )

                 -- partial sollution, may be usefull for a combined solution
                 else if count3 + count4 < count1 + count2 then
                 (
                   partialSolWithCoordMsgs := mk_(sd3, coordMsgs, ∅);
                 )
         );
   );

   -- Generate and check coordination time constraints for controllability issues
   if genCoordConstraints ∧ transmissionTime ≠ 0 ∧ responseTime ≠ 0 ∧ count1 > 0 then
       let tc = ⋃ genCoordTimeConstraints(sd2, VExt, E, transmissionTime,
responseTime),
           sd3 = simplifyTimeConstraints(μ(sd2, timeConstraints ↦ sd2.timeConstraints
∪ tc)),
           eval1_sd3 = evalInteraction1(sd3, VExt)
       in
       (
         if preserveValidLocalTraces3(sd3, P, eval1_sd3.validLocalTraces) then
           let eval2_sd3 = evalInteraction2(sd3, enforceObservability,
enforceControllability, eval1_sd3),
                   count4 = # removeCoordinationEvents(eval2_sd3.uncheckableTraces,
suffix),
                   count3 = # removeCoordinationEvents(eval2_sd3.unintendedTraces,
suffix)

           in -- full solution, just return
              if count3 + count4 = 0 then
              (
                return {mk_(sd3, ∅, sd3.timeConstraints \ sd2.timeConstraints)};
              )
              else if count3 + count4 < count1 + count2 then
              (
                partialSolWithCoordTc := mk_(sd3, ∅, sd3.timeConstraints \
sd2.timeConstraints);
              )
       );

    -- try a combined solution (could also search incrementally)
    if partialSolWithCoordMsgs ≠ nil ∧ partialSolWithCoordTc ≠ nil then
       let mk_(sd3, -, t) = partialSolWithCoordTc,
           mk_(-, m, -) = partialSolWithCoordMsgs,
           sd4 = insertMessages(sd3, m),
           eval1_sd4 = evalInteraction1(sd4, nil)
       in if preserveValidLocalTraces2(sd4, P, m, eval1_sd4.validTCTraces) then
            let eval2_sd4 = evalInteraction2(sd4, enforceObservability,
enforceControllability, eval1_sd4),
                count4 = # removeCoordinationEvents(eval2_sd4.uncheckableTraces, m),
                count3 = # removeCoordinationEvents(eval2_sd4.unintendedTraces, m)
            in if count3 + count4 = 0 then
               (
                 -- minimize solution and terminate
                 let minMsgs = minimizeCoordMsgs(sd4, P, m, enforceObservability,
enforceControllability)
                 in return {mk_(sd4, minMsgs, t)};
               )
  );
```

```
  -- Failed
  return ∅
);

-- Simplified version, with fewer parameters, still being used for testing.
operations
protected static pure genCoordinationMessages: Interaction × 𝔹 ⟶ EnforcementSolution-set
genCoordinationMessages(sd, controllabilityOnly) ≜
genCoordinationFeatures(sd, ¬ controllabilityOnly, true, true, false, 0, 0, 1,
LocationMultiplier, nil);


/****** Computation of relative positions of inserted coordination messages *******/

-- Auxiliary types to represent relative positions.
types
PositionType = <Before> | <After>;
PositionTarget = MessageEnd | CombinedFragment | InteractionOperand;
RelativePosition ::
         positionType: PositionType
         positionTarget: PositionTarget;
MessageEnd ::
          event: EventType
          signature: String
          lifeline: Lifeline
          location: ℕ;

-- Obtains the relative position of an event (message end) in an interaction.
functions
getRelativePosition: MessageEnd × Interaction ⟶ [RelativePosition]
getRelativePosition(e, sd) ≜
   let messageEnds = ⋃{
        {mk_MessageEnd(<Send>, m.signature, m.sendEvent.#1, m.sendEvent.#2),
         mk_MessageEnd(<Receive>, m.signature, m.receiveEvent.#1, m.receiveEvent.#2)}
        | m ∈ sd.messages},
       before = {me | me ∈ messageEnds • me.lifeline = e.lifeline ∧ me.location <
e.location},
       after = {me | me ∈ messageEnds • me.lifeline = e.lifeline ∧ me.location >
e.location},
       closestBefore = if before = ∅ then nil
                       else ι me ∈ before • ∄ me2 ∈ before • me2.location > me.location,
       closestAfter = if after = ∅ then nil
                      else ι me ∈ after • ∄ me2 ∈ after • me2.location < me.location
   in if closestBefore = nil then
         if closestAfter = nil then nil
         else mk_RelativePosition(<Before>, closestAfter)
      else mk_RelativePosition(<After>, closestBefore);

types
CoordMsgEnd = MessageEnd × [RelativePosition];

-- Obtains the relative positions of the events (message end) in a message.
functions
getRelativePosCoordMessage: Message × Interaction ⟶ CoordMsgEnd*
getRelativePosCoordMessage(m, sd) ≜
   let s = mk_MessageEnd(<Send>, m.signature, m.sendEvent.#1, m.sendEvent.#2),
       r = mk_MessageEnd(<Receive>, m.signature, m.receiveEvent.#1, m.receiveEvent.#2)
```

```
   in [ mk_(s, getRelativePosition(s, sd)), mk_(r, getRelativePosition(r, sd))];

-- Represents a coordination event (message end) in a string
static coordEvent2str: CoordMsgEnd ⟶ String
coordEvent2str(mk_(e, pos)) ≜
   (if e.event = <Send> then "!" else "?") ↷ e.signature ↷ "@" ↷ e.lifeline.name
   ↷ (if pos = nil then ""
      else (if pos.positionType = <Before> then "B" else "A")
           ↷ pos.positionTarget.signature);


/********** Main functions, with results in string **********************/

-- Generates coordinations messages and time constraints to make an interaction locally
-- controllable and/or locally observable.
-- Input parameters:
--   sd: interaction
--   enforceObservability: enforce local observability
--   enforceControllability: enforce local controllability
--   genCoordMessages: generate coordination messages
--   genCoordConstraints: generate coordination time constraints
--   transmissionTime: default transmission time to consider for generation of time
--       constraints
--   responseTime: default response time to consider for generation of time constraints
-- Returns:
--   A string representation of the coordination messages and time constraints.
public genCoordinationFeaturesStr: Interaction × 𝔹 × 𝔹 × 𝔹 × 𝔹 × nat1 × nat1 ⟶ String
genCoordinationFeaturesStr(sd, enforceObservability, enforceControllability,
genCoordMessages, genCoordConstraints, transmissionTime, responseTime) ≜
  let sd1 = mkMsgsTimedAndIndexed(sd),
      sd2 = renumberInteraction(sd1, LocationMultiplier),
      sol1 = genCoordinationFeatures(sd1, enforceObservability, enforceControllability,
genCoordMessages, genCoordConstraints, transmissionTime, responseTime, 1,
LocationMultiplier, nil),
      sol2 =  {mk_(sd3, {getRelativePosCoordMessage(m, sd2) | m ∈ s}, t)
                 | mk_(sd3, s, t) ∈ sol1}
  in set2str[Interaction × CoordMsgEnd*-set × TimeConstraint-set](
       sol2,
       λ mk_(sd3, s,t): Interaction × CoordMsgEnd*-set × TimeConstraint-set •
         pair2str[CoordMsgEnd*-set, TimeConstraint-set] (mk_(s,t),
              λ s: CoordMsgEnd*-set • set2str[CoordMsgEnd*](s,
                      λ sq: CoordMsgEnd* • seq2str[CoordMsgEnd](sq, coordEvent2str)),
              λ t: TimeConstraint-set •
                     set2str[TimeConstraint](t,
                        λ t1: TimeConstraint • timeConstraint2str(sd3, t1)))
     );

-- Simplified version of genCoordinationFeaturesStr, with fewer input parameters.
public genCoordinationMessagesStr: Interaction × 𝔹 × Duration × Duration  ⟶ String
genCoordinationMessagesStr(sd, controllabilityOnly, transmissionTime, responseTime) ≜
 genCoordinationFeaturesStr(sd, ¬ controllabilityOnly, true, true, true,
transmissionTime, responseTime);

-- Simplified version of genCoordinationFeaturesStr, with fewer input parameters.
public genCoordinationMessagesStr2: Interaction × 𝔹 ⟶ String
genCoordinationMessagesStr2(sd, controllabilityOnly) ≜
   let sd1 = mkMsgsTimedAndIndexed(sd),
       sd2 = renumberInteraction(sd1, LocationMultiplier),
       sol1 = genCoordinationMessages(sd1, controllabilityOnly),
```

```
      sol2 =  {{getRelativePosCoordMessage(m, sd2) | m ∈ s} | mk_(-, s, -) ∈ sol1}
   in set2str[CoordMsgEnd*-set](sol2,
        λ s: CoordMsgEnd*-set • set2str[CoordMsgEnd*](s,
                 λ sq: CoordMsgEnd+ • seq2str[CoordMsgEnd](sq, coordEvent2str)));

end Enforcement
```

12. Class TestCases

```
/**
 * Test cases.
 */

class TestCases is subclass of SequenceDiagrams, Traces, ValidTraces, Observability,
Controllability, Enforcement, ConformanceChecking

operations

/*** Auxiliary assertion checking primitives and test helpers ***/

-- Simulates assertion checking by reducing it to post-condition checking.
-- If 'arg' does not hold, a post-condition violation will be signaled.
protected assertTrue: 𝔹 ⟶ ()
assertTrue(arg) ≙
  return
post arg;

protected assertFalse: 𝔹 ⟶ ()
assertFalse(arg) ≙
  return
post ¬ arg;

-- Simulates assertion checking by reducing it to post-condition checking.
-- If values are not equal, prints a message in the console and generates
-- a post-conditions violation.
protected assertEqual: ? × ? ⟶ ()
assertEqual(expected, actual) ≙
  if expected ≠ actual then (
    IO`print("Actual value (");
    IO`print(actual);
    IO`print(") different from expected (");
    IO`print(expected);
    IO`println(")\n")
)
post expected = actual;

functions

-- Generates time constraints with given minimum and maximum durations for a set of
messages.
public mkMsgTimeConstraints: Message-set × [Duration] × [Duration] ⟶ TimeConstraint-set
mkMsgTimeConstraints(messages, minDuratin, maxDuration) ≙
  {mk_TimeConstraint(t(s(m)), t(r(m)), minDuratin, maxDuration) | m ∈ messages};


/*************** Test scenarios without time constraints ***************/

operations

-- Simple scenario with two lifelines and two messages in opposite directions.
-- It has a single valid trace and is locally observable and controllable.
public testSimple() ≙
(
  let l1 = mkLifeline("L1"),
```

```
      l2 = mkLifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, ∅)
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
    assertEqual(<Pass>, finalConformanceChecking(sd1,
                        {l1 ↦ [s(m1), r(m2)], l2 ↦ [r(m1), s(m2)]}));
    assertTrue(checkNextEvent([s(m1)], r(m2), {[s(m1), r(m2)]}));
    assertFalse(checkNextEvent([], s(m2), {[r(m1), s(m2)]}));
    assertEqual({s(m1)}, nextSendEvents([], {[s(m1), r(m2)]}));
    assertTrue(isLocallyObservable(sd1));
    assertTrue(isLocallyControllable(sd1));
  )
);

-- Invalid scenario with two lifelines and two messages that cross each other.
public testImpossible() ≙
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l2, 2), mk_(l1, 1), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, ∅)
  in
  (
    assertEqual(∅, validTraces(sd1));
  )
);

-- Scenario with four lifelines and two independent messages.
-- It has several valid traces corresponding to different permutations of the same
events.
-- It is locally observable and controllable.
public testIndepMessages() ≙
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      l4 = mkLifeline("L4"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l3, 1), mk_(l4, 1), "m2"),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2}, ∅)
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)], [s(m1), s(m2), r(m1), r(m2)],
                 [s(m1), s(m2), r(m2), r(m1)], [s(m2), s(m1), r(m1), r(m2)],
                 [s(m2), s(m1), r(m2), r(m1)], [s(m2), r(m2), s(m1), r(m1)]},
               validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertTrue(isLocallyControllable(sd1));
  )
);

-- Scenarios with an 'opt' combined fragment with an optional message.
-- It is not locally observable, but can be refined with an acknowledgment message.
-- Corresponds to motivating example b).
public testOpt() ≙
```

```
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      o1 = mk_InteractionOperand(nil, {mk_(l1,1), mk_(l2,1)}, {mk_(l1,3), mk_(l2,3)}),
      f1 = mk_CombinedFragment(<opt>, [o1], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1}, {f1})
  in
  (
    assertEqual({[s(m1), r(m1)], []}, validTraces(sd1));
    assertEqual(<Pass>, finalConformanceChecking(sd1, {l1 ↦ [], l2 ↦ []}));
    assertEqual(<Pass>, finalConformanceChecking(sd1, {l1 ↦ [s(m1)], l2 ↦ [r(m1)]}));
    assertEqual(<Fail>, finalConformanceChecking(sd1, {l1 ↦ [s(m1)], l2 ↦ []}));
    assertTrue(isLocallyControllable(sd1));
    assertEqual({[s(m1)]}, uncheckableLocally(sd1));
    assertEqual("{{[!Ack_m1@L2Am1, ?Ack_m1@L1Am1]}}",
                genCoordinationMessagesStr2(sd1, false));
  )
);

-- Scenario with an 'alt' combined fragment with two alternative messages.
-- It is locally observable and controllable.
public testAlt() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l1, 4), mk_(l2, 4), "m2"),
      o1 = mk_InteractionOperand(nil, {mk_(l1,1), mk_(l2,1)}, {mk_(l1, 3), mk_(l2, 3)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1,3), mk_(l2,3)}, {mk_(l1, 5), mk_(l2, 5)}),
      f1 = mk_CombinedFragment(<alt>, [o1, o2], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {f1})
  in
  (
    assertEqual({[s(m1), r(m1)], [s(m2), r(m2)]}, validTraces(sd1));
    assertEqual(<Pass>, finalConformanceChecking(sd1, {l1 ↦ [s(m1)], l2 ↦ [r(m1)]}));
    assertTrue(isLocallyObservable(sd1));
    assertTrue(isLocallyControllable(sd1));
  )
);

-- Scenario with a 'strict' combined fragment that imposes an ordering between
-- otherwise unrelated message ends.
-- It is not locally observable nor controllable.
-- Such properties can be enforced with a coordination message.
public testStrict() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      m1 = mkMessage(1, mk_(l1, 3), mk_(l2, 3), "m1"),
      m2 = mkMessage(2, mk_(l3, 5), mk_(l2, 5), "m2"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2), mk_(l3, 2)},
                                 {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)},
                                {mk_(l1, 6), mk_(l2, 6), mk_(l3, 6)}),
      f1 = mk_CombinedFragment(<strict>, [o1, o2], {l1, l2, l3}),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2}, {f1})
  in
```

```
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
    assertEqual(<Inconclusive>, finalConformanceChecking(sd1,
                   {l1 ↦ [s(m1)], l2 ↦ [r(m1), r(m2)], l3 ↦ [s(m2)]}));
    assertEqual({[s(m1), s(m2), r(m1), r(m2)], [s(m2), s(m1), r(m1), r(m2)]},
                uncheckableLocally(sd1));
    assertEqual({[s(m1), s(m2)], [s(m2)]}, unintendedTraces(sd1));
    assertEqual(∅, missingTraces(sd1)) ;
    assertEqual("{{[!Ctrl1@L2Am1, ?Ctrl1@L3Bm2]}}",
               genCoordinationMessagesStr2(sd1, false));
  )
);

-- Scenario with a 'loop' combined fragment with a finite number of iterations
-- and a single message in each iteration.
-- According to the standard, there is a weak sequencing between iterations.
-- Because last iteration is optional and the message has no acknowledgment, the
-- scenario is not locally observable, but can be fixed with an acknowledgment message.
public testLoop() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      ctrl = mkMessage(2, mk_(l2, 3), mk_(l1, 3), "m2"),
      o1 = mk_InteractionOperand(mk_InteractionConstraint(1, 2, nil),
             {mk_(l1, 1), mk_(l2, 1)}, {mk_(l1, 4), mk_(l2, 4)}),
      f1 = mk_CombinedFragment(<loop>, [o1], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1}, {f1})
  in
  (
    assertEqual({[s(m1), r(m1)], [s(m1), r(m1), s(m1), r(m1)],
                 [s(m1), s(m1), r(m1), r(m1)]}, validTraces(sd1));
    assertEqual({[s(m1), s(m1), r(m1)], [s(m1), r(m1), s(m1)]},
                 uncheckableLocally(sd1));
    assertTrue(isLocallyControllable(sd1));
    assertEqual("{{[!Ack_m1@L2Am1, ?Ack_m1@L1Am1]}}",
               genCoordinationMessagesStr2(sd1, false));
  )
);

-- Combined scenario with nested 'alt' combined fragments.
-- It is locally observable and controllable.
public testAltNested() ≜
(
  let l1 = mkLifeline("User"),
      l2 = mkLifeline("Watch"),
      l3 = mkLifeline("Smartphone"),
      l4 = mkLifeline("WebServer"),
      o11 = mk_InteractionOperand(nil, {mk_(l1,2), mk_(l2, 2), mk_(l3, 2), mk_(l4, 2)},
                                       {mk_(l1,4), mk_(l2, 4), mk_(l3, 4), mk_(l4, 4)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4), mk_(l4, 4)},
                                     {mk_(l1,14), mk_(l2,14), mk_(l3,14), mk_(l4,14)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3, l4}),
      o21 = mk_InteractionOperand(nil, {mk_(l2, 6), mk_(l3, 6), mk_(l4, 6)},
                                       {mk_(l2, 8), mk_(l3, 8), mk_(l4, 8)}),
      o22 = mk_InteractionOperand(nil, {mk_(l2, 8), mk_(l3, 8), mk_(l4, 8)},
                                       {mk_(l2, 12), mk_(l3, 12), mk_(l4, 12)}),
      f2 = mk_CombinedFragment(<alt>, [o21, o22], {l2, l3, l4}),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
```

```
      m2 = mkMessage(2, mk_(l2, 3), mk_(l1, 3), "m2"),
      m3 = mkMessage(3, mk_(l2, 5), mk_(l3, 5), "m3"),
      m4 = mkMessage(4, mk_(l3, 7), mk_(l2, 7), "m4"),
      m5 = mkMessage(5, mk_(l3, 9), mk_(l4, 9), "m5"),
      m6 = mkMessage(6, mk_(l4, 10), mk_(l3, 10), "m6"),
      m7 = mkMessage(7, mk_(l3, 11), mk_(l2, 11), "m7"),
      m8 = mkMessage(8, mk_(l2, 13), mk_(l1, 13), "m8"),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2, m3, m4, m5, m6, m7, m8}, {f1, f2})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)],
                 [s(m1), r(m1), s(m3), r(m3), s(m4), r(m4), s(m8), r(m8)],
                 [s(m1), r(m1), s(m3), r(m3), s(m5), r(m5), s(m6), r(m6), s(m7), r(m7),
                  s(m8), r(m8)]},
             validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertTrue(isLocallyControllable(sd1));
  )
);

-- Scenario with a race condition.
-- It is not locally controllable, but can be enforced wih a coordination message.
public testRace() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l3, 2), mk_(l2, 2), "m2"),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2}, ∅)
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)], [s(m1), s(m2), r(m1), r(m2)],
                 [s(m2), s(m1), r(m1), r(m2)]}, validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertEqual({[s(m1), s(m2), r(m2)], [s(m2), s(m1), r(m2)], [s(m2), r(m2)]},
                unintendedTraces(sd1));
    assertEqual(∅, missingTraces(sd1));
    assertEqual("{{[!Ctrl1@L2Am1, ?Ctrl1@L3Bm2]}}",
                genCoordinationMessagesStr2(sd1, false));
  )
);

-- Another race condition, with 3 messages.
-- Corresponds to motivating example a).
-- Not locally controllable, but can be enforced with a coordination message
-- or a set of coordination time constraints.
public testRaceReceiveReceive() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l2, 2), mk_(l3, 2), "m2"),
      m3 = mkMessage(3, mk_(l1, 4), mk_(l3, 4), "m3"),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3}, ∅)
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)],
```

```
                 [s(m1), r(m1), s(m2), s(m3), r(m2), r(m3)],
                 [s(m1), r(m1), s(m3), s(m2), r(m2), r(m3)],
                 [s(m1), s(m3), r(m1), s(m2), r(m2), r(m3)]}, validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertEqual({[s(m1), r(m1), s(m2), s(m3), r(m3)],
                 [s(m1), r(m1), s(m3), s(m2), r(m3)],
                 [s(m1), r(m1), s(m3), r(m3)],
                 [s(m1), s(m3), r(m1), s(m2), r(m3)],
                 [s(m1), s(m3), r(m1), r(m3)],
                 [s(m1), s(m3), r(m3)]}, unintendedTraces(sd1));
    assertEqual("{{[!Ctrl1@L3Am2, ?Ctrl1@L1Am1]}}",
                genCoordinationMessagesStr2(sd1, false));
    assertEqual("{({}, {!m2@L2 - ?m1@L2 <= 1, ?m1@L2 - !m1@L1 <= 1, 4 <= !m3@L1 -
!m1@L1, ?m2@L3 - !m2@L2 <= 1})}",
                genCoordinationFeaturesStr(sd1, false, true, false, true, 1, 1));
  )
);

-- Similar to previous scenario, followed by 'alt' combined fragment,
-- with multiple violations of local observability and controllability.
-- Local controllability can be enforced with coordination time constraints.
public testRacePlusAlt() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l2, 2), mk_(l3, 2), "m2"),
      m3 = mkMessage(3, mk_(l1, 3), mk_(l3, 3), "m3"),
      m4 = mkMessage(4, mk_(l1, 5), mk_(l2, 5), "m4"),
      m5 = mkMessage(5, mk_(l2, 7), mk_(l3, 7), "m5"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)}, {mk_(l1, 6),
mk_(l2, 6), mk_(l3, 6)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 6), mk_(l2, 6), mk_(l3, 6)}, {mk_(l1, 8),
mk_(l2, 8), mk_(l3, 8)}),
      f1 = mk_CombinedFragment(<alt>, [o1, o2], {l1, l2, l3}),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4, m5}, {f1})
  in
  (
    assertEqual("{({}, {!m2@L2 - ?m1@L2 <= 1, ?m1@L2 - !m1@L1 <= 2, 6 <= !m3@L1 - !m1@L1
<= 7, ?m2@L3 - !m2@L2 <= 2, 11 <= !m5@L2 - !m2@L2, ?m3@L3 - !m3@L1 <= 2, !m4@L1 - !m3@L1
<= 1, ?m4@L2 - !m4@L1 <= 2})}",
      genCoordinationFeaturesStr(sd1, false, true, false, true, 2, 1));
  )
);

-- Yet another race condition, causing a local controllability problem
-- that can be solved with a coordination message or coordination time constraints.
public testRaceSendReceive() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l3, 1), "m1"),
      m2 = mkMessage(2, mk_(l3, 2), mk_(l2, 2), "m2"),
      m3 = mkMessage(3, mk_(l1, 3), mk_(l3, 3), "m3"),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3}, ∅)
  in
  (
```

```
    assertTrue(isLocallyObservable(sd1));
    assertFalse(isLocallyControllable(sd1));
    assertEqual("{({[!Ctrl1@L3Am2, ?Ctrl1@L1Am1]}, {})}",
        genCoordinationFeaturesStr(sd1, true, true, true, false, 2, 1));
    assertEqual("{({}, {!m2@L3 - ?m1@L3 <= 1, ?m1@L3 - !m1@L1 <= 2, 4 <= !m3@L1 -
!m1@L1})}",
        genCoordinationFeaturesStr(sd1, false, true, false, true, 2, 1));
  )
);

-- Scenario with non-local choice causing observability and controllability problems
-- that can be solved with coordinations messages.
-- Corresponds to motivating example d).
public testNonLocalChoice() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      l4 = mkLifeline("L4"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1), mk_(l4, 1)},
                                       {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4, 3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4, 3)},
                                       {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5), mk_(l4, 5)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3, l4}),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 2), mk_(l4, 2), "m2"),
      m3 = mkMessage(3, mk_(l1, 4), mk_(l2, 4), "m3"),
      m4 = mkMessage(4, mk_(l3, 4), mk_(l4, 4), "m4"),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2, m3, m4}, {f1})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)], [s(m1), s(m2), r(m1), r(m2)],
                 [s(m1), s(m2), r(m2), r(m1)],
                 [s(m2), s(m1), r(m1), r(m2)], [s(m2), s(m1), r(m2), r(m1)],
                 [s(m2), r(m2), s(m1), r(m1)],
                 [s(m3), r(m3), s(m4), r(m4)], [s(m3), s(m4), r(m3), r(m4)],
                 [s(m3), s(m4), r(m4), r(m3)],
                 [s(m4), s(m3), r(m3), r(m4)], [s(m4), s(m3), r(m4), r(m3)],
                 [s(m4), r(m4), s(m3), r(m3)]},
            validTraces(sd1));
    assertFalse(isLocallyObservable(sd1));
    assertEqual({[s(m1), r(m1), s(m4)], [s(m1), s(m4)], [s(m4), s(m1)],
                 [s(m4), r(m4), s(m1)], [s(m3), r(m3), s(m2)], [s(m3), s(m2)],
                 [s(m2), s(m3)], [s(m2), r(m2), s(m3)]},
            unintendedTraces(sd1));
    assertEqual("{{[!Ctrl1@L1Am1, ?Ctrl1@L3Bm2], [!Ctrl2@L1Am3, ?Ctrl2@L3Am2]}}",
                genCoordinationMessagesStr2(sd1, false));
  )
);

-- Race condition caused by message overtaking (allowed in UML standard).
public testRaceByMsgOvertaking() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(3, mk_(l1, 3), mk_(l2, 3), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, ∅)
  in
```

```
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)], [s(m1), s(m2), r(m1), r(m2)]},
                validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertEqual({[s(m1), s(m2), r(m2)]}, unintendedTraces(sd1));
    assertEqual("{({[!Ctrl1@L2Am1, ?Ctrl1@L1Am1]}, {})}",
         genCoordinationFeaturesStr(sd1, true, true, true, false, 2, 1));
    assertEqual("{({}, {?m1@L2 - !m1@L1 <= 2, 3 <= !m2@L1 - !m1@L1})}",
         genCoordinationFeaturesStr(sd1, false, true, false, true, 2, 1));
  )
);

-- Scenario with mutually exclusive emission and reception events
-- simultaneously enabled, involving 3 lifelines.
-- Can be enforced with coordination messages or coordination time constraints.
public testSendRecvEnabled() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2), mk_(l3, 2)},
                                       {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)},
                                       {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3}),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l2, 3), mk_(l1, 3), "m2"),
      m3 = mkMessage(3, mk_(l1, 6), mk_(l3, 6), "m3"),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3}, {f1})
  in
  (
    assertFalse(isLocallyObservable(sd1));
    assertFalse(isLocallyControllable(sd1));
    assertEqual("{({[!Ack_m3@L3Am3, ?Ack_m3@L1Am3], [!Ctrl1@L2Am2, ?Ctrl1@L1Am2]},
{})}",
         genCoordinationFeaturesStr(sd1, true, true, true, false, 2, 1));
    assertEqual("{({}, {!m2@L2 - ?m1@L2 <= 1, ?m1@L2 - !m1@L1 <= 2, 6 <= !m3@L1 -
!m1@L1, ?m2@L1 - !m2@L2 <= 2})}",
         genCoordinationFeaturesStr(sd1, false, true, false, true, 2, 1));
  )
);

-- Another scenario with mutually exclusive emission and reception events
-- simultaneously enabled, with 2 lifelines.
-- Can be enforced with coordination messages or coordination time constraints.
-- Corresponds to motivating example f).
public testSendRecvEnabled2() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2)},
                                       {mk_(l1, 4), mk_(l2, 4)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4)},
                                       {mk_(l1, 6), mk_(l2, 6)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2}),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessage(2, mk_(l2, 3), mk_(l1, 3), "m2"),
      m3 = mkMessage(3, mk_(l1, 5), mk_(l2, 5), "m3"),
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3}, {f1})
```

```
  in
  (
    assertFalse(isLocallyObservable(sd1));
    assertFalse(isLocallyControllable(sd1));
    assertEqual("{({[!Ctrl1@L2Am2, ?Ctrl1@L1Am2]}, {})}",
          genCoordinationFeaturesStr(sd1, true, true, true, false, 2, 1));
    assertEqual("{({}, {!m2@L2 - ?m1@L2 <= 1, ?m1@L2 - !m1@L1 <= 2, 6 <= !m3@L1 -
!m1@L1, ?m2@L1 - !m2@L2 <= 2})}",
          genCoordinationFeaturesStr(sd1, false, true, false, true, 2, 1));
  )
);

-- A scenario with two alternative messages unrelated (with 4 lifelines),
-- causing local observability and controllability problems.
public testUnintendedEmptyTrace() ≙
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      l4 = mkLifeline("L4"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1), mk_(l4, 1)},
                                       {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4, 3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3), mk_(l4, 3)},
                                       {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5), mk_(l4, 5)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3, l4}),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 4), mk_(l4, 4), "m2"),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2}, {f1}),

      e1 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m2", l3),
      e4 = mkEvent(<Receive>, "m2", l4)
  in
  (
    assertEqual({[s(m1), r(m1)], [s(m2), r(m2)]}, validTraces(sd1));
    assertFalse(isLocallyObservable(sd1));
    assertFalse(isLocallyControllable(sd1));
    assertEqual("{{[!Ctrl1@L1Am1, ?Ctrl1@L3Bm2]}}",
       genCoordinationMessagesStr2(sd1, true /*controllability only*/));
    assertEqual("{{[!Ack_Ctrl1@L3Bm2, ?Ack_Ctrl1@L1Am1], [!Ack_m1@L2Am1, ?Ack_m1@L1Am1],
[!Ack_m2@L4Am2, ?Ack_m2@L3Am2], [!Ctrl1@L1Am1, ?Ctrl1@L3Bm2]}}",
       genCoordinationMessagesStr2(sd1, false));
  )
);

-- A scenario with two alternative messages partially unrelated (with 3 lifelines),
-- causing local observability and controllability problems.
public testUnintendedEmptyTrace2() ≙
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1)},
                                       {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)},
                                       {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2, l3}),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
```

```
      m2 = mkMessage(2, mk_(l3, 4), mk_(l2, 4), "m1"),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2}, {f1})
  in
  (
    assertEqual({[s(m1), r(m1)], [s(m2), r(m2)]}, validTraces(sd1));
    assertFalse(isLocallyObservable(sd1));
    assertFalse(isLocallyControllable(sd1));
    assertEqual("{{[!Ctrl1@L1Am1, ?Ctrl1@L3Bm1]}}",
                genCoordinationMessagesStr2(sd1, true));
  )
);

-- Example with unintendedTraces with invalidStop but not other problems
-- (at least one sends).
public testUnintendedEmptyTrace3() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1)},
                                       {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)},
                                       {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5)}),
      o13 = mk_InteractionOperand(nil, {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5)},
                                       {mk_(l1, 11), mk_(l2, 11), mk_(l3, 11)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12, o13], {l1, l2, l3}),
      o21 = mk_InteractionOperand(nil, {mk_(l2, 6)}, {mk_(l2, 8)}),
      o22 = mk_InteractionOperand(nil, {mk_(l2, 8)}, {mk_(l2, 10)}),
      f2 = mk_CombinedFragment(<par>, [o21, o22], {l2}),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 4), mk_(l2, 4), "m1"),
      m3 = mkMessage(3, mk_(l1, 7), mk_(l2, 7), "m1"),
      m4 = mkMessage(4, mk_(l3, 9), mk_(l2, 9), "m1"),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4}, {f1, f2}),

      e1 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m1", l3)
  in
  (
    assertEqual({[e1, e2], [e3, e2], [e1, e2, e3, e2], [e3, e2, e1, e2],
                 [e3, e1, e2, e2], [e1, e3, e2, e2]}, validTraces(sd1));
    assertEqual({[e1, e3, e2], [e3, e1, e2], [e3, e2, e1], [e1, e2, e3]},
                uncheckableLocally(sd1));
    assertEqual({[]}, unintendedTraces(sd1));
    -- violates assumption of biunivoc relation between send and receive events
  )
);

-- Scenario with two alternative messages in opposite directions.
public testWhoSends() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      o11 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1)},
                                       {mk_(l1, 3), mk_(l2, 3)}),
      o12 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3)},
                                       {mk_(l1, 5), mk_(l2, 5)}),
      f1 = mk_CombinedFragment(<alt>, [o11, o12], {l1, l2}),
```

```
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l2, 4), mk_(l1, 4), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {f1}),

      e1 = mkEvent(<Send>, "m1", l1),
      e2 = mkEvent(<Receive>, "m1", l2),
      e3 = mkEvent(<Send>, "m2", l2),
      e4 = mkEvent(<Receive>, "m2", l1)
  in
  (
    assertEqual({[s(m1), r(m1)], [s(m2), r(m2)]}, validTraces(sd1));
    assertEqual({[s(m1), s(m2)], [s(m2), s(m1)]}, uncheckableLocally(sd1));
    assertEqual({[], [s(m1), s(m2)], [s(m2), s(m1)]}, unintendedTraces(sd1));
    assertEqual(∅, missingTraces(sd1)) ;
    assertEqual("{{[!Ctrl1@L1Am1, ?Ctrl1@L2Am1]}}",
                genCoordinationMessagesStr2(sd1, false));
  )
);

-- Conformance checking of timed traces showing different verdicts.
-- The scenario is not time constrained.
public testVerdictWithTimestamps() ≙
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      m1 = mkMessage(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessage(2, mk_(l3, 4), mk_(l2, 4), "m2"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1), mk_(l3, 1)},
                                  {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)},
                                  {mk_(l1, 5), mk_(l2, 5), mk_(l3, 5)}),
      f1 = mk_CombinedFragment(<strict>, [o1, o2], {l1, l2, l3}),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2}, {f1}),

      te1 = mkEvent(<Send>, "m1", l1, 10),
      te2 = mkEvent(<Receive>, "m1", l2, 20),
      te3a = mkEvent(<Send>, "m2", l3, 20 + MaxClockSkew),
      te3b = mkEvent(<Send>, "m2", l3, 20 + MaxClockSkew + 1),
      te3c = mkEvent(<Send>, "m2", l3, 20 - MaxClockSkew - 1),
      te3d = mkEvent(<Send>, "m2", l3, 20 - MaxClockSkew),
      te4 = mkEvent(<Receive>, "m2", l2, 20 + MaxClockSkew + 2)
  in
  (
    assertEqual(<Inconclusive>, finalConformanceChecking(sd1,
                                {l1 ↦ [s(m1)], l2 ↦ [r(m1), r(m2)], l3 ↦ [s(m2)]}));
    assertEqual(<Inconclusive>, timedFinalConformanceChecking(sd1,
                                {l1 ↦ [te1], l2 ↦ [te2, te4], l3 ↦ [te3a]}));
    assertEqual(<Pass>, timedFinalConformanceChecking(sd1,
                                {l1 ↦ [te1], l2 ↦ [te2, te4], l3 ↦ [te3b]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
                                {l1 ↦ [te1], l2 ↦ [te2, te4], l3 ↦ [te3c]}));
    assertEqual(<Inconclusive>, timedFinalConformanceChecking(sd1,
                                {l1 ↦ [te1], l2 ↦ [te2, te4], l3 ↦ [te3d]}));
  )
);


/*************** Test scenarios with time constraints ***************/
```

```
-- Scenario with a non-locally controllable roundtrip time constraint.
public testRoundtripConstraint() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, ∅,
                        {mk_TimeConstraint(t(r(m1)), t(s(m2)), 0, 2),
                         mk_TimeConstraint(t(s(m1)), t(r(m2)), 0, 5)}),
      e1 = mk_Event(<Send>, "m1", l1, 1),
      e2 = mk_Event(<Receive>, "m1", l2, 2),
      e3 = mk_Event(<Send>, "m2", l2, 3), -- meets first constraint
      e4a = mk_Event(<Receive>, "m2", l1, 6), -- meets second constraint
      e4b = mk_Event(<Receive>, "m2", l1, 7) -- violates second constraint

  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
    assertEqual(∅, uncheckableLocally(sd1));
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, unintendedTraces(sd1)); -- cannot assure
2nd constraint
    assertEqual(∅, missingTraces(sd1)) ;

    assertTrue(timedCheckNextEvent([e1], e4a, projectTCTraces(validTimedTraces(sd1),
l1)));
    assertFalse(timedCheckNextEvent([e1], e4b, projectTCTraces(validTimedTraces(sd1),
l1)));

    assertEqual({mk_(s(m2), mk_(2, 4))},
           nextSendEventsTimed([e2], projectTCTraces(validTimedTraces(sd1), l2)));

    assertEqual(<Pass>, timedFinalConformanceChecking(sd1,
                                 {l1 ↦ [e1, e4a], l2 ↦ [e2, e3]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
                                {l1 ↦ [e1, e4b], l2 ↦ [e2, e3]}));

    assertEqual("{({}, {?m1@L2 - !m1@L1 <= 1, ?m2@L1 - !m2@L2 <= 1})}",
           genCoordinationMessagesStr(sd1, true, 1, 1));
  )
);

-- Scenario to check that, in the presence of multiple timed events refering to the same
-- timestamp variable (e.g., in a loop), it's the most recent occurrence that prevails.
public testTimeConstraintInLoop() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 2), mk_(l2, 2), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 3), mk_(l1, 3), "m2"),
      o1 = mk_InteractionOperand(mk_InteractionConstraint(1, 2, nil),
                      {mk_(l1, 1), mk_(l2, 1)}, {mk_(l1, 4), mk_(l2, 4)}),
      f1 = mk_CombinedFragment(<loop>, [o1], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, {f1},
                        {mk_TimeConstraint(t(r(m1)), t(s(m2)), nil, 2),
                         mk_TimeConstraint(t(s(m1)), t(r(m2)), nil, 5)}),

      te1 = mk_Event(<Send>, "m1", l1, 1),
```

```
      te2 = mk_Event(<Receive>, "m1", l2, 2),
      te3 = mk_Event(<Send>, "m2", l2, 3),
      te4a = mk_Event(<Receive>, "m2", l1, 6),
      te4b = mk_Event(<Receive>, "m2", l1, 7),

      te21 = mk_Event(<Send>, "m1", l1, 11),
      te22 = mk_Event(<Receive>, "m1", l2, 12),
      te23 = mk_Event(<Send>, "m2", l2, 13),
      te24a = mk_Event(<Receive>, "m2", l1, 16),
      te24b = mk_Event(<Receive>, "m2", l1, 17)
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)],
                 [s(m1), r(m1), s(m2), r(m2), s(m1), r(m1), s(m2), r(m2)]},
                validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertEqual({[s(m1), r(m1), s(m2), r(m2)],
                 [s(m1), r(m1), s(m2), r(m2), s(m1), r(m1), s(m2), r(m2)]},
                unintendedTraces(sd1));

    assertTrue(timedCheckNextEvent([te1], te4a,
                               projectTCTraces(validTimedTraces(sd1), l1)));
    assertFalse(timedCheckNextEvent([te1], te4b,
                                projectTCTraces(validTimedTraces(sd1), l1)));

    assertEqual({mk_(s(m2), mk_(2, 4))},
        nextSendEventsTimed([te2], projectTCTraces(validTimedTraces(sd1), l2)));

    assertEqual(<Pass>, timedFinalConformanceChecking(sd1,
                   {l1 ↦ [te1, te4a], l2 ↦ [te2, te3]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
                   {l1 ↦ [te1, te4b], l2 ↦ [te2, te3]}));
    assertEqual(<Pass>, timedFinalConformanceChecking(sd1,
                   {l1 ↦ [te1, te4a, te21, te24a], l2 ↦ [te2, te3, te22, te23]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
                   {l1 ↦ [te1, te4a, te21, te24b], l2 ↦ [te2, te3, te22, te23]}));

    assertEqual("{({}, {?m1@L2 - !m1@L1 <= 1, ?m2@L1 - !m2@L2 <= 1})}",
           genCoordinationMessagesStr(sd1, true, 1, 1));
  )
);

-- Scenario with intra and inter-lifeline time constraints, showing inconclusive
verdicts.
public testInterLifelineTimeConstraints() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, ∅,
                    {mk_TimeConstraint(t(s(m1)), t(r(m1)), 0, 2000),
                     mk_TimeConstraint(t(r(m1)), t(s(m2)), 0, 2000),
                     mk_TimeConstraint(t(s(m2)), t(r(m2)), 0, 2000),
                     mk_TimeConstraint(t(s(m1)), t(r(m2)), 0, 5000)}),

      te1 = mk_Event(<Send>, "m1", l1, 1000),
      te2a = mk_Event(<Receive>, "m1", l2, 2000),
      te2b = mk_Event(<Receive>, "m1", l2, 4000),
```

```
      te3 = mk_Event(<Send>, "m2", l2, 4000),
      te4a = mk_Event(<Receive>, "m2", l1, 6000 - 10),
      te4b = mk_Event(<Receive>, "m2", l1, 7000),
      te4c = mk_Event(<Receive>, "m2", l1, 6000)
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, unintendedTraces(sd1)); -- violating
time constraint s(m1)-r(m2)
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, uncheckableLocally(sd1)); -- because of
inter-lifeline constraints
    assertEqual(∅, missingTraces(sd1)) ;

    assertTrue(timedCheckNextEvent([te1], te4a, projectTCTraces(validTimedTraces(sd1),
l1)));
    assertFalse(timedCheckNextEvent([te1], te4b, projectTCTraces(validTimedTraces(sd1),
l1)));

    assertEqual({mk_(s(m2), mk_(2000, 4000))},
           nextSendEventsTimed([te2a], projectTCTraces(validTimedTraces(sd1), l2)));

    assertEqual(<Pass>, timedFinalConformanceChecking(sd1,
                {l1 ↦ [te1, te4a], l2 ↦ [te2a, te3]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
               {l1 ↦ [te1, te4a], l2 ↦ [te2b, te3]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
               {l1 ↦ [te1, te4b], l2 ↦ [te2a, te3]}));
    assertEqual(<Inconclusive>, timedFinalConformanceChecking(sd1,
               {l1 ↦ [te1, te4c], l2 ↦ [te2a, te3]}));
  )
);

-- Example of restricting valid traces based on time constraints.
public testIsLocallyObservableTimed() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),
      l4 = mkLifeline("L4"),
      m1 = mkMessageTimed(1, mk_(l2, 2), mk_(l1, 2), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 4), mk_(l3, 4), "m2"),
      m3 = mkMessageTimed(3, mk_(l3, 6), mk_(l4, 6), "m3"),
      m4 = mkMessageTimed(4, mk_(l4, 7), mk_(l3, 7), "m4"),
      m5 = mkMessageTimed(5, mk_(l3, 10), mk_(l2, 10), "m5"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 1), mk_(l2, 1)}, {mk_(l1, 3), mk_(l2,
3)}),
      o2 = mk_InteractionOperand(nil, {mk_(l3, 5), mk_(l4, 5)}, {mk_(l3, 8), mk_(l4,
8)}),
      o3 = mk_InteractionOperand(nil, {mk_(l2, 9), mk_(l3, 9)}, {mk_(l2, 11), mk_(l3,
11)}),
      f1 = mk_CombinedFragment(<opt>, [o1], {l1, l2}),
      f2 = mk_CombinedFragment(<opt>, [o2], {l3, l4}),
      f3 = mk_CombinedFragment(<opt>, [o3], {l2, l3}),
      sd1 = mkInteraction({l1, l2, l3, l4}, {m1, m2, m3, m4, m5}, {f1, f2, f3},
               {mk_TimeConstraint(t(s(m1)), t(r(m1)), nil, 1000),
                mk_TimeConstraint(t(s(m1)), t(s(m2)), 2000, nil),
                mk_TimeConstraint(t(r(m3)), t(s(m4)), 10000,nil),
                mk_TimeConstraint(t(s(m1)), t(r(m5)), nil, 5000)}),
      e1 = s(m1), e2 = r(m1),
```

```
      e3 = s(m2), e4 = r(m2),
      e5 = s(m3), e6 = r(m3),
      e7 = s(m4), e8 = r(m4),
      e9 = s(m5), e10 = r(m5)
  in
  (
    assertEqual({[e1, e2, e3, e4], [e1, e2, e3, e4, e5, e6, e7, e8], [e1, e2, e3, e4,
e9, e10],
      [e3, e4], [e3, e4, e5, e6, e7, e8],  [e3, e4, e5, e6, e7, e8, e9, e10], [e3, e4,
e9, e10]},
      validTraces(sd1));

    let t = [e1, e2, e3, e4, e5, e6, e7, e8, e9, e10] in
    (
      assertFalse(t ∈ uncheckableLocally(sd1));
      assertTrue(t ∈ uncheckableLocallyUntimed(sd1))
    );

    assertTrue({[e1, e3, e2, e4], [e1, e3, e4, e2]} ⊆ uncheckableLocally(sd1));
    assertTrue({[e1, e3, e2, e4], [e1, e3, e4, e2]} ⊆ uncheckableLocallyUntimed(sd1));
    assertEqual(∅, missingTraces(sd1));
  )
);

-- Example of non-controllability because of roundtrip constraint.
public testNonLocallyControlableTimed() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, ∅,
                          {mk_TimeConstraint(t(s(m1)), t(r(m2)), 0, 1000)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
    assertTrue(isLocallyObservable(sd1));
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, unintendedTraces(sd1));
    assertEqual(∅, missingTraces(sd1))
  )
);

-- Example in which two optional fragments become mutually exclusive
-- and controllable because of time constraints.
public testStrangeControllableTimed() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),

      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l1, 3), mk_(l2, 3), "m2"),
      m3 = mkMessageTimed(3, mk_(l2, 4), mk_(l1, 4), "m3"),
      m4 = mkMessageTimed(4, mk_(l2, 7), mk_(l1, 7), "m4"),

      o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2)}, {mk_(l1, 5), mk_(l2,
5)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 6), mk_(l2, 6)}, {mk_(l1, 8), mk_(l2,
8)}),
      f1 = mk_CombinedFragment(<opt>, [o1], {l1, l2}),
```

```
      f2 = mk_CombinedFragment(<opt>, [o2], {l1, l2}),

      sd1 = mkInteraction({l1, l2}, {m1, m2, m3, m4}, {f1, f2},
        mkMsgTimeConstraints({m1, m2, m3, m4}, 0, 1) ∪
        {mk_TimeConstraint(t(s(m1)), t(s(m2)), 7, nil),
         mk_TimeConstraint(t(r(m2)), t(s(m3)), 0, 1),
         mk_TimeConstraint(t(r(m1)), t(s(m4)), 0, 4),
         mk_TimeConstraint(t(s(m2)), t(r(m3)), 0, 5)}), -- roundtrip
      e1 = s(m1), e2 = r(m1),
      e3 = s(m2), e4 = r(m2),
      e5 = s(m3), e6 = r(m3),
      e7 = s(m4), e8 = r(m4)
  in
  (
    assertEqual({[e1, e2], [e1, e2, e3, e4, e5, e6], [e1, e2, e7, e8]},
                validTraces(sd1));
    assertTrue(isLocallyControllable(sd1));
    assertFalse(isLocallyObservable(sd1));
  )
);

-- Example of race condition solved with time constraints.
public testSendableFirst() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      l3 = mkLifeline("L3"),

      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l1, 2), mk_(l3, 2), "m2"),
      m3 = mkMessageTimed(3, mk_(l2, 3), mk_(l3, 3), "m3"),

      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3}, ∅,
            mkMsgTimeConstraints({m1, m2, m3}, 0, 1) ∪
            {mk_TimeConstraint(t(s(m1)), t(s(m2)), 2, 4),
             mk_TimeConstraint(t(r(m1)), t(s(m3)), 8, nil)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)]}, validTraces(sd1));
    assertTrue(isLocallyControllable(sd1));
    assertFalse(isLocallyObservable(sd1)); -- because of inter-lifeline constraints
  )
);

-- Example of controllability problem caused by mutually exclusive events
-- simultaneously enabled solved with the addition of time constraints.
public testSendableFirst2() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l1, 3), mk_(l2, 3), "m2"),
      m3 = mkMessageTimed(3, mk_(l2, 5), mk_(l1, 5), "m3"),
      m4 = mkMessageTimed(4, mk_(l1, 6), mk_(l2, 6), "m4"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2)}, {mk_(l1, 4), mk_(l2,
4)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4)}, {mk_(l1, 7), mk_(l2,
7)}),
      f1 = mk_CombinedFragment(<alt>, [o1, o2], {l1, l2}),
```

```
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3, m4}, {f1},
        mkMsgTimeConstraints({m1, m2, m3, m4}, 0, 1) ∪
        {mk_TimeConstraint(t(s(m1)), t(s(m2)), 2, 5),
         mk_TimeConstraint(t(r(m1)), t(s(m3)), 8, nil),
         mk_TimeConstraint(t(r(m3)), t(s(m4)), 0, 3),
         mk_TimeConstraint(t(s(m3)), t(r(m4)), 0, 5)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)],
                 [s(m1), r(m1), s(m3), r(m3), s(m4), r(m4)]},
                validTraces(sd1));
    assertTrue(isLocallyControllable(sd1));
  )
);

-- Example of intra-lifeline time constraint that causes controllability problems:
-- a maximum delay is defined between two send events, with an unconstrained event in
-- between (in this case, a reception event).
public testSendRecvSendConstraint() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      m3 = mkMessageTimed(3, mk_(l1, 3), mk_(l2, 3), "m3"),
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3}, ∅,
             {mk_TimeConstraint(t(s(m1)), t(s(m3)), nil, 5000)}),
      sd2 = mkInteraction({l1, l2}, {m1, m2, m3}, ∅,
             {mk_TimeConstraint(t(s(m1)), t(r(m1)), 0, 1000),
              mk_TimeConstraint(t(r(m1)), t(s(m2)), 0, 2000),
              mk_TimeConstraint(t(s(m2)), t(r(m2)), 0, 1000),
              mk_TimeConstraint(t(s(m1)), t(s(m3)), 0, 5000)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)]}, validTraces(sd1));
    assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)]}, validTraces(sd2));
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, unintendedTraces(sd1));
    assertEqual(∅, unintendedTraces(sd2));
  )
);

-- Similar to testSendRecvSendConstraint, but now with a send event in between.
public testSendSendSendConstraint() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l1, 2), mk_(l2, 2), "m2"),
      m3 = mkMessageTimed(3, mk_(l1, 3), mk_(l2, 3), "m3"),
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3}, ∅,
                {mk_TimeConstraint(t(s(m1)), t(s(m3)), 0, 6000)}),
      sd2 = mkInteraction({l1, l2}, {m1, m2, m3}, ∅, {
          mk_TimeConstraint(t(s(m1)), t(r(m1)), nil, 1000),
          mk_TimeConstraint(t(s(m2)), t(r(m2)), nil, 1000),
          mk_TimeConstraint(t(s(m1)), t(s(m2)), 2000, 3000),
          mk_TimeConstraint(t(s(m2)), t(s(m3)), 2000, 3000),
          mk_TimeConstraint(t(s(m1)), t(s(m3)), 0, 6000)})
  in
  (
```

```
    assertEqual({
      [s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)],
      [s(m1), r(m1), s(m2), s(m3), r(m2), r(m3)],
      [s(m1), s(m2), r(m1), r(m2), s(m3), r(m3)],
      [s(m1), s(m2), r(m1), s(m3), r(m2), r(m3)],
      [s(m1), s(m2), s(m3), r(m1), r(m2), r(m3)]}, validTraces(sd1));

    assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)]}, validTraces(sd2));

    let U = unintendedTraces(sd1) in (
      assertTrue([s(m1), s(m2), r(m2)] ∈ U); --message overtaking
      assertTrue([s(m1), r(m1), s(m2), r(m2)] ∉ U);-- invalid termination
      assertTrue([s(m1), s(m2), r(m1), r(m2)] ∉ U);-- invalid termination
    );

    assertEqual(∅, unintendedTraces(sd2));
  )
);

-- Test case for a bug fixed in the satisfiability checking algorithm.
public testBugFixCheckSatisfiability() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      m3 = mkMessageTimed(3, mk_(l1, 3), mk_(l2, 3), "m3"),
      m4 = mkMessageTimed(4, mk_(l1, 4), mk_(l2, 4), "m4"),
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3, m4}, ∅, {
          mk_TimeConstraint(t(r(m2)), t(s(m3)), 0, 1000),
          mk_TimeConstraint(t(r(m2)), t(s(m4)), 0, 1000),
          mk_TimeConstraint(t(s(m1)), t(s(m3)), 0, 10000),
          mk_TimeConstraint(t(s(m1)), t(s(m4)), 12000, nil)})
  in
  (
    assertEqual(∅, validTraces(sd1));
  )
);

-- Example of non-locally controllable round-trip constraint.
public testRcvConstraint() ≜
(
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "m2"),
      sd1 = mkInteraction({l1, l2}, {m1, m2}, ∅,
                          {mk_TimeConstraint(t(s(m1)), t(r(m2)), nil, 4000)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, validTraces(sd1));
    assertEqual({[s(m1), r(m1), s(m2), r(m2)]}, unintendedTraces(sd1));
    assertEqual(∅, missingTraces(sd1));
  )
);

-- Examples in which the system may remain in an invalid quiescent state.
public testMayRemainQuiescentTimed() ≜
(
```

```
  let l1 = mkLifeline("L1"),
      l2 = mkLifeline("L2"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "m1"),
      m2 = mkMessageTimed(2, mk_(l2, 3), mk_(l1, 3), "m2"),
      m3 = mkMessageTimed(3, mk_(l1, 5), mk_(l2, 5), "m3"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2)}, {mk_(l1, 4), mk_(l2,
4)}),
      f1 = mk_CombinedFragment(<opt>, [o1], {l1, l2}),
      sd1 = mkInteraction({l1, l2}, {m1, m2, m3}, {f1},
              {mk_TimeConstraint(t(s(m1)), t(r(m1)), nil, 1)}), --to force timed version
      sd2 = mkInteraction({l1, l2}, {m1, m2, m3}, {f1},
              {mk_TimeConstraint(t(r(m1)), t(s(m2)), nil, 2),
               mk_TimeConstraint(t(r(m1)), t(r(m3)), nil, 4)}),
      sd3 = mkInteraction({l1, l2}, {m1, m2, m3}, {f1},
              {mk_TimeConstraint(t(s(m1)), t(r(m1)), nil, 1),
               mk_TimeConstraint(t(r(m1)), t(s(m2)), nil, 2),
               mk_TimeConstraint(t(s(m2)), t(r(m2)), nil, 1),
               mk_TimeConstraint(t(s(m1)), t(s(m3)), 5, 6)})
  in
  (
    assertEqual({[s(m1), r(m1), s(m2), r(m2), s(m3), r(m3)],
                 [s(m1), r(m1), s(m3), r(m3)], [s(m1), s(m3), r(m1), r(m3)]},
                validTraces(sd1));
    assertTrue([s(m1), r(m1)] ∈ unintendedTraces(sd1));
    assertTrue([s(m1), r(m1)] ∈ unintendedTraces(sd2));
    assertEqual(∅, unintendedTraces(sd3));
    assertEqual(∅, missingTraces(sd1));
    assertEqual(∅, missingTraces(sd2));
    assertEqual(∅, missingTraces(sd3));
  )
);


/**************** Case studies ****************/

-- Fall detection scenario from an ambient-assisted living (AAL) ecosystem.
-- The scenario is locally controllable with the specified time constraints,
-- but would not in the absence of such constraints.
public testFallDetection() ≙
(
  let l1 = mkLifeline("Care_Receiver"),
      l2 = mkLifeline("Fall_Detection_App"),
      l3 = mkLifeline("AAL4ALL_Portal"),
      m1 = mkMessageTimed(1, mk_(l1, 1), mk_(l2, 1), "fall_signal"),
      m2 = mkMessageTimed(2, mk_(l2, 2), mk_(l1, 2), "confirm?"),
      m3 = mkMessageTimed(3, mk_(l1, 4), mk_(l2, 4), "yes"),
      m4 = mkMessageTimed(4, mk_(l2, 5), mk_(l3, 5), "notify_fall"),
      m5 = mkMessageTimed(5, mk_(l1, 7), mk_(l2, 7), "no"),
      m6 = mkMessageTimed(6, mk_(l2, 9), mk_(l3, 9), "notify_possible_fall"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 3), mk_(l2, 3), mk_(l3, 3)},
                                 {mk_(l1, 6), mk_(l2, 6), mk_(l3, 6)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 6), mk_(l2, 6), mk_(l3, 6)},
                                 {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8)}),
      o3 = mk_InteractionOperand(nil, {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8)},
                                 {mk_(l1, 10), mk_(l2, 10), mk_(l3, 10)}),
      f1 = mk_CombinedFragment(<alt>, [o1, o2, o3], {l1, l2, l3}),
      tcs =  {mk_TimeConstraint(t(s(m2)), t(r(m2)), 0, 1000),
                     mk_TimeConstraint(t(s(m3)), t(r(m3)), 0, 1000),
                     mk_TimeConstraint(t(s(m5)), t(r(m5)), 0, 1000),
```

```
                    mk_TimeConstraint(t(r(m2)), t(s(m3)), 0, 10000),
                    mk_TimeConstraint(t(r(m2)), t(s(m5)), 0, 10000),
                    mk_TimeConstraint(t(s(m2)), t(s(m6)), 13000, nil)},
    sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4, m5, m6}, {f1}, tcs),
    sd2 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4, m5, m6}, {f1}, ∅),
    e1 = s(m1), e2 = r(m1),
    e3 = s(m2), e4 = r(m2),
    e5 = s(m3), e6 = r(m3),
    e7 = s(m4), e8 = r(m4),
    e9 = s(m5), e10 = r(m5),
    e11 = s(m6), e12 = r(m6),

    e1a = mk_Event(<Send>, "fall_signal", l1, 0),
    e2a = mk_Event(<Receive>, "fall_signal", l2, 2000),
    e3a = mk_Event(<Send>, "confirm?", l2, 4000),
    e4a = mk_Event(<Receive>, "confirm?", l1, 4200),
    e5a = mk_Event(<Send>, "yes", l1, 14200),
    e6a = mk_Event(<Receive>, "yes", l2, 14500),
    e7a = mk_Event(<Send>, "notify_fall", l2, 14600),
    e8a = mk_Event(<Receive>, "notify_fall", l3, 16000),

    e6b = mk_Event(<Receive>, "yes", l2, 15200),
    e7b = mk_Event(<Send>, "notify_fall", l2, 15600),

    e6c = mk_Event(<Receive>, "yes", l2, 18000),
    e7c = mk_Event(<Send>, "notify_fall", l2, 18600),
    e8c = mk_Event(<Receive>, "notify_fall", l3, 19000),

    e4d = mk_Event(<Receive>, "confirm?", l1, 16800),
    e11d = mk_Event(<Send>, "notify_possible_fall", l2, 17000),
    e12d = mk_Event(<Receive>, "notify_possible_fall", l3, 18000)

  in
  (
    assertEqual({[e1, e2, e3, e4, e5, e6, e7, e8],
                 [e1, e2, e3, e4, e9, e10],
                 [e1, e2, e3, e4, e11, e12]},
                validTraces(sd1));

    MaxClockSkew := 500;
    assertEqual(<Pass>, timedFinalConformanceChecking(sd1,
                 {l1 ↦ [e1a, e4a, e5a], l2 ↦ [e2a, e3a, e6a, e7a], l3 ↦ [e8a]}));
    assertEqual(<Inconclusive>, timedFinalConformanceChecking(sd1,
                 {l1 ↦ [e1a, e4a, e5a], l2 ↦ [e2a, e3a, e6b, e7b], l3 ↦ [e8a]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
                {l1 ↦ [e1a, e4a, e5a], l2 ↦ [e2a, e3a, e6c, e7c], l3 ↦ [e8c]}));
    assertEqual(<Fail>, timedFinalConformanceChecking(sd1,
                {l1 ↦ [e1a, e4d], l2 ↦ [e2a, e3a, e11d], l3 ↦ [e12d]}));
    MaxClockSkew := 10;

    assertTrue(isLocallyControllable(sd1));
    assertFalse(isLocallyControllable(sd2));
    assertFalse(isLocallyObservable(sd1)); -- because of optional messages without ack
  )
);

-- Scenario of driving license renewal with a race condition.
public testDrivingLicenseRenewal() ≜
(
```

```
  let l1 = mkLifeline("DriverApp"),
      l2 = mkLifeline("DrivingLicenseIssuerApp"),
      l3 = mkLifeline("ElectronicPaymentService"),
      m1 = mkMessage(1, mk_(l1, 1), mk_(l2, 1), "requestDriverLicenseRenewal"),
      m2 = mkMessage(2, mk_(l2, 3), mk_(l1, 3), "paymentReference"),
      m3 = mkMessage(3, mk_(l2, 5), mk_(l3, 5), "referenceValidation"),
      m4 = mkMessage(4, mk_(l1, 9), mk_(l3, 9), "electronicPayment"),
      ctrl1 = mkMessage(5, mk_(l3, 7), mk_(l1, 7), "ctrl"),
      o1 = mk_InteractionOperand(nil, {mk_(l1, 2), mk_(l2, 2), mk_(l3, 2)},
                                   {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)}),
      o2 = mk_InteractionOperand(nil, {mk_(l1, 4), mk_(l2, 4), mk_(l3, 4)},
                                  {mk_(l1, 8), mk_(l2, 8), mk_(l3, 8)}),
      f1 = mk_CombinedFragment(<par>, [o1, o2], {l1, l2, l3}),
      sd1 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4}, {f1}),
      sd2 = mkInteraction({l1, l2, l3}, {m1, m2, m3, m4, ctrl1}, {f1})
  in
  (
     assertTrue(isLocallyObservable(sd1));
     assertTrue(isLocallyObservable(sd2));
     assertFalse(isLocallyControllable(sd1));
     assertTrue(isLocallyControllable(sd2));
     assertEqual("{{[!Ctrl1@ElectronicPaymentServiceAreferenceValidation,
?Ctrl1@DriverAppApaymentReference]}}",
                 genCoordinationMessagesStr2(sd1, false));
  )
);

-- Scenario from a paper in the International Journal of Parallel, Emergent and
-- Distributed Systems.
public testDistributedTestingPaper() ≙
(
  let actor1 = mkActor("Actor1"),
      port1 = mkLifeline("Port1"),
      actor2 = mkActor("Actor2"),
      port2 = mkLifeline("Port2"),
      actor3 = mkActor("Actor3"),
      port3 = mkLifeline("Port3"),

      L = {actor1, port1, actor2, port2, actor3, port3},

      a1 = mkMessageTimedSynch(1, mk_(actor1, 3), mk_(port1, 3), "a1"),
      x1 = mkMessageTimedSynch(2, mk_(port1, 5), mk_(actor1, 5), "x1"),
      i1 = mkMessageTimed(3, mk_(port1, 6), mk_(port2, 6), "i1"),
      y1 = mkMessageTimedSynch(4, mk_(port2, 8), mk_(actor2, 8), "y1"),

      b1 = mkMessageTimedSynch(5, mk_(actor2, 11), mk_(port2, 11), "b1"),
      y2 = mkMessageTimedSynch(6, mk_(port2, 13), mk_(actor2, 13), "y2"),
      i2 = mkMessageTimed(7, mk_(port2, 14), mk_(port1, 14), "i2"),
      x2a = mkMessageTimedSynch(8, mk_(port1, 16), mk_(actor1, 16), "x2a"),

      b2 = mkMessageTimedSynch(9, mk_(actor2, 18), mk_(port2, 18), "b2"),
      i3 = mkMessageTimed(10, mk_(port2, 19), mk_(port1, 19), "i3"),
      x2b = mkMessageTimedSynch(11, mk_(port1, 20), mk_(actor1, 20), "x2b"),

      c1 = mkMessageTimedSynch(12, mk_(actor3, 23), mk_(port3, 23), "c1"),
      z1 = mkMessageTimedSynch(13, mk_(port3, 25), mk_(actor3, 25), "z1"),
      i4 = mkMessageTimed(14, mk_(port3, 26), mk_(port1, 26), "i4"),
      x3 = mkMessageTimedSynch(15, mk_(port1, 28), mk_(actor1, 28), "x3"),
```

```
      c2 = mkMessageTimedSynch(16, mk_(actor3, 30), mk_(port3, 30), "c2"),
      z2 = mkMessageTimedSynch(17, mk_(port3, 32), mk_(actor3, 32), "z2"),

      i5 = mkMessageTimed(18, mk_(port3, 33), mk_(port2, 33), "i5"),
      y3 = mkMessageTimedSynch(19, mk_(port2, 34), mk_(actor2, 34), "y3"),
      i6 = mkMessageTimed(20, mk_(port3, 36), mk_(port1, 36), "i6"),
      x4 = mkMessageTimedSynch(21, mk_(port1, 38), mk_(actor1, 38), "x4"),

      synchMsgs = {a1, x1, y1 , b1, x2a, y2, b2, x2b, c1 , z1 , x3, c2 /*,z2,x4,y3*/},
      internalMsgs = {i1, i2 , i3, i4 /*, i5, i6*/},

      -- strict operands
      o1 = mk_InteractionOperand(nil, {mk_(a, 2) | a ∈ L}, {mk_(a, 9) | a ∈ L}),
      o2 = mk_InteractionOperand(nil, {mk_(a, 9) | a ∈ L}, {mk_(a, 22) | a ∈ L}),
      o3 = mk_InteractionOperand(nil, {mk_(a, 22) | a ∈ L}, {mk_(a, 29) | a ∈ L}),
      o4 = mk_InteractionOperand(nil, {mk_(a, 29) | a ∈ L}, {mk_(a, 39) | a ∈ L}),

      -- alt operands
      o5 = mk_InteractionOperand(nil, {mk_(a, 10) | a ∈ L}, {mk_(a, 17) | a ∈ L}),
      o6 = mk_InteractionOperand(nil, {mk_(a, 17) | a ∈ L}, {mk_(a, 21) | a ∈ L}),

      -- loop operand
      o7 = mk_InteractionOperand(mk_InteractionConstraint(1, 1, nil),
                                 {mk_(a, 1) | a ∈ L}, {mk_(a, 40) | a ∈ L}),

      f1 = mk_CombinedFragment(<strict>, [o1, o2, o3, o4], L),
      f2 = mk_CombinedFragment(<alt>, [o5, o6], L),
      f3 = mk_CombinedFragment(<sloop>, [o7], L),
      sd1 = mkInteraction(L, synchMsgs ∪ internalMsgs, {f1, f2, f3},
               mkMsgTimeConstraints(synchMsgs, nil, 0))
  in
  (
     assertEqual(2, # validTraces(sd1));
  )
);


/***********  Entry points *************/

public testAll() ≜
(
    -- Scenarios without time constraints
    testSimple();
    testImpossible();
    testIndepMessages();
    testOpt();
    testAlt();
    testStrict();
    testLoop();
    testAltNested();
    testRace();
    testRaceReceiveReceive();
    testRacePlusAlt();
    testRaceSendReceive();
    testNonLocalChoice();
    testRaceByMsgOvertaking();
    testSendRecvEnabled();
    testSendRecvEnabled2();
```

```
    testUnintendedEmptyTrace();
    testUnintendedEmptyTrace2();
    testUnintendedEmptyTrace3();
    testWhoSends();
    testVerdictWithTimestamps();

    -- Scenarios with time constraints
    testRoundtripConstraint();
    testTimeConstraintInLoop();
    testInterLifelineTimeConstraints();
    testVerdictWithTimestamps();
    testIsLocallyObservableTimed();
    testNonLocallyControlableTimed();
    testStrangeControllableTimed();
    testSendableFirst();
    testSendableFirst2();
    testSendRecvSendConstraint();
    testSendSendSendConstraint();
    testBugFixCheckSatisfiability();
    testRcvConstraint();
    testMayRemainQuiescentTimed();

    -- Case studies
    testFallDetection();
    testDrivingLicenseRenewal();
    testDistributedTestingPaper();
);

end TestCases
```

13. References

1. VDM-10 Language Manual, Peter Gorm Larsen *et al*, Overture Technical Report Series No. TR-001, Feb 2018
2. Local Observability and Controllability Analysis of Test Scenarios for Time-constrained Distributed Systems, Bruno Lima, FEUP, December 2020 (*PhD Thesis*)

References

AAL4ALL. AAL4ALL - Official Website, 2015. URL http://www.aal4all.org/.

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

S Akshay, Paul Gastin, Madhavan Mukund, and K Narayan Kumar. Checking conformance for time-constrained scenario-based specifications. *Theoretical Computer Science*, 594:24–43, 2015.

Nauman Bin Ali, Kai Petersen, and Mika Mäntylä. Testing highly complex system of systems: an industrial case study. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 211–220. ACM, 2012.

Shaukat Ali, Lionel C. Briand, Muhammad Jaffar-ur Rehman, Hajra Asghar, Muhammad Zohaib Z. Iqbal, and Aamer Nadeem. A State-based Approach to Integration Testing Based on UML Models. *Inf. Softw. Technol.*, 49(11-12):1087–1106, November 2007. ISSN 0950-5849. doi: 10.1016/j.infsof.2006.11.002. URL http://dx.doi.org/10.1016/j.infsof.2006.11.002.

Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

Salma Azzouzi, Sara Hsaini, and My El Hassan Charaf. A synchronized test control execution model of distributed systems. *International Journal of Grid and High Performance Computing (IJGHPC)*, 12(1):1–17, 2020.

Josip Babić. Model-Based Approach to Real-Time Embedded Control Systems Development with Legacy Components Integration. 2014.

Francesca Basanieri and Antonia Bertolino. A practical approach to UML-based derivation of integration tests. 2000.

Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

Sebastian Benz. Combining test case generation for component and integration testing. In *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, pages 23–33. ACM, 2007.

Antonia Bertolino, Eda Marchetti, and Andrea Polini. Integration of "Components" to test software components. *Electronic Notes in Theoretical Computer Science*, 82(6):44–54, 2003.

Antonia Bertolino, Eda Marchetti, and Henry Muccini. Introducing a reasonably complete and coherent approach for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116:85–97, 2005.

Matthias Beyer, Winfried Dulz, and Fenhua Zhen. Automated TTCN-3 test case generation by means of UML sequence diagrams and Markov chains. In *Test Symposium, 2003. ATS 2003. 12th Asian*, pages 102–105. IEEE, 2003.

Barry Boehm. Some Future Software Engineering Opportunities and Challenges. In Sebastian Nanz, editor, *The Future of Software Engineering*, pages 1–32. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-15186-6. doi: 10.1007/978-3-642-15187-3_1. URL http://dx.doi.org/10.1007/978-3-642-15187-3_1.

Sergiy Boroday, Alexandre Petrenko, and Andreas Ulrich. Implementing MSC Tests with Quiescence Observation. In *Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, TESTCOM '09/FATES '09, pages 49–65, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-05030-5. doi: 10.1007/978-3-642-05031-2_4.

Lionel C Briand, Yvan Labiche, and Yihong Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 287–296. IEEE, 2001.

Miroslav Bures, Tomas Cerny, and Bestoun S Ahmed. Internet of things: Current challenges in the quality assurance and testing methods. In *International Conference on Information Science and Applications*, pages 625–634. Springer, 2018.

Marco Canini, Vojin Jovanović, Daniele Venzano, Dejan Novaković, and Dejan Kostić. Online Testing of Federated and Heterogeneous Distributed Systems. *SIGCOMM Comput. Commun. Rev.*, 41(4):434–435, August 2011. ISSN 0146-4833. doi: 10.1145/2043164.2018507. URL http://doi.acm.org/10.1145/2043164.2018507.

Yuting Chen, Shaoying Liu, and Fumiko Nagoya. An approach to integration testing based on data flow specifications. In *Theoretical Aspects of Computing-ICTAC 2004*, pages 235–249. Springer, 2005.

Mind Commerce. Connected device market for consumer, enterprise, and industrial iot devices by use case, device type, application, region, and country 2020 - 2025, 2020.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

João António Custódio Soares, Bruno Lima, and João Pascoal Faria. Automatic model transformation from uml sequence diagrams to coloured petri nets. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, pages 668–679. SCITEPRESS-Science and Technology Publications, Lda, 2018.

Zhen Ru Dai, Jens Grabowski, Helmut Neukirchen, and Holger Pals. Model-based testing with UML applied to a roaming algorithm for Bluetooth devices. *Journal of Zhejiang University Science*, 5(11):1327–1335, 2004.

Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal methods in system design*, 19(1):45–80, 2001.

Karnig Derderian, Robert M Hierons, Mark Harman, and Qiang Guo. Estimating the feasibility of transition paths in extended finite state machines. *Automated Software Engineering*, 17(1): 33, 2010.

Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36. ACM, 2007.

DoD. Systems Engineering Guide for Systems of Systems. Technical report, Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems and Software Engineering Version 1.0, 2008.

E Durr and Jan Van Katwijk. Vdm++, a formal specification language for object-oriented designs. In *CompEuro'92.'Computer Systems and Software Engineering', Proceedings.*, pages 214–219. IEEE, 1992.

Stephen H Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2):97–111, 2001.

Christof Efkemann. *A Framework for Model-based Testing of Integrated Modular Avionics*. PhD thesis, Universität Bremen, 2014.

Kirill Fakhroutdinov. Bank atm, Jun 2013. URL https://www.uml-diagrams.org/bank-atm-uml-state-machine-diagram-example.html.

João Pascoal Faria and Ana C. R. Paiva. A toolset for conformance testing against UML sequence diagrams based on event-driven colored Petri nets. *International Journal on Software Tools for Technology Transfer*, pages 1–20, 2014. ISSN 1433-2779. doi: 10.1007/s10009-014-0354-x. URL http://dx.doi.org/10.1007/s10009-014-0354-x.

João Pascoal Faria, Bruno Lima, Tiago Boldt Sousa, and Angelo Martins. A Testing and Certification Methodology for an Open Ambient-Assisted Living Ecosystem. *International Journal of E-Health and Medical Communications (IJEHMC)*, 5(4):90–107, 2014. doi: 10.4018/ijehmc.2014100106.

John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005. ISBN 1852338814.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

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.

Ahmad Nauman Ghazi, Kai Petersen, and Jürgen Börstler. *Software Quality. Software and Systems Quality in Distributed and Mobile Environments: 7th International Conference, SWQD 2015,*

Vienna, Austria, January 20-23, 2015, Proceedings, chapter Heterogeneous Systems Testing Techniques: An Exploratory Survey, pages 67–85. Springer International Publishing, Cham, 2015. ISBN 978-3-319-13251-8. doi: 10.1007/978-3-319-13251-8_5. URL http://dx.doi.org/10.1007/978-3-319-13251-8_5.

Wolfgang Grieskamp. *Formal Approaches to Software Testing and Runtime Verification: First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, chapter Multi-paradigmatic Model-Based Testing, pages 1–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006a. ISBN 978-3-540-49703-5. doi: 10.1007/11940197_1.

Wolfgang Grieskamp. Multi-paradigmatic model-based testing. In *Formal Approaches to Software Testing and Runtime Verification*, pages 1–19. Springer, 2006b.

Hans-Gerhard Gross. *Component-Based Software Testing with UML*. Springer Berlin Heidelberg, 2005.

Hans-Gerhard Gross, Ina Schieferdecker, and George Din. Model-based built-in tests. *Electronic Notes in Theoretical Computer Science*, 111:161–182, 2005.

Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. UML-based integration testing. In *ACM SIGSOFT Software Engineering Notes*, volume 25, pages 60–70. ACM, 2000.

Philipp Helle and Wladimir Schamai. Towards an Integrated Methodology for the Development and Testing of Complex Systems-with Example. 2014.

Robert M. Hierons. Overcoming controllability problems in distributed testing from an input output transition system. *Distributed Computing*, 25(1):63–81, 2012. ISSN 1432-0452. doi: 10.1007/s00446-011-0153-5.

Robert M. Hierons. Combining Centralised and Distributed Testing. *ACM Trans. Softw. Eng. Methodol.*, 24(1):5:1–5:29, October 2014. ISSN 1049-331X. doi: 10.1145/2661296. URL http://doi.acm.org/10.1145/2661296.

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

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

IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990. doi: 10.1109/IEEESTD.1990.101064.

ISTQB. ISTQB Worldwide Software Testing Practices Report 2015-2016. Technical report, 2016. URL http://www.istqb.org/references/surveys/istqb-worldwide-software-testing-practices-report-2015-2016.html.

ISTQB. ISTQB Worldwide Software Testing Practices Report 2017-2018. Technical report, 2018. URL http://www.istqb.org/references/surveys/istqb-worldwide-software-testing-practices-report-2015-2016.html.

ISTQB. International Software Testing Qualifications Board. http://www.istqb.org/, March 2020.

Abu Zafer Javed, Paul A Strooper, and Geoffrey N Watson. Automated generation of test cases using model-driven architecture. In *Proceedings of the Second International Workshop on Automation of Software Test*, page 3. IEEE Computer Society, 2007.

Kurt Jensen, LarsMichael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007. ISSN 1433-2779. doi: 10.1007/s10009-007-0038-x. URL http://dx.doi.org/10.1007/s10009-007-0038-x.

Abdul Salam Kalaji, Robert Mark Hierons, and Stephen Swift. Generating feasible transition paths for testing from an extended finite state machine (efsm). In *2009 international conference on software testing verification and validation*, pages 230–239. IEEE, 2009.

Ahmed Khoumsi. A temporal approach for testing distributed systems. *IEEE Transactions on Software Engineering*, 28(11):1085–1103, 2002.

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

Mariam Lahami and Moez Krichen. Test isolation policy for safe runtime validation of evolvable software systems. In Sumitra Reddy and Mohamed Jmaiel, editors, *2013 Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 377–382. IEEE Computer Society, 2013.

Mariam Lahami, Fairouz Fakhfakh, Moez Krichen, and Mohamed Jmaiel. Towards a TTCN-3 test system for runtime testing of adaptable and distributed systems. In Brian Nielsen and Carsten Weise, editors, *24th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2012)*, volume 7641 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2012a.

Mariam Lahami, Moez Krichen, Mariam Bouchakwa, and Mohamed Jmaiel. Using knapsack problem model to design a resource aware test architecture for adaptable and distributed systems. In Brian Nielsen and Carsten Weise, editors, *24th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2012)*, volume 7641 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 2012b.

Peter G. Larsen, Kenneth Lausdahl, Nick Battle, John Fitzgeral, Sune Wolff, Shin Sahra, Marcel Verhoef, Peter Tran-Jørgensen, Tomohiro Oda, and Paul Chisholm. VDM-10 Language Manual. Technical report, 2016.

Johan Lilius and Iván Porres Paltor. Formalising uml state machines for model checking. In *International Conference on the Unified Modeling Language*, pages 430–444. Springer, 1999.

B. M. C. Lima and J. C. P. Faria. Towards decentralized conformance checking in model-based testing of distributed systems. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 356–365, March 2017. doi: 10.1109/ICSTW.2017.64.

Bruno Lima and Joao Pascoal Faria. An approach for automated scenario-based testing of distributed and heterogeneous systems. In *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 1, pages 1–10. IEEE, 2015.

Bruno Lima and João Pascoal Faria. *Software Technologies: 10th International Joint Conference, ICSOFT 2015, Colmar, France, July 20-22, 2015, Revised Selected Papers*, chapter Automated Testing of Distributed and Heterogeneous Systems Based on UML Sequence Diagrams, pages 380–396. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30142-6. doi: 10.1007/978-3-319-30142-6_21. URL http://dx.doi.org/10.1007/978-3-319-30142-6_21.

Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong, and Zheng Guo. Generating test cases from UML activity diagram based on gray-box method. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 284–291. IEEE, 2004.

Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI*, volume 8, pages 423–437, 2008.

Meihemutijiang Maimaiti. *Towards Development of Overture/VDM++ to Java Code Generator*. PhD thesis, Aarhus Universitet, Datalogisk Institut, 2011.

Marius Mikucionis, Kim G Larsen, and Brian Nielsen. T-uppaal: Online model-based testing of real-time systems. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 396–397. IEEE, 2004.

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

Harlan D Mills, Michael Dyer, and Richard C Linger. Cleanroom software engineering. 1987.

B. Mitchell. Resolving race conditions in asynchronous partial order scenarios. *IEEE Transactions on Software Engineering*, 31(9):767–784, Sept 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.104.

F. Moutai, S. Hsaini, S. Azzouzi, and M. E. Hassan Charaf. Testing distributed cloud: A case study. In *2019 International Symposium on Advanced Electrical and Communication Technologies (ISAECT)*, pages 1–5, 2019.

Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

Mikko Nieminen and Tomi Raty. Adaptable Design for Root Cause Analysis of a Model-Based Software Testing Process. In *Information Technology-New Generations (ITNG), 2015 12th International Conference on*, pages 379–384. IEEE, 2015.

OMG. OMG Unified Modeling LanguageTM (OMG UML) Version 2.5.1, Superstructure. Technical report, Object Management Group, 2017.

Oracle. Java SE 12, December 2019. URL https://www.oracle.com/technetwork/java/javase/overview/index.html.

Thomas Ostrand. White-Box Testing. *Encyclopedia of Software Engineering*, 2002.

Overture community. Overture Tool, 2020. URL https://www.overturetool.org/.

B Polgár, I Ráth, Z Szatmári, Á Horváth, and I Majzik. Model-based integration, execution and certification of development tool-chains. *Model Driven Tool and Process Integration*, 35, 2009.

Rudolf Ramler and Klaus Wolfmaier. Economic perspectives in test automation: Balancing automated and manual testing with opportunity cost. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 85–91, New York, NY, USA, 2006. ACM. ISBN 1-59593-408-1. doi: 10.1145/1138929.1138946. URL http://doi.acm.org/10.1145/1138929.1138946.

Sana Rao, Hosney Jahan, and Dongmei Liu. A search-based approach for test suite generation from extended finite state machines. In *2016 International Conference on Progress in Informatics and Computing (PIC)*, pages 82–87. IEEE, 2016.

Sacha Reis, Andreas Metzger, and Klaus Pohl. Integration testing in software product line engineering: a model-based technique. In *Fundamental Approaches to Software Engineering*, pages 321–335. Springer, 2007.

Debra J Richardson and Alexander L Wolf. Software testing at the architectural level. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, pages 68–71. ACM, 1996.

Michael Scheetz, Anneliese von Mayrhauser, Robert France, Eric Dahlman, and Adele E Howe. Generating test cases from an OO model with an AI planning system. In *ISSRE*, page 250. IEEE, 1999.

Stephan Schulz, Jukka Honkola, and Antti Huima. Towards model-based testing with architecture models. In *Engineering of Computer-Based Systems, 2007. ECBS'07. 14th Annual IEEE International Conference and Workshops on the*, pages 495–502. IEEE, 2007.

sepp.med gmbh. MBTsuite, 2020. URL https://mbtsuite.com/.

Avik Sinha and Carol Smidts. An experimental evaluation of a higher-ordered-typed-functional specification-based test-generation technique. *Empirical Software Engineering*, 11(2):173–202, 2006.

Dehla Sokenou et al. Generating Test Sequences from UML Sequence Diagrams and State Diagrams. In *GI Jahrestagung (2)*, pages 236–240. Citeseer, 2006.

STAF. Software Testing Automation Framework (STAF), December 2014. URL http://staf.sourceforge.net/.

QM Tani and A Petrenko. Input/output automata'. In *Testing of Communicating Systems: Proceedings of the IFIP TC6 11th International Workshop on Testing of Communicating Systems (IWTCS'98) August 31-September 2, 1998, Tomsk, Russia*, volume 3, page 83. Springer, 2013.

Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, National Institute of Standards and Technology, 2002.

TestLink Development Team. TestLink Open Source Test Management, 2020. URL http://testlink.org/.

C. Torens and L. Ebrecht. RemoteTest: A Framework for Testing Distributed Systems. In *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, pages 441–446, Aug 2010. doi: 10.1109/ICSEA.2010.75.

Jan Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.

Andreas Ulrich and Hartmut König. Architectures for Testing Distributed Systems. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Testing of Communicating Systems*, volume 21 of *IFIP — The International Federation for Information Processing*, pages 93–108. Springer US, 1999. ISBN 978-1-4757-6699-8. doi: 10.1007/978-0-387-35567-2_7. URL http://dx.doi.org/10.1007/978-0-387-35567-2_7.

Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software testing, verification and reliability*, 22(5):297–312, 2012.

Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal methods and testing*, pages 39–76. Springer, 2008.

Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

Jiacun Wang. *Timed Petri nets: Theory and application*, volume 9. Springer Science & Business Media, 2012.

Stephan Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines*. PhD thesis, Humboldt University of Berlin, 2010.

Sebastian Wieczorek, Vitaly Kozyura, Andreas Roth, Michael Leuschel, Jens Bendisposto, Daniel Plagge, and Ina Schieferdecker. Applying model checking to generate model-based integration tests from choreography models. In *Testing of Software and Communication Systems*, pages 179–194. Springer, 2009.

Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical research methods in software engineering. In *Empirical methods and studies in software engineering*, pages 7–23. Springer, 2003.

Ye Wu, Mei-Hwa Chen, and Jeff Offutt. UML-based integration testing for component-based software. In *COTS-Based Software Systems*, pages 251–260. Springer, 2003.

Dianxiang Xu and Weifeng Xu. State-based incremental testing of aspect-oriented programs. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 180–189. ACM, 2006.

Justyna Zander, Ina Schieferdecker, and Pieter J Mosterman. *Model-based testing for embedded systems*. CRC press, 2017.

Fuyuan Zhang, Zhengwei Qi, Haibing Guan, Xuezheng Liu, Mao Yang, and Zheng Zhang. FiLM: A Runtime Monitoring Tool for Distributed Systems. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 40–46, July 2009. doi: 10.1109/SSIRI.2009.55.

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